==

Java - 8 Features

==

1. Lambda Expression
2. Functional Interfaces
3. Default methods in interfaces
4. Static methods in interfaces
    a. >> Predefined Functional Interfaces: (java.util.function)
    b. Predicate
    c. Function
    d. Consumer
    e. Supplier
5. Method Reference & Consumer Reference by double colon(::) Operator
6. Stream API
7. Date and Time API (Joda API || joda.org)

Java - 1.8V Intentions are:

1. To simplify the programming
2. To utilize functional programming benefits in Java
3. To enable parallel Programming

Lambda Expression:

By using **Lambda** Expression in Java:

To enable functional programming in java

Write more readable, maintainable & concise code

To use APIs very easily and effectively

To enable parallel processing also

Rules for simplifying the lambda expression:

Method name is no need

Modifier is not required

Type also not requires

Ex:

Without lambda expression:

```
public void m1() { sop("hello"); }
```

with lambda expression:

```
() -> {
sop("hello");
}
() -> { sop("hello"); }
```

```
//if in the lambda expression can content only one statement
then the {} are optional.
```

Ex 3:

```
 public String str(String str) { return str; }
() -> sop("hello");
```

Ex 2:

```
public void add(inta, int b)
{
     sop(a+b);

}

//lambda expression
(int a, int b) -> {sop(a+b)}


//if in the lambda expression can content only one statement
then the {} are optional.

(int a, int b) -> sop(a+b)


//Type Inference: usually we can specify the type of
parameter, If compiler expect the type based on context, then
we can remove type [Type Interface]

(a, b) -> sop(a+b);
```

Ex 3:

```
//a java method into lambda expression from String to int
public int getLength(String s)
{
     return s.length();
}


//lambda expression
(String s) -> {return s.length();}

//if in the lambda expression can content only one statement
then the {} are optional.

(String s) -> return s.length();


//Type Inference: by compiler automatically based on the
context then we can remove types also

(s) -> return s.length();


//If lambda expression return something then we can remove
return keyword

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

```
//if it is taking one parameter(input) '()' are optional

S -> s.length():
```

Functional Interface:

Once we write lambda expression to invoke that lambda expression we required Functional Interface.

An Functional interface contain only one abstract method and any number of default and Static methods.

To indicate that this is the functional interface we have to use annotation i.e. @FunctionalInterface

Bu using annotation compiler will check this is a Functional interface and if not it will give the compile time error.

Ex 1:

```
@FunctionalInterface

interface Interf
{
     public void m1();

     default void m2()
     {

     }

     public static void m3()
     {

     }
}
```

Ex 2:

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

     default void m2();
}

//CE: Unexpected @FunctionalInterface annotation
multiple non-overriding abstarct methods present in interface
Interf1

Ex 3:
@FunctionalInterface

interface Interf1
{
}

//CE:Unexpected @FunctionalInterface annotation
no abstract method found in interface Interf2
```

Functional Interface w.r.t. Inheritance

Case 1:

If an interface extend 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
{

}
```

Case 2:

In the child interface we can define exactly same parent interface abstract method.

Ex:

```
@Functional Interface
```

```
interface P
{
     public void methodOne();
}

@Functional Interface
interface C extends P
{
     public void methodOne();
}
```

Case 3:

In the child interface we can’t define any new abstract methods otherwise child interface won’t be

Functional Interface and if we are trying to use @Functional Interface annotation then compiler gives an error message.

Ex:

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

@FunctionalInterface
interface C extends P
{
     public void methodTwo();
}
//CE: Unexpected @FunctionalInterface annotation
multiple non-overriding abstarct methods present in interface C
```

Case 4:

```
@FunctionalInterface

interface P

{

     public void methodOne();

}

interface C extends P

{

     public void methodTwo();

}

//we are not defined @FunctionalInterface This's Normal
interface so that code compiles without error
```

In the above example in both parent & child interface we can write any number of default methods and there are no restrictions.

Restrictions are applicable only for abstract methods.

Invoking Lambda Expression by using Functional Interface

Once we write Lambda expressions to invoke it's functionality, then Functional Interface is required.

We can use Functional Interface reference to refer Lambda Expression.

Where ever Functional Interface concept is applicable there we can use Lambda Expressions.

EX 1:

```
//   Without Lambda Expression
interface Interf
{
     public void m1();
}

class Demo implements Interf
{
	public void m1()
	{
		System.out.println("Normal method implementation");
	}
}

public class Test
{

	public static void main(String[] args) {

		Interf i = new Demo();
		i.m1();
	}

}
output: Normal method implementation


//	With Lambda Expression
interface Interf
{
	public void m1();
}
class Test
{
	public static void main(String[] args) {
		Interf i = () -> System.out.println("Lambda
Expresssion implementation");
		i.m1();
	}
}
output: Lambda Expresssion implementation
```

Ex 2:

```
//	Without Lambda Expression
interface Interf
{
	public void add(int a, int b);
}

class Demo implements Interf
{
	public void add(int a, int b)
	{
		System.out.println("The sum by normal
implementation: " + (a+b));
```

```
     }
}

public class Test
{

     public static void main(String[] args) {

          Interf i = new Demo();
          i.add(10, 20);
          i.add(100, 200);
     }

}

output:
     The sum by normal implementation: 30
     The sum by normal implementation: 300
```

```
//   Without Lambda Expression
interface Interf
{
     public void add(int a, int b);
}
class Test
{
     public static void main(String[] args)
     {
          Interf i = (a, b) -> System.out.println("The Sum: "
+ (a+b));
          i.add(10, 20);
          i.add(100, 200);
     }
}

The Sum: 30
The Sum: 300
```

Note: in the above example we can't provide any type to the parameter but compiler will guess the type based on context, it is called Type Inference

Ex 3:

```
//   Without Lambda Expression
interface Interf
{
     public int getLength(String s);
}

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

public class Test
{

     public static void main(String[] args) {

          Interf i = new Demo();
          System.out.println(i.getLength("Navin"));
          System.out.println(i.getLength("Without Lambda"));
     }

}
output:
5
14


//   With Lambda Expression
interface Interf
{
     public int getLength(String s);
}

class Test
{
     public static void main(String[] args)
     {
//          Interf i = (String s)-> {return s.length();};
//          Interf i = (String s)-> return s.length();
//          Interf i = (String s)->  s.length();
//          Interf i = (s)-> s.length();

          Interf i = s->s.length();

          System.out.println(i.getLength("Navin5"));
```

```
		System.out.println(i.getLength("With Lambda 
Expression"));
	}
}

output:
6
22
```

Ex 4:

```
//   Without Lambda Expression

interface Interf

{

	public int squareIt(int x);

}


class Demo implements Interf

{

	public int squareIt(int x)

	{

		return x*x;

	}

}


public class Test

{


	public static void main(String[] args) {


		Interf i = new Demo();

		System.out.println(i.squareIt(4));

		System.out.println(i.squareIt(5));

	}
}
output:
16
25

//   With Lambda Expression
interface Interf
{
     public int squareIt(int x);
}
class Test
{
     public static void main(String[] args)
     {
//        Interf i = (int x) -> {return x*x;};
//        Interf i = (int x) ->  x*x;
//        Interf i = (x) ->  x*x;
          Interf i = x ->  x*x;

          System.out.println(i.squareIt(4));

          System.out.println(i.squareIt(5));
     }
}

output:
16
25
```

Ex 5:

```
//   Without Lambda Expression implements Thread

class MyRunnable implements Runnable
{
     public void run()
     {
          for(int i = 0 ; i<10; i++)
          {
               System.out.println("child Thread");
          }
     }
}

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

          Runnable r = new MyRunnable();
          Thread t = new Thread(r);
          t.start();

          for(int i = 0 ; i<10; i++)
          {
               System.out.println("Main Thread");
          }
     }
}
output:
     child Thread
     child Thread
     child Thread
     child Thread
     child Thread
     child Thread
     child Thread
     child Thread
     child Thread
     child Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
```

```
//   With Lambda Expression implements Thread


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

          Runnable r = () -> {for(int i=0; i<10; i++)
                                                {

     System.out.println("Lambda Child Thread");
                                                }
          };
          Thread t = new Thread(r);
          t.start();

          for(int i = 0 ; i<10; i++)
          {
               System.out.println("Lambda Main Thread");
          }
     }
}
```

```
output:
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Main Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
     Lambda Child Thread
```

Functional Interface, Lambda Expression Summary:

It should contains exactly one abstract method (SAM single abstract method).

It can contain any number of default and static methods.

It acts as a type for lambda expression.

Ex: Interf I = () -> sopln("HELLO");

It can be used to invoke lambda expression

Ex: i.m1();

Case 1:

Why functional Interface should contains only one abstract method?

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

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

		Interf I = (i)-> System.out.println(i*i);
		I.m1(10);
	}
}
```

//CE: incompatible type Interf is not a functional interface

Multiple non-overriding abstract methods in Interface Interf

So this is why Functional interface must contain only single anstract method

Case 2:

What is the advantage of @FunctionalInterface annotation?

It indicates that this interface is used for lambda expression purpose don't add second abstract method.

If we are mention a interface with @FunctionalInterface one more person can’t add the one more abstract method to that interface for this @FunctionalInterface is mentioned

Collection Overview:

A group of objects represented as single entity is called collection.

List(I)

Set(I)

Map(I)

List(I):

If we want to represent a group of objects as a single entity where duplicate objects are allowed and insertion order is preserved then we should go for List.

1. Insertion order is preserved

2. Duplicate objects are allowed

The main implementation classes of List interface are:

1. ArrayList 2. LinkedList 3. Vector 4. Stack

Demo Program to describe List Properties:

```
import java.util.ArrayList;

class Test
{
	public static void main(String[] args)
	{
		ArrayList<String> l = new ArrayList<String>();
		l.add("Sunny");
		l.add("Bunny");
		l.add("Chinny");
		l.add("Sunny");
		System.out.println(l);
```

```
     }
}
output: [Sunny, Bunny, Chinny, Sunny]
```

Note: List(may be ArrayList,LinkedList,Vector or Stack) never talks about sorting order. If we want

sorting for the list then we should use Collections class sort() method.

Collecttions.sort(list)==>meant for Default Natural Sorting Order

Collections.sort(list,Comparator)==>meant for Customized Sorting Order

Set(I):

If we want to represent a group of objects as a single entity where duplicate objects are not allowed and insertion order is not preserved then we should go for Set.

1. Insertion order is not preserved

2. Duplicate objects are not allowed. If we are trying to add duplicates then we won't get any error, just

add() method returns false

The following are important Set implementation classes

1. HashSet 2.TreeSet etc

Demo Program for Set:

