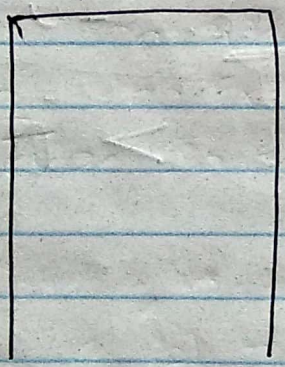
Java 8

Topics :

1) λ-expressions
2) Functional interfaces
3) Default & Static methods in Interface
4) java.util.function
   a) Predicate<T> → test( )
   b) Function<T,R> → apply( )
   c) Consumer<T> → accept( )
   d) Supplier<R> → get( )
5) :: Double colon operator
   ↳ Method Reference
   → Constructor Reference
6) Streams
7) Date & Time ApI

Web.xml

1) We have to ~~proide~~ provide mapping blw filterdispatcher and URL pattern



STS

Lect No → 31, 32, 33, 34, 35, 36, 37 39, 44, 50, 51, 58, 62, 63, 89, 96, 97, 98

Java 8

Lambda Expression :)

- It is anonymous function
  - → Not having method name
  - → " " return type
  - → " " access modifiers

e.g (I) `public void m1() { SOP("Hi"); }` ⇒ `() → { S.O.P("Hello"); }`

e.g
```
public void add(int a, int b)
{
    S.O.P(a+b);
}
```

① `(int a, int b) → {S.O.P (a+b);}`

② `(int a, in b) → S.O.P (a+b);`

③ `(a, b) → S.O.P(a+b);`

Note: if only one Line is there you can remove curly braces.

e.g (II)
```
public int getNumber(String s)
{
    SOP(s.length());
}
```

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

λ → `(String s) → return s.length();`

Note: Without curly braces we should not use return statement.

λ → `(s) → s.length();`

→ when one parameter is being passed

λ → `s → s.length();`

(I) Properties of Lamdbda Expression

1) A λ expressions can take any no. of parameters e.g
```
() -> sop("Hello");
(s) -> s.length();
(a,b) -> sop (a+b);
```

2) If multiple parameters present then there parameters should be represented with comma separated
e.g (a, b) → sop(a+b);

3) If only one parameter available then parenthesis are optional
```
λ (s) -> s.length();
λ  s -> s.length();
```

4) Usually we can specify the type of parameter of compiler expect the type based on context, then we can remove type
[Type Inference]

```
eg (int a, int b) -> sop(a+b)
λ -> (a, b) -> sop(a+b);
```

5) Similar to method body, λ-expression body can contain any number of statements. If multiple statements are there then we should enclose within curly braces
```
e.g () -> {
          ≡
          ≡
          };
```

If body contains only one statement then curly braces are optional
e.g () → sop("Hello");

6) If λ expression return something then we can remove (return) keyword
s → s.length();

Functional Interface

→ It contains single abstract method (SAM) and can have multiple default or static methods

e.g Runnable Interface → contains only run() method
Callable " → " " call() "
ActionListener " → " " actionPerformed()"
Comparable " → " " compareTo() "

@FunctionalInterface → annotation

```
interface Parent
{
    public void height();

    default void smile()
    {
    }

    public static void m3()
    {
    }
}
```

Since, this interface contains only one abstract method
It is Functional Interface

```
interface Parent
{
    public void height();
    public void smile();
}
```

It contains more than one abstract methods.
This is not Functional interface

Lec-7) Functional Interface w.r.t Inheritance

Case 1) If an interface extends Functional Interface and child interface does not contain any abstract method, then child interface is always Functional Interface

e.g
```
@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

e.g
```
@FunctionalInterface
interface P
{
  public void m1();
}
@FunctionalInterface
interface C extends P
{
  public void m1();
}
```

case 3) In the child interface we can't define any new abstract methods otherwise we will get compile time error.

e.g
```
@FunctionalInterface
interface P
{
  public void m1();
}
@FunctionalInterface
interface C extends P
{
  public void m2();
}
```

CE: Unexpected @FunctionalInterface annotation, multiple non-overriding abstract methods found in interface 'C'

Case 4)
```
@FunctionalInterface
interface P
{
  public void m1();
}
interface C extends P
{
  public void m2();
}
```

No annotation

Note: This code will get compile

Lecture 8 Invoking λ Expressions By using Functional Interface

without λ expression

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

class Demo implements Parent
{
  public void m1()
  {
    S.O.P("m1 implementation");
  }
}

Class Main
{
  p. s. v. main(String a[])
  {
    Parent p= new Demo();
    p. m1();
  }
}
```

With λ expression

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

Class Main
{
  p.s.v. main(String a[])
  {
  Parent p= ()-> S.O.P("m1()");
  p. m1();
  }
}
```

Without λ expression

```
interface Father
{
  public void add(int a,int b);
}

class Demo implements Father
{
  @Override
  public void add(int a, int b)
  {
    S.o.p (a+b);
  }
}

Class Test
{
  p.s.v.m(      )
  {
    Father f = new Demo();
    f.add(10,20);
    f.add(40,60);
  }
}
```

O/p: 30
100

Eg

```
Class Test
{
  P.S.V. main(String a[])
  {
    Runnable r=()->{ For(int i=0; i<5; i++)
                     {
                       S.O.P("Child thread");
                     }
                   };
    Thread t = new Thread (r); //Till here only One main thread is present
    t.start();
```

↳ After this Line another child thread starts simultaneously.

