# Java 1.8

# Java 1.8

## Purpose of Java 1.8

1. To Simplify Programming
2. To utilize functional programming Benefits in Java
3. To Enable Parallel programming (Processing) in Java

- 1.0 ✓
- 1.1
- 1.2 ✓
- 1.3
- 1.4
- 1.5 ✓
- 1.6
- 1.7
- 1.8 ✓

Java 1.8 is one of the major and more prestigious version from Java. In this tutorials we covered every topic in details on the board and on the System with live execution. Definitely you can feel like you are learning in class room directly from the instructor. As the Part of this Course we are covering the following 11 topics

1. Lambda Expressions
2. functional interfaces
3. Default method in interface
4. Static methods in interfaces
5. Predicate
6. Function
7. Consumer
8. Supplier
9. Method Refference & Constructor Reference by (::) Double Colon operater
10. Stream API.
11. Date & Time API (JODA API)

# 1. Lambda Expression

How many Languages are using the Lambda Expression.

① LISP (first programing language which use the Lambda Expression)
② C
③ C++
④ objective C
⑤ C# .net
⑥ SCALA
⑦ RUBY
⑧ Python

finally Java Also using Lambda

## Benefits of Lambda Expression

① To Enable functional programing in Java
② To write more readable, maintaible & Concise code
③ To use API's very Easily & effectivelly.
④ To Enable Parallel Processing

## How to write Lambda Expression

- Lambda Expression is an Anonymose function.
- Anonymose function is a function which is not having any name.

— Not having any name
— Not having modifiers
— Not having any return type.

Example :- Normal Java | Lambda Expression

① Normal Java

```
Public void m1()
{
   S.o.p("Hello");
}
```

Lambda Expression

```
() → { S.o.P("Hello"); }
```

② Normal Java

```
Public void add(int a, int b)
{
   S.o.pln(a+b);
}
```

Lambda Expression

```
(int a, int b) → { S.o.pln(a+b); }
```

③ Normal Java

```
Public int getLength(String s)
{
  return s.length();
}
```

Lambda Expression

```
(String s) → { return s.length(); }
```

④

```
Public void m1()
{
  S.o.pln("Hello");
}
```

```
() → { S.o.pln("Hello"); }
        OR
() → S.o.pln("Hello");
```

- If the body of the Lambda Expression Contain only one statement then curly Braces are not required

⑤

```
Public void add(int a, int b)
{
  S.o.pln(a+b);
}
```

```
(int a, int b) → S.o.pln(a+b);
        OR
(a,b) → S.o.pln(a+b);   (Type inference)
```

- Based on the context sometime Compiler able to guess type automatically

6:-

```
Public int getLength(String s)
{
 return s.length();
}
```

```
(String s) -> { return s.length(); }
        OR
(s) -> return s.length();
        OR
(s) -> s.length();
        OR
s -> s.length();
```

## Characteristics/properties of Lambda Expression

(1) A λ-Expression can take any no. of Parameters

Ex:-
```
() -> s.o.pln("Hello");
(s) -> s.length();
(a,b) -> s.o.pln(a+b);
```

(2) If multiple parameters present then these Parameters should be seperated with , .

Ex:- (a, b) -> s.o.pln(a+b);

(3) If only one parameter available then Paranthesis are optional.

Ex:- (s) -> s.length(); ⟹ s -> s.length();

(4) Usally we can specify the type of parameter, if Compiler expect the type based on context, then we can remove type [Type Inference].

Ex:-
```
(int a, int b) -> s.o.pln(a+b);
        ⇓
(a,b) -> s.o.pln(a+b);
```

⑤ Similarly to body (method), λ-expression body can contain any number of statements, if multiple statements are there then we should enclose with in curly braces.

Ex:-

```
() ⟶ {
        Stmt 1;
        Stmt 2;
        Stmt 3;
     };
```

If body contains only one statement then curly braces are optional.

Ex:- `() ⟶ S.o.println("Hello");`

⑥ If λ-expression return something then we can remove return keyword.

`s ⟶ s.length();`

# Functional Interface

- Its Contain Single Abstract method (SAM)
- We use the function interface to invoke the Lambda-Expression.

## Example of functional Interface :-

Runnable ← Contains only run() method
Callable – Contains only call() method
ActionListener – containing only actionPerformed() method
Comparable – containing only compareTo() method

- functional interface can Contain any number of "Default" and static method. which contain the Body in the interface. Restriction only for Abstract method. which will be single in functional interface.

### Functional interface

```
interface Interf
{
 public void m1();

default void m2()
{

}

 Public static void m3()
{

}

}
```

### Non functional Interface

```
interface Interf
 {

  Public void m1();
  Public void m2();

 }
```

## @FunctionalInterface Annotation

With this annotation we can make Interface Explicitly-FunctionalInterface.

Example :-

① 
```
@FunctionalInterface
interface Interf
{
  Public void m1();
  default void m2()
  {
  }
  Public Static void m3()
  {
  }
}
```
It is valid.

② 
```
@FunctionalInterface
interface Interf1
{
   Public void m1();
   Public void m2();
}
```
It is Invalid.

CE :- Unexpected @FunctionalInterface annotation
multiple non-overriding abstract method present in interface Interf1

③ 
```
@FunctionalInterface
interface Interf2
{
}
```
It is Invalid

CE :- unexpected @FunctionalInterface annotation,
no abstract method found in interface Interf2.

## Functional Interface with respect to Inheritence

**Case1:-** If an interface extends Functional Interface and child interface does not contain any abstract method, then child interface is always functional interface.

Ex:-

```
@FunctionalInterface
interface P
{
 Public void m1();
}

@FunctionalInterface
 interface c extends P
 {
 }
```

It is valid ✓

**Case 2:-** In the child interface, we can define exactly same parent interface abstract method.

Ex:-

```
@FunctionalInterface
interface P
{
 Public void m1();
}

@FunctionalInterface
 interface c extends P
 {
  Public void m1();
 }
```

It is valid ✓

**Case3:-** In the child interface we can't define any new abstract methods otherwise we will get Compile time Error.

Ex:-

```
@FunctionInterface
interface P
{
 public void m1();
}

@FunctionInterface
interface C extends P
{
 public void m2();
}
```

✗

gt is invalid

CE: unexpected @FunctionalInterface annotation

multiple non-overriding abstract methods found in interface 'C'.

**Case-4**

```
@FunctionalInterface
interface P
{
 public void m1();
}

interface C extends P
{
 public void m2();
}
```

gt is valid

because in child interface we did not declare @FunctionalInterface annotation. so gts normal interface.

# Invoking Lambda Expression By using Funtional-Interface

## Without λ-Expression

```
interface Interf
{
  Public void m1();
}
```

```
Class Demo implements Interf
{
  Public void m1()
  {
  Sopln("m1() method implementation");
  }
}
```

```
Class Test
{
  Public static void main(String[] args)
  {
    Interf i = new Demo();
      i.m1();
  }
}
```

## With λ-expression

```
interface Interf {
  public void m1();
}
```

```
Class Test {
  P s v main (String[] args) {
  Interf i = () -> S.opln("m1() method implementation");
    i.m1();
} }
```

Example 2 :-

```
interface Interf
{
  Public int squareIt(int x);
}

class Test2
{
  Public static void main(String[] args)
  {
     Interf i = x -> x*x;
     System.out.println(i.squareIt(10));
     System.out.println(i.squareIt(20));
  }
}
```

Output : 100
400

Example :-3

Without λ-expression

```
interface Interf
{
  Public void add(int a, int b);
}

class Demo implements Interf
{
  public void add(int a, int b)
  {
    S.O.Pln("The Sum :" + (a+b));
  }
```

```
Class Test1
{
  Public static void main(String[] args)
  {
    Interf i = new Demo();
      i.add(10,20);          // The Sum: 30
      i.add(100,200);        // The Sum: 300
  }
}
```

With λ-Expression

```
interface Interf
{
  Public void add(int a, int b);
}
```

```
Class Test
{
  Public static void main(String[] args)
  {
    Interf i = (a,b) -> Sopln("The Sum:" +(a+b));
        i.add(10,20);
        i.add(100,200);
  }
}
```

Output: The Sum: 30
The Sum: 300

## Example -4

### Without · λ-Expression

```
Interface Interf
{
  Public int getLength(String S);
}

class Demo implements Interf
{
  Public int getLength(String S)
  {
    return s.length();
  }
}

class Test {
  P. S. V main(String[] args){
   Interf i= new Demo();
    S.O.Pln(i.getLength("Hello"));
    S.O.Pln(i.getLength("without Lambda"));
  }
```

### With λ-Expression

```
interface Interf {
  Public ~~static void~~ int getLength(String S);
}

class Test {
  P S V main(String() args)
  {
    Interf i = S → s.length();
   S.O.Pln(i.getLength("Hello"));
   S.O.Pln(i.getLength("without Lambda"));
} }
```

## Example-5 → Runnable Interface

### Without λ-Expression :-

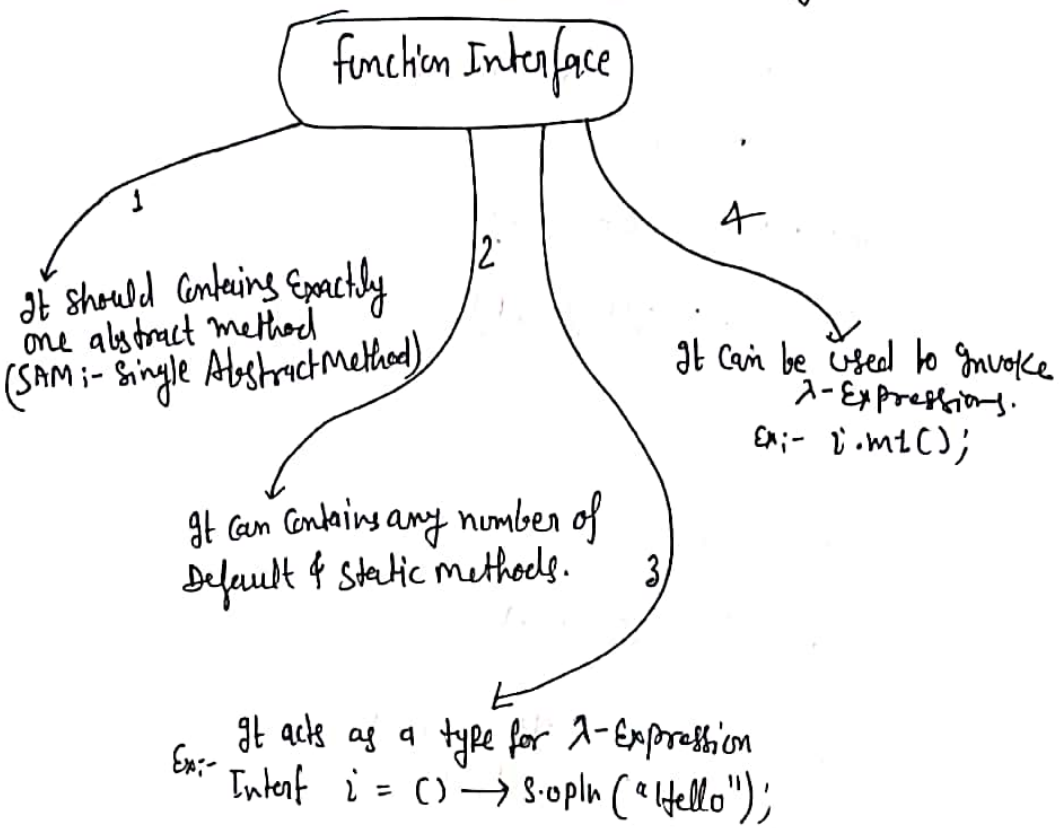
```
class myRunnable implements Runnable
{
  public void run()
  {
      for(int i=0; i<10; i++)
      {
        S.o.pln("child thread");
      }
  }
}
```

```
class ThreadsDemo1
{
  P S V main(String[] args)
  {
    Runnable r = new myRunnable();
    Thread t = new Thread(r);
      t.start();
    S.o.pln("main Thread");
    for(int i=0; i<10; i++)
    {
      S.o.pln("main Thread");
    }
  }
}
```

### With λ-Expression

```
class ThreadDemo1
{
  P. S. V main(String[] args)
  {
    Runnable r = () -> { for(int i=0; i<10; i++)
                         { S.o.pln("child Thread");
                       };};
    Thread t= new Thread(r);
      t.start();
    for(int i=0; i<10; i++)
      { S.o.pln("main Thread");
      }
  }
}
```

# Functional Interface, Lambda Expression Summary




Q:- Case 1- Why functional interface should Contains only one abstract method ?

Ans:-

It is Valid

```
interface Interf{
Public void m1(int i);
}

Interf I = i -->S.o.pln(i*i);
    I.m1(10);
    I.m1(20);
```

It is not valid

```
interface Interf
 {
  Public void m1(int i);
  Public void m2(int i);
 }

 Interf I = i -->S.o.pln(i*i);
```

CE: incompatible types Interf is not a functional interface.
multiple non-overriding abstract methods in interface Interf

Q→ Case 2- What is the advantage of @FunctionalInterface Annotation?

Ans:- The advantage of this is that it indicate that this interface is a functional interface and it Contains only a single Abstract method. If we are going to add second abstract method then it will raise the CE.