```
import java.util.HashSet;

class Test
{
     public static void main(String[] args)
     {
          HashSet<String> l = new HashSet<String>();
          l.add("Sunny");
          l.add("Bunny");
          l.add("Chinny");
          l.add("Sunny");
          System.out.println(l);
     }
}
```

```
//output: [Chinny, Bunny, Sunny]
```

Note: In the case of Set, if we want sorting order then we should go for: TreeSet

Map(I):

If we want to represent a group of objects as key-value pairs then we should go for Map Concept

Eg:

Rollno-->Name

mobilenumber-->address

The important implementation classes of Map are:

1. HashMap 2. TreeMap etc

Demo Program for Map:

```
import java.util.HashMap;

class Test {
     public static void main(String[] args) {
          HashMap<String, String> m = new HashMap<String,
String>();
          m.put("A", "Apple");
          m.put("Z", "Zebra");
          m.put("Kholi", "Java");
          m.put("B", "Boy");
          m.put("T", "Tiger");
          System.out.println(m);
     }
}
Output: {A=Apple, B=Boy, T=Tiger, Z=Zebra, Kholi=Java}
```

Comparator Interface and Compare method:

Comparator(I) contain only one method is compare() method

To define our own sorting (Customized sorting)

```
Public int compare(Object obj1, Object obj2)
     Return -1, if obj1 has to come before obj2
     Return +1, if obj1 has to come before obj2
     Return 0, if obj1 and obj2 are equal.
```

Sorted Collections:

1. Sorted List
2. Sorted Set
3. Sorted Map

1. Sorted List:

List(may be ArrayList,LinkedList,Vector or Stack) never talks about sorting order. If we want

sorting for the list then we should use Collections class sort() method.

Collecttions.sort(list)==>meant for Default Natural Sorting Order

For String objects: Alphabetical Order

For Numbers : Ascending order

Instead of Default natural sorting order if we want customized sorting order then we should go for

Comparator interface.

Comparator interface contains only one abstract method: compare()

Hence it is Functional interface.

public int compare(obj1,obj2)

returns -ve iff obj1 has to come before obj2

returns +ve iff obj1 has to come after obj2

returns 0 iff obj1 and obj2 are equal

Collections.sort(list,Comparator)==>meant for Customized Sorting Order

Sorting Elements of List without Lambda Expression:

Ex: Default natural sorting order

```
import java.util.ArrayList;
import java.util.Collections;

class Test
{
	public static void main(String[] args)
	{
		ArrayList<Integer> l = new ArrayList<Integer>();
		l.add(10);
		l.add(0);
		l.add(15);
		l.add(5);
		l.add(20);

		System.out.println("Before Sorting:" + l);
		Collections.sort(l);
		System.out.println("After Sorting:" + l);
	}
}
```

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

Demo Program to Sort elements of ArrayList according to Customized Sorting: Order(Descending Order):

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

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;
		}
//         Using Ternary Operator
           return (I1 > I2)? -1: (I1 < I2)?+1:0 ;
      }
}

class Test
{
      public static void main(String[] args)
      {
           ArrayList<Integer> l = new ArrayList<Integer>();
           l.add(10);
           l.add(0);
           l.add(15);
           l.add(5);
           l.add(20);

           System.out.println("Before Sorting:" + l);
           Collections.sort(l, new MyComparator());
           System.out.println("After Sorting:" + l);
      }
}

//output:
Before Sorting:[10, 0, 15, 5, 20]
After Sorting:[20, 15, 10, 5, 0]
```

Sorting with Lambda Expressions:

As Comparator is Functional interface, we can replace its implementation with Lambda Expression

```
Collections.sort(l,(I1,I2)->(I1<I2)?1:(I1>I2)?-1:0);
```

Demo Program to Sort elements of ArrayList according to Customized Sorting Order By using Lambda Expressions(Descending Order):

```
import java.util.ArrayList;
import java.util.Collections;

class Test
{
      public static void main(String[] args)
      {
           ArrayList<Integer> l = new ArrayList<Integer>();
           l.add(10);
           l.add(0);
```

```
            l.add(15);
            l.add(5);
            l.add(20);
            l.add(25);

            System.out.println("Before Sorting:" + l);
            Collections.sort(l, (I1, I2)-> (I1 > I2)?-
1:(I1<I2)?+1:0);
            System.out.println("After Sorting:" + l);
      }
}
//output:
Before Sorting:[10, 0, 15, 5, 20, 25]
After Sorting:[25, 20, 15, 10, 5, 0]
```

2. Sorted Set

In the case of Set, if we want Sorting order then we should go for TreeSet

1. TreeSet t = new TreeSet();

 This TreeSet object meant for default natural sorting order

2. TreeSet t = new TreeSet(Comparator c);

 This TreeSet object meant for Customized Sorting Order

Demo Program for Default Natural Sorting Order(Ascending Order):

```
import java.util.TreeSet;

class Test
{
      public static void main(String[] args)
      {
            TreeSet<Integer> t = new TreeSet<Integer>();
            t.add(10);
            t.add(0);
            t.add(15);
            t.add(5);
            t.add(20);
            System.out.println(t);
      }
}
output:[0, 5, 10, 15, 20]
```

Demo Program for Customized Sorting Order(Descending Order) with lambda Expression:

```
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(5);
          t.add(20);
          System.out.println(t);
     }
}
//output:[20, 15, 10, 5, 0]
```

3. Sorted Map:

In the case of Map, if we want default natural sorting order of keys then we should go for

TreeMap.

1. TreeMap m = new TreeMap();

This TreeMap object meant for default natural sorting order of keys

2. TreeMap t = new TreeMap(Comparator c);

This TreeMap object meant for Customized Sorting Order of keys

Demo Program for Default Natural Sorting Order(Ascending Order):

```
import java.util.TreeMap;

class Test
{
     public static void main(String[] args)
     {
          TreeMap<Integer, String> m = new TreeMap<Integer,
String>();
```

```
		m.put(100, "Kholi");
		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: {100=Kholi, 200=Chinny, 300=Bunny, 400=Pinny,
600=Sunny, 700=Vinny}
```

Demo Program for Customized Sorting Order(Descending Order):

```
import java.util.TreeMap;

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, "Kholi");
		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=Kholi}
```

Sorting to our own class objects with Lambda Expression:

```
import java.util.ArrayList;
import java.util.Collections;

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) {

//         Employee e1 = new Employee(100, "navin");
//         System.out.println(e1);

           ArrayList<Employee> l = new ArrayList<Employee>();
           l.add(new Employee(100, "Katrina"));
           l.add(new Employee(600, "Kareena"));
           l.add(new Employee(200, "Deepika"));
           l.add(new Employee(400, "Sunny"));
           l.add(new Employee(500, "Alia"));
           l.add(new Employee(300, "Mallika"));
           System.out.println("Before Sorting:");
           System.out.println(l);
           Collections.sort(l, (e1, e2) -> (e1.eno < e2.eno) ?
-1 : (e1.eno > e2.eno) ? 1 : 0);
           System.out.println("After Sorting:");
           System.out.println(l);
     }
}
```

```
output:
Before Sorting:
[100:Katrina, 600:Kareena, 200:Deepika, 400:Sunny, 500:Alia,
300:Mallika]

After Sorting:
[100:Katrina, 200:Deepika, 300:Mallika, 400:Sunny, 500:Alia,
600:Kareena]
```

Anonymous Inner Classes VS Lambda Expression:

Nameless inner class is by default consider as Anonymous Inner Classes.

We can replace anonymous class with lambda expression but may not every time.

Length of the code will be reduced

Complexity of program will reduced.

Ex:

```
//with anonymous inner class
class Test
{
     public static void main(String[] args) {

          Runnable r =  new Runnable()
          {
               public void run()
               {
                    for(int i=0; i<10; i++)
                    {
                         System.out.println("Child Thread");
                    }
               }
          };
          Thread t = new Thread(r);
          t.start();

          for(int i=0; i<10; i++)
          {
               System.out.println("Main Thread");
          }
     }
}



//with Lambda Expression
class Test
{
     public static void main(String[] args) {
          Runnable r = () -> {for(int i =0; i<10; i++)
                                   {

     System.out.println("Child Thread");
                                   }
                              };
               Thread t = new Thread(r);
               t.start();

               for(int i=0; i<10; i++)
               {
                    System.out.println("Main Thread");
          }
     }
}

output:
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Main Thread
     Child Thread
     Child Thread
     Child Thread
     Child Thread
     Child Thread
     Child Thread
     Child Thread
     Child Thread
     Child Thread
     Child Thread
     Main Thread
     Main Thread
     Main Thread
```

We can use lambda expression as argument also
Can I assign with functional interface reference variable? yes
Ex:

```java
//with Lambda Expression
class Test {
     public static void main(String[] args) {

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

          for (int i = 0; i < 10; i++) {
               System.out.println("Main Thread");
          }
     }
}
```

Output:

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
```

Anonymous Inner Classes VS Lambda Expression part-2:

Anonymous inner classes are more powerful than lambda expressions

The below all cases we can able to write innerclasses but in all the casses writing lambda expression is not possible

```
case 1:
//anonymous inner class that extends the concrete class
class Test
{

}
Test t = new Test()
{

};


case 2:
//anonymous inner class that extends the abstract class
abstract class Test
{

}
Test t = new Test()
{

};


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() {}
};


case 4:
//anonymous inner class that implements an interface which
contain only one abstract method
//in this particular case only it can possible to replace
anonymous inner class with lambda expression

interface Test
{
	public void m1();
}
Test t = new Test()
{
	public void m1() {}

};
```

Lambda expression can implements an functional interface only (an interface it contains only one abstract method() is called functional interface).

Anonymous Inner Classes VS Lambda Expression part-3:

With in the anonymous inner class it is possible to declare instance variable? yes

Inside anonymous inner class this always refers current inner class instance variable only

Ex:

```
interface Interf
{
	public void m1();
}
class Test
{
	int x = 888;
	public void m2() //instance method of Test class
	{
		Interf i = new Interf()
		{
			int x =999; //instance variable of anonymous
inner class
			public void m1()
			{
				System.out.println(this.x);//999
				System.out.println(Test.this.x);//888
			}
		};
		i.m1();
	}
	public static void main(String[] args)
	{
		Test t = new Test();
		t.m2();
	}
}
output: 999
888
```

→Inside lambda expression it is not possible do declare any instance variable,

The declared variables are seems like local variable

Ex:

```
interface Interf
{
	public void m1();
}
class Test
{
	int x = 888;
	public void m2() //instance method of Test class
	{
		Interf i = () -> {
			int x = 999;
			System.out.println(this.x); //it refers current
outer class ans: 888
		};
		i.m1();
	}
	public static void main(String[] args)
```