```
    For(int i=0; i<5; i++)
    {
      S.O.P("Main thread");
    }
```

→ Main thread will execute

Note:- Functional Interface

1) It should contains exactly one abstract method Single Abstract Method (SAM).

2) It may contain multiple default methods and Static methods.

3) It acts as a type for λ-expression

e.g:

```
Inter i = () -> S.O.P("Hello");
```

Inter ↓ Type

Functional Interface

int a = 10; → It a int type

String s = "Hello"; → It is a string type

imp

4) It can be used to invoke λ-expression
e.g i.m1();

5) Functional Interface acts as a type for λ expression

imp

6) Why Functional Interface should contains only one abstract method ?

```
@FunctionalInterface
interface Parent
{
   public void m1(int i);
}

Class Test
{  P.S.V.m
   {
      Parent p = i -> i*i;
      Parent P = i -> S.O.p(i*i);
      p.m1(10);
      p.m1(20);
   }
}
```

Compiler mapped the λ expression with m1 method through that it comes to know that i is of int type

Through this λ expression is invoked

o/p : 100
: 400

7) Only for Functional Interface we can write λ expression implementation

e.g If functional Interface contains more than one abstract method then see example

```
@FunctionalInterface
interface Parent
{
    public void m1(int i);
    public void m2(int i);
}

class Test
{
    p.s.v.m main
    {
        Parent p = i -> s.o.p(i*i);
        p.m1(10);
        p.m2(10);
    }
}
```

It won't allow to add this

This is for confusing it won't work

In order to map λ expression with Parent interface, immediately you will get error that Parent interface must be Functional Interface

Lecture 16: Sorting Elements of Arraylist Using Lambda expression

```
List<Integer> l = new ArrayList<>();
l.add(10);
l.add(20);
l.add(5);
l.add(30);
l.add(0);
S.OP => [10,20, 5, 30, 0];
```

(I) By default approach to sort number is ascending order

```
Collections.sort(l);
```

O/p: [0, 5, 10, 20, 30]

II To sort number in descending order

step 1) use comparator interface which is functional interface bcoz it has one abstract method name compare(Object O_1, Object O_2)

```
~~interface class~~ MyComparator implements
class MyComparator implements Comparator<Integer>
{
  public int compare(Integer I1, , Integer I2)
  {
    return (I1>I2)?-1:(I1<I2)?1:0;
  }
}
```

ternary Operator

```
Class Test{
  ~~public~~ p.s.v.m{
      Comparator<Integer> c = new MyComaparto()
      Collections.sort(l, c);
```

Using lambda Expression

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

~~Streams~~ Streams

java.utils.stream.	java.io.stream
→ This is to perform operation on collection	This stream is used perform read and write operation

Collection

→ To represent group of objects as a single entity is called collection.

Stream

→ To process data from collection object is called stream.

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

1) Configuration → Filter mechanism
                 → map mechanism

2) Processing

1) Filtering

1) If we want to filter elements from the collection based on some boolean condition, then we should go for filtering.

2) We can configure filter by using filter() method of Stream interface

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

It can be boolean valued function or λ expression

<T, R>
→ Return type
→ Class Object

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)

e.g Stream s = c.stream().map(i → i*2);

Lec: 83 Various methods of Stream

1) Processing by collect() method
→ This method collects the elements from the stream and adding to the specified collection.

Collection interface ka method

```
List<String> list1 = list.stream().filter(i->i.length>6).
                     collect(Collectors.toList());
```

stream interface ke andar method hai

```
List<String> list2 = list.stream().map(s->s+"India")
                     .collect(Collectors.toList());
```

Collectors → class
toList() → static method

```
Long count = list.stream().filter(i->i.length()>6).
             .count();
```

stream interface ke andar hai

1) collect()
2) count()
3) sorted()
4) min() & max()
5) forEach()
6) toArray()

Lec-24 Default Methods Inside interfaces

Till jara 1.7

```
interface Interf
{
   public abstract void m1();
   public static final int x=10;
}
```

From jara 1.8