# Sorting Elements of Arraylist without λ-Expression

Example:-1→

```
Class Test{
Public static void main(String[] args)
{
 ArrayList<Integer> I = new ArrayList<Integer>();
 I.add(10);
 I.add(0);
 I.add(15);
 I.add(5);
 I.add(20);
S.o.pln("BeforeSorting:" +I);
        //Before sorting : [10, 0, 15, 5, 20]
 Collections.sort(I);
 System.out.println("After Sorting:" +I);
        //After Sorting : [0, 5, 10, 15, 20]
 }
}
```

Example:-2→

```
Class MyComparator implements Comparator<Integer>
{ public int Compare(Integer I1, Integer I2)
 { if(I1>I2)
   { return -1;
   } else if(I1<I2)
     { return +1;
     } else {
         return 0;
     }
 }
}
```

Instead of this code we can use Ternary Operator like below.

```
{
return (I1>I2)? -1:
        (I1<I2)?1:0;
}
```

```
Class Test {
Public static void main(String[] args)
 { ArrayList<Integer> I = new ArrayList<Integer>();
 I.add(10); I.add(0); I.add(15); I.add(5); I.add(20);
 S.o.P("Before Sorting"+I);
 Collections.sort(I, new MyComparator());
 S.o.pln("After Sorting"+I);
 }}
```

Output: Before Sorting: [10, 0, 15, 5, 20]
After Sorting: [20, 15, 10, 5, 0]

# Sorting Elements of ArrayList with Lamda(λ)-Expression

Example:-

```
class Test
{
  Public static void main(String[] args)
  {
    ArrayList<Integer> I = new ArrayList<Integer>();
      I.add(10);
      I.add(0);
      I.add(15);
      I.add(5);
      I.add(20);
      I.add(25);
    System.out.println("Before Sorting:" +I);
    Collections.sort(I, (I1,I2) → (I1>I2)?-1:
                                (I1<I2)?1:0);
      System.out.println("After Sorting:" +I);
  }
}
```

Output:-

Before Sorting : [10, 0, 15, 5, 20, 25]

After Sorting : [25, 20, 15, 10, 5, 0]

# Sorting Elements of TreeSet with Lambda Expression

Example:

```
Class Test:
{
  Public static void main(String[] args)
  {
  TreeSet<Integer> t =
              new TreeSet<Integer>((I1,I2) →(I1>I2)?
                                  -1 : (I1<I2)?1:0);

    t.add(10);
    t.add(0);
    t.add(15);
    t.add(25);
    t.add(5);
    t.add(20);
   System.out.println(t);

  }
}
```

Output:-

[25, 20, 15, 10, 5, 0]

# Sorting Elements of TreeMap with Lambda Expression

Example:-

```
Class Test
{
 Public static void main(String[] args)
 {
 TreeMap<Integer,String> m = new
 TreeMap<Integer, String>(I1,I2) →(I1 >I2) ?-1:
                                 (I1 <I2) ? 1 : 0);
   m.put (100,"Durga");
   m.put(600,"Sunny");
   m.put(300,"Bunny");
    m.put(200,"Chinny");
    m.put(700,"Vinny");
    m.put(400,"Pinny");
    System.out.println(m);
   }
}
```

Output:-

{700=Vinny, 600=Sunny, 400=Pinny, 300=Bunny,
200=Chinny, 100=Durga}

# Sorting of our own class objects with λ-Expression

Example:-
```
class Employee
{
  int eno;
  String ename;

  Employee(int eno, String ename)
  {
    this.eno = eno;
    this.ename = ename;
  }

  public String toString()
  {
    return eno + ":" + ename;
  }
}

class Test {
  public static void main(String[] args)
  {
    ArrayList<Employee> I = new ArrayList<Employee>();
    I.add(new Employee(200, "Deepika"));
    I.add(new Employee(400, "Sunny"));
    I.add(new Employee(300, "Mallika"));
    I.add(new Employee(100, "Katrina"));
    S.O.println("Before Sorting:");
    S.O.println(I);
    Collections.sort(I, (e1, e2) -> (e1.eno < e2.eno) ? -1
                    : (e1.eno > e2.eno) ? 1 : 0);
    System.out.println("After Sorting");
    System.out.println(I);
  }
}
```

Output:
```
Before Sorting:
[200:Deepika, 400:Sunny, 300:Mallika, 100:Katrina]
After Sorting:
[100:Katrina, 200:Deepika, 300:Mallika, 400:Sunny]
```

## Anonymous Inner class Vs Lambda Expression

- A class which is not having any Name is called Anonymous Inner class.

Ex:
```
Runnable r = new Runnable(){
                Public void run()
                {
                  =
                }
            }
```

## with Anonymous Inner Class

```
class ThreadDemo1
{
  Public static void main(String[] args)
  {
    Runnable r= new Runnable()
    {
      public void run()
      {
        for(int i=0; i<10; i++)
        { S.O.pln ("child Thread");
        }}
    };
    Thread t= new Thread (r);
    t.start();
    for(int i =0; i<10; i++)
    {
      S.O.pln ("main Thread");
    }
  }
}
```

## With λ-Expression (Anonymous Inner Class).

```
Class ThreadDemo2
{
  Public static void main(String[] args)
  {
    Runnable r = () -> { for(int i=0; i<10; i++)
                         {
                           S.O.pln("Child Thread");
                         };
    Thread t = new Thread(r);
      t.start();
    for(int i=0; i<10; i++)
    {
      S.O.pln("main Thread");
    }
  }
}
```

OR

```
Class ThreadDemo2 {
 P.S.V.main(String args[]){
 Thread t = new Thread(()->{
                for(int i=0; i<10; i++)
                {S.O.pln("Child Thread");
              }};
   t.start();
   for(int i=0; i<10; i++){
       S.O.pln("main Thread");
   }}
  }}
```

## Anonymous Inner Class is Equal to λ-Expression

Annonymous Inner Class is not Equal to λ-Expression.

**CASE-1** Annonymous Inner Class that Extends Concrete class

```
Class Test {
}
  Test t = new Test()
           {
           ≡
           };
```

for this case λ-Expression can't use.

**CASE-2** Annonymous Inner Class that Extends abstract class.

```
abstract class Test {
}
Test t= new Test()
  {
  ≡
  };
```

for this case also λ-Expression can't use.

**CASE-3** Anonymous Inner class that implements an interface which contains multiple methods.

```
Interface Test
{ Public Void m1();
  Public Void m2();
  Public Void m3();
}
Test t= new Test()
{ Public Void m1(){ }
  Public Void m2(){ }
  Public Void m3(){ }
};
```

In this case also we can't use λ-Expression

**CASE-4** Anonymous Inner class implements an Interface which contains one abstract method.

```
Interface Test{
  Public void m1();
}

Test t = new Test()
 {
  Public void m1()
   {
    =
   }
 };
```

On this case we can use λ-Expression.

So finally we can say that

Anonymous Inner class != λ-Expression

## Difference Between Anonymous Inner Class and λ-Expression Based on "this" object

this. in Anonymous Inner class :-

```
interface Interf
{
 Public void m1();
}
Class Test
{ int x=888;
 Public void m2(){
    Interf i=new Interf()
     { int x=999; //instance variable
      Public void m1(){
        S.O.Pln(this.x); //999
      }
    };
    i.M1();
 }
  P.S.V main(String[] args){
     Test t= new Test();
      t.m2();
} }
```

So inside Anonymous Inner class 'this' always refers current Inner class object.

## "this" in λ-Expression :-

```
interface Intef
{
  public void m1();
}
class Test
{
  int x = 888;
  public void m2()
  {
    Intef i = () -> {
                      int x=999; // Local var
                      S.o.pln(this.x); // 888
                    };
    i.m1();
  }
  Public Static void main(String[] args)
  {
    Test t = new Test();
        t.m2();
  }
}
```

Inside λ-Expression "this" always refers outer class member only. because inside λ-Expression internally Object Data story not there.

# Difference Between Anonymous Inner Class & λ-Expression

| Anonymous Inner Class | λ-Expression |
|---|---|
| • It is a class without name | • It is a function without name. so this is call Anonymous function |
| • Anonymous Inner Class can Extend abstract & Concrete classes. | • λ-Expression can't extend abstract & Concrete classes |
| • Anonymous Inner Class can Implement an Interface that contains any number of abstract methods | • λ-Expression can implement an Interface which contains single abstract method (functional Interface) |
| • Inside Anonymous Inner class, we can declare instanse variables. | • Inside λ-Expression we can't declare instance variables. whatever variables we declared, are considered as Local-variables. |
| • Anonymous Inner Class can be Instantiated | • λ-Expression can't be instantiated. |
| • Inside Anonymous Inner class, 'this' always refers current anonymous Inner class object but not outer class object. | • Inside λ-Expression, 'this' always refers current outer class object, i.e. Enclosing class Object. |
| • Anonymous inner class is best choice if we want to handle multiple methods | • λ-Expression is the best choice if we want to handle interface with single Abstract method (functional Interface) |
| • for the Anonymous Inner class, at the time of Compilation, a Seperate .Class file will be generated | For the λ-Expression, at the time of Compilation, no seperate .class file will be generated. |
| • Memory will be allocated on demand whenever, we are creating object | • λ-Expression will reside in Permanent memory of JVM (Method Area) |

Example:-

```
interface Interf
{
  Public void m1();
}

Class Test
{
  int x = 10;
  Public void m2()
  {
    int y = 20;
    Interf i = () ->{
                    Sopln(x); // 10
                    Sopln(y); // 20
                   };
```

The Local variable referenced from λ-Expression are final or implicitly final. So the variable y is a Local variable and it is referenced from λ-Expression. So it will be the final. we can't change its value.

```
        i.m1();
    }
    Public static void main(String[] args)
    {
        Test t = new Test();
            t.m2();
    }
}
```

Example-2

```
Interface Interf
{
  public void m1();
}

class Test
{
   int x = 10;
  Public void m2()
  {
    int y=20;
    Interf i = () -> {
                    sopln(x); // 10
                    sopln(y); // 20
                    x=888;   ✓
                    y=999;  X → CE:- Local variables referenced from a λ-expression must be final or effectively final
                };
        i.m1();
  }
   Public static void main(String[] args)
   {
      Test t = new Test();
          t.m2();
   }
}
```

Compile time Error

## Advantages of λ-Expressions

1. We can Enable functional programming in Java.
2. We can reduce length of the code so that readibility will be improved.
3. We can resolve complexity of Anonymous Inner classes until some extent.
4. We can handle procedures/functions just like values.
5. We can pass procedures/functions as arguments.
6. Easier to use updated APIs and Libraries.
7. Enable support for parallel processing.

## Default Methods Inside Interfaces

Example:- • By using default keyword we can declare default method.

```
interface Interf
{
  default void m1()
  {
    Sopln("default method");
  }
}
```

```
Class Test implements Interf
{
  p.s.v main(String[] args)
  {
    Test t = new Test();
    t.m1(); //default method
  }
}
```

here we are not overriding default method m1 in Test class, and same default method going to access.

```
Class Test implements Interf
{
  public void m1()
  { Sopln("my own implementation"); }
  p s v main(String[] args)
  { Test t = new Test();
    t.m1(); // my own implementation
  }
}
```

here we are overriding default m1 method in Test class. So the implemented method will be accessed.

## Default method with respect to multiple inheritence

```
interface Left
{
  default void m1()
  {
    Sopln("Left Default method");
  }
}
```

```
Interface Right
{
  default void m1()
  {
    Sopln("Right Default method");
  }
}
```

```
Class Test implements Left, Right
{
}
```

Output:- Compile time Error : (Ambiguity Problem)
Class Test inherits unrelated defaults for m1() from types Left and Right

Solution of above problem;

```
Class Test implements Left, Right
{
    public void m1()
    {
      //Sopln("my own Implementation");
      // Left.super.m1();
    } // Right.super.m1();

    public static void main(String[] args)
    {
      Test t = new Test();
      t.m1();
    }
}
```

Output: my own implementation.

## Difference between interface with default methods & Abstract Class.

| Interface with Default Method | Abstract class |
|---|---|
| ① Inside interface every variables is always public, static and final we can not declare instance variables. | ① Inside Abstract class we can declare instance variables, which are required to the child class. |
| ② Interface never talkes about state of object | ② Abstract class can talk about state of object. |
| ③ Inside interface we can't declare constructors | ③ Inside abstract class we can declare constructors |
| ④ Inside interface we can't declare instance and static blocks | ④ Inside abstract class, we can declare Instance & static blocks |
| ⑤ Functional interface with default method can refer Lambda (λ)-Expression. | ⑤ Abstract class can't refer Lambda λ-Expression |
| ⑥ Inside interface we can't override object class methods | ⑦ Inside Abstract class we can override object class methods. |

## Static methods inside Interface :-

Example:- Static method we use to define general utility methods. And interface static method by default never be available in implementation class.

```
Interface Interf
{
  Public static void m1()
  {
    S.o.pln("interface static method");
  }
}

Class Test implements Interf
{
  Public static void main(String[] args)
  {
      // m1();          X
      Test t= new Test();
      // t.m1();        X
      // Test.m1();     X
      Interf.m1();      ✓
  }
}
```

with interface name only we can call static method. there are no other way to call static method in Java 1.8.