```
    {
        Test t = new Test();
        t.m2();
    }
}
```

Output: 888

Conclusion:

Inside anonymous inner class is it possible to declare instance variable

Inside the lambda expression it is not possible to declare instance variable, whatever the variable declared inside the lambda expression by default Considered as local variable

Inside anonymous inner class this always refer current inner class object only

Inside lambda expression this always refer current Outer class object only

Differences between Anonymous Inner class & Lambda expressions:

Anonymous inner class != Lambda Expression

Anonymous Inner class	**Lambda expressions**
It is a class without name	It is a function without name (anonymous function)
Anonymous inner class can extend abstract and concrete classes	Lambda expression can’t extend abstract and concrete classes
Anonymous inner class can implements an interface that contains any number of abstract methods	Lambda expression can implement an interface which contains single abstract method(functional interface)
Inside anonymous inner class we can declare instance variable	Inside Lambda expression we can’t declare instance variables what ever variables declared are considered as local variables
Anonymous inner class can be instantiated	Lambda expression cannot be instantiated

Inside anonymous inner class, this always refers current anonymous inner class object but not outer class object	Inside Lambda expression this always refers current outer class object i.e. enclosing class object
Anonymous inner class is best choice if we want handle multiple methods	Lambda 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 separate .class file will be generated	For the Lambda expression at the time of compilation no separate .class file will be generated
Will be allocated on demand, whenever we are create object	Lambda expression will reside in permanent memory of JUM(method area)

```
//From the lambda expression we can access enclosing method
varibale enclosing class varibale directly

interface Interf
{
	public void m1();
}
class Test
{
	int x =10;
	public void m2() //instance method of Test class
	{
		int y =20;
		Interf i = ()-> {
//			x = 888; // it is valid
//			y = 999; //CE: local variable is final
			System.out.println(x);
			System.out.println(y);
		};
		i.m1();
	}
	public static void main(String[] args)
	{
		Test t = new Test();
		t.m2();
	}
}
```

```
output:
     10
     20
```

Any local variable which referenced from lambda expression is always final weather declare or not, effectively it is final.

Any variable if you are using in local variable implicitly it is final.

From lambda expression we can access class level and method level variables, but the method level local variables referenced from lambda expression must be final or effectively final.

Advantages of Lambda Expression:

- We can enable functional programming in java
- We can reduce length of code so that readability will be improved
- we can resolve complexity of anonymous inner classes until some extent
- we can handle procedures/functions just like values
- we can pass procedures/functions as arguments
- easier to use updated APIs and Libraries
- Enable support for parallel processing

Default Methods and Static Methods in Interfaces:

Default Methods inside interfaces:

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

Every variable declared inside interface is always public static final whether we are declaring or not.

But from 1.8 version onwards in addition to these, we can declare default concrete methods also inside interface, which are also known as defender methods.

We can declare default method with the keyword “default” as follows syntax:

```
     default void m1()
     {
          System.out.println("Default method");
     }
```

Ex:

```
//   From the lambda expression we can access enclosing method
varibale enclosing class varibale directly
interface Interf
{
     default void m1()
     {
          System.out.println("Default method");
     }
}



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


//output: Default method
```

Ex:

```
From the lambda expression we can access enclosing method
varibale enclosing class varibale directly by overriding

interface Interf
{
     default void m1()
     {
          System.out.println("Default method");
     }
}

class Test implements Interf
{
//   overriding default method of interface
     public void m1()
     {
```

```
                System.out.println("My own implementation");
        }
        public static void main(String[] args)
        {
                Test t = new Test();
                t.m1();
        }
}
// output: My own implementation
```

Difference between interface with Default Methods and Abstract Classes:

Default method vs multiple inheritance:

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

Ex:

```
//default methdos with multiple inheritance

interface Left
{
        default void m1()
        {
                System.out.println("Left Default Method");
        }
}

interface Right
{
        default void m1()
        {
                System.out.println("Right Default Method");
        }
}

class Test implements Left, Right
{
        public void m1()
        {
//              overriding method call
                System.out.println("My own implementation");
//              Left interface method call
                Left.super.m1();
//          Right interface method call
            Right.super.m1();
      }
      public static void main(String[] args) {

            Test t = new Test();
            t.m1();
      }
}
```

```
output:
      My own implementation
      Left Default Method
      Right Default Method
```

Differences between interface with default methods and abstract class

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

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

Interface with default method != abstract class

Static methods inside interface:

From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.

Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can’t call interface static methods.

We should call interface static methods by using interface name.

The purpose defining static methods inside an interface is just for utility purpose.

Ex:

```
interface Interf
{
	public static void m1()
	{
		System.out.println("interface static method");
	}
}
class Test implements Interf
{
	public static void main(String[] args) {

		Interf.m1(); //by using interface name
	}
}
```

Output: interface static method

Interface static methods with respect to Overriding

```
//overriding concept is not applicable for interface static method but if we want we can use exactly same method but it is not overriding
case 1:
interface Interf
{
	public static void m1()
	{

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

     }
}


case 2:
interface Interf
{
     public static void m1()
     {

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

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

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

     }
}


case 3:

interface Interf
{
     public static void m1()
     {

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

Main method inside the interface:

```
//from 1.8v onwords we can run interface directly from the
command prompt why because inside interface we are declaring
static methods including main method also
// Interf.java
interface Interf
{
	public static void main(String[] args) {
		System.out.println("Interface main method");
	}
}
```

//Use command prompt

Predefined Functional Interfaces – Predicate:

Lambda Expressions

Functional Interfaces

Default methods inside the interfaces

Static methods inside the interfaces

Predefined Functional Interfaces – java.util.function;

Predicate

Function

Consumer

Supplier, etc……

Predicate(I):

A predicate is a function with a single argument and returns boolean value.

To implement predicate functions in Java, Oracle people introduced Predicate interface in **1.8 version** (i.e.,Predicate<T>).

Predicate interface present in **Java.util.function** package.

It's a functional interface and it contains only one abstract method i.e., test()

Ex:

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

As predicate is a functional interface and hence it can refers lambda expression

Ex:1 Write a predicate to check whether the given integer is greater than 10 or not.
Ex:

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

          ||

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

I -> I > 10; //lambda expression
          ||

Predicate<Integer> p = I -> I>10;
System.out.println(p.test(100)); //true
System.out.println(p.test(5)); //false
```

Ex:

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

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

          Predicate<Integer> p = I -> I>10;
          System.out.println(p.test(200));
          System.out.println(p.test(5));
     //   System.out.println(p.test("navin")); //CE:
incompatible types
     }
}
```

```
output:
     true
     false
```

1 Write a predicate to check the length of given string is greater than 6 or not.

```
 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("naveen"));
          System.out.println(p.test("navin"));
     }
}
```

```
output:
     true
     false
```

2 write a predicate to check whether the given collection is empty or not.

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

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));
          ArrayList l2 = new ArrayList();
          System.out.println(p.test(l2));
     }
}
output:
     false
     true
```

Predicate Joining:

It's possible to join predicates into a single predicate by using the following methods.

     and()

     or()

     negate()

these are exactly same as logical AND ,OR complement operators

predicate joining

P1: Given number greater than 10?

P2: is even numer?

P1.negate()

P1.and(P2)

P1.or(P2)

These are default methods present in predicate interface

     and()

     or()

     negate()

Ex:

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

```
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;

		System.out.println("The numbers greater than 10 are:
");
		m1(p1, x);

		System.out.println("The even numbers are: ");
		m1(p2, x);

		System.out.println("The numbers not greater than 10:
");
		m1(p1.negate(), x);

		System.out.println("The numbers greater than 10 and
even are: ");
		m1(p1.and(p2), x);

		System.out.println("The numbers greater than 10 or
even are: ");
		m1(p1.or(p2), x);

	}
	public static void m1(Predicate<Integer>p, int[]x)
	{
		for(int x1: x)
		{
			if(p.test(x1))
				System.out.println(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:
0
5
```

```
10
The numbers greater than 10 and even are:
20
30
The numbers greater than 10 or even are:
0
10
15
20
25
30
```

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

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

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

		String[] names = {"sunny", "kajal", "malika",
"katrina", "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:
		kahal
		katrina
		kareena
```

Predicate Example to remove null values and empty strings from the given list:

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

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

		String[] names = { "Kholi", "", 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 valid Names:");
		System.out.println(list);
	}
}
output:
	The List of valid Names:
		[Kholi, Ravi, Shiva]
```

Ex: Program for User Authentication by using Predicate:

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

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("Navin")&& u.pwd.equals("jadi");
```

```
		Scanner sc = new Scanner(System.in);

		System.out.println("Enter Username :");
		String username = sc.next();

		System.out.println("Enter pwd :");
		String pwd = sc.next();

		User user = new User(username, pwd);
				if(p.test(user))
		{
			System.out.println("Valid User, You get all 
Services");
		}
		else
		{
			System.out.println("Invalid User, Please login 
again");
		}


	}
}

output:
	Enter Username :
		Navin
		Enter pwd :
		jadi
		Valid User, You get all Services
```

Ex: Program to check whether SoftwareEngineer is allowed into pub or not by using Predicate?

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

class SoftwareEngineer
{
	String name;
	int age;
	boolean isHavingGF;

	SoftwareEngineer(String name, int age, boolean 
isHavingGF)
	{
		this.name = name;
		this.age = age;
		this.isHavingGF = isHavingGF;
	}
```

```
    public String toString()
    {
        return name;
    }
}

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

        SoftwareEngineer[] list = { new
SoftwareEngineer("Kholi", 60, false),
                                             new
SoftwareEngineer("Sunil", 25, true),
                                             new
SoftwareEngineer("Sayan", 26, true),
                                             new
SoftwareEngineer("Subbu", 28, false),
                                             new
SoftwareEngineer("Ravi", 17, true)
                                             };

        Predicate<SoftwareEngineer> allowed = se ->
se.age>=18 && se.isHavingGF == true;
        System.out.println("The Allowed Members into Pub
are:");
        for(SoftwareEngineer se: list)
        {
            if(allowed.test(se))
            {
                System.out.println(se);
            }
        }
    }
}

output:
    The Allowed Members into Pub are:
        Sunil
        Sayan
```

Ex: Employee Management Application:

Test.java

```
package com.naveen;

