# Advanced JAVA(Main Part - 4)

**JDK8 Features**

java 1.7 ----> july 2011

|=> 3 years of hardwork by Oracle Community

java 1.8 ----> March 2014

Major Version of java earlier was JDK1.5 with features like

a. Annotations

b. Enum

c. Wrapper classes

d. foreach loop.....

After JDK1.5Version being the major development SUNMS didn't focussed on releasing major things in java. **But when java was sold to Oracle communicty**, Oracle people gave more importance for java to come up with "Major changes" in programming.

Before JDK1.8 => java ---------> Object Oriented Programming[Oops]

After JDK1.8 => java ---------> Object Oriented programming[Ooops, **Functional Aspects of Programming**]

**Features**

a. Lambda Expression

b. Functional Interfaces

c. Default Methods

d. Predicates,Supplier,Consumer

e. Double colon operation[::]

f. Stream API

g. Date and Time API

h. Optional API

**How to write Lambda Expression?**

While writing lambda expression

a. method name is not required

b. return type of method is not required

c. access modifier for a method is not required

d. if a body of a method contains only one instruction then {} is also optional.

e. No need to give datatype for parameters also.

f. If it contains only one arguments, then don't specify paranthesis() also.

**g. If a method is returning something then we need not use return keyword also to return the value.**

**Best way to understand lamda expression → just global search → +++++**

1. **Method with no parameters, no return type.**

```
public void m1()
{
    System.out.println("hello");
}
() -> System.out.println("hello");
```

2. **Method with parameters, no return type**

**a)-**

```
public void m1(int a, int b)
{
    System.out.println(a+b)
}
(a,b)-> System.out.println(a+b)
```

**b)-**

```
public void m1(String str)
{
    System.out.println(str.toUpperCase());
}
str -> System.out.println(str.toUpperCase());
```

3. **Method with parameters and return type**

```
public String m1(String str)
{
    return str.toUpperCase();
}
str-> str.toUpperCase();
```

**Note:**

1. Similar to method body,lamda expression can have single statements or multiple statements.
2. if multiple statements are there then we need to keep those statements under {}.
3. **After writing Lambda Expression,we can call that expression,but to call that Lambda Expression we need "FunctionalInterfaces".**

**Functional Interface**

=> If an interface conatins only one abstract method then such type of interfaces are called as "Functional interface".

=> To indicate an interface is "FunctionalInterface",they gave one Annotation **"@FunctionalInterface".**

**eg:: Runnable**

```
interface Runnable
{
    void run();
}
```

**eg:: Callable**

```
interface Callable
{
    T call();
}
```

**eg:: Comparable**

```
interface Comparable
{
    int compareTo();
}
```

**CaseStudies**

1. **Valid**

```
@FunctionalInterface
interface Interf
{
    void m1();
}
```

2. **In-Valid**

```
@FunctionalInterface
interface Interf
{
    void m1();
    void m2();
}
```

3. **In-Valid**

```
@FunctionalInterface
interface Interf
{
}
```

4. **Valid**

```
@FunctionalInterface
interface Interf1
{
    void m1();
}
@FunctionalInterface
interface Interf2 extends Interf1
{
}
```

5. **Valid**

```
@FunctionalInterface
interface Interf1
{
    void m1();
}
@FunctionalInterface
interface Interf2 extends Interf1
{
    void m1();
}
```

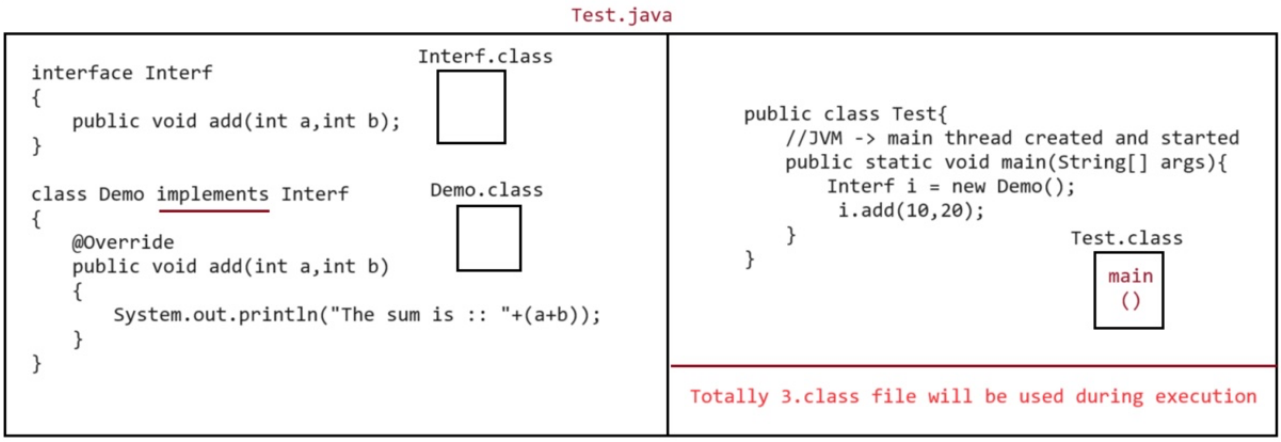
6. **In-Valid**

```
@FunctionalInterface
interface Interf1
{
    void m1();
}
```

```
@FunctionalInterface
interface Interf2 extends Interf1
{
    void m2();
}
```

7.**Valid**

```
@FunctionalInterface
interface Interf1
{
    void m1();
}
interface Interf2 extends Interf1
{
    void m2();
}
```




**three .class files will be created(not only two)**

**Working with Lambda Expression through Functional Interfaces → without the below approach we have to follow above mentioned approach(image) which includes three .class files.**

**1)- Why only on abstract method in Functional Interface??**

**Ans → because compiler will get confused with the following syntax -->**

**Interf i = (a,b)-> System.out.println("The sum is :: "+(a+b));**

**i.add(10,20);**

**if multiple abstract methods with same number of parameters are there.**

**eg::**

```
@FunctionalInterface
interface Interf
{
    public void add(int a,int b);
}
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args){
        //Interface called Interf add(int a,int b) :: void
        //Binding Lambda-Expression
        Interf i = (a,b)-> System.out.println("The sum is :: "+(a+b));
        i.add(10,20);
    }
}
```

**output**

**The sum is :: 30**

**2)- Only two .class files will be created for this program and it will require less processing, increasing the performance**

**Lambda Expression**

```
@FunctionalInterface
interface Interf
{
    public int square(int x);
}
class Demo implements Interf
{
    @Override
    public int square(int x)
    {
        return (x*x);
    }
}
public class Test{
```

```
    //JVM -> main thread created and started
    public static void main(String[] args){
        //Traditional Approach
        Interf i = new Demo();
        int result=i.square(5);
        System.out.println("The square is :: "+result);

        System.out.println();

        //Interface called Interf square(int x) :: void
        //Binding Lambda-Expression
        Interf i1 = x->x*x;
        System.out.println("The square is :: "+i1.square(100));
    }
}
```

**Output**

**The square is :: 25**

**The square is :: 10000**

**eg#2.**

```
/*
public interface java.lang.Runnable { //Runnable class also has only one abstract method
    public abstract void run();
}
*/
class MyRunnable implements Runnable
{
    @Override
    public void run()
    {
        //logic for thread
        for (int i = 0;i<10 ;i++ )
        {
            System.out.println("Child Thread....");
        }
    }
}
```

```
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        //Traditional Approach of working with OOPS
        Runnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();
        for (int i = 1;i<=10 ;i++ )
        {
            System.out.println("Main Thread");
        }

        System.out.println();

        System.in.read(); // after enter only it runs next set of instructions in console

        //New Approach in java :: Functional Programming
        //Interface called Runnable run() :: void
        //Binding Lambda-Expression
        Runnable r1=()->{
            for (int i =0;i<5 ;i++ )
            {
                System.out.println("Lambda Expression :: Thread");
            }
        };
        Thread t1 = new Thread(r1);
        t1.start();
        for (int i = 1;i<=10 ;i++ )
        {
            System.out.println("Main Thread");
        }
    }
}
```