# Interface Static Methods with respect to Overriding

- As we know that Interface static methods by defaults will not be available inside the implementation class. So overriding concept is not applicable for interface static methods. If you want, you can define exactly same methods inside child class. there will not be any problem at all. It is perfectly valid but not overriding.

Example:-①

```
interface Interf
{
  public static void m1()
  {
  }
}
```

```
class Test implements Interf
{
   public static void m1()
   {
   }
}
```

It is valid ✓

Example:-2

```
interface Interf
{
  public static void m1()
  {
  }
}
```

```
class Test implements Interf
{
  public void m1() { }
}
```

It is valid ✓

Example:-3

```
Interface Interf
{
  public static void m1() { }
}
class Test implements Interf
{
   private static void m1() { }
}
```

It is valid ✓

Example:- 4-

It is valid

```
interface Interf
{
  public static void main(String[] args)
  {
     Sopln("Interface main method")
  }
}
```

- from Java 1.8 version onwards we can declare main method inside Interface & call the main method from command prompt also.

```
Javac Interf.java
Java Interf.
```

## Predefined Functional Interfaces - Predicate

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

} Predefined functional Interfaces in Java 1.8

### 1. Predicate :-

1. It introduce in 1.8 version
2. It present in java.util.function Package
3. It contain only one method test().

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

Example:- without λ Expression

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

- With λ Expression

```
(Integer I) -> {
                  if (I > 10)
                    return true;
                  else
                    return false;
                };
```

⇓ OR

```
I -> I > 10;
```

- With predicate function Interface

```
Predicate <Integer> p = I -> I > 10;

Sopln (P.test(100)); // true
Sopln (P.test(5)); // false
```

Example:-

```
import java.util.function.*;
Class Test
{
  P S V main (String[] args)
  {
    Predicate <Integer> P = I -> I > 10;
    Sopln (P.test(100)); // true
    Sopln (P.test(5)); // false.
  }
}
```

Example :-

```
import java.util.function.Predicate;
class Test
{
  Public static void main (String[] args)
  {
    Predicate <String> p = s -> s.length() > 5;
    System.out.println (p.test ("abcdef")); // true
    System.out.println (P.test ("abc")); // false
  }
}
```

Example:-

```
import java.util.function.Predicate;
import java.util.*;
Public class Test
{
 Public static void main(String[] args)
 {
   Predicate<Collection> p = c -> c.isEmpty();
   ArrayList l1 = new ArrayList();
     l1.add("A");
    System.out.println(P.test(l1)); // false
   ArrayList l2 = new ArrayList();
    System.out.println(P.test(l2)); // true
} }
```

## Predefined Functional Interfaces – Predicate Joining

Predicate functional Interface contains 3 default methods.

① negate();
② and();
③ or();

} Default methods inside Predicate. these methods use for joining the Predicate.

④ test(); } this is single abstract method inside Predicate interface.

Example:-

```
Public class Test
{
  Public static void main(String[] args)
  {
    int[] x={0, 5, 10, 15, 20, 25, 30}
    Predicate<Integer> P1 = i -> i>10;
    Predicate<Integer> P2 = i -> i%2 ==0;
   S.o.pln("The members greater than 10 are:");
     m1(P1,x);
```

```
Soplm("The even numbers are;");
    m1(P2, x);
Soplm("The member not greater than 10;");
  m1(P1.negate();x);
Soplm("The numbers greater than 10 and even are:");
  m1(P1.and(P2), x);
S.oplm("The numbers greater than 10 or even are:");
  m1(P1.or(P2), x);
}
Public static void m1(Predicate<Integer> P, int[] x)
 {
   for(int x1 : x)
    {
      if (P.test(x1))
        S.oplm(x1);
    }
  }
}
```

Output:-

```
The numbers Greater than 10 are:
15 20 25 30
The even numbers are:
0 10 20 30
The numbers Not Greater than 10 are;
0 5 10
The numbers Greater than 10 and even numbers are;
20 30
The numbers Greater than 10 or even numbers are;
0 10 15 20 25 30
```

## Program to display names starts with 'K' by using - Predicate :-

Example:-

```
import java.util.function.Predicate;
Class Test
{
 Public Static Void main(String[] args)
 {
   String[] names={"Sunny", "Kajal", "Mallika",
                   "Katerina", "Kareena"}
 Predicate<String> startsWithK = s -> s.charAt(0)=='K';
 System.out.println("The Names starts with K are:");
   for( String s : names)
   {
     if (startsWithK.test(s))
     {
       System.out.println(s);
     }
   }
 }
}
```

Output:

```
The Names starts with K are:
Kajal
Katrina
Kareena
```

# Predicate Example to Remove null values and Empty string from the given List?

Example:

```
import java.util.ArrayList;
import java.util.function.Predicate;
Class Test
{
Public static void main(String[] args)
  {
   String[] names = {"Durga", "", null, "Ravi", "", "Shiva",
                     null};
  Predicate<String> P = s -> s != null && s.length() != 0;
  ArrayList<String> list = new ArrayList<String>();
     for( String s : names)
     {
        if (P.test (s))
        {
          list.add(s);
        }
     }
    System.out.println ( "The List of Vaid Names:");
      System.out.println(list);
  }
}
```

Output:

```
The List of valid Names:
[Durga, Ravi, Shiva]
```

- Program for User Authentication by using Predicate

Example:

```
import java.util.function.Predicate;
import java.util.Scanner;
Class User
{
  String username;
  String Pwd;
  User(String username, String Pwd):
    {
      this.username = username;
      this.pwd = pwd;
    }
}
Class Test
{
  Public static void main(String[] args)
  {
  Predicate<User> P = u -> {u.username.equals("durga") &&
                            u.pwd.equals("java");
                           }
  Scanner sc = new Scanner(System.in);
    System.out.println("Enter User name:").
     String username = sc.next();
    System.out.println("Enter Password:");
     String pwd = sc.next();
     User user = new User(Username, pwd);
       if(P.test(User))
       {
        System.out.println("valid user can avail all services");
       }else{
        System.out.println("invalid user, you can't avail services");
  } }
```

output:

Enter user Name:
durga
Enter password:
Java
Valid user can avail all services.

} for valid user

Enter user Name:
durga
Enter Password:
anushka
Invalid user you can't avail services

} for Invalid user

- Program to check whether Software Engineer is allowed into Pub or not?

Example:-

```
import java.util.function.Predicate;
Class SoftwareEngineer
{
 String name;
  int age;
 Boolean isHavingCrf;
SoftwareEngineer(String name, int age, boolean isHavingCrf)
 {
    this.name = name;
    this.age = age
    this.isHavingCrf = isHavingCrf;
 }
Public String toString()
 {
    return name;
 }

}
```

## Employee Management Application By using Predicate :-

Example :-

```
import java.util.ArrayList;
class Employee
{
 String name;
 String designation;
 double salary;
 String city;

Employee(String name, String designation, double salary, String city)
 {
   this.name = name;
   this.designation = designation;
   this.salary = salary;
   this.city = city;
 }

  Public string toString()                    → for .2 decimal Point
  {                                    %0.2f
    String s = String.formate("%s, %s, %f, %s)", name, designation,
                                        salary, city);
      return s;
  }

}

   Class Test
   {
      Public static void main(String[] args)
      {
ArrayList<Employee> list = new ArrayList<Employee>();
  Employee e1= new Employee("Durga", "CEO", 30000, "Hyderabad");
  Empolee e2= new Employee("Sunny", "manager, 20000, "Bangulure");
    list.add(e1);
    list.add(e2);
```

## Employee Management Application By using Predicate :-

Example :-

```
import java.util.ArrayList;
class Employee:
{
 String name;
 String designation;
 double salary;
 String city;

Employee(String name, String designation, double salary, String city)
 {
   this.name = name;
   this.designation = designation;
   this.salary = salary;
   this.city = city;
 }

  Public string toString()
  {
    String s = String.formate("%s, %s, %f, %s)", name, designation,
                                %0.2f → for 2 decimal Point
                                        Salary, city);
      return s;
  }

}

   Class Test
   {
      Public static void main(String[] args)
      {
ArrayList<Employee> list = new ArrayList<Employee>();
  Employee e1= new Employee("Durga", "CEO", 30000, "Hyderabad");
  Empolee e2= new Employee("Sunny", "Manager, 20000, "Bangulure");
    list.add(e1);
    list.add(e2);
```

```
    System.out.println(list);
  }
}
```

Output:-

```
[(Durga, CEO, 30000.00, Hyderabad),
 (Sunny, Manager, 20000.00, Bangalore)]
```

## With using Predicate for above Example

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

  Predicate<Employee> P1 = emp -> emp.designation.equals("Manager");
System.out.println("Mangers Information");
    display(P1, list);

Predicate<Employee> P2 = emp -> emp.city.equals("Banglore");
  System.out.println("Bangalore Employee Information");
    display(P2, list);

    public static void Populate(ArrayList<Employee> list)
    {
    list.add(new Employee("Durga", "CEO", 30000, "Hyderabad"));
    list.add(new Employee("Sunny", "Manager", 20000, "Hyderabad"));
    list.add(new Employee("Mallika", "Manager", 20000, "Bangalore"));
    list.add(new Employee("Kareena", "Lead", 15000, "Hyderabad"));
    list.add(new Employee("Katrina", "Lead", 15000, "Bangalore"));
    list.add(new Employee("Anushka", "Developer", 10000, "Hyderabad"));
    list.add(new Employee("Kanishka", "Developer", 10000, "Hyderabad"));
    list.add(new Employee("Soumya", "Developer", 10000, "Bangalore"));
    list.add(new Employee("Ramya", "Developer", 10000, "Bangalore"));
}
```

```
Public static void display(Predicate<Employee> p, ArrayList<Employee> list)
 {
   for(Employee e: list)
   {
     if(p.test(e))
      {
        System.out.println(e);
      }
   }
  System.out.println("*********************");
 }
}
```

Output:

```
Managers Information:
[Sunny ;Manager , 20000.00 , Hyderabad]
[Mallika , Manager, 20000.00, Bangalore]
**********
Bangalore Employees Information:
[Mallika , Manager, 20000.00 , Bangalore]
[Katrina , Lead , 15000.00 , Bangalore]
[Soumya , Developer , 10000.00, Bangalore]
[Ramya , Developer , 10000.00 , Bangalore]
**********
```

## We can use some other Predicates for Last Example:

```
{ Predicate<Employee> p3 = emp -> emp.Salary < 20000;
  System.out.println("All Employee Information whose salary < 20000:");
  display(p3, list);
```

~~Predicate<Employee> p4 = emp -> emp.~~

```
{ System.out.println("All mangers from Bangalore to give Pink Slip:");
  display(p1.and(p2), list);
{ System.out.println("All Employee who are mangers or Salary < 20000:");
  display(p1.or(p3), list);
{ System.out.println("All Employee who are not Managers:");
  display(p1.negate(), list);
```

## Predicate Interface isEqual() method

```
interface Predicate<T>
{
    public boolean test(T t);
    and()
    or()      } Default methods
    negate

    Public static Predicate isEqual(T t); } static Method
}
```

Example:

```
import java.util.function.predicate;
class Test
{
```

```
Public Static Void main(String[] args)
{
    Predicate<String> p = Predicate.isEqual("DurgaSoft");
    System.out.println(p.test("DurgaSoft")); //true
    System.out.println(p.test("Mallika")); //false
}
}
```

Output:

True
false.

Example:

```
Predicate<Employee> isCEO = Predicate.isEqual(new Employee("Durga",
                "CEO", 30000, "Hyderabad"));
Employee e1 = new("Durga", "CEO", 30000, "Hyderabad");
Employee e2 = new("Sunny", "Manager", 20000, "Hyderabad");
System.out.println(isCEO.test(e1)); //true
System.out.println(isCEO.test(e2)); //false.
```

# Predefined functional Interfaces – Function

- To Perform certain operation and to return some result, we should go for Function.
- Function can take 2 type Parameters First one represent input argument type and second one represent return type

`Function<T, R>`

- Function interface defines one abstract method called

`apply()`

- Public R apply(T t)
- Function can return any type of value.

Example:

```
import java.util.function.Function;
Public class Test
{
 Public static void main(String[] args)
 {
  Function<String, Integer> f = s ⟶ s.length();
  System.out.println(f.apply("durga");
  System.out.println(f.apply("durgasoft"));
 }
}
```

Output:

```
5
9
```

Example:

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

output:

```
25
100
```

* Program to remove spaces present in the given String by using Function

Example:

```
import java.util.function.*;
Class Test
{
  Public static void main(String[] args)
  {
    String s = "durga software solutions hyderabad";
    Function<String, String> f = s1 -> s1.replaceAll(" ", "");
    System.out.println(f.apply(s));
}}
```

output:

```
durgasoftwaresolutionshyderabad
```

*Example:- Count spaces from the String using Function;

```
import java.util.function.*;
class Test
{
 Public static void main(String [] args)
 {
   String s = "durga software solutions hyderabad";
   Function<String, Integer> f = s1 -> s1.length() - s1.replaceAll(" ","")
                                                      .length();
   System.out.println(f.apply(s));
 }
}
```

output:-

3

* Program to find Student Grade by using Function

Example:-

```
import java.util.function.*;
import java.util.*;
Class Student
{
   String name;
   int marks;
   student(string name, int marks)
   {
      this.name = name;
      this.marks = marks;
   }
}
class Test
{
    Public static void main(String[] args)
    {
      ArrayList <student> l = new ArrayList<student>();
        Populate(l);
```

```
Function<Student, String> f = s -> {
        int marks = s.marks;
        if(marks >= 80)
        {
          return "A[Dichtinction]";
        }
        else if(marks >= 60)
        {
          return "B[First class]";
        }
        else if (marks >= 50)
        {
          return "C[second class]";
        }
        else if (marks >= 35)
        {
          return "D[Third class]";
        }
        else
        {
          return "E[failed]";
        }
    };
    for (Student s : l)
    {
     Soutpln("Student Name:"+s.name);
     Soutpln("Student Marks:"+s.marks);
     Soutpln("Student Grade:"+f.apply(s));
     Soutpln();
    }
}
Public static void populate(ArrayList<Student> l)
{
    l.add(new Student("Sunny", 100));
    l.add(new Student("Bunny", 65));
```

```
        l.add(new Student("Chinny", 55));
        l.add(new Student("Vinny", 45));
        l.add(new Student("Pinny", 25));
    }
}
```

output:

```
Student Name: Sunny
Student Marks: 100
Student Grade: A[Distinction]

Student Name: Bunny
Student Marks: 65
Student Grade: B[First class]

Student Name: Chinny
Student Marks: 55
Student Grade: C[Second class]

Student Name: Vinny
Student Marks: 45
Student Grade: D[Third class]

Student Name: Pinny
Student Marks: 25
Student Grade: E[Failed].
```

*Need to add predicate in above program where we will check student marks is >=60. those student details need to print.