```
interface Interf
{
   default void m1();
   {
      s.o.p("default method");
   }
}
```

→ It is Keyword, It is not access modifier.

Directly calling interface method:

```
class Test implements Interf
{
   p.s.v.m(String a[])
   {
      Test t= new Test();
      t.m1();
   }
}
```

o/p: default method

Overriding the default method, providing own imple~

```
class Test implements Interf
{
   @Override → optional (likho or mat likho)
   public void m1();
   {
      s.o.p("Custom implementat");
   }
   p.s. void main(String [] a)
   {
      Test t = new test();
      t.m1();
   }
}
```

o/p: Custom implementation

Lec 25) Default methods w.r.t multiple inheritance

```
interface Left
{
  default void mob()
  {
    S.O.p("Left D.Method");
  }
}
```

```
interface Right
{
  default void mob()
  {
    S.O.p("Right Default Method");
  }
}
```

```
class Test implements Left, Right
{
  ambiguity problem
}
```

o/p: CE : Class Test inherits unrelated defaults for mob() from type Left and Right

Solution:

```
Class Test implements Left, Right
{
  public void mob()
  {
1  S.O.p("My own implementation");
2  Left.super.mob();
3  Right.super.mob();
  }
  Public static void main(String args[])
  {
    Test t = new Test();
Case1 : t.mob(); //output : My Own implementation
  }
```

If you want to use defaut methods

Output case 1) My Own implementation
case 2) Left Default method
case 3) Righ Default method

Lec-26 Difference between I.

Interface with default methods	Abstract Class
1) Inside interface every variable is always public, static and final, we cannot declare instance variable.	2) Inside Abstract class we can declare instance variables, which are required to the child class.
2) We cannot declare constructors	2) We can declare constructor in order to initilize instance variables
3) We cannot declare instance blocks and static blocks	3) We can declare instance blocks & static blocks
4) Function interface with single abstract method can use lambda expression.	4) We cannot use lambda expression.
5) Inside interface we can't override Object class method.	5) Inside Abstract class we can override Object class method.

Static methods inside interface

1) The purpose of providing static method inside interface is for general utility method that may be required commonly across application.

```
interface Interf
{
  public static void mob()
  {
    SOP("interface static method");
  }
}

Class Test
{
  P.S.V main(String a[])
  {
    Interf.mob(); // you can directly called
  }
```

Note: Class Test is not implementing interface still it can call

↓ Only way to call

No need of implementing

Note: 1) Overriding w.r.t to interface static method is not applicable

2) interface static method is not available for the implementation class, due to this overriding is not applicable here

Ex 1)

```
interface Interf
{
    public static void mob()
    {

    }
}
Class Test implements Interf
{
    public static void mob()
    {

    }
}
```

This is not overriding

You cannot override static method.

Note: I can write the same name of the method in implementation class, it is valid but you cannot say overriding

```
Class Parent
{
    public static void walk()
    {

    }
}
Class Child extends Parent
{
    @Override
    public void walk()
    {

    }
}
```

Not you cannot override ~~make~~ Static method of parent class to non static in child class

Compile Time error.

```
interface Interf
{
    public static void mob()
    {

    }
}
Class Test implements Interf
{
    public void mob()
    {

    }
}
```

Here mob() is not overriding method of parent class, due to this it is valid code.

Ex 3:

```
class Parent
{
  public static void mob()
  {
  }
}
class Test implements Interf
{
  @Override
  private static void mod()
  {
  }
}
```

You cannot reduce the visibility of the parent method

Compile time Error

```
interface Interf
{
  public static void mob()
  {
  }
}
class Test implements Interf
{
  private static void mob()
  {
  }
}
```

Static method of interface you cannot override.
Implementation class has it's own method mob().
This code run successfully.

Note: From 1.8 Java, we can directly run from command line

```
interface Interf
{
  p.s.v. main(String a[])
  {
    s.o.p("Interface main method");
  }
}
```

→ static method is allowed in interface

→ javac Interef.java
→ java Interf

~~Predicate interface~~

1) Predefined functional Interface.

a) Functional Interface → Single Abstract Method
→ Multiple default Method
→ Multiple static method

java.util.function
1) Predicate<T>
2) Function<T, R>
3) Consumer<T
4) Supplier<R

E.g of Functional Interface till 1.7 Java
1) Comparable ⇒ compareTo()
2) Comparator ⇒ compare()
3) Runnable ⇒ run()
4) Callable ⇒ call()
5) ActionListener ⇒ actionPerformed()

Predicate<T>

```
interface Predicate<T>
{
    boolean test(T t);   // only 1 arguement is allowed.
}
```

E.g 1) To check number is greater than 10.

refer

```
Predicate<Integer> p = i -> i > 10;
p.test(11);  // true
p.test(5);   // false
```

Lambda expression can be refer by functional interface. or
Functional Interface ⇒ can refer Lambda expression

E.g 2)