**Anonymous Inner class ->**

**Inner classes**

```
class Outer
{
```

```
    //static varaibles
    //instance variables

    //blocks :: instance, static
    //methods :: instance, static

    class Inner
    {
    }
}
class Outer
{
    class Inner
    {
        public void m1()
        {
            System.out.println("From Inner class");
        }
    }
}
```

**Two classes will be formed →**

**Outer.class**

**Outer$Inner.class**

=> Sometimes we declare inner class without name such type of inner classes are called as **"Ananymous Inner class".**

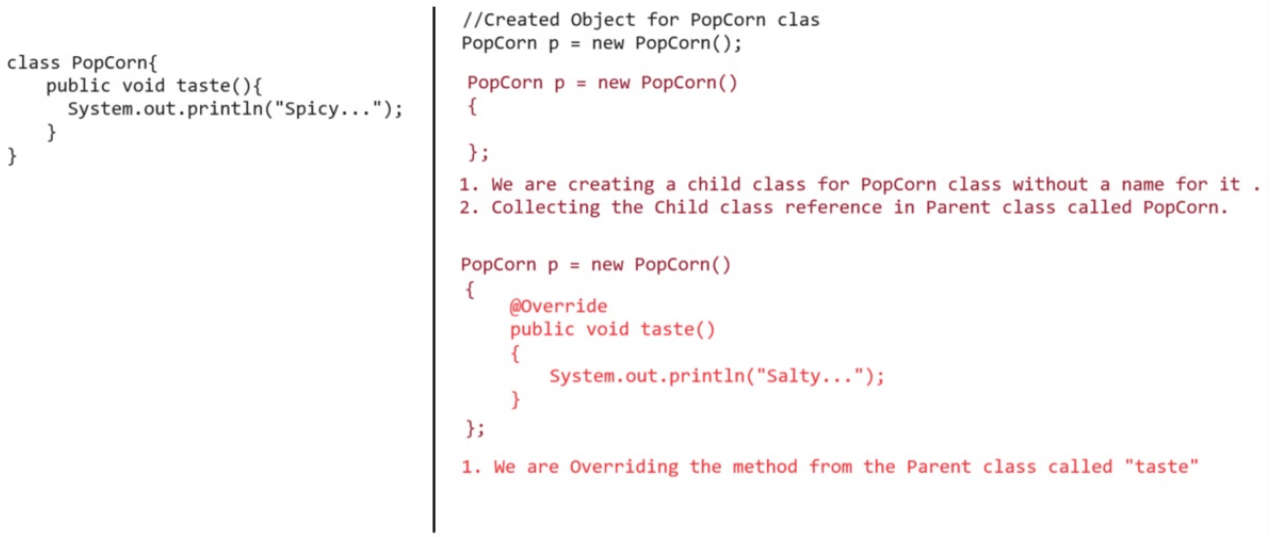
=> Main objective of Ananymous Inner class is "just for instance use".

=> There are 3 types of Ananymous Inner class

1. Anonymous inner class extends a class.
2. Anonymous inner class implements an interface.
3. Anonymous inner class that is defined inside method argument.

**If we have a class and it has a method(taste) with defination, now i want to change the implementation of the methods. This can be acheived by creating a new class and inside of which we can overide the method taste by creating its object.**

**ShotCut ->**




In above image P is a reference of parent class. Three .class will be created which are ->

1)- PopCorn.class

2)- Test$1.class **// Since the inner class have no name so this is how compiler shows that class which means the first inner class of Test class without the name.**

3)- Test.class

1. **Anonymous inner class extends a class.**

**eg#1.**

```
class PopCorn
{
    public void taste()
    {
        System.out.println("Spicy...");
    }
}
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        PopCorn p = new PopCorn()
        {
            @Override
            public void taste()
            {
                System.out.println("Salty...");
                brandName();   //child specific method can be called within the scope only
```

```
            }
            public void brandName()
            {
                System.out.println("MacD");
            }
        };
        p.taste();
        // p.brandName(); // will not work as p is the reference of parent and with it we cant access child specific methods.
        System.out.println();
        PopCorn p1 = new PopCorn();
        p1.taste();
    }
}
```

**output**

**Salty...**

**MacD**

**Spicy...**

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Try to understand lamda expression as →**

```
PopCorn p = new PopCorn()
{
    @Override
    public void taste()
    {
        System.out.println("Salty...");
        brandName();   //child specific method can be called within the scope only
    }
    public void brandName()
    {
        System.out.println("MacD");
    }
};
```

Now since writing new PopCorn() is something that even a compiler can do, so remove everything including method name and @override and then → PopCorn p =  () → {

```
                System.out.println("Salty...");
```

```
        brandName();
        }
```

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**eg#2.**

```
/*
    @FunctionalInterface
    interface Runnable
    {
        void run();
    }
    public class Thread implements Runnable
    {
    public void start(){
        1. register the thread with T.S
        2. perform low level activities
        3. invoke run()
    }
    @Override
    public void run(){
    }
}
*/
class MyThread extends Thread
{
    @Override
    public void run(){
        //logic for a thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("child thread...");
        }
    }
}
public class Test{
    //JVM -> main thread created and started
```

```
public static void main(String[] args)throws Exception{
    Thread t = new MyThread();
    t.start();
    //logic for main thread
    for (int i =0;i<5 ;i++ )
    {
        System.out.println("parent thread...");
    }

    System.out.println();
    System.in.read();

    Thread t1 =new Thread()
    {
        @Override
        public void run(){
            //logic for a thread
            for (int i =0;i<5 ;i++ )
            {
                System.out.println("Child thread::Anonymous Inner class");
            }
        }
    };
    t1.start();
    //logic for main thread
    for (int i =0;i<5 ;i++ )
    {
        System.out.println("Parent thread:: Ananymous Inner class");
    }

    System.out.println();
    System.in.read();

    Runnable r = ()->{
        //logic for a thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("Child thread::Lambda Expression");
        }
```

```
        };
        new Thread(r).start();
        //logic for main thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("Parent thread:: Lambda Expression");
        }
    }
}
```

**Output**

```
D:\OctBatchMicroservices>java Test
parent thread...
parent thread...
parent thread...
parent thread...
parent thread...
child thread...
child thread...
child thread...
child thread...
child thread...
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Child thread:: Anonymous Inner class
Child thread:: Anonymous Inner class
Child thread:: Anonymous Inner class
Child thread:: Anonymous Inner class
Child thread:: Anonymous Inner class
Parent thread:: Ananymous Inner class
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
```

Parent thread:: Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

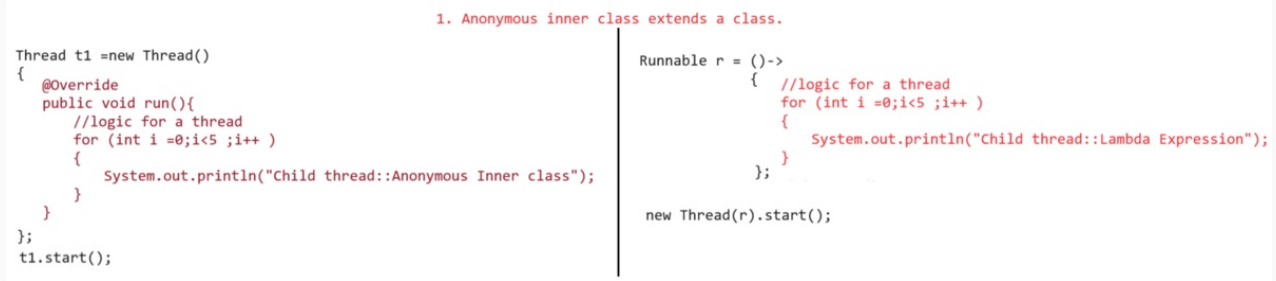
**Total classes →**

MyThread.class

Test$1.class

Test.class

**(no classes formed for lamda expressions.)**




2. **Anonymous inner class implements an interface.**

/*

@FunctionalInterface

interface Runnable

{

void run();