```
Predicate<Student> p = s.marks.>=60;
for(Student s : l)
{
    if(p.test(s))
    {
        soutpln("Student Name: "+s.name);
        soutpln("Student Marks: "+s.marks);
        soutpln("Student Grade: "+f.apply(s));
        soutpln();
    }
}
```

output:
Student Name: Sunny
Student Marks: 100
Student Grade: A[Distinction]

Student Name: Bunny
Student Marks: 65
Student Grade: B[First class]

## * Program to find Total Monthly Salary of All Employees by using Function

Example:

```
import java.util.*;
import java.util.function.*;
class Employee
{
  String names;
  double salary;
  Employee(String name, double salary)
  {
     this.name = name;
     this.salary = salary;
  }
 Public String toString()
 {
    return name + ":" + salary;
  }
}
  class Test
 {
   Public static void main(String[] args)
    {
     ArrayList<Employee> l = new ArrayList<Employee>();
      Populate(l);
      Soutpln(l);
      Function<ArrayList<Employee>, Double> f = ll ->{
            double total=0;
        for(Employee e : ll)
        {
           total = total + e.salary;
        }
        return total;
      };
```

```
System.out.println("The total salary of this month:"+
                                    f.apply(l));
}

Public Static Void populate(ArrayList<Employee> l)
{
    l.add(new Employee("Sunny", 1000));
    l.add(new Employee("Bunny", 2000));
    l.add(new Employee("Chinny", 3000));
    l.add(new Employee("Pinny", 4000));
    l.add(new Employee("Vinny", 5000));
  }
}
```

Output:

```
[Sunny:1000.0, Bunny:2000.0, Chinny:3000.0
Pinny:4000.0, Vinny:5000.0]
The total salary of this month: 15000.0
```

## * Program to perform Salary Increment for Employees by using Predicate & Function

Example:

```
import java.util.*;
import java.util.function.*;
Class Employee
{
   String name;
   double salary;
  Employee(String name, double salary)
  {
     this.name=name;
     this.salary = salary;
  }
```

```
Public String toString()
 {
   return name+":"+salary;
 }
}
class Test
{
  Public static void main(String[] args)
  {
    ArrayList<Employee> l = new ArrayList<Employee>();
        Populate(l);
    System.out.println("Before Increment:");
    System.out.println(l);

    Predicate<Employee> p = e -> e.salary<3500;
    Function<Employee, Employee> f = e -> {
                e.salary = e.salary+477;
               return e;
     };
    System.out.println("After Increment:");

    ArrayList<Employee> l2 = new ArrayList<Employee>();
       for(Employee e: l)
       {
          if(p.test(e))
          {
            l2.add(f.apply(e));
          }
       }
       System.out.println(l);
       System.out.println("Employees with incremented salary:");
       System.out.println(l2);
  }
```

```
Public static void populate(ArrayList<Employee> l)
{
    l.add(new Employee("Sunny", 1000));
    l.add(new Employee("Bunny", 2000));
    l.add(new Employee("Chinny", 3000));
    l.add(new Employee("Pinny", 4000));
    l.add(new Employee("Vinny", 5000));
    l.add(new Employee("Durga", 10000));
}
}
```

Output:

```
Before Increment:
[Sunny:1000.0, Bunny:2000.0, Chinny:3000.0,
 Pinny:4000.0, Vinny:5000.0, Durga:10000]
After Increment:
[Sunny:1477.0, Bunny:2477.0, Chinny:3477.0,
 Pinny:4000.0, Vinny:5000.0, Durga:10000]
Employees with incremented Salary:
[Sunny:1477.0.
 Bunny:2477.0
 Chinny:3477.0
]
```

## Function Chaining:

function interface contain 2 defaults methods. to use function chaining.

① f1.andThen(f2) → f1 will be applied followed by f2
② f1.compose(f2) → f2 will be applied followed by f1

here f1 & f2 are 2 functions.

**Example:-**

```
import java.util.function.*;
Class Test
{
    Public Static void main(String[] args)
    {
        Function<String, String> f1 = s -> s.toUpperCase();
        Function<String, String> f2 = s -> s.substring(0,9);

        Sopln("The Result of f1 : "+ f1.apply("AishwaryaAbhi"));
        Sopln("The Result of f2 :"+f2.apply("AishwaryaAbhi"));
        Sopln("The Result of f1.andThen(f2):"+ f1.andThen(f2).
                                    apply("AishwaryaAbhi"));
        Sopln("The Result of f1.compose(f2);"+ f1.compose(f2).
                                    apply("AishwaryaAbhi"));
    }
}
```

**output:**

```
The Result of f1 : AISHWARYAABHI
The Result of f2 : Aishwarya
The Result of f1.andThen(f2) : AISHWARYA
The Result of f1.compose(f2) : AISHWARYA
```

## * Program to Demonstrate the difference between andThen() and Compose();

Example:

```
import java.util.function.*;
Class Test
{
  Public static void main(String[] args)
  {
    Function<Integer, Integer> f1 = i -> i+i;
    Function<Integer, Integer> f2 = i -> i*i*i;
    System.out.println(f1.andThen(f2).apply(2));
    System.out.println(f1.compose(f2).apply(2));
  }
}
```

Output:

```
64
16
```

## * Program for User Authentication by using Function Chaining

Example:-

```
import java.util.function.*;
import java.util.*;
class Test
{
  Public static void main(String[] args)
  {
    Function<String, String> f1 = s -> s.toLowerCase();
    Function<String, String> f2 = s -> s.substring(0, 5);
```

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter User name:");
String username = sc.next();
System.out.println("Enter Password:");
String pwd = sc.next();
if(f1.andThen(f2).apply(username).equals("durga")
        && pwd.equals("Java")){
  {
     System.out.println("Valid User");
  }
 else
 {
   System.out.println("Invalid User");
 }
}
}
```

output:

```
Enter User Name:
durga
Enter Password:
Java
Valid User
```

```
Enter User Name:
durga software solutions
Enter Password:
Java
Valid User
```

```
Enter User Name:
DURGA TECHNOLOGIES
Enter Password:
Java
Valid User
```

```
Enter User Name:
JavaJava
Enter Password:
Java
Invalid User.
```

## * Function interface Static Method : identity()

Function Interface contains Below methods.

apply() } abstract method
andThen() } Default method
compose() }
identity() } static method

* whatever the input we are providing same output will be provided.

- static <T> Function<T, T> identity()
- Returns a function that always returns its input argument.

Example :-

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        Function<String, String> f1 = Function.identity();
        String s2 = f1.apply("durga");
        System.out.println(s2);
    }
}
```

output :- durga

# * Predefined Functional Interfaces - Consumer

- Consumer Interface is a part of java.util.function Package which has been introduced since Java 8, to implement functional programming in Java. It represents a function which takes in one argument and produces a result. However these kind of functions don't return any value.

Hence this functional interface which takes in one generic namely:-

- T: denotes the type of the input argument to the operation.

The lambda expression assigned to an object of Consumer type is used to define its accept() which eventually applies the given operation on its argument. Consumers are useful when it not needed to return any value as they are expected to operate via side-effects.

```
interface Consumer<T>   // T → input type
{
    void accept(T t);
}
```

## Functions in Consumer Interface :-

The Consumer interface consists of the following two functions:

1. void accept(T t)
   - this method takes in one parameter:
     - t - the input argument
   - this method does not return any value.

2. default Consumer<T> andThen(Consumer<? super T> after)

   This method accepts a parameter after which is the Consumer to be applied after the current one.
   - this method throws NullPointerException if the after operation is null.

## Example :- use of accept method.

```
import java.util.function.Consumer;
Public class Test
{
    Public static void main(String[] args)
    {
```

```
        Consumer<String> c = s -> System.out.println(s);
          c.accept("Hello");
          c.accept("DURGASOFT");
      }
  }
```

output:

```
Hello
DURGASOFT
```

**Example:-** Program to display Movie information by Using Consumer

```
import java.util.function.*;
import java.util.*;
Class Movie
{
   String name;
   String hero;
   String heroine;
   Movie(String name, String hero, String heroine)
   {
         this.name = name;
         this.hero = hero;
         this.heroine = heroine;
   }
}

Class Test
{
   Public static void main(String[] args)
   {
     ArrayList<Movie> l = new ArrayList<Movie>();
      Populate(l);
     Consumer<Movie> c = m -> {
      System.out.println("Movie Name:" + m.name);
      System.out.println("Movie Hero:" + m.hero);
      System.out.println("Movie Heroine:" + m.heroine);
     }; System.out.println();
```

```
    for (Movie m : l)
     {
       c.accept(m);
     }
  }
  Public static void populate(ArrayList<Movie> l)
  {
      l.add(new Movie("Bahubali", "Prabhas", "Anushka"));
      l.add( new Movie("Rayees", "Sharukh", "Sunny"));
      l.add( new movie("Dangal", "Ameer", "Ritu"));
      l.add( new Movie("Sultan", "Salman", "Anushka"));
   }
}
```

output:

```
Movie Name: Bahubali
Movie Hero: Prabhas
Movie Heroine: Anushka

Movie Name: Rayees
Movie Hero: Sharukh
Movie Heroine: Sunny

Movie Name: Dangal
Movie Hero: Ameer
Movie Heroine: Ritu
```

* Program to display Student information by using Predicate, Function and Consumer

```
import java.util.function.*;
import java.util.*;
class Student
{
  String name;
  int marks;
```

```
Student(String name, int marks)
 {
    this.name = name;
    this.marks = marks;
  }
}
class Test
{
   Public static void main(String[] args)
   {
    ArrayList<Student> l = new ArrayList<Student>();
     Populate(l);
    Predicate<Student> P = s -> s.marks >=60;
     Function<Student, String> f = s -> {
         int marks = s.marks;
         if(marks >= 80)
         {
           return "A[Dictinction]";
         }
         else if(marks>=60)
         {
           return "B[First class]";
         }
         else if(marks >= 50)
         {
           return "C[Second class]";
         }
         else if(marks >=35)
         {
           return "D[Third class]";
         }
         else
         {
           return "E[failed]";
         }
    };
```

```
Consumer<Student> c=s->{
    System.out.println("Student Name:"+s.name);
    System.out.println("Student Marks:"+s.marks);
    System.out.println("Student Grade:"+f.apply(s));
    System.out.println();
};
for(Student s : l)
{
    if(p.test(s))
    {
        c.accept(s);
    }
}
}
Public static void populate(ArrayList<Student> l)
{
    l.add(new Student("Sunny", 100));
    l.add(new student("Bunny", 65));
    l.add(new student("Chinny", 55));
    l.add(new Student("Vinny", 45));
    l.add(new Student("Pinny", 25));
}
}
```

Output:

```
Student Name: Sunny
Student Marks: 100
Student Grade: A[Distinction]

Student Name: Bunny
Student Marks: 65
Student Grade: B[First class]

Student Name: Chinny
Student Marks: 55
Student Grade: C[Second class]

Student Name: Vinny
Student Marks: 45
Student Grade: D[Third class]

Student Name: Pinny
Student Marks: 25
Student Grade: E[failed]
```

## Consumer chaining: -

for Consumer chaining we use andThen() method and it is the default method.

- default Consumer<T> andThen (Consumer<? super T> after)

Example:

```
import java.util.function.*;
import java.util.*;
Class Movie
{
   String name;
   String result;
   Movie(String name, String result)
   {
     this.name = name;
     this.result = result;
   }
}
Class Test
{
  Public static void main(String[] args)
  {
  Consumer<Movie> c1 = m -> System.out.println(
                "Movie:" + m.name + " is ready to release");
  Consumer<Movie> c2 = m -> System.out.println("Movie:"+
                m.name+" is just Release and it is:"
                + m.result);
  Consumer<Movie> c3 = m -> System.out.println("Movie:"+
                m.name + " information storing in the database");
   Consumer<Movie> chainedC = c1.andThen(c2).andThen(c3);