```
import java.util.function.*;
Class Test
{
  public static void main(String args[])
  {
    Predicate<String> p = s -> s.length() > 5;
      sop (p.test("balmukund"); // true
      sop (p.test("abcd"); // false
  }
}
```

Ex. 3 To check whether list object is null

Logic => `Predicate<Collection> p = c -> c.isEmpty();`

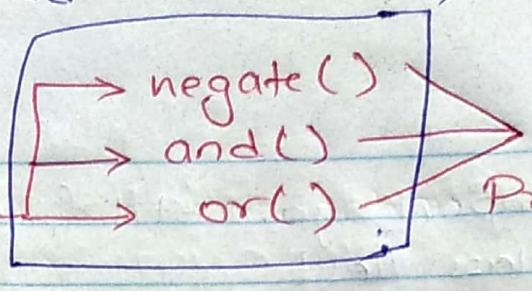
```
Class Test
{
  p. s. v. main(String args[])
  {
    Predicate<Collection> p = c -> c.isEmpty();

    ArrayList<String> a1 = new ArrayList<>();
      a1.add("abc");
    System.out.println(p.test(a1); // false

    ArrayList<String> a2 = new ArrayList<>();

    System.out.println(p.test(a2)); // true
```

all are default method Predicate<T> interface



Predicate Joining

Ex 1) P1 : Given number greater than 10
P2 : Given number must be even number

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

Class Test
{
  public static void main(String a[])
  {
  Predicate<Integer> P1 = i -> i>10;
  Predicate<Integer> P2 = i -> i%2==0;
          P1.test(12); // true
          P2.test(12); // true

  Predicate<Integer> P = P1.and(P2);
          P.test(12); //true

  Predicate<Integer> Pg = P1.or(P2);
          P.test(9); // false

  Predicate<Integer> P5 = P1.negate();
```

# @FunctionalInterface
public interface Predicate<T> {
p.a. boolean test(T t); → SAM
default Predicate<T> and(Predicate P){};
default Predicate<T> or(Predicate P){};
" " negate(){};

Function<T, R> → andThen() → default method
→ compose() → default method
→ identity() → static method

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

eg 1
```
import java.util.function.Function;
class Test
{
  p.s.v. main(String args[])
  {
    Function<String, Integer> f = s -> s.length();
      s.o.p(f.apply("durga"); // 5
      s.o.p(f.apply("balmukund"); // 9
  }
}
```

eg 2 Return square of given number

```
Function<Integer, Integer> f = i -> i*i;
  s.o.p(f.apply(5)); //25
  s.o.p(f.apply(2)); //4
```

Predicate	Function
1) It is used for to check condition which returns boolean result.	1) To perform certain operat^n which return some value as a result.
2) It takes one input parameter	2) It takes 2 parameters. Input type and return type both we have to specify.
3) One abstract method test( )	3) One abstract method apply( ).
4) public boolean test(T t)	4) public R apply(T t)
5) It will return only boolean value.	5) It can return any type of value.

Function Chaining

f_1 & f_2

① f_1.andThen(f_2) => f_1 will be applied first then f_2 will be applied

② f_1.compose(f_2)

⇓

f_2 first, then f_1 will be applied

Ex. Example of Function chainning

```
import java.util.function.Function;
Class Test
{
  P.S.V.M(  )
  {
  Function<String, String> f1 = S -> S.toUpperCase();
  Function<String, String> f2 = S -> S.substring(0,9);
   S.O.P(f1.apply("balmukund Singh");
   S.O.P(f2.apply("balmukundSingh");

  Function<String, String> f3 = f1.apply(f2);
   S.O.P(f3.apply("balmukundSingh");

S.O.P(f1.compose.(f2).apply("balmukundSingh");
```

O/P : BALMUKUND

Ex. 2

```
Function<Integer, Integer> f1 = i -> i+i;
Function<Integer, Integer> f2 = i -> i*i*i;
S.O.P(f1.apply(f2))
S.O.P(f1.andThen(f2).apply(2)); // 64

S.O.P(f1.compose(f2).apply(2)); // 16
```

Function static method

```
interface Function<T,R>
{
    public static<T> Function<T,T> identity()
    {
    }
}
```

e.g

```
import java.util.function.*;
class Test
{
    p.s.v.m(String a[])
    {
        Function<String,String> f = Function.identity();
        String s = f.apply("durga");
        S.O.P(s);        // durga
```

Consumer <T>

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

It won't return anything.