}

public class Thread implements Runnable

{

public void start(){

1. register the thread with T.S
2. perform low level activities
3. invoke run()

}

@Override

public void run(){

```
    }
}
*/
class MyRunnable implements Runnable
{
    @Override
    public void run(){
        //logic for a thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("child thread...");
        }
    }
}
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        Runnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();
        //logic for main thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("parent thread...");
        }

        System.out.println();
        System.in.read();

        /*
            here we are creating an object of a class which impements
            Runnable interface and there is no name for that class.
        */
        Runnable r1 = new Runnable()
        {
            @Override
            public void run()
            {
```

```
                //logic for a thread
                for (int i =0;i<5 ;i++ )
                {
                    System.out.println("Child thread::Ananymous Inner class");
                }
            }
        };
        new Thread(r1).start();
        //logic for main thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("Parent thread::Ananymous Inner class");
        }

        System.out.println();
        System.in.read();

        Runnable r2 = ()->{
            //logic for a thread
            for (int i =0;i<5 ;i++ )
            {
                System.out.println("Child thread::Lambda Expression");
            }
        };
        new Thread(r2).start();
        //logic for main thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("Parent thread:: Lambda Expression");
        }
    }
}
```

**Output**

```
D:\OctBatchMicroservices>java Test
parent thread...
parent thread...
parent thread...
parent thread...
```

parent thread...

child thread...

child thread...

child thread...

child thread...

child thread...

Parent thread:: Ananymous Inner class

Parent thread:: Ananymous Inner class

Parent thread:: Ananymous Inner class

Parent thread:: Ananymous Inner class

Child thread::Anonymous Inner class

Child thread::Anonymous Inner class

Child thread::Anonymous Inner class

Child thread::Anonymous Inner class

Child thread::Anonymous Inner class

Parent thread:: Ananymous Inner class

Parent thread:: Lambda Expression

Parent thread:: Lambda Expression

Parent thread:: Lambda Expression

Parent thread:: Lambda Expression

Parent thread:: Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

Child thread::Lambda Expression

1. Anonymous inner class extends a class.

```
Thread t1 =new Thread()
{
   @Override
   public void run(){
       //logic for a thread
       for (int i =0;i<5 ;i++ )
       {
           System.out.println("Child thread::Anonymous Inner class");
       }
   }
};
t1.start();
```

```
Runnable r = ()->
            {   //logic for a thread
                for (int i =0;i<5 ;i++ )
                {
                    System.out.println("Child thread::Lambda Expression");
                }
            };

new Thread(r).start();
```

2. Anonymous inner class implements an interface.

```
Runnable r1 = new Runnable()
{
   @Override
   public void run()
   {
       //logic for a thread
       for (int i =0;i<5 ;i++ )
       {
           System.out.println("Child thread::Ananymous Inner class");
       }
   }
};
new Thread(r1).start();
```

```
Runnable r = ()->
            {   //logic for a thread
                for (int i =0;i<5 ;i++ )
                {
                    System.out.println("Child thread::Lambda Expression");
                }
            };

new Thread(r).start();
```

3. **Anonymous inner class that is defined inside method argument(just compare 2nd and 3rd)**

```
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        new Thread(
        new Runnable()
        {
            @Override
            public void run()
            {
                //logic for a thread
                for (int i =0;i<5 ;i++ )
                {
                    System.out.println("Child thread::Ananymous Inner class");
                }
            }
        }).start();
        //logic for main thread
        for (int i =0;i<5 ;i++ )
        {
            System.out.println("Parent thread::Ananymous Inner class");
        }
    }
```

}

**Output**

Parent thread::Ananymous Inner class

Parent thread::Ananymous Inner class

Parent thread::Ananymous Inner class

Parent thread::Ananymous Inner class

Parent thread::Ananymous Inner class

Child thread::Ananymous Inner class

Child thread::Ananymous Inner class

Child thread::Ananymous Inner class

Child thread::Ananymous Inner class

Child thread::Ananymous Inner class

Anonymous inner class that is defined inside method argument

```
new Thread(
            new Runnable()
            {
                    @Override
                    public void run()
                    {
                            //logic for a thread
                            for (int i =0;i<5 ;i++ )
                            {
                                System.out.println("Child thread::Ananymous Inner class");
                            }
                    }
            }
        ).start();
```

**Working with Lambda Expression**

1. Inside lambda expression we can declare varaibles those variables are treated as local variables.
2. Within Lambda expression we can access instance variables of that class using "this" keyword.
3. Inside Lambda expression "this" would refer to Current class object.

**eg#1.**

```
@FunctionalInterface
    interface Interf
    {
        void m1();
    }
```

```
    public class Test{
        //instance variable
        int x= 777;
        //instance method
        public void m2()
        {
            Interf i = () -> {
                int x = 888;
                System.out.println(x);       //888
                System.out.println(this.x);  //777
            };
            i.m1();
        }
        //JVM -> main thread created and started
        public static void main(String[] args)throws Exception{
            new Test().m2();
    }
}
```

**eg#2.**

**Inside a method the variables are local variables, but if we write a lambda expression inside a method,then those local variables inside lambda expression will be treated as "final" variables, if we try to change the value it would result in "CE".()**

**Reason →**

**When you use a lambda expression inside a method, the compiler generates an instance of a functional interface, which represents the lambda expression's behavior. This instance may hold references to the local variables that are captured from the enclosing method's scope.**

**To ensure predictable behavior and avoid potential issues related to concurrent access and state changes, the Java compiler requires captured variables to be effectively final. This means that you can assign a value to the variable only once, and you cannot modify it afterwards.By enforcing this restriction, the compiler ensures that the captured variables behave consistently within the lambda expression, even if the lambda expression is executed asynchronously or in a multi-threaded context. It also helps prevent unintended side effects and makes the code easier to reason about.**

```
@FunctionalInterface
interface Interf
{
    void m1();
}
```

```
public class Test{
    //instance variable
    int x= 10;
    //instance method
    public void m2()
    {
        //local variable[Inside lambda they are final]
        int y = 20;
        Interf i = () -> {
            System.out.println(x);//10
            System.out.println(y);//20
            x = 100;
            System.out.println(x);//100
            y = 200;   //CE: y is final
            System.out.println(y);
        };
        i.m1();
        y = 200;
        System.out.println(y);//200
    }
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        new Test().m2();
    }
}
```

**Special features in interface from JDK1.8V**

**from chatgpt -->**

Default methods, also known as Defender methods or Virtual methods, were introduced in Java 8 primarily to provide support for backward compatibility when evolving interfaces. Let's delve deeper into how these methods accomplish this goal with an example.

Consider a scenario where you have an existing interface `Shape` that declares a method `calculateArea()`. Various classes such as `Circle`, `Rectangle`, and `Triangle` implement this interface, each providing its own implementation of the `calculateArea()` method.

```
interface Shape {
    double calculateArea();
}
```

```
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

}

class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }

}

// Similar implementation for Triangle, etc.`
```

**Now, let's say you want to add a new method `calculatePerimeter()` to the `Shape` interface. In Java versions before 8, adding a new method to an interface would break all classes implementing that interface, as they would be required to provide an implementation for the new method. This situation makes evolving interfaces challenging without causing widespread disruption.**

This is where default methods come into play. With default methods, you can provide a default implementation for the new method in the interface itself. Implementing classes that do not override this default method will automatically inherit the default implementation. This ensures backward compatibility with existing implementations.

Here's how you can modify the `Shape` interface to add the `calculatePerimeter()` method using a default method:

```
interface Shape {
    double calculateArea();

    // Default method for calculating perimeter
    default double calculatePerimeter() {
        return 0.0; // Default implementation, might not be meaningful
    }

}
```

Now, classes implementing the `Shape` interface will not be required to provide an implementation for the new `calculatePerimeter()` method unless they specifically want to customize its behavior. This ensures that existing code continues to work without modification, providing backward compatibility.