   Movie m1 = new Movie("Bahubali", "Hit");
    chainedC.accept(m1);
  }}
```

Output:-

Movie : Bahubali is ready to release
Movie : Bahubali is Just Released and it is : Hit
Movie : Bahubali information storing in the database

## *Supplier Interface:

The Supplier interface is a part of the java.util.function pacakge which has been introduced since Java 8, to implement functional Programming in Java. It represents a function which does not take in any argument but produces a value of type T.

Hence this functional interface takes in only one generic namely:-

- T: denotes the type of the result.

The lambda expression assigned to an object of supplier type is used to define its get() which eventully produce a value. Suppliers are useful when we don't need to supply any value and obtain a result at the same time.

The Supplier interface consists of only one function:

1. get()

The method does not take in any argument but produces a value of type T.

Syntax: R get()

This method returns a value of type R.

Return type

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

## * Program to get System Date by using Supplier Interface

```
import java.util.function.Supplier;
import java.util.Date;
Public class Test
{
   Public static void main(String[] args)
   {
      Supplier<Date>  s = () → new Date();
      System.out.println(s.get());
   }
}
```

Output: Sat Aug 11 11:17:36 IST 2018

## * Program to get Random Name by Using Supplier

```
import java.util.function.Supplier;
class Test
{
   Public static void main(String[] args)
   {
      Supplier<String> s = () → {
         String[] s1 = {"Sunny","Bunny","Chinny","Vinny"};
         int x = (int)(Math.random()*4);
         return s1[x];
      };
   System.out.println(s.get());
   }
}
```

Output:
Chinny
Bunny
Vinny
} it will generate the Random name

## * Program to get Random OTP by using Supplier

```
import java.util.function.*;
Class Test
{
  Public Static Void main (String[] args)
  {
     Supplier<String>  otps = () -> {
      String otp = "";
      for( int i =1; i<=6; i++)
      {
        otp =otp+ (int) (Math.random() *10);
      }
     return otp;
    };
   System.out.println(otps.get());
  }
}
```

Output:-

```
089733
------
061958
------
999586
```

## * Program to get Random Password by using Supplier:-

```
import java.util.function.*;
import java.util.*;
Class Test
{
    Public Static Void main(String[] args)
     {
      Supplier<Integer> d = () -> (int) (Math.random()*10);
```

69.

```
String symbols = "ABCDEFGHIJKLMNOPQRSTUVWXYZ#$@";
Supplier<character> c= ()→ symbols.charAt((int)(Math.random()*29));
String pwd ="";
for(int i =1; i<=8; i++)
{
    if(i%2==0)
    {
        pwd = pwd + cl.get();
    }
    else
    {
        pwd = pwd + c.get();
    }
}
return pwd;
};
System.out.println(s.get());
System.out.println(s.get()); System.out.println(s.get());
System.out.println(s.get()); System.out.println(s.get());
System.out.println(s.get()); System.out.println(s.get());
System.out.println(s.get()); System.out.println(s.get());
}
}
```

Output:

A8B5L6W1
N1R8X1Z1
K5F8@1W7
X2U1K5W7
1+999E8C2
T5P2P1N3
X2Q8O4#3
Q1U6G9!+9
P8W6X5J3

## * Comparison Table of Predicate, Function, Consumer and Supplier

### 1. Predicate :-

Property :-

a. Purpose :- To take some input and perform some conditional check.

b. Interface :-
```
interface Predicate<T>
{
....
}
```

c. Single Abstract Method (SAM) :-
```
Public boolean test(T t);
```

d. Default Methods :- and(), or(), negate()

e. Static Methods :- isEqual().

### 2. Function :-

Property :-

a. Purpose :- To take some input and perform required operation and return the result.

b. Interface :-
```
interface Function<T, R>
{....
}
```

c. Single Abstract Method (SAM) :-
```
Public R apply(T t);
```

d. Default Methods :- andThen(), Compose()

e. Static Methods :- identity()

## 3. CONSUMER :-

### Property :-

a. Purpose :- To consume some input and perform required operation but it will not return anything.

b. Interface :-
```
interface Consumer<T>
{
....
}
```

c. Single Abstract Method :- Public void accept(T t)

d. default Methods :- andThen()

e. Static Methods :- It don't have any static Methods.

## 4. Supplier :-

### Property :-

a. Purpose :- To Supply some value based on our requirement.

b. Interface :-
```
interface Supplier<R>
{
....
}
```

c. Single Abstract Methods :- Public R get();

d. default Methods :- It don't have default Methods

e. Static Methods :- It don't have static Methods.

## BiPredicate interface:-

- java.util.function.BiPredicate is a functional interface whose method is boolean test(T t, U u).
- The BiPredicate interface represents an operation that takes two arguments (T, U) and returns a boolean result.
- This is same as Predicate interface and also have same methods like Predicate. but Predicate take one argument and BiPredecate take two arguments.
- Methods in BiPredicate:-
  1. boolean test(T t, U u); } abstract method
  2. and() } defaults method
  3. or()
  4. negate() } Static method.

Example: test() method of the BiPredicate interface with lambda Expression.

```
import java.util.function.*;
class Test
{
  Public static void main(String args)
  {
    BiPredicate<Integer, Integer> P=(a,b)->(a+b)%2==0
    System.out.println(P.test(10,20));
    System.out.println(P.test(15,20));
  }
}
```

Output:-

true
False.

**Example:** Use of defaults methods/static Methods (and(), or() and negate()) of the Bipreclicate Interface with Lambda Expression.

```
import java.util.function.Bipredicate;
Public class Test
 {
   Public static void main(String[] args)
   {
   Bipredicate<Long, Long> P1 = (x, y) → x > y;
   BiPredicate<Long, Long> P2 = (x, y) → x == y;
  // Using and(), or() and negate()
  System.out.println(P1.and(P2).test(51, 51));
  System.out.println(P1.and(P2).test(61, 52));
  System.out.println(P1.negate().test(81, 11));
 }
}
```

**Output:**

```
false
true
false
```

**BiFunction Interface :-** It is a Part of java.util.function Package which has been introduced since Java 8, to implement function programming in Java. It represent a function which takes in two arguments and produce a result.

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

- This functional interface which takes in 3 parameters namely:
  - T: denotes the type of the first argument to the function
  - U: denotes the type of the second argument to the function
  - R: denotes the return type to the function.

- The Lambda Expression assigned to an Object of Bifunction type is used to define its apply() which eventually applies the given function on the 2 arguments. The main advantage of using a Bifunction is that it allows us to use 2 input arguments while in function we can only have 1 input argument.

## Functions in Bifunction Interface :

The Bifunction interface consists of the following two functions:

1. apply()

   Syntax: `R apply(T t, U u);` (R: return type, T: first argument, U: second argument)

2. addThen()

   Syntax:
   ```
   default <R>
       BiFunction <T, U, R>
           andThen (Function <? super R,
                              ? extends V> after)
   ```

- where R is the type of output of the after function, and of the Composed function.
- This method accepts a parameter after which is the function to be applied after this function is one.
- This method returns a Composed Function that first applies the current function first and then the after function.
- This method throws NullPointerException if the after function is null.

Note: The function being passed as the argument should be of type Function and not Bifunction.

• It returns a composed function where in the Parameterized function will be executed after the first one. If evaluation of either function throws an error, it is relayed to the caller of the composed function.

Example:-

```
import Java.util.function.*;
Class Test
{
  Public static void main(String[] args)
  {
   Bifunction<Integer, Integer, Integer> f = (a, b) -> a*b;
    System.out.println(f.apply(10,20));
    System.out.println(f.apply(100,200));
  }
}
```

output: 200
20000

Example:-

```
Import Java.util.function.*;
Class Test
{
  Public static void main(String[] args)
  {
  Bifunction<Integer, Integer, Integer> f1 = (a,b) -> a+b;
      f1 = f1.andThen(a -> 2*a);
  System.out.println(f1.apply(2,3));
  Bifunction<Integer, Integer, Integer> f2 = (a,b) -> a*b;
     f2 = f2.andThen(a -> 3*a);
}} System.out.println(f2.apply(2,3));
```

Output:- 10
18

Example:-

```
Impot java.util.function.Bifunction;
  Public class Test
  {
    Public Static Void main(String args[])
    {
     Bifunction<Integer, Integer, Integer> f1 = (a,b)->a+b;
     try{
       f1 = f1.apndthen(null);
      System.out.println(f1.apply(2,3));
     }
     catch(Exception e){
       System.out.println("Exception: "+e);
    }}
}
```

Output;

Exception: java.lang.NullPointerException.

Note:
- It is not possible to add a Bifunction to an Existing Bifunction using andThen()
- Bifunction interface is useful when it is needed 2 Parameters, unlike function interface which allows to Pass only one. However; it is posible to cascade two or more function objects to form a Bifunction but in that case it won't be possible to use both parameters at the same time. This is utility of Bifunction.

Example: ~~Creation of Student object by taking name and Rollno as input with Bifunction.~~

Calculate Monthly Salary with Employee & TimeSheet by using Bifunction.

```
import java.util.function.*;
import java.util.*;
class Employee
{
 int eno;
 String name;
 double dailywage;
 Employee(int eno, String name, double dailywage)
 {
    this.eno = eno;
    this.name = name;
    this.dailywage = dailywage;
 }
}
class TimeSheet
{
  int eno;
  int days;
  Timesheet(int eno, int days)
   {
     this.eno = eno;
     this.days = days;
   }
}
Class Test
{
 BiFunction<Employee, Timesheet, Double> f = (e,t) ->
                    e.dailywage * t.days;
 Employee e = new Employee(101, "Durga", 15000);
 TimeSheet t = new Timesheet(101, 25);
 System.out.println("Employee Monthly Salary:" +f.apply
                                          (e,t));
 }
}
```

Output:- Employee Monthly Salary: 37500.0

**Example:-** Creation of Student object by taking Name and Rollno as input with BiFunction.

```
import java.util.function.*;
import java.util.*;
class Student
{
  String name;
  int rollno;
  Student(String name, int rollno)
  {
    this.name = name;
    this.rollno = rollno;
  }
}

Class Test
{
  Public static void main(String[] args)
  {
  ArrayList<Student> l = new ArrayList<Student>();
  BiFunction<String, Integer, Student> f
        =(name, rollno)-> new Student(name, rollno);
   l.add(f.apply("Durga", 100);
   l.add(f.apply("Ravi", 200);
   l.add(f.apply("Shiva", 300);
   l.add(f.apply("Pavan", 400);
   for(Student s: l)
   {
     System.out.println("Student Name: "s.name);
     System.out.println("Student Rollno: "s.rollno);
     system.out.println();
  }}
```

**Output:**

| | |
|---|---|
| Student Name: Durga | Student Name: Ravi |
| Student Rollno: 100 | Student Rollno: 200 |
| Student Name: Shiva | Student Name: Pavan |
| Student Rollno: 300 | Student Rollno: 400 |

## BiConsumer :-

- The BiConsumer interface is a part of the java.util.function package which has been introduced since Java 8, to implement functional programming in Java. It represents a function which takes in two arguments and produces a result. However these kind of functions don't return any value.
- The functional interface takes in two generics, namely :-
  - T: denotes the type of the first input argument to the operation
  - U: denotes the type of the second input argument to the operation.
- The Lambda expression assigned to an object of BiConsumer type is used to define its accept() which eventually applies the given operation on its arguments.
- BiConsumers are useful when it is not required to return any value as they are expected to operate via side-effects.

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

### Functions in BiConsumer Interface :

The BiConsumer interface consists of the following two functions

① **accept() :-**

- This method accepts two values and performs the operation on the given arguments.

Syntax : `void accept(T t, U u)`

- `void` — This method does not return any value.
- `T t` — first input argument
- `U u` — second input argument

② **andThen() :**

It returns a composed BiConsumer wherein the parameterized BiConsumer will be executed after the first one. If evaluation of either operation throws an error, it is relayed to the caller of the composed operation.

- Note:- The operation being passed as the argument should be of type BiConsumer.

Syntax:

```
default BiConsumer<T, U>
    andThen(BiConsumer<? super T, ? super U> after)
```

- This method accepts a parameter after which is the BiConsumer to be applied after the current one.
- This method returns a composed BiConsumer that first applies the current operation first and then the after operation.
- This method throws NullPointerException if the after operation is null.

Example:-

```
import java.util.function.*;
class Test
{
  Public static void main(String[] args)
  {
   BiConsumer<String, String> c =
              (s1, s2) -> System.out.println(s1+s2);
      c.accept("durga", "soft");
  }
}
```

output:

durgasoft

Example:- Demo Program to increment Employee salary by using Bifunction & BiConsumer:

```
import java.util.function.*;
import java.util.*;
class Employee
{
  String name;
  String salary;
  Employee(String name, double salary)
  {
    this.name = name;
    this.salary = salary;
  }
}
Class Test
{
  Public static void main(String[] args)
  {
   ArrayList<Employee> l = new ArrayList<Employee>();
   Bifunction<String, Double, Employee> f =
              (name, Salary) -> new Employee(name, Salary);
    l.add(f.apply("Durga", 1000.0));
    l.add(f.apply("Sunny", 2000.0));
    l.add(f.apply("Bunny", 3000.0));
    l.add(f.apply("Chinny", 4000.0));
BiConsumer<Employee, Double> c = (e, d) -> e.salary
                                  = e.salary + d;
   for(Employee e: l)
   {
      e.accept(e, 500.0);
   }
   for(Employee e: l)
   {
```