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

```
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.format("[%s,%s,%.2f,%s]", name,
designation, salary, city);
		return s;
	}
	public boolean equals(Object obj)
	{
		Employee e=(Employee)obj;

	if(name.equals(e.name)&&designation.equals(e.designation)
&&salary==e.salary && city==e.city)
		{
			return true;
		}
		else
		{
			return false;
		}
	}
}

class Test
{
	public static void main(String[] args)
	{
		ArrayList<Employee> list= new ArrayList<Employee>();
		populate(list);
//		System.out.println(list);

//		add the Employee object one by one
//		Employee e1 = new Employee("Navin", "Developer",
30000, "Hyderabad");
//		Employee e2 = new Employee("Shiva", "Tester", 40000,
"Chennai");
```

```
//          list.add(e1);
//          list.add(e2);
//          System.out.println(list);

            Predicate<Employee> p1 = emp ->
emp.designation.equals("Manager");
            System.out.println("Mangers Inforamation: ");
            display(p1, list);

            Predicate<Employee> p2=emp-
>emp.city.equals("Bangalore");
            System.out.println("Bangalore Employees
Information:");
            display(p2,list);

            Predicate<Employee> p3 = emp ->emp.salary<20000;
            System.out.println("Employees whose slaray <20000 To
Give Increment:");
            display(p3,list);

            System.out.println("All Managers from Bangalore city
for Pink Slip:");
            display(p1.and(p2),list);

            System.out.println("All Employees Information who
are either Managers or salary <20000");
            display(p1.or(p3),list);

            System.out.println("All Employees Information who
are not managers:");
            display(p1.negate(),list);


            Predicate<Employee> isCEO=Predicate.isEqual(new
Employee("Kholi","CEO",30000,"Hyderabad"));

            Employee e1=new
Employee("Kholi","CEO",30000,"Hyderabad");
            Employee e2=new
Employee("Sunny","Manager",20000,"Hyderabad");
            System.out.println(isCEO.test(e1));//true
            System.out.println(isCEO.test(e2));//false

      }
       public static void populate(ArrayList<Employee> list)
       {
       list.add(new Employee("Kholi","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("Kanushka","Developer",10000,"Hyderabad"));
	list.add(new 
Employee("Sowmya","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:

Mangers Inforamation:
[Sunny,Manager,20000.00,Hyderabad]
[Mallika,Manager,20000.00,Bangalore]
**************************************************
Bangalore Employees Information:
[Mallika,Manager,20000.00,Bangalore]
[Katrina,Lead,15000.00,Bangalore]
[Sowmya,Developer,10000.00,Bangalore]
[Ramya,Developer,10000.00,Bangalore]
**************************************************
Employees whose slaray <20000 To Give Increment:
[Kareena,Lead,15000.00,Hyderabad]
[Katrina,Lead,15000.00,Bangalore]
[Anushka,Developer,10000.00,Hyderabad]
[Kanushka,Developer,10000.00,Hyderabad]
[Sowmya,Developer,10000.00,Bangalore]
[Ramya,Developer,10000.00,Bangalore]
**************************************************
```

```
All Managers from Bangalore city for Pink Slip:
[Mallika,Manager,20000.00,Bangalore]
**************************************************
All Employees Information who are either Managers or salary
<20000
[Sunny,Manager,20000.00,Hyderabad]
[Mallika,Manager,20000.00,Bangalore]
[Kareena,Lead,15000.00,Hyderabad]
[Katrina,Lead,15000.00,Bangalore]
[Anushka,Developer,10000.00,Hyderabad]
[Kanushka,Developer,10000.00,Hyderabad]
[Sowmya,Developer,10000.00,Bangalore]
[Ramya,Developer,10000.00,Bangalore]
**************************************************
All Employees Information who are not managers:
[Kholi,CEO,30000.00,Hyderabad]
[Kareena,Lead,15000.00,Hyderabad]
[Katrina,Lead,15000.00,Bangalore]
[Anushka,Developer,10000.00,Hyderabad]
[Kanushka,Developer,10000.00,Hyderabad]
[Sowmya,Developer,10000.00,Bangalore]
[Ramya,Developer,10000.00,Bangalore]
**************************************************
true
false
```

Syntax: the methods in predict interface and static method present in Predicate interface.

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

     and()
     or()
     negate()

     public static Predicate isEqual(T t);
}
```

Ex: isEqual example

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

class Test
{
     public static void main(String[] args) {
          Predicate<String> p =
Predicate.isEqual("Navinkumar");
          System.out.println(p.test("Navinkumar"));
          System.out.println(p.test("Kumar"));
     }
}

output:
     true
     false
```

Functions:

Functions are exactly same as predicates except that functions can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.

To implement functions oracle people introduced Function interface in 1.8version.

Function interface present in Java.util.function package.

Functional interface contains only one method i.e., apply()

```
interface function(T,R) {

 public R apply(T t);

}
```

Assignment: Write a function to find length of given input string.

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

class Test {

     public static void main(String[] args)
     {
//          Function<String, Integer> f = s -> s.length();
//          System.out.println(f.apply("Navin")); //5
//          System.out.println(f.apply("JavaTutorials")); //13
```

```
          Function<Integer, Integer> f = i -> i*i;
          System.out.println(f.apply(5)); //25
          System.out.println(f.apply(15)); //225
     }
}
```

Differences between predicate and function

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

Differences between predicate and function:

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


//Function
//where T = input parameter, R=Return Type
interface Function<T, R>
{
     R apply(T t);
}
     Function<String, Integer> f= s -> s.length();
```

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

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

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

          String s = "Naveen kumar from Hyderabad";
          Function<String, String> f = s1 -> s1.replaceAll("
", "");
          System.out.println(f.apply(s));
     }
}
output: NaveenkumarfromHyderabad
```

Ex: Program to find Number of spaces present in the given String by using Function:

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

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

          String s = "Naveen kumar from Hyderabad";
          Function<String, Integer> f = s1 -> s1.length()-
s1.replaceAll(" ", "").length();
          System.out.println(f.apply(s));
     }
}

//output: 3
```

Ex: Program to find Student Grade by using Function:

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

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[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]";
			}
		 };
		 for(Student s: l)
		 {
			System.out.println("Name :"+ s.name);
			System.out.println("Marks :"+ s.marks);
			System.out.println("Grade :"+ f.apply(s));
			System.out.println();
		 }
	}

	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:
	Name :Sunny
```

```
     Marks :100
     Grade :A[Dictinction]

     Name :Bunny
     Marks :65
     Grade :B[First Class]

     Name :Chinny
     Marks :55
     Grade :C[Second Class]

     Name :Vinny
     Marks :45
     Grade :D[Third Class]

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

Ex: Program to find Students Information including Grade by using Function whose marks are >=60:

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

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[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]";
                    }
               };
               Predicate<Student> p =s -> s.marks >= 60;

               for(Student s: l)
               {
                    if(p.test(s))
                    {
                         System.out.println("Name :"+ s.name);
                         System.out.println("Marks :"+ s.marks);
                         System.out.println("Grade :"+
f.apply(s));
                         System.out.println();
                    }
               }
     }

     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:
     Name :Sunny
     Marks :100
     Grade :A[Dictinction]

     Name :Bunny
     Marks :65
     Grade :B[First Class]
```

Ex: Progarm to find Total Monthly Salary of All Employees by using Function:

```
package com.naveen;

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(l);
		Function<ArrayList<Employee>, Double> f = l1 -> {
			double total = 0;
			for (Employee e : l1)
			{
				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));
```

```
          l.add(new Employee("navin", 10000));
     }
}
```

```
output: [Sunny:1000.0, Bunny:2000.0, Chinny:3000.0,
Pinny:4000.0, Vinny:5000.0, navin:10000.0]
          The total salary of this month:25000.0
```

EX: Progarm to perform Salary Increment for Employees by using

Predicate & Function:

```
package com.naveen;

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))
               {
                    f.apply(e);
                    l2.add(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("navin", 10000));
     }
}
```

```
output:
     Before Increment:
          [Sunny:1000.0, Bunny:2000.0, Chinny:3000.0,
Pinny:4000.0, Vinny:5000.0, navin:10000.0]
          After Increment:
          [Sunny:1477.0, Bunny:2477.0, Chinny:3477.0,
Pinny:4000.0, Vinny:5000.0, navin:10000.0]
          Employees with incremented salary:
          [Sunny:1477.0, Bunny:2477.0, Chinny:3477.0]
```

Function Chaining:

We can combine multiple functions together to form more complex functions.For this Function interface defines the following 2 default methods:

f1.andThen(f2): First f1 will be applied and then for the result f2 will be applied.

f1.compose(f2)===>First f2 will be applied and then for the result f1 will be applied.

Ex: Demo Program-1 for Function Chaining

```
package com.naveen;

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);

          System.out.println("The Result of f1:" + 
f1.apply("AishwaryaAbhi"));
          System.out.println("The Result of f2:" + 
f2.apply("AishwaryaAbhi"));
          System.out.println("The Result of f1.andThen(f2):" + 
f1.andThen(f2).apply("AishwaryaAbhi"));
          System.out.println("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
```

Ex: Demo program to Demonstrate the difference between andThen() and compose():

```
package com.naveen;

import java.util.function.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
```

Ex: Demo Program for Function Chaining:

```
package com.naveen;

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("navin")
&& pwd.equals("java"))
          {
               System.out.println("Valid User");
          }
          else
          {
               System.out.println("Invalid User");
          }
     }
}
```

```
output:
     Enter User Name:
          navinKumarJadi
          Enter Password:
          java
          Valid User
```

Function interface Static Method : identity()

Function interface contains a static method.

```
static <T> Function<T,T> identity()
```

Returns a function that always returns its input argument.

Ex:

```
package com.naveen;

import java.util.function.*;

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

		Function<String,String> f1= Function.identity();
		String s2= f1.apply("Navin");
		System.out.println(s2);
	}
}
```

```
output: Navin
```

Function contain:

Abstract () methods -> Apply()

Default() methods -> andThen()

compose()

static() methods -> identity()

Consumer Introduction:

Sometimes our requirment is we have to provide some input value, perform certain operation, but not required to return anything, then we should go for Consumer.i.e Consumer can be used to consume object and perform certain operation.

Consumer Functional Interface contains one abstract method accept.