For example, if we have a class `Circle` implementing `Shape`, it can continue to work without any changes even after the addition of the `calculatePerimeter()` method to the `Shape` interface. It will inherit the default implementation provided by the interface.

```
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    // No need to provide implementation for calculatePerimeter()
}
```

In summary, default methods in interfaces enable the addition of new methods to interfaces without breaking existing implementations, thereby providing backward compatibility and facilitating the evolution of APIs. They offer a powerful tool for Java developers to extend

from video -->

**Till JDK1.7V**

a. Inside interface we can write

1. method -> by default they are public and abstract
2. variable -> by default they are public static final.

**From JDK1.8V**

=> It is possible to write concrete methods also in interface.

**=> To write concrete methods we need to use "default" keyword.**

=> These methods are called as **"Defeneder methods/Virtual methods**[To give support for Backward Compatibility]".

=> By writing these methods the implementation class won't get affected.

=> These methods will be available to implementation class object directly, if the child class is not happy with the implemenation then we can change the implemenation of default methods[means we can override the method].

=> These methods will have some dummy implementation which might be required for implementation class.

**eg#1.**

```
//JDK1.8V
@FunctionalInterface
interface Car
{
    public int noOfWheels();
    default void engineMake()
    {
        System.out.println("ENGINE MAKE GOOD FROM :: TATA");
    }
}
class Nexon implements Car
{
    @Override
    public int noOfWheels()
    {
        return 6;
    }
    @Override
    public void engineMake()
    {
        System.out.println("ENGINE MAKE GOOD FROM :: MARUTHI");
    }
}
public class Test{
    public static void main(String[] args){
        Car car = new Nexon();
        car.engineMake();
        int wheels=car.noOfWheels();
        System.out.println("No of wheels is :: "+wheels);
    }
}
```

**Output**

**ENGINE MAKE GOOD FROM :: MARUTHI**

**No of wheels is :: 6**

**eg#2.**

```
@FunctionalInterface
```

```
interface Car
{
    public int noOfWheels();
    default void engineMake()
    {
        System.out.println("ENGINE MAKE GOOD FROM :: TATA");
    }
    default String toString()
    {
        return "Hey Default method from Interface";
    }
}
```

**Note: Methods of object(like toString()) class will be by default available to every implementation class, so we should not bring those methods through "default" methods of an interface.**

**Note: Default methods in interface would lead to "DiamondShaped" problem in Multiple inheritance**

**eg#3**

```
interface Left{
    default void info(){
        System.out.println("From Left");
    }
}
interface Right{
    default void info(){
        System.out.println("From Right");
    }
}
class Demo implements Left,Right{}
public class TestApp {
    public static void main(String[] args) {
        Demo d =new Demo();
        d.info();//CE: ambiguity
    }
}
```

**Solution :: Compulsorily we need to override default method in implementation class.**

**Note: To get the facility of interface default methods in overdiden method we use the following syntaxinterfaceName.super.methodName();(just to make it different from class syntax as it is an interface.)**

**eg#4.**

```
interface Left{
    default void info(){
        System.out.println("From Left");
    }
}
interface Right{
    default void info(){
        System.out.println("From Right");
    }
}
class Demo implements Left,Right{
    @Override
    public void info()
    {
        Left.super.info();
        Right.super.info();
        System.out.println("From Implementation class...");
    }
}
public class TestApp {
    public static void main(String[] args) {
        Demo d =new Demo();
        d.info();
    }
    }
```

**Output**

**From Left**

**From Right**

**From Implementation class...**

**Conclusions :: Functional interface can have any no of default methods, but we need to have only one "abstract method".**

**static methods inside interface**

=> It is possible to write static methods inside interface.

=> These methods are called as "Helper/utility" methods.

**=> These methods by default won't be available to implementation class, to use this methods we need to use "InterfaceName".**

**=> Static methods won't be inherited to Implementation class, so Overriding is not possible.**

**eg#1.**

```
interface Vehicle
{
    //public abstract methods
    String getBrand();
    String speedUp();
    String speedDown();

    //default methods
    default String turnAlarmOn()
    {
        return "Turning the Vehicle alaram on...";
    }
    default String turnAlarmOff()
    {
        return "Turning the Vehicle alaram of...";
    }

    //static methods :: utility methods/helper methods
    public static void cleanVehicle()
    {
        System.out.println("Clean the Vehicle Properly....");
    }
}
class Car implements Vehicle
{
    private String brand;
    Car(String brand){
        this.brand = brand;
    }
    @Override
    public String getBrand(){
```

```
        return brand;
    }
    @Override
    public String speedUp(){
        return "The car is speeding up...";
    }
    @Override
    public String speedDown(){
        return "The car is speeding down...";
    }
}
public class TestApp {
    public static void main(String[] args) {
        Vehicle car= new Car("Nexon");

        //abstract methods
        System.out.println(car.getBrand());
        System.out.println(car.speedUp());
        System.out.println(car.speedDown());

        //default methods
        System.out.println(car.turnAlarmOn());
        System.out.println(car.turnAlarmOff());

        //Utility method
        Vehicle.cleanVehicle();
    }
}
```

**Nexon**

**The car is speeding up...**

**The car is speeding down...**

**Turning the Vehicle alaram on...**

**Turning the Vehicle alaram of...**

**Clean the Vehicle Properly...**

**Case1:**

interface Interf1

```
{
    public static void m1(){}
}
public class TestApp implements Interf1{
    @Override
    public static void m1(){}
    public static void main(String[] args) {
    }
}
```

**Output:: CE**

**Case2:**

```
interface Interf1
{
    public static void m1(){}
}
public class TestApp implements Interf1{
    @Override
    public void m1(){}
    public static void main(String[] args) {
    }
}
```

**Output :: CE**

**Case3::**

```
interface Interf1
{
    private void m1(){}
}
public class TestApp implements Interf1{
    @Override
    public void m1(){}
    public static void main(String[] args) {
    }
}
```

**Output: CE (private modifiers are not allowed)**

**Note: Since static methods can be a part of interface, we can write main method which is static inside the interface.(main includes the driving code,jvm doesn't looks for main method only inside of class it search for it in whole file.)**

```
interface Interf
{
    //utility methods :: JDK1.8
    public static void main(String[] args)
    {
        System.out.println("Main method in interface");
    }
}
```

**output**

**D:\OctBatchMicroservices>javac Interf.java**

**D:\OctBatchMicroservices>java Interf**

**Main method in interface**

**interfaces of jdk1.8V**

```
public interface java.util.function.Predicate
{
    public abstract boolean test(T);
    public default java.util.function.Predicate and(java.util.function.Predicate super T>);
    public default java.util.function.Predicate negate();
    public default java.util.function.Predicate or(java.util.function.Predicate super T>);
    public static java.util.function.Predicate isEqual(java.lang.Object);
    public static java.util.function.Predicate
    not(java.util.function.Predicate<? super T>);
}
```

**Note:**

It is an functional interface present in java.util.function package.

It is an interface which contains only one abstract method called test(T).

Since the interface Predicate is functional interface,we can invoke the method using "LambdaExpression".