```
System.out.println("Employee Name:"+e.name);
System.out.println("Employee Salary:"+e.Salary);
System.out.println();
  }
 }
}
```

Output:

```
Employee Name : Durga
Employee Salary : 1500.0
&
Employee Name : Sunny
Employee Salary : 2500.0
```

```
Employee Name : Bunny
Employee Salary : 3500.0

Employee Name : Chinny
Employee Salary : 4500.0
```

Example:-

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

Public class Main{

Public Static void main(String args[])
{
  List<Integer> list1 = new ArrayList<Integer>();
   list1.add(2);
   list1.add(1);
   list1.add(3);

  List<Integer> list2 = new ArrayList<Integer>();
   list2.add(2);
   list2.add(1);
   list2.add(2);
```

03.

```
BiConsumer<List<Integer>,List<Integer>> equals =
        (list1 , list2) -> {
        if(list1.size() != list2.size()){
          System.out.println("false");
        }
        else{
          for(int i = 0; i < list1.size(); i++){
            if (list1.get(i) != list2.get(i) {
              System.out.println ("false");
              return;
            }
            system.out.println("true");
          }
        };

BiConsumer<List<Integer>, List.<Integer>> disp =
            (list1 , list2) -> {

        list1.stream().forEach(a -> System.out.print(a +" "));
        System.out.println();
        list2.stream().forEach(a -> System.out.println(a +" "));
        System.out.println();
    };

    equals.andThen(disp).accept( list1 , list2);
  }
}
```

Output:

```
false
2 1 3
2 1 2
```

# Comparision table Between one argument & Two argument Fenctional Interface

| One Argument Fenctional Interface | Two Argument fenctional Interface |
|---|---|
| 1. interface Predicate<T><br>{<br>Public boolean test(T t);<br>Default Predicate and(Predicate P);<br>Default Predicate or(Predicate P)<br>Default predicate negate();<br>Static predicate isEqual(object O);<br>} | 1. interface BiPredicate<T, U><br>{<br>Public boolean test(T t, U u);<br>Default BiPredicate and(BiPredicate P);<br>Default BiPredicate or(BiPredicate P);<br>Default BiPredicate negate();<br>} |
| 2. interface Fenction<T, R)<br>{<br>Public R apply(T t);<br>Default function andThen(Function f)<br>Default Function Compose(Function f)<br>Static Function identity()<br>} | 2. interface BiFunction<T, U, R><br>{<br>Public R apply(T t, U u);<br>Default BiFunction andThen(<br>Function f)<br>} |
| 3. interface Consumer<T><br>{<br>Public void accept(T t);<br>Default Consumer andThen(<br>Consumer c);<br>} | 3. Interface BiConsumer<T, U><br>{<br>Public void accept(T t,<br>U u);<br>Default BiConsumer andThen(<br>BiConsumer c);<br>} |

Note :- Suplier Interface Never take the any Input argument so here we did not mentioned Suplier methods.

# Primitive type Function Interfaces :-

## ① Autoboxing :-

Incompatible Type error

Integer I = 10 → until 1.4 version it was invalid after 1.5 it is valid and it is called Autoboxing.

↓ after compilation

Integer I = Integer.valueOf(10);

Internally Compiler is responsible to replace this code.

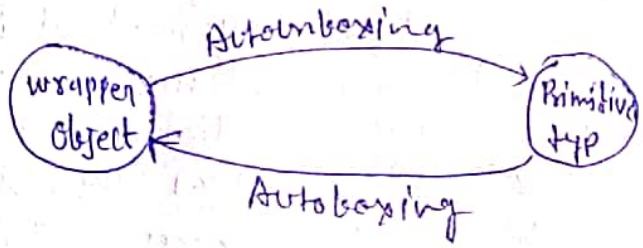
## ② Auto Unboxing :-

Integer I = new Integer(10);
int x = I;

→ this was invalid Unitil 1.4 V but 1.5 V onward this valid and it is called Auto-unboxing.

int x = I.intValue();

Internally Compiler is responsible do replace this Code.




## ③ Generics - Type Parameter :

AL<Integer> l = new AL<Integer>(); ✓
→ Type Parameter

AL<int> l = new AL<int>(); ✗

- for the Type Parameter only allowed Object type Primitive type don't allow to pass.

# Need of Primitive Type functional Interfaces



Example:-

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

```
1.  int[] x = {0, 5, 10, 15, 20, 25}
2.  Predicate<Integer> P = i ⟶ i%2 == 0;
3.     for(int x1 : x)
4.     {
5.        if(P.test(x1))
6.        {
7.           Sopln(x1);
8.        }
9.     }
```

Integer value convert into int value

int {Autounboxing}

Integer {Autoboxing}

int value convert into Integer

Performance wise above code is not up to the Mark.

Above code have below problem.

- At line 2. the predicate taking the Integer type Parameter but our input argument (at line 1) is int type. So at line 5 when we pass int value and it will execute the Predicate that time the int will convert into Interger Type. It is called Autoboxing. to Execute the Expression (i%2==0) here i required the int type value then again Integer will convert into int type. this Conversion is called Autounboxing. So for all input Parameter if this Conversion will be happened then Performance will be degrade. To improve the Performance Primitive Type fenctional Interface Come into Picture.

Note:- Normal fenctional interfaces not best suitable for the primitive type.

Example:-

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

```
{ Function<Integer, Integer> f = i -> i*i;
  Sopln(f.apply(10));
}
```

① Converting int value To Integer {AutoBoxing}

② Converting Integer to int value {AutoUnboxing}

③ after executing the Expression (i*i) the result will come in int but the return type is Integer Type so again the int value will be converted into Integer {AutoBoxing}

- Due to these Conversion Performance will be degrade. To improve this Performance we never use the normal functional Interfaces for Primitive.

## Primitive Type Functional Interfaces for Predicate

① interface IntPredicate
{
  Public boolean test(int i);
} // remainings methods will be same.

Example:- IntPredicate

```
import java.util.function.*;
class Test{
Public static void main(String[] args)
{
    int[] x ={0,5,10,15,20,25};
    IntPredicate p = i -> i%2 ==0
    for(int x1 : x)
    {
      if(p.test(x1))
      {
        System.out.println(x1);
      }
    }
}
}
```

output: 0
10
20

(2) interface LongPredicate
{
Public boolean test(long d);
= // other methods are same
}

(3) interface DoublePredicate
{
Public boolean test(double d);
= // remaining methods will be same
}

## Demo Programes about Primitive Type functional Interfaces for Function

1. Example:-

```
imports java.util.function.*;
class Test
{
  Public static void main(String[] args)
  {
    IntFunction<Integer> f=i->{i*i;};
    System.out.println(f.apply(5));
  }
}
```

Output: 25

2. Example :-

```
imports java.util.function.*;
Class Test {
  Public static void main(String[] args)
  {
    ToIntFunction<String> f=s->s.length();
    System.out.println(f.applyAsInt("durga"));
  }
}
```

Output:- 5

3. Example:-

```
imports java.util.function.*;
Class Test {
  Public Static void main(String[] args)
  {
    IntToDoubleFunction f = i -> Math.sqrt(i);
    System.out.println(f.applyAsDouble(5));
  }
}
```

Output: 25

# All 15 primitive Type functional Interfaces for function

1. IntFunction<R>
   Public R apply(int i);
2. LongFunction<R>
   Public R apply(long l);
3. DoubleFunction<R>
   Public R apply(Double d);

- It is used to control on input type
- return type can be any type

4. ToIntFunction<T>
   Public int applyAsInt(T t);
5. ToLongFunction<T>
   Public long applyAsLong(T t);
6. ToDoubleFunction<T>
   Public double applyAsDouble(T t);

- It is used to control on return type
- input can be any type

7. IntToLongFunction
   Public long applyAsLong(int i);
8. IntToDoubleFunction
   Public double applyAsDouble(int i);
9. LongToIntFunction
   Public int applyAsInt(long l);
10. LongToDoubleFunction
    Public double applyAsDouble(long l);
11. DoubleToIntFunction
    Public int applyAsInt(double d);
12. DoubleToLongFunction
    Public long applyAsLong(double d);

To control on Both Input & return type here both input & return type will be always Primitive type

13. ToIntBiFunction<T, U>
    Public int applyAsInt(T t, U u);

14. ToLongBiFunction<T, U>
    Public long applyAsLong(T t, U u);

15. ToDoubleBiFunction<T, U>
    Public double applyAsDouble(T t, U u);

- To Control on return type
- InPut Type can be any Type.

## Primitive Type Finctional Interfaces for Consumer

### 1. IntConsumer:-

```
interface IntConsumer
{
  Public void accept(int i);
}
```

### 2. LongConsumer:-

```
interface LongConsumer
{
  Public void accept(long l);
}
```

### 3. DoubleConsumer:-

```
interface DoubleConsumer
{
  Public void accept(double d);
}
```

Example:-
```
import java.util.function.*;
class Test
{
 public static void main(String[] args)
 {
 IntConsumer c = i -> System.out.println("The
                    square of i:"+(i*i));
 c.accept(10);
 }
}
```

Output:- The Square of i : 100

## Primitive Type functional Interfaces for BiConsumer

1. ObjIntConsumer<T>

```
public void accept(T t, int i);
```

2. ObjLongConsumer<T>

```
public void accept(T t, long l);
```

3. ObjDoubleConsumer<T>

```
public void accept(T t, double d);
```

above functional Interfaces are the primitive versions of BiConsumer. These functions are control on input type where one input type always be primitive type and second input always be object type.

See the Example on next Page

Example:-

```
import java.util.*;
import java.util.function.*;
class Employee
{
  String name;
  double salary;
  Employee(String name, double salary)
  {
     this.name = name;
     this.salary = salary;
  }
}
class Test
{
  Public static void main(String[] args)
  {
   ArrayList<Employee> l = new ArrayList<Employee>();
   Populate(l);
   ObjDoubleConsumer<Employee> c = (e, d) ->
                                   e.salary = e.salary + d;
   for (Employee e : l)
   {
     c.accept (e, 500.0);
   }
   for(Employee e : l)
   {
     System.out.println("Employee Name:" + e.name);
     System.out.println("Employee salary:" + e.salary);
     System.out.println();
   }
}
```

```
Public static void Populate(ArrayList<Employee> l)
{
    l.add(new Employee("Durga", 1000));
    l.add(new Employee("Sunny", 2000));
    l.add(new Employee("Bunny", 3000));
    l.add(new Employee("Chinny", 4000));
    l.add(new Employee("Pinny", 5000));
}
```

Output:

```
Employee Name: Durga
Employee Salary: 1500.0

Employee Name: Sunny
Employee Salary: 2500.0

Employee Name: Bunny
Employee Salary: 3500.0

Employee Name: Chinny
Employee Salary: 4500.0

Employee Name: Pinny
Employee Salary: 5500.0
```

# Primitive Functional Interfaces for Supplier

## 1. IntSupplier:-

```
interface IntSupplier
{
  public int getAsInt();
}
```

## 2. DoubleSupplier:-

```
interface DoubleSupplier
{
  public double getAsDouble();
}
```

## 3. LongSupplier:-

```
interface LongSupplier
{
  public long getAsLong();
}
```

## 4. BooleanSupplier:-

```
interface BooleanSupplier
{
  public boolean getAsBoolean();
}
```

**Example:-**

```
import 'java.util.function.*';
class Test
{
public static void main(String[] args)
{
IntSupplier s=()->(int)(Math.random()*10);
String obj="";
```

```
for(int i=0; i<6; i++)
  {
    otp = otp + s.getAsInt();
  }
System.out.println("The 6 digit OTP:"+otp);
System.out.println("The 6 digit OTP:"+otp);
}
}
```

Output;

The 6 digit OTP : 432696
The 6 digit OTP : 456966

## UnaryOperator and its Primitive Functional Interfaces:

* Input and return type are same type then we should go for UnaryOperator.
* Is a child of Function<T, T>

Example:-

```
import java.util.function.*;
Class Test
{
  Public static void main(String[] args)
  {
    UnaryOperator<Integer> f = i -> i*i;
    System.out.println(f.apply(10));
  }
}
```

Output: 100

Note:- In the above Program is having Autoboxing & Autounboxing. due to this Performence issue will be down. so we can go for primitive version of UnaryOperator instead of normal UnaryOperator.