```
 interface Consumer<T>

 {

 public void accept(T t);

}
```

Demo Program-1 for Consumer:

```
package com.naveen;

import java.util.function.Consumer;

class Test
{
	public static void main(String[] args) {
		Consumer<String> c = s -> System.out.println(s);
		c.accept("Hello");
		c.accept("NaveenKumar");
	}
}
Hello
NaveenKumar
```

Ex: Demo Program-2 to display Movie Information by using Consumer:

```
package com.naveen;

import java.util.ArrayList;
import java.util.function.Consumer;

class Movie
{
	String name;
	String hero;
	String heroine;
	Movie(String name, String hero, String heroine)
	{
		this.name = name;
		this.hero = hero;
		this.heroine = heroine;
	}
	public String toString()
	{
		return name;
	}
}

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

     Movie name: Sultan
     Movie hero: Salman
     Movie heroine: Anushka
```

Ex: Demo Program-3 for Predicate, Function & Consumer:

```
package com.naveen;

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("kunny", 75));
		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[Dictinction]

Student Name:kunny
Student Marks:75
Student Grade:B[First Class]

Student Name:Bunny
Student Marks:65
Student Grade:B[First Class]
```

Consumer Chaining:

Just like Predicate Chaining and Function Chaining, Consumer Chaining is also possible. For this Consumer Functional Interface contains default method andThen().

c1.andThen(c2).andThen(c3).accept(s)

First Consumer c1 will be applied followed by c2 and c3.

Ex: Demo Program for Consumer Chaining:

```
package com.naveen;

import java.util.function.Consumer;

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 released and it is: " + m.result);
            Consumer<Movie> c3 = m -> System.out.println("Movie
:" + m.name + " information storing in database");

            Consumer<Movie> chainedC =
c1.andThen(c2).andThen(c3);
            Movie m = new Movie("Bahubali", "Hit");
            chainedC.accept(m);

            System.out.println();
            Movie m1 = new Movie("Chathrapathi", "Flop");
            chainedC.accept(m1);
      }
}

//Output
Movie :Bahubali is ready to release
Movie :Bahubali is just released and it is: Hit
Movie :Bahubali information storing in database

Movie :Chathrapathi is ready to release
Movie :Chathrapathi is just released and it is: Flop
Movie :Chathrapathi information storing in database
```

Consumer contain:

abstract () methods -> accept()

default() methods -> andThen()

Supplier(I) Introduction:

Sometimes our requirement is we have to get some value based on some operation like

supply Student object

Supply Random Name

Supply Random OTP

Supply Random Password

etc

For this type of requirements we should go for Supplier.

Supplier can be used to supply items (objects).

Supplier won't take any input and it will always supply objects.

Supplier Functional Interface contains only one method get().

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

Supplier Functional interface does not contain any **default** and **static** methods.

Ex: Demo Program-1 For Supplier to generate Random Name:

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

//output:
Bunny vinny vinny
Bunny chinny sunny
sunny vinny vinny
vinny Bunny chinny
```

Ex: Demo Program-2 For Supplier to supply System Date:

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

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

          Supplier<Date> s = ()-> new Date();
          System.out.println(s.get());
     }
}
//output: Tue Jun 06 16:26:34 IST 2023
```

Ex: Demo Program-3 For Supplier to supply 6-digit Random OTP:

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

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

          Supplier<String> s = ()-> {
               String otp = "";
               for(int i =0; i<6; i++)
               {
                    otp = otp+(int)(Math.random()*10);
               }
               return otp;
          };
          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:
259607
236229
733185
405670
518509
192031
```

Ex: Demo Program-4 For Supplier to supply Random Passwords:

Rules:

1. length should be 8 characters

2. 2,4,6,8 places only digits

3. 1,3,5,7 only Capital Uppercase characters,@,#,$

```
package com.naveen;

import java.util.function.Supplier;

class Test
{
	public static void main(String[] args)
	{
		Supplier<String> s = () ->
		{
			Supplier<Integer> d = () -> (int)
(Math.random() * 10);
			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 + d.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());
	}
}
```

```
output:
	W7Z1V2L5
	J9N6#1P2
	R7T8$2S9
	J1M8M8J4
	W8F4T2V5
```

Comparison Table of Predicate, Function, Consumer and Supplier

Property	Predicate	Function	Consumer	Supplier
1) Purpose	To take some Input and perform some conditional checks	To take some Input and perform required Operation and return the result	To consume some Input and perform required Operation. It won't return anything.	To supply some Value base on our Requirement
2) Interface Declaration	interface Predicate { ::::::::::::: }	interface Function <T, R> { ::::::::::::: }	interface Consumer <T> { ::::::::::::: }	interface Supplier <R> { ::::::::::::: }
3) Signle Abstract Method (SAM)	Public Boolean test(T t);	Public R apply(T t);	Public void accept(T t);	Public R get();
4) Default Methods	Predicate Joining: And(), or(), negate()	Function chainging: andThen(), compose()	Consumer Chainging: andThen()	-
5) Static Method	isEqual()	Identity()	-	-

Two-Argument (Bi) Functional Interfaces – BiPredicate, BiFunction & BiConsumer

Need of Two-Argument (Bi) Functional Interfaces:

Normal Functional Interfaces (Predicate, Function and Consumer) can accept only one input argument. But sometimes our programming requirement is to accept two input arguments, then we should go for two-argument functional interfaces. The following functional interfaces can take 2 input arguments.

1. BiPredicate

2. BiFunction

3. BiConsumer

Predicate<Integer> p = i -> i%2 ==0; => BiPredicate

Function<Integer, Integer> f = i -> i*i; => BiFunction

Consumer<String> c = s -> sysout(s); => BiConsumer

Supplier<Date> s = () -> new Date(); //it never take any argument so no BiSupplier

1. BiPredicate(I):

Normal Predicate can take only one input argument and perform some conditional check.

Sometimes our programming requirement is we have to take 2 input arguments and perform some conditional check, for this requirement we should go for BiPredicate.

BiPredicate is exactly same as Predicate except that it will take 2 input arguments.

It applicable default methods also

And()

Or()

Negate()

```
interface Predicate<T>

{

     Public Boolean test(T t);

}


interface BiPredicate<T, U>

{

     Public Boolean test(T t, U u);

}
```

Ex: To check the sum of 2 given integers is even or not by using **BiPredicate:**

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

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)); //true
          System.out.println(p.test(15, 20)); //false
     }
}
```

2.BiFunction:

Normal Function can take only one input argument and perform required operation and returns the result. The result need not be boolean type.

But sometimes our programming requirement to accept 2 input values and perform required operation and should return the result. Then we should go for BiFunction.

BiFunction is exactly same as funtion except that it will take 2 input arguments.

```
interface BiFunction<T,U,R>

{

      public R apply(T t,U u);

 //default method andThen()

}


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

interface BiFunction<T, U, R>
```

```
{
	public R apply(T t, U u);
}
```

To find product of 2 given integers by using BiFunction:

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

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

		BiFunction<Integer, Integer, Integer> f = (a, b) -> 
a*b;
		System.out.println(f.apply(10, 20)); //200
		System.out.println(f.apply(15, 20)); //300
	}
}
```

Ex: To create Student Object by taking name and rollno as input by using **BiFunction:**

```
package com.naveen;

import java.util.ArrayList;
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("Kholi",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("name :" + s.name );
			System.out.println("Rollno :" + s.rollno );
			System.out.println();
		}
	}
}
output:
	name :Kholi
	Rollno :100

	name :Ravi
	Rollno :200

	name :Shiva
	Rollno :300

	name :Pavan
	Rollno :400
```

Ex: To calculate Monthly Salary with Employee and TimeSheet objects as input By using BiFunction:

```
package com.naveen;

import java.util.function.BiFunction;
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
{
	public static void main(String[] args)
	{
		BiFunction<Employee,TimeSheet,Double> f=(e,t)-
>e.dailywage*t.days;

		Employee e= new Employee(101,"Kholi",1500);
		TimeSheet t = new TimeSheet(101, 25);

		System.out.println("Monthly Salary: " + f.apply(e,
t));


	}
}

output:
	Monthly Salary: 37500.0
```

BiConsumer:

Normal Consumer can take only one input argument and perform required operation and won't return any result.

But sometimes our programming requirement to accept 2 input values and perform required operation and not required to return any result. Then we should go for BiConsumer.

BiConsumer is exactly same as Consumer except that it will take 2 input arguments.

```
interface BiConsumer<T,U>

{

 public void accept(T t,U u);
 //default method andThen()
}


interface Consumer<T>
{
     public void accept(T t);
     //default method: andThen()
}

interface BiConsumer<T U>
{
     public void accept(T t, U u);
     //default method: andThen()
}
```

Ex: Program to accept 2 String values and print result of concatenation by using **BiConsumer:**

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

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

          BiConsumer<String, String> c = (s1, s2) ->
System.out.println(s1+s2);
          c.accept("Navin", "Jyothi"); //NavinJyothi
     }
}
```

Ex: Demo Program to increment employee Salary by using **BiConsumer:**

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

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);

		//extra line not required if it mention also no 
result change
		BiFunction<String, Double, Employee> f = (name, 
	salary) -> new Employee(name, salary);


		BiConsumer<Employee, Double> c = (e, increment) -> 
e.salary = e.salary + increment;
		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("Kholi", 1000));
		l.add(new Employee("Sunny", 2000));
		l.add(new Employee("Bunny", 3000));
		l.add(new Employee("Chinny", 4000));
	}
}
output:
	Employee Name:Kholi
	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
```

Comparison Table between One argument and Two argument Functional Interfaces:

One Argument Functional Interface	Two Argument Functional Interface

`interface Predicate<T>`
`{`
` public boolean test(T t);`
` default Predicate and(Predicate P)`
` default Predicate or(Predicate P)`
` default Predicate negate()`
` static Predicate isEqual(Object o)`
`}`
	`interface BiPredicate<T, U>`
	`{`
	` public boolean test(T t, U u);`
	` default BiPredicate and(BiPredicate P)`
	` default BiPredicate or(BiPredicate P)`
	` default BiPredicate negate()`
	`//no static methods`
	`}`

`interface Function<T, R>`
`{`
` public R apply(T t);`
` default Function andThen(Function F)`
` default Function compose(Function F)`
` static Function identify()`
`}`
	`interface BiFunction<T, U, R>`
	`{`
	` public R apply(T t, U u);`
	`default BiFunction andThen(Function F)`
	`}`

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

Primitive Type Functional Interfaces, Unary Operator and Binary Operator:

Primitive Type Functional Interfaces

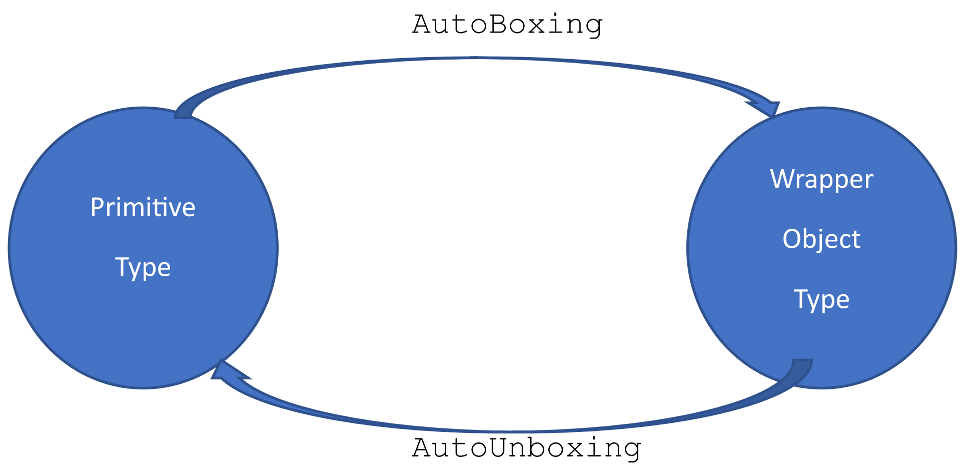
What is the Need of Primitive Type Functional Interfaces?