**Write a Predicate to check whether the given integer is greater than 10 or not**

**1)- Traditional Approach(OOPS)**

```
public boolean test(Integer i)
{
    if(i>10)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

**2)- Lambda Expression**

i -> i > 10;  → equivalent to → **i → i > 10 ? return true : return false**

**eg#1.**

```
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        //Binded to test(Integer) :: boolean
        Predicate p = i->i>10;
        System.out.println("Result is :: "+p.test(10));
        System.out.println("Result is :: "+p.test(100));
        System.out.println("Result is :: "+p.test(-5));
        System.out.println("Result is :: "+p.test("sachin"));//CE
    }
}
```

**Output**

**Result is :: false**

**Result is :: true**

**Result is :: false**

**eg#2.**

Write a predicate to check the length of the given String is greater than 3 or not.

Input :: MI, RCB,CSK,LSG,RR

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

```
    public static void main(String[] args){
        //Binded to test(Integer) :: boolean
        Predicate p = name-> name.length() >=3;
        System.out.println("Result is :: "+p.test("MI"));
        System.out.println("Result is :: "+p.test("CSK"));
        System.out.println("Result is :: "+p.test("RCB"));
        System.out.println("Result is :: "+p.test("RR"));
    }
}
```

Output

**Result is :: false**

**Result is :: true**

**Result is :: true**

**Result is :: false**

**eg#3.**

**Write a predicate to check whether the given Array number is less than 18 or not?**

**int[] arr = {10,20,30,40,50,60};**

**eg#4.**

**Write a predicate to check whether the given Array elements are even in number or not?**

```
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        int[] arr = {0,5,10,15,20,25,30};
        //Binded to test(Integer) :: boolean
        Predicate p1= i -> i < 18;
        Predicate p2= i -> i%2 == 0;

        System.out.print("Numbers which are less than 18:: ");
        performOperation(p1,arr);
        System.out.println();

        System.out.print("Numbers which are even :: ");
        performOperation(p2,arr);
        System.out.println();
    }
```

```
    public static void performOperation(Predicate<Integer>p,int[] arr)
    {
        for (int data:arr)
        {
            if (p.test(data))
            {
                System.out.print(data+"\t");
            }
        }
        System.out.println();
    }
}
```

**Predicate Joining**

It is possible to join predicates into single predicate by using the following methods

a. and()

b. or()

c. negate()

These are exactly same as logical operators like AND,OR,NOT

**eg#1.**

```
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        int[] arr = {0,5,10,15,20,25,30};
        //Binded to test(Integer) :: boolean
        Predicate p1= i -> i < 18;
        Predicate p2= i -> i%2 == 0;

        System.out.print("Numbers which are less than 18:: ");
        performOperation(p1,arr);
        System.out.println();

        System.out.print("Numbers which are even :: ");
        performOperation(p2,arr);
        System.out.println();

        System.out.print("Numbers which are not less than 18:: ");
        performOperation(p1.negate(),arr);
```

```
        System.out.println();

        System.out.print("Numbers which are less than 18 and even numbers:: ");
        performOperation(p1.and(p2),arr);
        System.out.println();

        System.out.print("Numbers which are less than 18 or even numbers:: ");
        performOperation(p1.or(p2),arr);
        System.out.println();
    }
    public static void performOperation(Predicate p,int[] arr)
    {
        for (int data:arr)
        {
            if (p.test(data))
            {
                System.out.print(data+"\t");
            }
        }
        System.out.println();
    }
}
```

**Output**

**Numbers which are less than 18:: 0 5 10 15**

**Numbers which are even :: 0 10 20 30**

**Numbers which are not less than 18:: 20 25 30**

**Numbers which are less than 18 and even numbers:: 0 10**

**Numbers which are less than 18 or even numbers:: 0 5 10 15 20 30**

**Write a predicate to check whether the age of student is less than 30 or not?**

**eg#1.**

```
import java.util.function.*;
class Student
{
    private String name;
    private Integer age;
    Student(String name,Integer age)
```

```
    {
        this.name = name;
        this.age = age;
    }
    public String getName(){
        return name;
    }
    public Integer getAge(){
        return age;
    }
}
public class Test{
    public static void main(String[] args){
        Student[] std = new Student[3];
        Student std1 = new Student("sachin",25);
        Student std2 = new Student("dravid",31);
        Student std3 = new Student("kohli",28);
        std[0] = std1;
        std[1] = std2;
        std[2] = std3;

        Predicate p = student -> student.getAge()<30;
        performOperation(p,std);
    }
    public static void performOperation(Predicatep,Student[] students)
    {
        int count = 0;
        for (Student student: students )
        {
            if(p.test(student))
                count++;
        }
        System.out.println("No of students whose age is less < 30 is :: "+count);
    }
}
```

**Output**

**No of students whose age is less < 30 is :: 2**

**Function**

=> They are exactly same as Predicate, except that functions can return any type of Result.

=> It is present inside a package called "java.util.function.Function"

=> It contains only one method called " R apply(T)".

**=> Since Function is a FunctionalInterface, we can refer it through "Lambda Expression".**

```
public interface java.util.function.Function<T, R> {
    public abstract R apply(T);
    public default java.util.function.Function
    compose(java.util.function.Function<? super V, ? extends T>);
    public default java.util.function.Function
    andThen(java.util.function.Function<? super R, ? extends V>);
    public static java.util.function.Function identity();
}
```

Write a function to display the length of the given String?

```
public int apply(String data)
{
    int count=data.length();
    return count;
}
```

**Lambda Expression**

```
data-> data.length();
```

**eg#1.**

```
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        Function<String,Integer> f = s-> s.length();
        System.out.println("Length of the String is :: "+f.apply("dhoni"));
        System.out.println("Length of the String is :: "+f.apply("sachin"));
    }
}
```

**Output**

**Length of the String is :: 5**

**Length of the String is :: 6**

**Write a Function to get the result as half of the given supplied input?**

```
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        Function<Integer,Double> f = i->i/2.0;
        System.out.println(f.apply(10));
        System.out.println(f.apply(5));
    }
}
```

**Output**

**5.0**

**2.5**

**Chaining**

**eg#1.**

```
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        Function<String,Integer> f1 = s-> s.length();
        Function<Integer,Integer> f2 = i->i*2;
        System.out.println(f1.andThen(f2).apply("sachin")); //first f1 than f2 and value is from apply
        System.out.println(f1.andThen(f2).apply("RCB"));
    }
}
```

**Output**

**12**

**6**

**eg#2.**

```
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        Function<Integer,Double> f1 = i->i/2.0;
        Function<Integer,Integer> f2 = i->i*3;
        System.out.println(f1.compose(f2).apply(5)); //first f2 than f1 and value is from apply
    }
```

}

**Output**

**7.5**

**What is the difference b/w Predicate(I) vs Function(I)**

**=> Predicate**

1. To implement conditional check.
2. Predicate can take only argument of any Type.

   Predicate<T>
3. Predicate defines only one method called test()

   public boolean test(T t)
4. **Predicate returns only boolean value**

**=> Function**

1. To perform some operation and return some result.
2. Function can take 2 Parameters

   Function<T,R>

   T:: Input type

   R:: Output type
3. Function defines only one method called apply()

   public R apply(T t)
4. **Function can return any type of value.**

**Note:**

Lambda Expression would be used to refer to Functional Interface. **We have alternative solution for Lambda Expression that is "Method and Constructor Reference".**

**Method and Constructor reference**

=> It can be used as an alternative to Lambda Expression.

**=> Since it can be used as an alternative we can use this only for "Functional interface".**

=> We can map functional interface methods through

1. Lambda expression
2. Method reference[userdefined methods like instance and static]

**=> Functional interface method and our specific methods should have same argument types, except this remaining things like return-type, methodName,accessmodifier is not important.**

**Syntax for instance methods**

ojbName::methodName

**Syntax for static methods**

ClassName::methodName

**Logic using Lambda Expression**

```
Runnable r = ()->{
    for(int i=1;i<=5;i++)
    {
        System.out.println("child thread");
    }
};
```

**Logic using Method reference**

```
public class Test{
    //Developer :: Karthik
    //UserDefined static method
    public static void logicForThread()
    {
        //logic for child thread
        for (int i =0;i<5 ;i++)
        {
            System.out.println("Child thread...");
        }
    }
}
```

**//binded to void run()(below line syntax is same as void run. logicForThread → return type is void and arguments are 0. Below we will see if we pass arguments or will have different return type.So this is actually the meaning of binding(will only be valid if both are same.) )**

**Ans => Functional interface method and our specific methods should have same argument types, except this remaining things like return-type, methodName,accessmodifier is not important.**

**Runnable r1=Test::logicForThread;**

One code can be written in mulitple forms

a. Traditional Approach(OOPS)

1. Ananymous inner class implements interface
2. Ananymous inner class extends a class
3. Ananmyous inner class passed as an argument.

b. Functional Programming

1. Lambda Expression **[build the code from the scratch]** for Functionalinterface[SAM]
2. MethodReference **[Reuse the code]** for Functional interface[SAM]

**(SAM - Single Abstract Method)**

**eg#1.**