## Primitive functional Interfaces Of UnaryOperator.

1. interface IntUnaryOperator

```
{
   Public int applyAsInt(int i);
}
```

2. LongUnaryOperator

```
interface LongUnaryOperator
{
   Public long applyAsLong(long l);
}
```

3. DoubleUnaryOperator

```
interface DoubleUnaryOperator
{
   Public double applyAsDouble(double d);
}
```

Example:

```
import java.util.function.*;
class Test
{
Public static void main(String[] args)
  {
     IntUnaryOperator f = i->i*i;
     System.out.println(f.applyAsInt(10));
  }
}
```

Output:- 100

## BinaryOperator :-

- The BinaryOperator interface<T> is a part of the java.util.function Package which has been introduced since Java 8, to implement functional programming in Java. It represents a binary operator which takes two operands and operates on them to produce a result. However, what distinguishes it from a normal Bifunction is that both of its arguments and return type are same.
- This functional interface which takes in one generic namely:
  - T denotes the type of the input arguments and the return value of the operation.
- The BinaryOperator<T> extends the Bifunction<T,T,T> type. So it inherits the following methods from the Bifunction Interface:
  - apply (T t, T u)
  - and Then (Function<? super R, ? extends V> after)

```
interface BinaryOperator<T>
{
    Public T apply(T t1, T t2);
}
```

Example:-

```
import java.util.function.*;
Class Test
{ Public static void main(String[] args)
  {
    BinaryOperator<String> b = (S1,S2) -> S1+S2;
    System.out.println(b.apply("durga", "software"));
} }
```

Output: durgasoftware

*in the last program we are having Autoboxing and Autounboxing which is create the performance program and it is not suitable. So instead of using normal BinaryOperator we will use Primitive Interfaces of BinaryOperater. So that the performance will be improved because we will not do there Autoboxing and Autounboxing.

## Primitive Functional Interfaces of BinaryOperater:

### 1. IntBinaryOperater:

```
interface IntBinaryOperater
{
  Public int applyAsInt(int i1, int i2);
}
```

### 2. LongBinaryOperater:

```
interface LongBinaryOperater
{
  Public long applyAsLong(long l1, long l2);
}
```

### 3. DoubleBinaryOperater:

```
interface DoubleBinaryOperater
{
  Public double applyAsDouble(double d1, double d2);
}
```

Example:

```
import java.util.function
Class Test {
  Public static void main(String args[])
  { IntBinaryOperater b = (i1, i2) -> i1+i2;
    System.out.println(b.applyAsInt(10,20));
}}
```

Output: 30

## Methods Reference By Double Colon(::) Operator:

In the Lambda Expression we provide the new implementation but in method reference we don't provide the new implementation. Already implemented method happy allow to reuse. So we can say method reference provide the code reusability. It is the feature of Method Reference. Method refence we denote with (::) double colon Operator.

Example:

```
Interface Interf
{
  Public void m1();
}
Public class Test
{
  public static void m2()
  {
   System.out.println("Implementation By
                       Method Reference");
  }
  Public static void main(String[] args)
  { Interf i = Test :: m2;
    i.m1();
  }
}
```

here we did not provide the implementation for m1 method instead of proving implementation we refer m2 method reference for M1 method. if I will call the m1() method then M2 Method will be executed.

Output:

Implementation By Method Reference

Note:- While using method reference we have one restriction. referer and refering method both method should have same argument type.

- functional interface can refer λ-expression to implement
- functional interface can also refer method reference also.
- method reference is the Alternative syntax to λ-expression

## Syntax for Method Reference :

### ① static method :-

```
Classname :: methodname
```

Eg:- Test :: m2

### ② instance method :-

```
Objectref :: methodname
```

Eg: Test t = new Test();
t :: m2

**Example:** Define a Thread by implementing Runnable interface with λ-expression :

```
Public class Test
{
  Public Static void main(String[] args)
  {
    Runnable r1 = ()->{                    // λ-Expression
                    for(int i=0; i<10; i++)
                    {System.out.println("Child Thread");
                    };}
    Thread t = new Thread(r1);
    t.start();
    for(int i=0; i<10; i++)
    {System.out.println("Main Thread");
    }
  }
}
```

Example:

## Define a Thread by implementing Runnable interface with method reference (::)

```
Public class Test
{
  Public void m1()
  {
    for(int i=0; i<10; i++)
    {
      System.out.println("Child Thread");
  } }
```

```
  Public static void main(String[] args)
  {
    Test t = new Test();
    Runnable r = t::m1;
                 → method reference (Instance)
    Thread t1 = new Thread(r);
      t1.start();
      for(int i=0; i<10; i++)
      {
        System.out.println("Main Thread");
      }
  }
}
```

Output:

| ① | ② |
|---|---|
| Main Thread | Child Thread |
| : | : |
| 10 | 10 |
| Child Thread | Main Thread |
| : | : |
| 10 | 10 |

## Constructor Reference By Double Colon (::) Operator

Constructor Reference also the alternate syntax of λ-Expression. It is same as method reference; argument type must be matched for Constructor reference also.

Example:

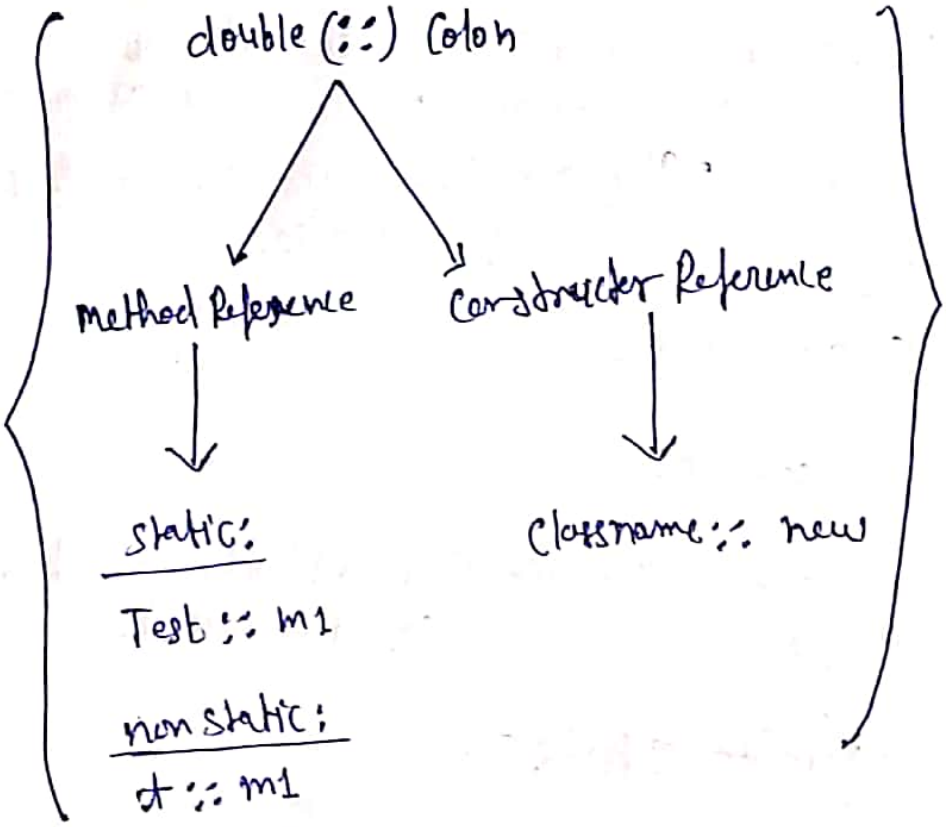
```
class Sample
{
  sample()
  {
  s.o.pln("sample class Constructor execution & Object Creation");
  }
}
```

```
Interface Test
{
  Public Static void main(String[] args)
  {
     Interf i = Sample::new;
       Sample s = i.get();
  }
}
```

```
Interface Interf
{
   Public Sample get();
}
```

Output:

Sample class Constructor execution & Object creation



# STREAMS API :-

- A Stream in Java can be defined as a sequence of elements from a source that supports aggregate operations on them. The source here refers to a collections or Arrays who provide data to a stream.
- Stream keeps the ordering of the data as it is in the source. The aggregate operations or bulk operations are operations which allow us to express common manipulations on stream elements easily and clearly.
- Before going Ahead, it is important to learn that Java 8 Streams are designed in such a way that most of the stream operations returns streams only. This help us creating chain of the stream operations. This is called pipe-lining.
- All of us have watch online videos on youtube or some other such website. When you start watching video, a small portion of file is first loaded into your computer and start playing. You don't need to download complete video before start playing it. This is called Streaming. I will try to relate this concept with respect to collections and differentiate with streams.
- At the basic level, the difference between Collections and Streams has to do with when things are Computed. A Collection is an in-memory data structure, which holds all the values that the data structure currently has - every Element in the collection has to be computed before it can be added to the collection. A stream is a conceptually fixed data structure in which elements are computed on demand. This gives rise to significant programing benifits. The idea is that a user will extract only the values they require from a stream, and these elements are only produced - invisibly to the user as and when required. This is a form of producer-Consumer relationship.

- In Java, java.util.Stream represents a Stream on which one or more operations can be performed. Stream operations are either intermediate or terminal. While terminal operations return a result of a certain type, intermediate operations return the stream itself. So you can chain multiple method calls in a row. Streams are created on a source, eg. a java.util. Collection like lists or sets (maps are not supported). Stream operations can either be executed sequential or parallel.

Based on the above points, if we list down the various Characteristics of Stream, they will be as follows:

- Not a data structure
- Designed for lambdas
- Do not support indexed access
- Can easily be outputted as arrays or lists.
- Lazy access supported.
- Parallelizable.

Example: without Stream (until Java 1.7v)

```
AL<Integer> l = new AL<Integer>();
l.add(0); l.add(10); l.add(20);
l.add(5); l.add(15); l.add(25);

List<Integer> lL = new AL<Integer>();
    for(Integer I1: l)
    {   if(I1%2==0)
        {lL.add(I1);
    } }

    Sop1(lL); // [0, 10, 20]
```

## With Stream (From 1.8 version onwards)

Example:

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
  Public static void main(String[] args)
  {
    ArrayList<Integer> l = new ArrayList<Integer>();
      l.add(0);
      l.add(10);
      l.add(20);
      l.add(5);
      l.add(15);
      l.add(25);
    System.out.println(l);
  List<Integer> l1 = l.stream().filter(i->i%2==0)
                      .collect(collectors.toList());
    System.out.println(l1);
  }
}
```

Output:

```
[0,10,20,5,15,25]

[0,10,20]
```

Example:

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
 Public static void main(String[] args)
 {
  ArrayList<Integer> l = new ArrayList<Integer>();
  l.add(0);
  l.add(10);
  l.add(20);
  l.add(5);
  l.add(15);
  l.add(25);

 System.out.println(l);
 List<Integer> l1 = l.stream().map(i -> i*2)
                .collect(Collectors.toList());
  System.out.println(l1);
  }
}
```

Output:

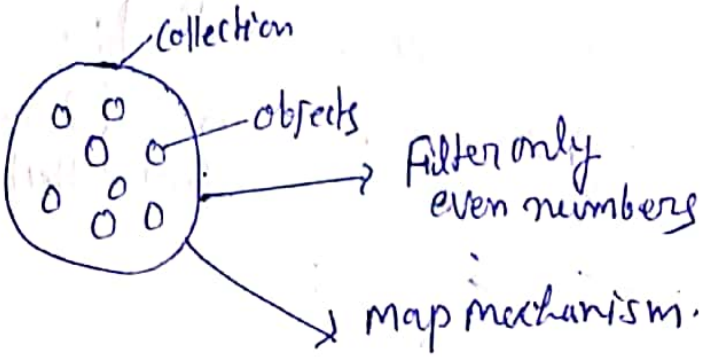
```
[0, 10, 20, 5, 15, 25]
[0, 20, 40, 10, 30, 50]
```

## ABout Stream :-

`Stream s = c.stream();`

- `c` → Any Collection object
- `stream()` → This method present inside Collection interface as default method.
- `Stream` → it is an interface
  - Present in Java.Util.Stream Package.
  - Its came in 1.8 version

- after creating the stream object its contains 2 stages to perform further operation.
  1. Configuration → Filter mechanism, Map mechanism
  2. processing




## Filtering :

* If we want to filter elements from the Collection based on some boolean condition, then we should go for filtering

* we can configure Filter by using filter() method of stream interface.

`Public Stream filter(Predicate<T> t)`

It can be boolean valued function or λ expression.

Eg:- `Stream s1= c.stream().filter(i -> i%2 == 0);`

## Mapping:-

*If we want to create a seperate new object for every object present in the collection based on some fenction then we should go for mapping mechanism.

* we can implement mapping by using map() method of Stream interface.

```
Public Stream map(Fenction<T, R> f)
```

Eg:-

```
Stream sl = c.stream().map(i -> i*2);
```

## Various Methods of Stream :-

### 1. Processing by Collect() method:-

- This method collects the elements from the stream and adding to the specified collection.

Example:

```
import java.util.*;
import java.util.stream.*;
Public class Test {
 public static void main(String[] args)
 {ArrayList<String> l = new ArrayList<String>();
    l.add("Pavan"); l.add("RaviTeja");
    l.add("chiranjeevi"); l.add("venkatesh");
    l.add("Nagarjuna");
  System.out.println(l);
List<String> l1 = l.stream().filter(s -> s.length()>=9)
                   .collect(Collectors.toList());
  System.out.println(l1);
}}
```

output: [Pavan, RAviTeja, chiranjeevi, venkatesh, Nagarjuna]
[chiranjeevi, Venkatesh, Nagarjuna]

**Example:-**

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
  Public static void main(String[] args)
  {
   ArrayList <String> l = new ArrayList<String>();
      l.add("Pavan");
      l.add("Raviteja");
      l.add("Chiranjeevi");
      l.add("Venkatesh");
      l.add("Nagarjuna");
     System.out.println(l)

List<String> l1 = l.stream().filter(s -> s.length()>=9)
                  .Collect(Collectors.toList());
System.out.println(l1);

List <String> l2 = l.stream().map(s -> s.toUpperCase()).
                  Collect(Collectors.toList());

   System.out.println(l2);
  }
}
```

Output:-

[Pavan, Raviteja, Chiranjeevi, Venkatesh, Nagarjuna]
[Chiranjeevi, Venkatesh, Nagarjuna]

~~[Pavan, Raviteja]~~

[PAVAN, RAVITEJA, CHIRANJEEVI, VENKATESH, NAGARJUNA]

## 2. Processing By Count() Method :

- This method returns the number of elements present in stream.