e.g 1)
```
Consumer<String> c = s -> s.o.p(s);
c.accept("Hello");   // Hello
c.accept("Durga");   // Durga
```

e.g 2) To display Student Details

```
Student s1 = new Student("Rahul", 28, "Mumbai");
Student s2 = new Student("Virat", "26", "Dehiradun");
ArrayList<Student> list = new ArrayList<>();
list.add(s1);
list.add(s2);
```

object

```
Consumer<Student> c = s -> {
    s.o.p(s.toString())
}

for(Student s : list)
{
    c.accept(s);
}
```

Consumer<T>

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

It won't return anything.

e.g 1)
```
Consumer<String> c = s -> s.o.p(s);
c.accept("Hello");   // Hello
c.accept("Durga");   // Durga
```

e.g 2) To display Student Details

```
Student s1 = new Student("Rahul", 28, "Mumbai");
Student s2 = new Student("Virat", "26", "Dehiradun");
ArrayList<Student> list = new ArrayList<>();
list.add(s1);
list.add(s2);
```

object

```
Consumer<Student> c = s -> {
    s.o.p(s.toString())
};

for(Student s : list)
{
    c.accept(s);
}
```

andThen() => default method of Consumer

Consumer Chaining

```
Student s1 = new Student("Virat", 28);
        s2 = "      "    ("Rahul", 26);
        s3 = "      "    ("Dhoni", 37);

AL<Student> list = new AL<>();
            list.add(s1);
            list.add(s2);
            list.add(s3);

Consumer<Student> c1 = std -> s.o.p(std.getName);
Consumer<Student> c2 = std -> s.o.p("storing data");
Consumer<Student> c3 = std -> s.o.p("showing details");

Consumer<Student> c4 = c1.andThen(c2).andThen(c3);

for(Student s : list)
{
    c4.accept(s);
}
```

☒ default Method
☒ static Method

# interface Supplier<R>
{
    R get(); → It only return value.
}
→ It is having only single abstract method.

Supplier interface doesn't have any default method and static Method.

e.g 1)

```
Class Test
{
    p. s. v. main ( )
    {
        Supplier<Date> s = ( ) → new Date();
    s.o.p(s.get());
    } }
}
```

→ Supplier doesn't require arguement

e.g 2)

```
Supplier<String> s = () → {
                    String arr[] = {"A", "B", "C", "D"}

                    int x = (int)(Math.random()*4);
                    return arr[x];
                };

s.o.p(s.get());
```

Note Math.random() generates value

0 ≤ x < 1

Minimum 0.00

Maximum 0.99

E.g 3 To generate OTP 6 digits

```
Supplier<Integer> s = () -> { string otp = "";
                              for(i=1; i<6; i++)
                              { int otp = (int)(Math.random() * 10);
                                otp
```

```
= () ->
{
    string otp = " ";

    for (i=0; i<6; i++)
    {
        otp = otp + (int)(Math.random()*10)
    }
    return otp;
};

s.o.p (s.get());  // you will get 6 digit OTP
```

E.g 4) To generate 8 characters random password

1) Length should be 8 characters
2) 2, 4, 6, 8 places only digits
3) 1, 3, 5, 7 places only uppercase alphabets
   symbols @, #, $

```
Supplier<String> s = () -> {

                        String pwd = " ";
Supplier<Integer> d = () -> (int)(Math.random()*10)
                       ↓
                     digit
```

```
String alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ@#$";
Supplier<Character> char = () ->
{
    Supplier<Integer> index = () -> (int)(Math.random() * 30);
    return alpha.charAt(index);
}

for(i=1; i<=8; i++)
{
    Predicate<Integer> p = i -> i==1 | i==3 | i==5 | i==7;
    if(p.test(i))
    {
        pwd = pwd + char.get();
    }
    else
    {
        pwd = pwd + digit.get();
    }
}
return pwd;
};

S.O.P(s.get())
```

O/P: A2B6#4$0

1 input parameter

1) Predicate<Integer> p = () → i % 2 == 0;

2) Function

3) Consumer

All are one arguement Functional Interface

default & Static Methods

2 input parameter

1) BiPredicate

2) BiFunction

3) BiConsumer

All are two arguements Functional Interface

default & Static methods

same hai

BiPredicate

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

```
interface BiPredicate<T,U>
{
  public boolean test(T t, U u);
}
```

two i/p parameters

E.g BiPredicate<Integer, Integer> p = (a, b) → (a+2) % 2 == 0;

```
S.O.p( p.test(10,2); // true
S.O.p( p.test(7, 4); // false
```

BiFunction

i/p, o/p

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

i/p, o/p

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

```
BiFunction<Integer, Integer, Integer> f = (a,b) ->
                                          a*b;
    S.O.p(f.apply(10,20)); //200
```

E.g 1) Creation of Student Object

```
BiFunction<String, Integer, Student> f = (name, rollno)
                  -> new Student(name, rollno);

AL<Student> list = new AL<>();
   list.add(f.apply("Durga", 5));
   list.add(f.apply("BMS", 6));
```