```
import java.util.function.*;
@FunctionalInterface
interface Interf
{
    public void m1(int i);
}
//logic is written by other developer
class Demo
{
    public int logicforReusing(int i)
    {
        System.out.println("Hey i gave the implementation.. reuse it ...");
        return 100;
    }
}
public class Test{
    public static void main(String[] args){
        //using lambda expression
        Interf i1 = i->System.out.println("coming from lambda Expression..."+i);
        i1.m1(10);

        //using method reference
        Demo d1 = new Demo();
        Interf i2 = d1::logicforReusing;
        i2.m1(100);
    }
}
```

**=> Functional interface method and our specific methods should have same argument types, except this remaining things like return-type, methodName,accessmodifier is not important.**

**Constructor reference**

=> we use the same :: operator to refer constructor also.

=> Classname obj = new ClassName();**//Traditional OOPS**

**Syntax**

ClassName :: new

**eg#1.**

```
import java.util.function.*;
@FunctionalInterface
interface Interf
{
    public Sample get(String s);
}
class Sample
{
    private String s;
    //Constructor
    Sample(String s)
    {
        this.s = s;
        System.out.println("Constructor Executed.... "+s);
    }
}
public class Test{
    public static void main(String[] args){
        //using lambda expression
        Interf i1= s->new Sample(s);
        i1.get("sachin :: using Lambda Expression");
        System.out.println();

        //using Constructor reference
        Interf i2 = Sample :: new ;
        i2.get("dhoni :: using Constructor Reference");
    }
}
```

**Output**

**Constructor Executed.... sachin :: using Lambda Expression**

**Constructor Executed.... dhoni :: using Constructor Reference**

**Optional API in JDK8**

=> It is given by Oracle to **avoid NullPointerException** in our program.

=> Normaly in realtime project every developer will not implement null checking in the program, as a result of which there would be NullPointerException in the program.

=> To avoid this we use "Optional" API in realtime project.

=> It is always a good practise to keep the data which needs to be processed inside "Optional" object only.

```
public final class java.util.Optional {
    public static java.util.Optional ofNullable(T);
    public static java.util.Optional empty();
    public static java.util.Optional of(T);
    public T get();
    public boolean isPresent();
    public boolean isEmpty();
    public java.util.Optional filter(java.util.function.Predicate);
    public java.util.Optional map(java.util.function.Function extends U>);
}
```

**eg#1.**

```
import java.util.*;
import java.util.*;
//API Development
class User
{
    //Written by Developer without Optional API::Shahid
    public String getUserById(Integer id)
    {
        if(id == 10)
            return "sachin";
        else if(id == 19)
            return "dravid";
        else if(id == 7)
            return "dhoni";
        else if(id == 18)
            return "kohli";
        else
            return null;
    }
```

```
    //Handling NullChecking through an API called "Optional"
    public Optional getUserNameById(Integer id)
    {
        String name = null;
        if(id == 10)
            name= "sachin";
        else if(id == 19)
            name= "dravid";
        else if(id == 7)
            name= "dhoni";
        else if(id == 18)
            name= "kohli";
        //returns Non-empty Optional object if a value is present otherwise return empty Optional object.
        return Optional.ofNullable(name);
    }
}
public class Test{
    public static void main(String[] args){
        //Written by Developer:: Karthik
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the id :: ");
        Integer id = scanner.nextInt();
        User user = new User();
        /*
            String userName=user.getUserById(id);
            System.out.println("Hello :: "+ userName.toUpperCase());
        */
        Optional optional=user.getUserNameById(id);
        if (optional.isPresent())
        {
            String record = optional.get();
            System.out.println("Hello :: " + record.toUpperCase());
        }
        else
        {
            System.out.println("No Data found...");
        }
    }
```

```
}
```

**if we run getUserById than on entering 100 as id it will throw nullpointerexception(why not try catch?? → 1)- nullpointerexception is unchecked exception 2)- It is really difficult to know what type of exception will a method throw which is written by someone else.)**

**Optional API methods**

public java.util.Optional filter(java.util.function.Predicate);

public java.util.Optional map(java.util.function.Function extends U>);

**eg#1.**

```
import java.util.*;
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        //Creating an Optional Object with data
        Optional nonEmptyGender=Optional.of("male");

        //Creating an Empty Optional Object
        Optional emptyGender=Optional.empty();
        System.out.println("Lambda Expression :: "+nonEmptyGender.map(s-> s.toUpperCase()));
        System.out.println("Method Reference"+nonEmptyGender.map(String::toUpperCase));

        System.out.println();

        System.out.println("Lambda Expression :: "+ emptyGender.map(s-> s.toUpperCase()));
        System.out.println("Method Reference :: "+emptyGender.map(String::toUpperCase));

        System.out.println();

        Predicate p1 = gender-> gender.equals("MALE");
        System.out.println(nonEmptyGender.filter(p1));
        System.out.println(nonEmptyGender.filter(gender→ gender.equals("male")));
        System.out.println(nonEmptyGender.filter(gender→ gender.equalsIgnoreCase("male")));
        System.out.println();
    }
}
```

**What is the difference b/w the follwing methods?**

1. **public boolean isPresent();**
   It returns true if the given Optional object is non-empty(data present) otherwise it returns false.
2. **public void ifPresent(java.util.function.Consumer<? super T>);**

**Consumer**

=> Sometimes our requirement is to provide some input value, perform certain operations but not requried to return anything.

=> Consumer can be used to consume the object and perform certain operation.

```
public interface java.util.function.Consumer {
    public abstract void accept(T);
    public default java.util.function.Consumer
    andThen(java.util.function.Consumer<? super T>);
}
```

```
import java.util.*;
import java.util.function.*;
import java.util.function.*;
public class Test{
    public static void main(String[] args){
        //Binded to accept(T):: void of Consumer
        Consumer c=s->System.out.println(s);
        c.accept("sachin");
        c.accept("dravid");

        System.out.println();

        Consumer c1 =System.out::println;
        c1.accept("shahid");
        c1.accept("karthik");
    }
}
```

**Output**

**sachin**

**dravid**

**shahid**

**karthik**

**Requirement**

**Display movie information using Consumer interface**

```
import java.util.*;
import java.util.function.*;
import java.util.function.*;
class Movie
{
    String name;
    String hero;
    String heroine;
    Movie(String name,String hero,String heroine)
    {
        this.name = name;
        this.hero = hero;
        this.heroine = heroine;
    }
}
public class Test{
    public static void main(String[] args){
        Movie[] movies = new Movie[4];
        addMovies(movies);
        Consumer c = movie-> {
            System.out.println("Name is :: "+movie.name);
            System.out.println("Hero is :: "+movie.hero);
            System.out.println("Heroine is :: "+movie.heroine);
            System.out.println();
        };
        for (Movie movie: movies )
        {
            c.accept(movie);
        }
    }
    public static void addMovies(Movie[] movies){
        Movie m1= new Movie("Salaar","Prabhas","ShruthiHasan");
        Movie m2= new Movie("Pushpa","AlluArjun","Rashmikha");
```

```
        Movie m3= new Movie("Sambahadhur","Vicky","katrinakaif");
        Movie m4= new Movie("Katera","Darshan","Nikitha");
        movies[0] = m1;
        movies[1] = m2;
        movies[2] = m3;
        movies[3] = m4;
    }
}
```

**Output**

**Name is :: Salaar**

**Hero is :: Prabhas**

**Heroine is :: ShruthiHasan**

**Name is :: Pushpa**

**Hero is :: AlluArjun**

**Heroine is :: Rashmikha**

**Name is :: Sambahadhur**

**Hero is :: Vicky**

**Heroine is :: katrinakaif**

**Name is :: Katera**

**Hero is :: Darshan**

**Heroine is :: Nikitha**

**Write a program that student details as input print the student details along with "Grade" Grade will be given based on the marks taken by the student**

marks>=80 [Distinction]

marks>=60 [FirstClass]

marks>=50 [SecondClass]

marks>=35 [ThirdClass]

Print only such students whose marks is greater than or equal to 65???

**Printing ----> Consumer [accept(student)]**

**Grade ----> Predicate [test(marks)]**

**Predicting Grade ----> Function<Student,String>::[apply]**

**eg#1.**

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