1. **Autoboxing:**

Automatic conversion from primitive type to Object type by compiler is called autoboxing.

2. **Autounboxing:**

Automatic conversion from object type to primitive type by compiler is called autounboxing



3. **Generic-Type Parameter**

In the case of generics, the type parameter is always object type and no chance of passing primitive type.

```
ArrayList<Integer> l = new ArrayList<Integer>();//valid

ArrayList<int> l = new ArrayList<int>();//invalid
```

Need of Primitive Functional Interfaces:

In the case of normal Functional interfaces (like Predicate, Function etc) input and return types are always Object types. If we pass primitive values then these primitive values will be converted to Object type (Auboxing), which creates performance problems.

Ex:

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

class Test
{
	public static void main(String[] args)
	{
//		int[] x = {0, 10, 15, 20, 25};
//
//		Predicate<Integer> p = i -> i%2 == 0;
//		for(int x1:x)
//		{
//			if(p.test(x1))
//			{
//				System.out.println(x1);//0 10 20
//			}
//		}

		Function<Integer, Integer> f = i -> i*i;
		System.out.println(f.apply(10)); //100
	}
}
```

In the above examples,6 times autoboxing and autounboxing happening which creates Performance problems.

To overcome this problem primitive functional interfaces introduced, which can always takes primitive types as input and return primitive types.Hence autoboxing and autounboxing won't be required,which improves performance.

Primitive Type Functional Interfaces for Predicate:

The following are various primitive Functional interfaces for Predicate.

1.IntPredicate-->always accepts input value of int type

2.LongPredicate-->always accepts input value of long type

3.DoublePredicate-->always accepts input value of double type

```
interface IntPredicate
{
	public boolean test(int i);
	// default methods: and(), or(), negate()
}

interface LongPredicate
{
	public boolean test(long l);
	// default methods: and(), or(), negate()
}

interface DoublePredicate
{
	public boolean test(double d);
	// default methods: and(), or(), negate()
}
```

Demo Program for IntPredicate:

```
import java.util.function.IntPredicate;

class Test
{
	public static void main(String[] args)
	{
		int[] x = { 0, 10, 15, 20, 25 };
		IntPredicate p = i -> i % 2 == 0;
		for (int x1 : x)
		{
			if (p.test(x1))
			{
				System.out.println(x1); // 0 10 20
			}
		}

	}
}
```

In the above example, autoboxing and autounboxing won't be performed internally.Hence performance wise improvements are there.

Primitive Type Functional Interfaces for Function:

The following are various primitive Type Functional Interfaces for Function

1. IntFunction<R>: can take int type as input and return any type

```
 public R apply(int i);
```

2. LongFunction<R>: can take long type as input and return any type

```
 public R apply(long l);
```

3. DoubleFunction<R>: can take double type as input and return any type

```
 public R apply(double d);
```

control on Return Type:

4. ToIntFunction<T>: It can take any type as input but always returns int type

```
 public int applyAsInt(T t)
```

5. ToLongFunction<T>: It can take any type as input but always returns long type

```
 public long applyAsLong(T t)
```

6. ToDoubleFunction<T>: It can take any type as input but always returns double type

```
 public int applyAsDouble(T t)
```

7. IntToLongFunction: It can take int type as input and returns long type

```
 public long applyAsLong(int i)
```

8. IntToDoubleFunction: It can take int type as input and returns long type

```
 public double applyAsDouble(int i)
```

9. LongToIntFunction: It can take long type as input and returns int type

```
 public int applyAsInt(long i)
```

10. LongToDoubleFunction: It can take long type as input and returns double type

```
 public int applyAsDouble(long i)
```

11. DoubleToIntFunction: It can take double type as input and returns int type

```
 public int applyAsInt(double i)
```

12. DoubleToLongFunction: It can take double type as input and returns long type

```
 public int applyAsLong(double i)
```

13. ToIntBiFunction<T, U>: return type must be int type but inputs can be anytype

```
 public int applyAsInt(T t, U u)
```

14. ToLongBiFunction<T, U>: return type must be long type but inputs can be anytype

```
 public long applyAsLong(T t, U u)
```

15. ToDoubleBiFunction<T, U>: return type must be double type but inputs can be anytype

```
 public double applyAsDouble(T t, U u)
```

Ex: Demo Program to find square of given integer by using Function:

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

class Test
{
     public static void main(String[] args)
     {
          //normal Function
          Function<Integer, Integer> f = i -> i*i;
          System.out.println(f.apply(5));//25
     }
}
```

Ex: Demo Program to find square of given integer by using IntFunction:

```
import java.util.function.IntFunction;

class Test
{
     public static void main(String[] args)
     {
//        IntFunction
          IntFunction<Integer> f = i -> i*i;
          System.out.println(f.apply(5));//25
     }
}
```

Ex: Demo Program to find length of the given String by using Function:

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

class Test
{
     public static void main(String[] args)
     {
//        normal Function
          Function<String, Integer> f = s -> s.length();
          System.out.println(f.apply("Navin"));//5
     }
}
```

Ex: Demo Program to find length of the given String by using **ToIntFunction:**

```
import java.util.function.ToIntFunction;
```

```
class Test
{
     public static void main(String[] args)
     {
//          ToIntFunction
          ToIntFunction<String> f = s -> s.length();
          System.out.println(f.applyAsInt("Navin")); //5
     }
}
```

Ex: Demo Program to find square root of given integer by using **Function** and **IntToDoubleFunction**:

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

class Test
{
     public static void main(String[] args)
     {
          Function<Integer,Double> f=i->Math.sqrt(i);
          System.out.println(f.apply(5)); //2.23606797749979
     }
 }
```

==

```
import java.util.function.IntToDoubleFunction;

class Test
{
     public static void main(String[] args)
     {
          IntToDoubleFunction f = i -> Math.sqrt(i);
          System.out.println(f.applyAsDouble(9));//3.0
     }
}
```

Primitive Version for Consumer:

The following 6 primitive versions available for Consumer:

1. IntConsumer

 public void accept(int value)

2. LongConsumer

 public void accept(long value)

3. DoubleConsumer

 public void accept(double value)

4. ObjIntConsumer<T>

 public void accept(T t,int value)

5. ObjLongConsumer<T>

 public void accept(T t,long value)

6. ObjDoubleConsumer<T>

 public void accept(T t,double value)

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

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

LongConsumer

```
accept(long l)

DoubleConsumer

accept(double d)
```

Ex: Demo Program for IntConsumer:

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

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

//         Consumer<Integer> c = i -> System.out.println("The
Square of i: "+ i*i);100
//         c.accept(10);

           IntConsumer c = i -> System.out.println("The Square
of i: "+ i*i);//100
           c.accept(10);
     }
}
```

```
//primitive version consumer
//These are suitable for two inputs one is primitive type and
one is object type
ObjIntConsumer<T>
     public void accept(T t, int i);

ObjLongConsumer<T>
     public void accept(T t, long l);

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

EX: Demo Program to increment employee Salary by using ObjDoubleConsumer:

```
package com.naveen;

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

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

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)
     {

//        BiConsumer<Employee, Double> c = (e,d) -> e.salary =
e.salary + d;
//        Employee e = new Employee("navin", 1000);
//        c.accept(e, 500.0);

          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("Kholi", 1000));
          l.add(new Employee("Sunny", 2000));
          l.add(new Employee("Bunny", 3000));
          l.add(new Employee("Chinny", 4000));
     }
}
output:
     Employee Name:Kholi
     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
```

Primitive Versions for Supplier:

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


//IntSupplier
interface IntSupplier
{
     public void getAsInt();
}
```

The following 4 primitive versions available for Supplier:

1. IntSupplier

 public int getAsInt();

2. LongSupplier

 public long getAsLong()

3. DoubleSupplier

 public double getAsDouble()

4. BooleanSupplier

 public boolean getAsBoolean()

Ex: Demo Program to generate 6 digit random OTP by using **IntSupplier**:

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

class Test
{
	public static void main(String[] args)
	{
		//normal Supplier for generating 6 digit otp
	/*	Supplier<Integer> s = ()-> (int)(Math.random()*10);
		String otp = "";
		for(int i=0; i<6; i++)
		{
			otp = otp+s.get();
		}
		System.out.println("The 6 digit OTP : " + otp);
		System.out.println("The 6 digit OTP : " + otp);
	*/

		//IntSupplier for generating 6 digit otp
		IntSupplier s = () -> (int)(Math.random()*10);
		String otp = "";
		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);
	}
}
The 6 digit OTP : 139991
```

UnaryOperator<T>:

If input and output are same type then we should go for UnaryOperator

It is child of Function<T,T>

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

```
//UnaryOperator<T>
interface UnaryOperator<T>
{
     public T apply(T t);
}
```

Ex:

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

class Test
{
     public static void main(String[] args)
     {
//         Function
//         Function<Integer, Integer> f = i -> i*i;
//         System.out.println(f.apply(10));//100

//         UnaryOperator
           UnaryOperator<Integer> o = i -> i*i;
           System.out.println(o.apply(10));//100
           System.out.println(o.apply(6));//36
     }
}
```

The primitive versions for UnaryOperator:

IntUnaryOperator:

```
 public int applyAsInt(int)
```

LongUnaryOperator:

```
 public long applyAsLong(long)
```

DoubleUnaryOperator:

```
 public double applyAsDouble(double)
```

IntUnaryOperator:

```
interface IntUnaryOperator
{
	public int applyAsInt(int i);
}
```

```
LongUnaryOperator
	public long applyAsLong(long l);

DoubleUnaryOperator
	public double applyAsDouble(double d);
```

Ex: Demo Program-1 for IntUnaryOperator:

```
import java.util.function.IntUnaryOperator;

class Test
{
	public static void main(String[] args)
	{
		IntUnaryOperator f = i -> i *i;
		System.out.println(f.applyAsInt(10));//100
		System.out.println(f.applyAsInt(8));//64
		System.out.println(f.applyAsInt(17));//289
	}
}
```

Ex: Demo Program-2 for IntUnaryOperator

```
import java.util.function.IntUnaryOperator;

class Test
{
	public static void main(String[] args)
	{
		IntUnaryOperator f1 = i -> i +1;
		System.out.println(f1.applyAsInt(4));//5