BiConsumer

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

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

BiConsumer<String, String> c = (s1, s2) -> s.o.p(s1+s2);

c.accept("Balmukund", "singh");

//BalmukundSingh

One argument Function Interfa

```
interface Predicate<T>
{
 public boolean test(T t)
 default Predicate and(Predicat P)
default Predicate or(Predicate P)
default Predicate negate()
 static Predicate isEqual(Object O)
}
```

```
interface Function<T, R>
{
public R apply(T t);
default Function andThen(Fu^n F)
default Fumction compose(Funct^n F);
static Function ideatity();
}
```

```
interface Consumer<T>
{
public void accept(T t);
default Consumer andThen(Cons^r C)
}
```

Two argument Funct^n Interface

```
interface BiPredicate<T, U>
{
public boolean test(T t, U u);
default BiPredicate and(BiPredt P);
default BiPredicate or(BiPredicate P);
 default BiPredicate negate();
```
— No — static — method
```
}
```

```
interface BiFunction<T, U, R>
{
 public R apply(T t, U u);
default BiFunction andThen(Fu^n F);
```
— No — compose — method
— No — identity - method
```
}
```

```
interface BiConsumer<T, U>
{
public void accept(T t, Uu);
default BiConsumer
          andThen(Consumer c);
```

Note: Supplier is not added here because it won't accept any input argument.

1) Autoboxing → Compiler automatically converts from P.T to O.T

→ Primitive type to Wrapper Object type;

Integer I = 10; ✗ 1.4V

⇓

1.5 V compiler → [Integer.I = Integer.valueOf(10);]

2) AutoUnboxing → Compiler automatically converts from Wrapper Object type to primitive type

→ Wrapper Object Type ⇒ Primitive Type

Integer I = new Integer(10);

int x = I;

↳ Compiler [int x = I.intValue();]

3) Generic Type Parameter

AL<int> al = new AL<int>(); ✗ CE

↑

You are not allowed to pass primitive type here, Only Wrapper class Object can be pass.

AL<Integer> list = new AL<Integer>(); ✓

Primitive Type Functional Interface for Predicate

```
interface IntPredicate
{
 public boolean test(int i);

}
import java.util.function.*;
class Test
{
 p.s.v. main(String a[])
 {
   int x[]={0,5,10,15,20,25};

IntPredicate p=i->i%2==0;
 for(int i=x)
 {
    if(p.test(i))
    {
      s.o.p(i);
    }
}}
}
```

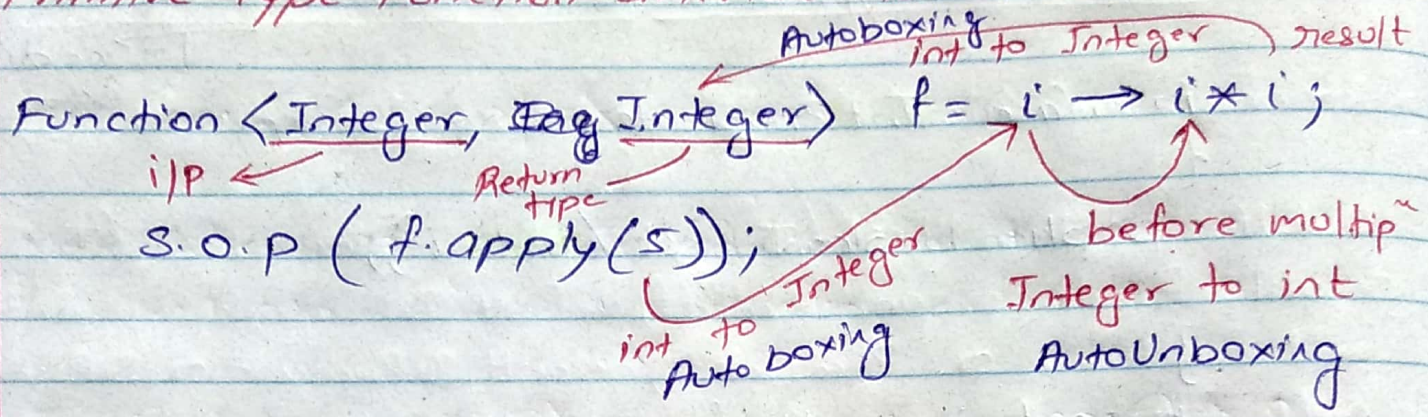
```
interface LongPredicate
{
public boolean test(Long l)

}
```