```
{
    String name;
    int marks;
    Student(String name,int marks)
    {
        this.name = name;
        this.marks = marks;
    }
}
public class Test{
    public static void main(String[] args){
        Student[] students = new Student[4];
        addStudents(students);
        Function<Student,String> f = s-> {
            int marks=s.marks;
            if (marks>=80)
                return "A[Distinction]";
            else if(marks>=60)
                return "B[FirstClass]";
            else if(marks>=50)
                return "C[SecondClass]";
            else if(marks>=35)
                return "D[ThirdClass]";
            else
                return "E[Failed]";
        };
        Consumer c = student-> {
            System.out.println("Name is :: "+student.name);
            System.out.println("Marks is :: "+student.marks);
            System.out.println("Grade is :: "+f.apply(student));
            System.out.println();
        };
        Predicate p = s-> s.marks>=65;
        for (Student s :students )
        {
            if(p.test(s))
                c.accept(s);
        }
    }
```

```
    public static void addStudents(Student[] students){
        Student s1 =new Student("Shahid",80);
        Student s2 =new Student("Karthik",65);
        Student s3 =new Student("Nitin",55);
        Student s4 =new Student("Naveen",45);
        students[0]=s1;
        students[1]=s2;
        students[2]=s3;
        students[3]=s4;
    }
}
```

**Output**

**Name is :: Shahid**

**Marks is :: 80**

**Grade is :: A[Distinction]**

**Name is :: Karthik**

**Marks is :: 65**

**Grade is :: B[FirstClass]**

**Inbuilt functional interfaces used in realtime Coding**

Predicate(logical condition) -> test(T) :: boolean

Fuction(Perform some operation and return result) -> apply(T) :: R

Consumer(Consume the input supplied by user) -> accept(T) :: void

Supplier(Supplies the data to the user) -> get() :: T

**Optional API**

```
public final class java.util.Optional {
    public static java.util.Optional empty();
    public static java.util.Optional of(T); :: returns the Optional object filled with non-null value
    public static java.util.Optional ofNullable(T); :: returns non-empty optional object if a value is present inside the object
    public T get();
    public boolean isPresent();
    public boolean isEmpty();
    public void ifPresent(java.util.function.Consumer<? super T>);
    public java.util.Optional filter(java.util.function.Predicate);
```

```
    public java.util.Optional map(java.util.function.Function extends U>);
    public T orElse(T);
    public T orElseGet(java.util.function.Supplier<? extends T>);
    public T orElseThrow();
    public T orElseThrow(java.util.function.Supplier<? extends X>) throws X;
}
```

**eg#1.**

```
import java.util.function.*;
import java.util.*;
public class Test{
    public static void main(String[] args){
        Optional empty = Optional.empty();
        System.out.println(empty.isPresent());

        System.out.println();

        String name= "nitin";
        if(name!=null)
            System.out.println(name.length());

        System.out.println();

        Optional opt = Optional.of("nitin");
        opt.ifPresent(str->System.out.println(str.length()));
    }
}
```

**Output**

**false**

**5**

**5**

**eg#2.**

```
import java.util.function.*;
import java.util.*;
public class Test{
    public static void main(String[] args){
        String nullName = "[sachin,51,true]";
```

```
        String result = Optional.ofNullable(nullName).orElse("Record not found");
        System.out.println(result);
    }
}
```

**Output**

**[sachin,51,true]**

**eg#3.**

```
import java.util.function.*;
import java.util.*;
public class Test{
    public static void main(String[] args){
        String nullName = null;
        String result = Optional.ofNullable(nullName).orElse("Record not found");
        System.out.println(result);
    }
}
```

**Output**

**Record not found**

**KeyPoints**

**orElse() ->** If the value is present, it would return the value otherwise use orElse() to return the value.

**orElseGet(Supplier) ->** If the value is present, it would return the value otherwise get it from Supplier API call.

**orElseThrow(Supplier) ->** If the value is present, it would return the value otherwise it throws an Exception.

**eg#1.**

```
import java.util.function.*;
import java.util.*;
class StudentRecordNotFoundException extends RuntimeException
{
    StudentRecordNotFoundException(String msg)
    {
        super(msg);
    }
```

```
}
public class Test{
    public static void main(String[] args){
        String nullValue = null;
        String result = Optional.ofNullable(nullValue).orElseGet(()->"not found");
        System.out.println(result);
        System.out.println();

        System.out.println(Optional.ofNullable(nullValue).orElseThrow(()->
            new StudentRecordNotFoundException("Record not found"))
        );
    }
}
```

**Supplier(I)**

Sometime our requirment is we have to get some value based on some operation like supply student object, supply otp, supply random password, supply randomName etc.... for this type of requirment we need to go for "Supplier".

-> Supplier won't take any input,but it will always supply objects.

```
public interface java.util.function.Supplier {
    public abstract T get();
}
```

**Note:** Math.random() :: double -> generates a number greater than or equal to 0.0 and lessthan 1.0

**eg#1.**

```
import java.util.function.*;
import java.util.*;
public class Test{
    public static void main(String[] args){
        //public abstract T get();
        Supplier s = () ->{
            String names[] = {"sachin","saurav","rahul","dhoni","kohli"};
            int index=(int)(Math.random()*5);
            System.out.print(index + "::");
            return names[index];
        };
        System.out.println(s.get());
        System.out.println(s.get());
```

```
        System.out.println(s.get());
    }
}
```

**Output**

**0::sachin**

**4::kohli**

**1::saurav**

**eg#2.**

```
import java.util.function.*;
import java.util.*;
public class Test{
    public static void main(String[] args){
        //public abstract T get();
        Supplier s = () ->{
            //Generating OTP(6 digits:: 0 to 9)
            String otp = "";
            for (int i =1;i<=6 ;i++ )
            {
                otp+=(int)(Math.random()*10);
            }
            return otp;
        };
    System.out.println(s.get());
    System.out.println(s.get());
    System.out.println(s.get());
    }
}
```

**Output**

**517276**

**947456**

**419555**

**eg#3.**

**Rules**

1. length should be 8 characters

2. 2,4,6,8 places only digits
3. 1,3,5,7 only capital letters and special symobls like @,#,$

```
import java.util.function.*;
import java.util.*;
public class Test{
public static void main(String[] args){
    //public abstract T get();
    Supplier s = () ->{
        String password = "";
        String symbols = "ABCDEFGHIJKLMNOPQRSTUVWXYZ@$#";
        Supplier i1=()->(int)(Math.random()*10);
        Supplierc1=()->symbols.charAt((int)(Math.random()*29));
        for (int i =1;i<=8 ;i++)
        {
                if (i%2==0)
                {
                    //even places
                    password+=i1.get();
                }
                else
                {
                    //odd places
                    password+=c1.get();
                }
        }
        return password;
        };
        System.out.println(s.get());
        System.out.println(s.get());
        System.out.println(s.get());
    }
}
```

**Output**

**V7R8Y3D3**

**L5D4C3K4**

**F4Z8@8Y5**

**StringJoiner class**

=> It is used to join 2 String seperated with a delimitor

=> Delimitor can be ,|,.....