		IntUnaryOperator f2 = i -> i *i;
		System.out.println(f2.applyAsInt(4));//16


	System.out.println(f1.andThen(f2).applyAsInt(4));//25
	}
}
```

BinaryOperator:

It is the child of BiFunction<T,T,T>

It is a specialized version of BiFunction

```
BinaryOperator<T>

 public T apply(T,T)
```

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

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

Ex:

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

class Test
{
	public static void main(String[] args)
	 {
//		BiFunction contain 2 inputs 1 return types are 
always same
		BiFunction<String,String,String> f=(s1,s2)->s1+s2;
		System.out.println(f.apply("Navin", 
"kumar"));//Navinkumar

//		Binary operator all 3 type parameter are same type 
then we should go for BinaryOperator
		BinaryOperator<String> b=(s1,s2)->s1+s2;
		System.out.println(b.apply("Navin", 
"Kumar"));//Navinkumar
	 }
}
```

The primitive versions for BinaryOperator:

1. IntBinaryOperator

```
 public int applyAsInt(int i,int j)
```

2. LongBinaryOperator

```
 public long applyAsLong(long l1,long l2)
```

3. DoubleBinaryOperator

```
 public double applyAsLong(double d1,double d2)
```

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



interface IntBinaryOperator
{
	public int applyAsInt(int i1, int i2);
}


interface LongBinaryOperator
{
	public int applyAsLong(long l1, long l2);
}


interface DoubleBinaryOperator
{
	public int applyAsDouble(double d1, double d2);
}
```

Ex:

```
import java.util.function.BinaryOperator;
import java.util.function.IntBinaryOperator;

class Test
{
	public static void main(String[] args)
	{
//		BinaryOperator<Integer> b = (i1, i2)-> i1 + i2;
//		System.out.println(b.apply(10, 20));//30


		IntBinaryOperator b = (i1, i2) -> (i1+i2);
```

```
        System.out.println(b.applyAsInt(10, 20));//30
    }
}
```

Method and Constructor references by using ::(double colon) operator

Functional Interface method can be mapped to our specified method by using :: (double colon)operator. This is called method reference.

Our specified method can be either static method or instance method.

Functional Interface method and our specified method should have same argument types, except this the remaining things like

returntype, methodname, modifiersetc are not required to match.

Syntax:

if our specified method is static method

Classname::methodName

if the method is instance method

Objref::methodName

Functional Interface can refer lambda expression and Functional Interface can also refer method reference.

Hence lambda expression can be replaced with method reference. Hence method reference is alternative syntax to lambda expression.

EX: Method Reference

```
interface Interf
{
    public void m1();
}
class Test
{
    public static void m2()
    {
         System.out.println("Implementation by Method 
Reference");
    }
    public static void main(String[] args)
    {
         Interf i = Test::m2; //method reference
         //m2() internally calls m1() method
         i.m1();
    }
}
```

output: Implementation by Method Reference

Method reference usage: Code reusability, already existing m2() method implementation I'm using for m1() method implementation, instead defining new implementation

- ➔ Method reference is alternative syntax to Lambda Expression

Functional Interface can refer Lambda expression

Functional Interface can refer method reference

Syntax for method reference:

Static():

```
     Classname::methodname
```

Ex: Test::m2

Instance():

```
     Objectref::methodname
```

Ex: Test t = new Test();

```
     t::m2
```

Ex: With Lambda Expression

```
//Extending Thread By using lambda expression
class Test
{
     public static void main(String[] args)
     {
          Runnable r = () -> {
               for(int i =0; i<10; i++)
```

```
                {
                        System.out.println("Child Thread");
                }
            };
            Thread t = new Thread(r);
            t.start();
            for(int i =0; i<10; i++)
            {
                System.out.println("Main Thread");
            }
    }
}
```

Ex: With Method Reference

```
//Extending Thread By using method reference
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(); //instance method so
object creation is mandatory
                Runnable r = t::m1;
                Thread t1 = new Thread(r);
                t1.start();
                for(int i = 0; i<10; i++)
                {
                        System.out.println("Main Thread");
                }
    }
}
output:
    Main Thread
    Main Thread
    Main Thread
    Main Thread
    Child Thread
    Child Thread
    Child Thread
    Child Thread
    Child Thread
    Child Thread
    Child Thread
    Child Thread
    Child Thread
    Child Thread
    Main Thread
    Main Thread
    Main Thread
    Main Thread
    Main Thread
    Main Thread
```

In the above example functional interface method m1() referring to Test class instance method m2(). The main advantage of method reference is we can use already existing code to implement functional interfaces (code reusability).

```
//Extending Thread By using static method reference
class Test
{
	public static void m1()
	{
		for(int i = 0; i<10; i++)
		{
			System.out.println("Child Thread");
		}
	}
	public static void main(String[] args)
	{

		Runnable r = Test::m1; //Static method so class
name is using
		Thread t1 = new Thread(r);
		t1.start();
		for(int i = 0; i<10; i++)
		{
			System.out.println("Main Thread");
		}
	}
}
```

In the above example Runnable interface run() method referring to Test class static method m1().

Method reference to Instance method:

Constructor References:

We can use :: (double colon)operator to refer constructors also

Syntax: classname :: new

Runnable r = Test::m1;

Ex:

Interf f = sample :: new;

functional interface f referring sample class constructor

- Functional reference can refer Lambda Expression and Functional reference can refer Method Reference and Constructor Reference so This Method reference and constructor reference are alternative for Lambda expression
- Input argument are Same type: the functional interface which argument we are taking the same argument typ should be use in reference method

If the Functional interface method can take int argument then the reference which method we are going to use we can take compulsory int argument only.

EX:

```
class Sample {
	private String s;

	Sample(String s)
	{
		this.s = s;
		System.out.println("Constructor Executed:" + s);
	}
}

interface Interf
{
	public Sample get(String s);
}

class Test
{
	public static void main(String[] args)
	{
//		lambda expression
		Interf f = s -> new Sample(s);
		f.get("From Lambda Expression");
//          Constructor reference
            Interf f1 = Sample::new;
            f1.get("From Constructor Reference");
      }
}

output:
      Constructor Executed:From Lambda Expression
      Constructor Executed:From Constructor Reference
```

Note: In method and constructor references compulsory the argument types must be matched.

Ex: with lambda Expresssion

```
class Sample
{
      Sample()
      {
            System.out.println("Sample Constructor Execution & 
Object Creation");
      }
}
interface Interf
{
      public Sample get();
}
class Test
{
            public static void main(String[] args)
            {
                  Interf i = () -> {
                        Sample s = new Sample();
                        return s;
                  };
                  Sample s1 = i.get();
      }
}
output: Sample Constructor Execution & Object Creation
```

Ex: with constructor reference

```
class Sample
{
      Sample()
      {
            System.out.println("Sample Constructor Execution & 
Object Creation");
      }
}
```

```
interface Interf
{
	public Sample get();
}
class Test
{
	public static void main(String[] args)
	{
		Interf i = Sample::new;
		Sample s = i.get();
	}
}
```

output: Sample Constructor Execution & Object Creation

Streams:

EX:

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

class Test
{
	public static void main(String[] args)
	{
		ArrayList<Integer> l = new ArrayList<Integer>();
		l.add(0);
		l.add(5);
		l.add(10);
		l.add(15);
		l.add(20);
		l.add(25);
		l.add(30);
		l.add(35);
		System.out.println(l);
//		With Stream (From 1.8v onwards)

		List<Integer> l1 = l.stream().filter(I-
>I%2==0).collect(Collectors.toList());

		System.out.println(l1);
	}
}
//output: [0, 5, 10, 15, 20, 25, 30, 35]
[0, 10, 20, 30]
```

Ex:

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

class Test
{
	public static void main(String[] args)
	{
		ArrayList<Integer> l = new ArrayList<Integer>();
		l.add(0);
		l.add(5);
		l.add(10);
		l.add(15);
		l.add(20);
		l.add(25);
		l.add(30);
		l.add(35);
		System.out.println(l);
//		With Stream (From 1.8v onwards)

		List<Integer> l1 = l.stream().map(I-
>I*2).collect(Collectors.toList());

		System.out.println(l1);
	}
}
```

```
outptu:
[0, 5, 10, 15, 20, 25, 30, 35]
[0, 10, 20, 30, 40, 50, 60, 70]
```

Streams:

To process objects of the collection, in 1.8 version Streams concept introduced.

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

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

What is the difference between collection and stream?

If we want to represent a group of individual objects as a single entity then We should go for collection.

If we want to process a group of objects from the collection then we should go for streams.

We can create a stream object to the collection by using stream() method of Collection interface.

stream() method is a default method added to the Collection in 1.8 version.

```
default Stream stream()
```

```
Ex: Stream s = c.stream();
```

Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.

We can process the objects in the following 2 phases

1.Configuration

2.Processing

1) Configuration:

We can configure either by using filter mechanism or by using map mechanism.

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)

here (Predicate<T> t) can be a boolean valued function/lambda expression

Ex:

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

Hence to filter elements of collection based on some Boolean condition we should go for filter()method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on some function then we should go for mapping mechanism

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

```
public Stream map (Function<T, R> f);
```

It can be lambda expression also

Ex:

```
Stream s = c.stream();
Stream s1 = s.map(i-> i+10);
```

Ex: `Stream s2 = c.stream().map(i->i*2);`

Once we performed configuration we can process objects by using several methods

Processing:

processing by collect() method

Processing by count()method

Processing by sorted()method

Processing by min() and max() methods

```
      forEach() method
      toArray() method
      Stream.of()method.
```

1.Processing by collect() method:

This method collects the elements from the stream and adding to the specified to the collection indicated (specified) by argument.

Ex 1: To collect only even numbers from the array list

Approach-1: **Without Streams**

```
import java.util.ArrayList;

class Test
{
     public static void main(String[] args) {
          ArrayList<Integer> l1 = new ArrayList<Integer>();
          for(int i=0; i<=10; i++)
          {
               l1.add(i);
          }
          System.out.println(l1);

          ArrayList<Integer> l2 = new ArrayList<Integer>();
          for(Integer i:l1)
          {
               if(i%2==0)
               {
                    l2.add(i);
               }
          }
          System.out.println(l2);
     }
}
output:
     [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
               [0, 2, 4, 6, 8, 10]
```

Approach-2: **With Streams**

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

```
class Test
{
     public static void main(String[] args) {
          ArrayList<Integer> l1 = new ArrayList<Integer>();
          for(int i=0; i<=10; i++)
          {
               l1.add(i);
          }
          System.out.println(l1);