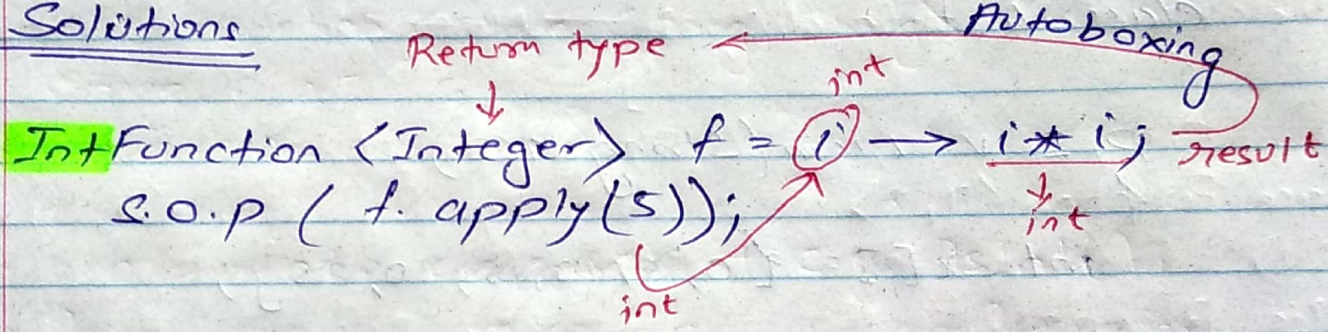
```
interface DoublePredicate
{
 public boolean test(double d);

}
```

Primitive Type Function Interface for Function



Solutions



e.g 1) Find length of string

ToIntFunction <String> f = s → s.length();

Return Type

input Type

s.o.p(f. applyAsInt("durga")); //5

return int value

e.g 2) find the square root of int value

IntToDoubleFunction f = i → Math.sqrt(i);

s.o.p(f.applyAsDouble(5));

Primitve Type function Interface for Consumer

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

IntConsumer c = i -> i*i;
S.O.p(C.accept(5));

2) interface LongConsumer
{
  public void accept(Long l);
}

3) interface DoubleConsumer
{
  public void accept(double d);
}

4) ObjIntConsumer<T>
  Public void accept(T t, int i);

Object Type

Primitve type

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

6) ObjDoubleCosumer<T>
  public void accept(T t, double d);

e.g
ObjDoubleConsumer<Employee> c = (e, d) ->
e.salary = e.salary + d;
e1 = new Emp("Durga", 1000);
S.O.p(C.accept(e1, 500.0));

Primitive type Function Interface for Supplier

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

```
IntSupplier s = () -> (int)(Math.random() * 10);

String otp = "";
for(i=0; i<6; i++)
{
  otp = otp + s.getAsInt();
}

return otp;
```

2) LongSupplier
    public long getAsLong();

3) DoubleSupplier
    public double getAsDouble();

4) BooleanSupplier
    public boolean getAsBoolean();

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

UnaryOperator and its Primitive Versions

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

UnaryOperator<T> → child of Function<T,T> interface

Note: You can only use UnaryOperator input type parameter = Return type Parameter

e.g 1)

```
UnaryOperator<Integer> o = i -> i*i
s.o.p(o.apply(5)); //25
```

Autoboxing of result

AutoUnboxing

Autoboxing

Solution

e.g 2)

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

IntUnaryOperator u = i -> i*i;
s.o.p(u.applyAsInt(5));
```

3) LongUnaryOperator

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

4) DoubleUnaryOperator

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

BinaryOperator

1) BiFunction<T, U, R>

Note: 2 i/p parameters are different
1 o/p " is also different

2) BiFunction<T, T, T>
e.g BiFunction<String, String, String>
i/p Return type

Note: When all are argument and return type is same better go for BinaryOperator

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

e.g 1 BinaryOperator<String> bo = (s1, s2) -> {s1 + s2};
bo.apply("Durga", "soft");
// Durgasoft

e.g 2) BinaryOperator<Integer> bo = (i1, i2) -> i1 * i2;
bo.apply(5, 10);
Autoboxing
Autoboxing
Auto Unboxing

Primitive Versions for BinaryOperator

1)
```
interface IntBinaryOperator
{
    public int applyAsInt(int i1, int i2);
}

IntBinaryOperator b = (i1, i2) -> {i1 + i2};
    S.O.P(b.apply(5, 10));
// 15
```

2) LongBinaryOperator
```
    public long applyAsLong(Long L1, Long L2)
```

3) DoubleBinaryOperator
```
    public double applyAsDouble(double d1, double d2);
```

Method Reference & Constructor Reference by :: operator

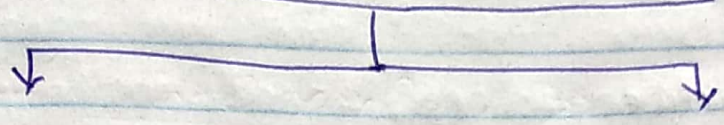
```
interface Interf
{
  public void mob();
}
 Class Test
 {
  p.s.v. main( )
  {
    Interf i= () -> { s.o.p("λ-Expression");
                      s.o.p("New topic");
                    };
} }
```

* By Method Reference for code reuseability