```
public final class java.util.StringJoiner {
    public java.util.StringJoiner(java.lang.CharSequence);
    public java.util.StringJoiner(java.lang.CharSequence, java.lang.CharSequence,
    java.lang.CharSequence);
    public java.util.StringJoiner setEmptyValue(java.lang.CharSequence);
    public java.lang.String toString();
    public java.util.StringJoiner add(java.lang.CharSequence);
    public java.util.StringJoiner merge(java.util.StringJoiner);
    public int length();
}
```

**eg#1.**

```
import java.util.*;
public class Test{
    public static void main(String[] args){
        delimitorDemonstration();
        addingPrefixAndSuffix();
        mergeTwoStringJoiners();
    }
    public static void addingPrefixAndSuffix()
    {
        StringJoiner sj =new StringJoiner(" , " , " [ " , " ] "); //,-delimiter,[-suffix,]-prefix
        sj.add("sachin");
        sj.add("saurav");
        sj.add("dhoni");
        sj.add("kohli");
        sj.add("dravid");
    System.out.println(sj);
    }
    public static void delimitorDemonstration()
    {
        StringJoiner sj =new StringJoiner(",");
        sj.add("sachin");
        sj.add("saurav");
        sj.add("dhoni");
```

```
        sj.add("kohli");
        sj.add("dravid");
        System.out.println(sj);

        System.out.println();

        sj =new StringJoiner("|");
        sj.add("sachin");
        sj.add("saurav");
        sj.add("dhoni");
        sj.add("kohli");
        sj.add("dravid");
        System.out.println(sj);
    }
    public static void mergeTwoStringJoiners()
    {
        StringJoiner sj1 =new StringJoiner(",","[","]");
        sj1.add("sachin");
        sj1.add("saurav");
        sj1.add("dravid");
        StringJoiner sj2 =new StringJoiner(",","[","]");
        sj2.add("dhoni");
        sj2.add("kohli");
        sj2.add("rohith");
        StringJoiner sj3 = sj1.merge(sj2);
        System.out.println(sj3);
    }
}
```

**Output**

**sachin,saurav,dhoni,kohli,dravid**

**sachin|saurav|dhoni|kohli|dravid**

**[sachin,saurav,dhoni,kohli,dravid]**

**[sachin,saurav,dravid,dhoni,kohli,rohith]**

**eg#2.**

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

```
    public static void main(String[] args){
        stringJoinerMethods();
    }
    public static void stringJoinerMethods()
    {
        StringJoiner sj =new StringJoiner(",");
        System.out.println(sj);
        sj.setEmptyValue("It is empty");
        System.out.println(sj);  //It is empty
        sj.add("sachin");
        sj.add("dravid");
        System.out.println(sj);  //sachin,dravid
        int length = sj.length();
        System.out.println(length);  //13
        String data = sj.toString();
        System.out.println(data.charAt(3));
        System.out.println(data.toUpperCase());
    }
}
```

**Output**

**It is empty**

**sachin,dravid**

**13**

**h**

**SACHIN,DRAVID**

**Date and Time API**

Untill JDK1.7Version to handle the data and time related information, we use classes like

a. Date

b. Calendar

c. TimeZone etc....

But these classes are not up to the mark w.r.t performance and convinence.

=>To Overcome this problem in JDK1.8V Oracle team had introducted an API called **"JODA-API"**.

=>This API is developed by an organisation called "joda.org' and it is present inside the package called "java.time" package.

**JODA-API**

a. LocalDate

b. LocalTime

c. LocalDateTime

**Methods of LocalDate**

public static java.time.LocalDate of(int, java.time.Month, int);

public static java.time.LocalDate of(int, int, int);

**Methods of LocalTime**

public static java.time.LocalTime of(int, int);

public static java.time.LocalTime of(int, int, int);

public static java.time.LocalTime of(int, int, int, int);

**Methods of LocalDateTime**

public static java.time.LocalDateTime of(int, java.time.Month, int, int, int);

public static java.time.LocalDateTime of(int, java.time.Month, int, int, int, int);

public static java.time.LocalDateTime of(int, java.time.Month, int, int, int, int,int);

public static java.time.LocalDateTime of(int, int, int, int, int);

public static java.time.LocalDateTime of(int, int, int, int, int, int);

public static java.time.LocalDateTime of(int, int, int, int, int, int, int);

**eg#1.**

```
import java.time.*;
public class Test{
    public static void main(String[] args){
        LocalDate ld = LocalDate.now();
        System.out.println(ld);
        System.out.println("Year is :: "+ld.getYear());
        System.out.println("Month is :: "+ld.getMonthValue());
        System.out.println("Date is :: "+ld.getDayOfMonth());

        System.out.println();

        LocalTime lt = LocalTime.now();
        System.out.println(lt);
        System.out.println("HOUR is :: "+lt.getHour());
        System.out.println("MIN is :: "+lt.getMinute());
```

```
        System.out.println("SECONDS is :: "+lt.getSecond());
        System.out.println("NANO is :: "+lt.getNano());

        System.out.println();

        LocalDateTime ldt = LocalDateTime.now();
        System.out.println(ldt);
    }
}
```

**Output**

**2024-01-21**

**Year is :: 2024**

**Month is :: 1**

**Date is :: 21**

**12:57:24.688**

**HOUR is :: 12**

**MIN is :: 57**

**SECONDS is :: 24**

**NANO is :: 688000000**

**2024-01-21T12:57:24.688**

**eg#2.**

**To create the date and time object as per our needs we use the following methods**

```
import java.time.*;
public class Test{
    public static void main(String[] args){
        LocalDate ld = LocalDate.of(1993,Month.JANUARY,03);
        System.out.println(ld);
        LocalTime lt = LocalTime.of(19,45,35);
        System.out.println(lt);
        LocalDateTime ldt = LocalDateTime.of(1996,Month.FEBRUARY,24,9,30,45);
        System.out.println(ldt);
    }
}
```

**Output**

**1993-01-03**

**19:45:35**

**1996-02-24T09:30:45**

**eg#3.**

**To represent Zone we use ZoneId Object**

Few ZoneId

America/Toronto

Asia/Singapore

Australia/Lindeman

America/Los_Angeles

```
import java.time.*;
import java.util.*;
public class Test{
    public static void main(String[] args){
        ZoneId zone = ZoneId.systemDefault();
        System.out.println(zone);//Asia/Calcutta
        System.out.println();
        ZoneId zoneId= ZoneId.of("Australia/Melbourne");
        ZonedDateTime zdt = ZonedDateTime.now(zoneId);
        System.out.println(zdt);
    }
}
```

**Output**

**Asia/Calcutta**

**2024-01-21T17:54:14.654+10:00[Australia/Lindeman]**

**Period Object**

**=> It is used to represent quantity of time**

**eg#1.**

```
import java.time.*;
import java.util.*;
public class Test{
    public static void main(String[] args){
```

```
        LocalDate todayDate = LocalDate.now();
        LocalDate birthDate = LocalDate.of(1993,1,03);
        Period period=Period.between(birthDate,todayDate);
        System.out.println(period);
        System.out.println("Your age is :: "+period.getYears() +" Months is :: "+period.getMonths() +"
        Days is :: "+period.getDays());
        System.out.println();
        System.out.printf("\nAge is %d years %d months %d days\n",
        period.getYears(),period.getMonths(),period.getDays());
    }
}
```

**Output**

**P31Y18D**

**Your age is 31 Months is 0 Days is :: 18**

**Age is 31 years 0 months 18 days**

**Write a program to check wheter the given year is leap year or not?**

**Logic::**

Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400.

**eg#1.**

```
import java.time.*;
public class Test{
    public static void main(String[] args){
        //Converting String to Integer type
        Integer year = Integer.parseInt(args[0]);
        Year checkLeapear = Year.of(year);
        String output = checkLeapYear.isLeap() ?
        year + " is a LeapYear " :
        year + " is not a LeapYear ";
        System.out.println(output);
    }
}
```

**Output**

**D:\OctBatchMicroservices>javac Test.java**

**D:\OctBatchMicroservices>java Test 2023**

**2023 is not a LeapYear**

**D:\OctBatchMicroservices>java Test 2024**

**2024 is a LeapYear**

**D:\OctBatchMicroservices>java Test 2020**

**2020 is a LeapYear**