          List<Integer> l2 = l1.stream().filter(i ->
i%2==0).collect(Collectors.toList());
          System.out.println(l2);
     }
}
//output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 2, 4, 6, 8, 10]
```

Ex: Program for map() and collect() Method

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

class Test
{
     public static void main(String[] args) {
          ArrayList<String> l = new ArrayList<String>();
          l.add("navin");
          l.add("jyothi");
          l.add("monksha");
          l.add("shwasa");
          l.add("shreyanshi");
          System.out.println(l);

          List<String> l2 =l.stream().map(s-
>s.toUpperCase()).collect(Collectors.toList());
          System.out.println(l2);

     }
}
//output:
[navin, jyothi, monksha, shwasa, shreyanshi]
[NAVIN, JYOTHI, MONKSHA, SHWASA, SHREYANSHI]
```

Ex: example for collect() method with filter() and map()

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

class Test
{
     public static void main(String[] args)
     {
          ArrayList<String> l = new ArrayList<String>();
          l.add("Pavan");
          l.add("Cheeranjevi");
          l.add("Balakrishna");
          l.add("RaviTeja");
          l.add("Nagarjuna");
          l.add("Venaktesh");
          System.out.println(l);
          System.out.println();

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

          List<String> l2 = l.stream().map(s-
>s.toUpperCase()).collect(Collectors.toList());
          System.out.println(l2);
     }
}
output:
     [Pavan, Cheeranjevi, Balakrishna, RaviTeja, Nagarjuna,
Venaktesh]

     [Cheeranjevi, Balakrishna, Nagarjuna, Venaktesh]

     [PAVAN, CHEERANJEVI, BALAKRISHNA, RAVITEJA, NAGARJUNA,
VENAKTESH]
```

2.Processing by count()method

This method returns number of elements present in the stream.

```
public long count()
```

Ex:

```
long count = l.stream().filter(s ->s.length()==5).count();
sop("The number of 5 length strings is:"+count);
```

Ex: Filter with count

```
import java.util.ArrayList;

class Test
{
     public static void main(String[] args)
     {
          ArrayList<String> l = new ArrayList<String>();
          l.add("Pavan");
          l.add("Cheeranjevi");
          l.add("Balakrishna");
          l.add("RaviTeja");
          l.add("Nagarjuna");
          l.add("Venaktesh");
          System.out.println(l);
          System.out.println();

          long count = l.stream().filter(s-
>s.length()>=9).count();
          System.out.println("The Number of Strings who's
length >= 9: " + count);


     }
}
//output:
[Pavan, Cheeranjevi, Balakrishna, RaviTeja, Nagarjuna,
Venaktesh]

The Number of Strings who's length >= 9: 4
```

3.Processing by sorted()method:

If we sort the elements present inside stream then we should go for sorted() method.

We can use sorted() method to sort elements inside Stream.

the sorting can either default natural sorting order or customized sorting order specified by comparator.

sorted()- default natural sorting order

```
sorted(Comparator c)-customized sorting order.
```

Ex:

```
List<String>
l3=l.stream().sorted().collect(Collectors.toList());

sop("according to default natural sorting order:"+l3);


List<String> l4=l.stream().sorted((s1,s2) -> -
s1.compareTo(s2)).collect(Collectors.toList());

sop("according to customized sorting order:"+l4);
```

Ex:

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


class Test
{
	public static void main(String[] args)
	{
		ArrayList<Integer> l = new ArrayList<Integer>();
		l.add(0);
		l.add(15);
		l.add(20);
		l.add(5);
		l.add(10);
		l.add(25);
		System.out.println(l);
		System.out.println();

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

//		List<Integer> l2 = l.stream().sorted((i1,i2)-> -
i1.compareTo(i2)).collect(Collectors.toList());
		List<Integer> l2 = l.stream().sorted((i1,i2)->
i2.compareTo(i1)).collect(Collectors.toList());
		System.out.println("List According to Customized
Sorting order: ");
		System.out.println(l2);
	}
}
//output:
[0, 15, 20, 5, 10, 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

min(Comparator c)
returns minimum value according to specified comparator.

max(Comparator c)
returns maximum value according to specified comparator

Ex:

```
String min=l.stream().min((s1,s2) -> s1.compareTo(s2)).get();
sop("minimum value is:"+min);

String max=l.stream().max((s1,s2) -> s1.compareTo(s2)).get();
sop("maximum value is:"+max);
```

Ex:

```
import java.util.ArrayList;

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);
		System.out.println();

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

```
          Integer max = l.stream().max((i1,i2)-
>i1.compareTo(i2)).get();
          System.out.println("Maximum Value: " + max);
     }
}
//output:
[0, 10, 20, 5, 15, 25]

Minimum Value: 0
Maximum Value: 25
```

5.forEach() method

This method will not return anything.
This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

Ex:

```
l.stream().forEach(s->sop(s));
l3.stream().forEach(System.out:: println);
```

EX:

```
import java.util.ArrayList;

class Test
{
     public static void main(String[] args)
     {
          ArrayList<String> l = new ArrayList<String>();
          l.add("A");
          l.add("BB");
          l.add("CCC");
          System.out.println(l);

//        by using forEach() method
          l.stream().forEach(s -> System.out.println(s));

//        by using method reference
          l.stream().forEach(System.out::println);
     }
}
//output:
[A, BB, CCC]
A
BB
CCC
A
BB
```

CCC

6.toArray() method

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

```
Integer[] ir = l1.stream().toArray(Integer[] :: new);

for(Integer i: ir) {
 sop(i);
 }
```

Ex:

```
import java.util.ArrayList;

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]
0
10
20
5
15
25
```

7.Stream.of()method

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

Ex:

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

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

Ex:

```
package com.naveen;


import java.util.stream.Stream;

class Test
{
     public static void main(String[] args)
     {
//         for Strings
           Stream<Integer> s = Stream.of(9, 99, 999, 9999, 
99999);
           s.forEach(System.out::println);

//         for Arrays
           Double[] d = {10.0, 10.1, 10.2, 10.3, 10.4};
           Stream<Double> s1 = Stream.of(d);
           s1.forEach(System.out::println);
     }
}
//output:
9
99
999
9999
99999
10.0
10.1
10.2
10.3
10.4
```

Date and Time API: (Joda-Time API):

Until Java 1.7version the classes present in Java.util package to handle Date and Time (like Date,Calendar, TimeZoneetc) are not up to the mark with respect to convenience and performance.

To overcome this problem in the 1.8version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of Java.time package.

# program for to display System Date and time.

Ex:
```
import java.time.LocalDate;
import java.time.LocalTime;

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:
2023-06-07
15:49:15.271580500
```

Once we get LocalDate object we can call the following methods on that object to retrieve Day,month
and year values separately.

```
import java.time.*;

class Test
{
	public static void main(String[] args)
	{
		LocalDate date = LocalDate.now();
		System.out.println(date);

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

```
            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:

2023-06-07
15:59:19.640108900
7...6...2023
7-6-2023
```

Once we get LocalTime object we can call the following methods on that object.

```
import java.time.*;

class Test
{
      public static void main(String[] args)
      {
            LocalTime time = LocalTime.now();

            int h = time.getHour();
            int m = time.getMinute();
            int s = time.getSecond();
            int n = time.getNano();

            System.out.printf("%d:%d:%d:%d",h,m,s,n);
      }
}
//output: 16:7:24:435667800
```

If we want to represent both Date and Time then we should go for LocalDateTime object.

```
import java.time.LocalDateTime;
class Test
{
      public static void main(String[] args)
      {
            LocalDateTime dt = LocalDateTime.now();
            System.out.println(dt);
      }
}
//output: 2023-06-07T16:12:13.754253100
```

```
import java.time.LocalDateTime;
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 yy = dt.getYear();
          System.out.printf("%d-%d-%d", dd,mm,yy);
          System.out.println();

          int h = dt.getHour();
          int m = dt.getMinute();
          int s = dt.getSecond();
          int n = dt.getNano();
          System.out.printf("%d-%d-%d-%d", h,m,s,n);

     }
}
//output:
2023-06-07T16:20:28.755318600
7-6-2023
16-20-28-755318600
```

We can represent a particular Date and Time by using LocalDateTime object as follows.

Ex:

```
 LocalDateTime dt1 =
LocalDateTime.of(1995,Month.APRIL,28,12,45);
 sop(dt1);
```

Ex:

```
 LocalDateTime dt1=LocalDateTime.of(1995,04,28,12,45);
 Sop(dt1);
 Sop("After six months:"+dt.plusMonths(6));
 Sop("Before six months:"+dt.minusMonths(6));
```

Ex:

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

```
class Test
{
     public static void main(String[] args)
     {
//          LocalDateTime dt = LocalDateTime.of(1997,
Month.NOVEMBER, 18, 12, 45);
          LocalDateTime dt = LocalDateTime.of(1997, 11, 01,
01, 45);
          System.out.println(dt);

          System.out.println("After Six Months : " +
dt.plusMonths(6));
          System.out.println("Before Six Months : " +
dt.minusMonths(6));

     }
}
//output:
1997-11-01T01:45
After Six Months : 1998-05-01T01:45
Before Six Months : 1997-05-01T01:45
```

Period Object:

Period object can be used to represent quantity of time

Ex:

```
 LocalDate today = LocalDate.now();

 LocalDate birthday = LocalDate.of(1989,06,15);

 Period p = Period.between(birthday,today);

 System.out.printf("age is %d year %d months %d
days",p.getYears(),p.getMonths(),p.getDays());
```

Ex:

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

          LocalDate birthday = LocalDate.of(1997, 11, 01);
          LocalDate today = LocalDate.now();
          Period p = Period.between(birthday, today);
          System.out.printf("Your age is %d years %d Months
and %d Days", p.getYears(), p.getMonths(), p.getDays());

          LocalDate deathday = LocalDate.of(1997+60, 11, 01);
          Period p1 = Period.between(today, deathday);
          int d =
p1.getYears()*365+p1.getMonths()*30+p1.getDays();
          System.out.printf("\nYou will be on earth only %d
Days, \nHurry Up to do more important things", d);
     }
}
//output:
Your age is 25 years 7 Months and 6 Days
You will be on earth only 12555 Days,
Hurry Up to do more important things
```

Ex: write a program to check the given year is leap year or not

```
import java.time.Year;
import java.util.Scanner;

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 :
1997
1997 year is Not Leap year
```

To Represent Zone:

ZoneId object can be used to represent Zone

Ex:

```
import java.time.ZoneId;
class Test
{
	public static void main(String[] args)
	{
		ZoneId zone = ZoneId.systemDefault();
		System.out.println(zone);

	}
}
//output: Asia/Calcutta
```

We can create ZoneId for a particular zone as follows

Ex:

```
import java.time.*;

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
2023-06-07 T04:43:45.914673500-07:00[America/Los_Angeles]
```