```
interface Interf
{ public void mob();
}
class Test
{ public static void map( )
  {
    s.o.p("map method");
  }
  public static void main(String a[])
  {
     Interef i = Test::map;
        i.map();
  }
}
```

Since, map is a static method, that's y it is called as `Test::map` → this also provide the implementation of map method to the mob method

Functional Interface



λ-Expression

Method Reference

Syntax:

a) For static method

```
class_name :: method_name;
```

b) For instance method

```
object_ref :: method_name;
```

e.g

```
Test t = new Test();
t::mapb();
```

Note: Interface single abstract method can be implemented by 3 ways

```
Interf i = new InterfImplClass();
         = λ-Expression
         = Method Reference
```

Class Constructor Reference (::)

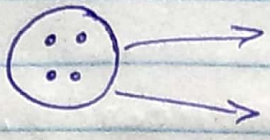
```
Class Employee
{
  private string name;
  private string age;
  public Employee(string name, string age);
  {
    this.name = name;
    this.age = age;
  }
}
```

```
interface Interf
{
  public Employee get();
}
```

```
Class Test
{
  p.s.v. main( )
  {
    Interf i = Employee::new;
    Employee e1 = i.get();
  }
}
```

Using lambda Expression

```
Class Test
{ p.s.v. main( )
  {
    Interf i = () -> { Employee e = new Emp();
                       return e;
                     };
  Employee e1 = i.get();
```



→ Method Reference
→ Constructor Reference

λ- Expression can be Replaced by → Method Reference / Constructor Reference

Date & Time API

```
import java.time.*;

LocalDate date = LocalDate.now();
    S.O.p(date);        // 2019-06-30

LocalTime time = LocalTime.now();
    S.O.p(time);        // 09:40:24.623
```

To customize date format

```
int dd = date.getDayOfMonth();
int mm = date.getMonthValue();
int yyyy = date.getYear();

S.O.P(dd+"-"+mm+"-"+yyyy);
System.out.printf("%d-%d-%d", dd, mm, yyyy);
```

To customize time format

time → getHour()
time → getMinute()
time → getSecond()
time → getNano()

LocalDateTime

```
LocalDateTime dt = LocalDateTime.now();
    S.O.p(dt);
// 2019-06-30 10:18:56.287

    int dd = dt.getDayOfMonth();
    int mm = dt.getMonthValue();
    int yy = dt.getYear();
printf ("%d-%d-%d", dd, mm, yy);

    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);
```

To find particular date & time

```
LocalDateTime dt = LocalDateTime.of(yy, mm, dd, h, m, s, u);
```

e.g

```
LocalDateTime dt = LocalDateTime.of(1960, 7, 6);
    S.O.p (dt)   // 1960-07-06

S.O.p("After six months: " + dt.plusMonths(6));
S.O.p("Befor six months: " + dt.minusMonths(6));
```

Period

```
LocalDate birth = LocalDate.of(1989, 8, 28);
Local Date today = LocalDate.now();
Period p = Period.between(birth, today);

S.O.printf("Your age is : %d", p.getYears());
S.O.printf("and days is : %d", p.getDays());
```

Year class

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter year number");
int n = sc.nextInt();
Year y = Year.of(n);
```

Year class

```
if(y.isLeap())
   S.O.printf("Leap year %d", n);
else
   s.o.printf("%d Not a leap year", n);
```

ZoneId Class

```
ZoneId zone = ZoneId.systemDefault();
   S.O.p(zone);          // Asia/Culcutta

ZoneId la = ZoneId.of("America/Los_Angeles");
ZonedDateTime zt = ZonedDateTime.now(la);
S.O.p(zt);
```

1) To get system current time

```
LocalDate dt = LocalDate.now();
S.O.p(dt);
```

2) To get particular date with year

```
LocalDate dt = LocalDate.of(yyyy, mm, dd);
```

Beside there range value you will get error DateTimeException

- Range => 1 to 12
- Range => 1 to 31

Parse & Format method

e.g 1

```
LocalDateTime dt = LocalDateTime.parse("2014-05-04T13:45:45.000");
String s = dt.format(DateTimeFormatter.ISO_DATE_TIME);
S.O.p(s)
```

e.g 2

```
LocalDate date1 = LocalDate.now();
LocalDate date2 = LocalDate.of(2019, 5, 5);
LocalDate date3 = LocalDate.parse("2019-05-05");
S.O.p(date1); // 2019-05-05
S.O.p(date2); // 2019-05-05
S.O.p(date3); // 2019-05-05
```