```
Public long Count()
```

~~AL<String>~~

Example:-

```
import java.util.*;
import java.util.stream.*;
Public Class Test
{
  Public static void main(String[] args)
  {
   ArrayList<String> l = new ArrayList<String>();
   l.add("Pavan");
   l.add("Ravi");
   l.add("Chiranjeevi");
   l.add("Venkatesh");
   l.add("Nagarjuna");
   System.out.println(l);
long count = l.stream().filter(s->s.length()>=9)
                       .count();
System.out.println("The number of strings whose length
                    >=9:" +count);
  }
}
```

Output:-

```
The number of strings whose length >=9: 3
```

## 3. Processing by Sorted() method:

* We can use sorted() method to sort elements inside stream.
* we can sort either based on default natural sorting order or based on our own sorting (Customized) order specified by Comparator Object.

1. sorted() :- For default natural sorting order.
2. sorted(Comparator c) :- For Customized sorting order.

### Example:-

### 1. Default natural sorting order:-

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
 Public static void main(String[] args)
 {
  ArrayList<Integer> l = new ArrayList<Integer>();
   l.add(0);
   l.add(10);
   l.add(20);
   l.add(5);
   l.add(15);
   l.add(25);
   System.out.println(l);

 List<Integer> l1 = l.stream().sorted().collect(
                        Collectors.toList());
 System.out.println("List According to Default
                     Natural Sorting order:"+l1);
 }
}
```

Output: [0, 10, 20, 5, 15, 25]
List According to Default Natural Sorting order: [0, 5, 10, 15, 20, 25]

Example:-

## 2. Customized Sorting order:-

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
  Public static void main(String[] args)
  {
    ArrayList<Integer> l = new ArrayList<Integer>();
    l.add(0);
    l.add(10);
    l.add(20);
    l.add(5);
    l.add(15);
    l.add(25);
    System.out.println(l);

    List<Integer> l1 = l.stream().sorted().collect(Collectors.
                                                   toList());
    System.out.println("List According to default Natural Sorting
                        order:" + l1);

    List<Integer> l2 = l.stream().sorted((i1, i2) ->
                        -i1.compareTo(i2)).collect(Collectors.toList());
    System.out.println("List According to Customized Sorting order:"
                                                   +l2);
                        or
    List<Integer> l2 = l.stream().sorted((i1, i2) ->
                        i2.compareTo(i1)).collect(Collectors.toList());
  }
```

output: [0, 10, 20, 5, 15, 25]

List According to Default Natural Sorting order:
[0, 5, 10, 15, 20, 25]
List According to Customized Sorting order:
[25, 20, 15, 10, 5, 0]

## 4. Processing by min() and max() methods:

### I. min(Comparator c):

- Returns minimum value according to specified Comparator

### II. max(Comparator c):

- Returns maximum value according to specified Comparator

**Example: min(Comparator c);** –

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
  Public static void main(String[] args)
  {
  ArrayList<Integer> l = new ArrayList<Integer>();
  l.add(0);
  l.add(10);
  l.add(20);
  l.add(5);
  l.add(15);
  l.add(25);
  System.out.println(l);

Integer min = l.stream().min((i1, i2) ->
                    i1.compareTo(i2)).get();
System.out.println("minimum value is : " +min);
}
}
```

**output:-** [0, 10, 20, 5, 15, 25]

minimum value is : 0

Example:- max(Comparator c);

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
 Public Static void main(String args[])
 {
  ArrayList<Integer> l = new ArrayList<Integer>();
  l.add(0);
  l.add(10);
  l.add(20);
  l.add(5);
  l.add(15);
  l.add(25);
  System.out.println(l);

 Integer min= l.stream().min((i1,i2)-> i1.compareTo
                                (i2)).get();
  System.out.println("Minimum value is :"+min);

 Integer max= l.stream().max((i1,i2) -> i1.
                        compareTo(i2)).get();
  System.out.println("Maximum value is :"+max);
 }
}
```

output: [0, 10, 20, 5, 15, 25]

Minimum value is : 0

Maximum value is : 25

## ⑤ Processing by Using forEach() method:-

* This method will not return anything
* This method can take Lamda Expression as argument and apply that Lambda Expression for each element present in Stream.

Example:

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
  Public Static void main (String[] args)
  {
    ArrayList<String> l = new ArrayList
                          <String>();
    l.add("A");
    l.add("BB");
    l.add("CCC");
   l.stream().forEach (S -> System.out.println(S));
   l.stream().forEach(System.out::println);
  }
}
```

output:-

```
A
BB
CCC

A
BB
CCC
```

## 6. Processing by toArray() method:

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

Example:-

```
import java.util.*;
import java.util.stream.*;
Public class Test
{
  Public static void main(String[] args)
  {
  ArrayList<Integer> l = new ArrayList<Integer>();
   l.add(0);
   l.add(10);
   l.add(20);
   l.add(5);
   l.add(15);
   l.add(25);
  System.out.println(l);
Integer[] array = l.stream().toArray(Integer[]::new);
 for(Integer x : array)
   {
     System.out.println(x);
  }}
}
```

Output:

```
0
10
20
5
15
25
```

## ⑦ Stream.of() method

- We can also apply Stream for group of values & for arrays.

### ① For group of values:

```
Stream<Integer> s = Stream.of(9,99,999,9999);
        s.forEach(System.out::println);
```

Example:

```
import java.util.stream.*;
Public class Test
{
Public static void main(String[] args)
 {
  Stream<Integer> s= Stream.of(9,99,999,9999);
     s.forEach(System.out::println);
  }
}
```

Output:
```
9
99
999
9999
```

### ② For Arrays:

Example:

```
import java.util.stream.*;
Public class Test
{
 Public static void main(String[] args)
 {
Stream<Integer> s = Stream.of(9,99,999,9999);
```

```
s.forEach(System.out::println);
Double[] d = {10.0, 10.1, 10.2, 10.3, 10.4};
Stream<Double> s1 = Stream.of(d);
   s1.forEach(System.out::println);
}
}
```

Output:

```
g
g g
g g g
g g g g
10.0
10.1
10.2
10.3
10.4
```

# Date & Time API (Joda-Time API)

- This API Developed by Joda.org. So the name is given Joda-Time API as well.
- Its come in Java 1.8 version
- Until 1.7 version whatever the class we were using for Date & time were not efficient and lots of method were deprecated & performance also down. due to these Problem new API came in Java 1.8 version.

Example: Print CurrentDate & CurrentTime:

```
import java.time.*;
Public class Test
{
  Public static void main(String[] args)
  {
    LocalDate date = LocalDate.now();
    System.out.println(date);

    LocalTime time = LocalTime.now();
    System.out.println(time);

  }
}
```

Output:

```
2019-07-29
09:40:24.623
```

Note: LocalDate & LocalTime class are present inside java.time Package.

## Print Day, Month & Year seperatelly.

Example:

```
import java.time.*;
Public class Test
{
 public static void main (String[] args)
 {
  LocalDate date = LocalDate.now();
   System.out.println(date);
  int dd = date.getDayOfMonth();
  int mm = date.getMonthValue();
  int yyyy = date.getYear();

 System.out.println(dd+"..."+mm+"..."+yyyy);
System.out.printf("%d-%d-%d", dd, mm, yyyy);
 }
}
```

Output:

```
2019-07-29      //default format
29...07...2019  // Customized format
29-7-2019       // Customized format
```

## Print hour, Minute, Second & Nanosecond:

Example:

```
import java.time.*;
Public class Test
{
  Public static void main(String[] args)
  {
    LocalDate
    LocalTime time = LocalTime.now();

    int h = time.getHour();
    int m = time.getMinute();
    int s = time.getSecond();
    int ns = time.Nano();

  System.out.printf("%d:%d:%d:%d", h,m,s,ns);

  }
}
```

Output

```
10:3:54:325000000
```

## LocalDateTime:

To use this class we can get Date and Time both combined & all feature which was present in LocalDate & LocalTime class.

Example:

```
import java.time.*;
Public class Test
{
  Public static void main(String[] args)
  {
    LocalDateTime dt = LocalDateTime.now();
    System.out.println(dt);
    int dd = dt.getDayOfMonth();
    int mm = dt.getMonthValue();
    int yyyy = dt.getYear();
  System.out.printf("Date:%d-%d-%d", dd, mm, yyyy);

    int h = dt.getHour();
    int m = dt.getMonth();
    int s = dt.getSecond();
    int n = dt.getNano();

    System.out.printf("\nTime:%d:%d:%d:%d", h,m,s,n);

  }
}
```

Output:

```
2019-07-29T10:24:40.511
Date: 29-7-2019
Time: 10:24:40:511000000
```

## Example: LocalDateTime class Methods:

```
import java.time.*;
Public class Test
{
  Public static void main(String[] args)
  {
   LocalDateTime dt = LocalDateTime.of
                (1995, Month.MAY, 28, 12,45);
    System.out.println(dt);
  System.out.println("After Six Months:"+
                dt.plusMonths(6));
  System.out.println("Before Six Months:"+
                dt.minusMonths(6));
  }
}
```

Output:

```
1995-05-28T12:45
After Six Months: 1995-11-28T12:45
Before Six Months: 1994-11-28T12:45
```

# Date and Time API Introduction Period & year

## • Period between two Date:

Example:-

```
import java.time.*;
Public Class Test.
{
 Public static void main(String[] args)
 {
  LocalDate birthday = LocalDate.of(1989, 8, 28);
  LocalDate today = LocalDate.now();
  Period p = Period.between(birthday, today);
  System.out.printf("Age is %d years %d
          Months %d Days", p.getYears(),
            p.getMonths(), p.getDays());
 }
}
```

Output: Age is 27 years 10 Months 10 Days

Example:- we can add below code as well in above Example:

```
LocalDate deathday = LocalDate.of(1989+60, 06, 15);
Period p1 = Period.between(today, deathday);
int d = p1.getYears()*365 + p1.getMonths()*30 +
                              p1.getDays();
System.out.printf("In you will be on the earth only %d Days..
  Hurry up to do more important Things", d);
```

Output: Age is 27 years 10 Months 10 Days
you will be on the earth only 11695 Days.. Hurry up to do more important things.

## • Year :-

Example:

```
import java.time.*;
import java.util.*;
Public class Test
{
  Public Static Void main(String[] args)
  {
   Scanner sc = new Scanner(System.in);
  System.out.println("Enter year Number:");
   int n = sc.nextInt();
   Year y = Year.of(n);
   if(y.isLeap())
   {
   System.out.printf("%d year is Leap year", n);

   }
   else
   {
   System.out.printf("%d year is Not Leap year", n);
   }
  }
}
```

Output :-

```
Enter year Number:
1900
1900 year is Leap year
```

```
Enter year Number:
2002
2002 year is Not Leap year.
```

# Date and Time API introduction ZoneId, ZonedDateTime

**ZoneId :** To print the current zone.

Example;

```
import java.time.*;
Public Class Test
{
 Public static Void main(String[] args)
 {
  ZoneId zone = ZoneId.SystemDefault();
   System.out.println(Zone);
 }
}
```

Output: Asia/Calcutta

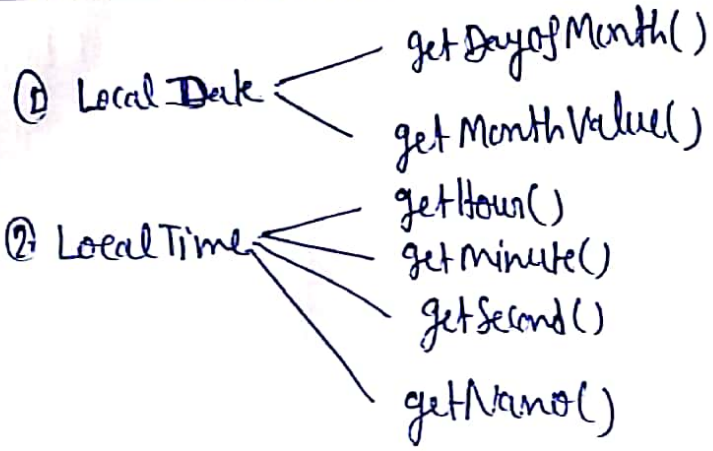
**ZonedDateTime:** To print Date Time in given Zone.

Example:

```
import Java.time.*;
Public class Test
{ Public Static Void main(String[] args)
  { ZoneId zone = ZoneId.SystemDefault();
    System.out.println(Zone);
  ZoneId la = ZoneId.Of("America/Los-Angeles");
  ZonedDateTime zt = ZonedDateTime.now(la);
   System.out.println(zt);
 }
}
```

Output: Asia/Calcutta
2017-07-15T22:57:50.933-07:00[America/Los-Angeles]

## Summerize Joda Time API:




3 LocalDateTime

4 Period

5 Year

6 ZoneId

7 ZonedDateTime.