Java 8 New Features

java 7 – July 28th 2011
2 Years 7 Months 18 Days

Java 8 - March 18th 2014

Java 9 - September 22nd 2016

Java 10 - 2018

After Java 1.5version, Java 8 is the next major version.
Before Java 8, sun people gave importance only for objects but in 1.8version oracle people gave the importance for functional aspects of programming to bring its benefits to Java.ie it doesn't mean Java is functional oriented programming language.

Java 8 New Features:

1) **Lambda Expression**
2) **Functional Interfaces**
3) **Default methods**
4) **Predicates**
5) **Functions**
6) **Double colon operator (::)**
7) **Stream API**
8) **Date and Time API**

Etc…..

Lambda (λ) Expression

☀ Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly this concepts started using in programming world. "LISP" is the first programming which uses Lambda Expression.

☀ The other languages which uses lambda expressions are:
- C#.Net
- C Objective
- C
- C++
- Python
- Ruby etc.

and finally in Java also.

☀ The Main Objective of Lambda Expression is to bring benefits of functional programming into Java.

What is Lambda Expression (λ):

- Lambda Expression is just an anonymous (nameless) function. That means the function which doesn't have the name, return type and access modifiers.
- Lambda Expression also known as anonymous functions or closures.

Ex: 1

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

```
() → {
        sop("hello");
    }
() → { sop("hello"); }
() → sop("hello");
```

Ex:2

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

```
(inta, int b) → sop(a+b);
```

- If the type of the parameter can be decided by compiler automatically based on the context then we can remove types also.
- The above Lambda expression we can rewrite as (a,b) → sop (a+b);

Ex: 3

```
public String str(String str) {
        return str;
}
```

(String str) → return str;

↓

(str) → str;

Conclusions:

1) A lambda expression can have zero or more number of parameters (arguments).

   Ex:
   () → sop("hello");
   (int a) → sop(a);
   (inta, int b) → return a+b;

2) Usually we can specify type of parameter. If the compiler expects the type based on the context then we can remove type. i.e., programmer is not required.

   Ex:
   (inta, int b) → sop(a+b);

   ↓

   (a,b) → sop(a+b);

3) If multiple parameters present then these parameters should be separated with comma (,).

4) If zero number of parameters available then we have to use empty parameter [like ()].

   Ex: () → sop("hello");

5) If only one parameter is available and if the compiler can expect the type then we can remove the type and parenthesis also.

   Ex:
   (int a) → sop(a);

   ↓

   (a)→ sop(a);

   ↓

   A → sop(a);

6) Similar to method body lambda expression body also can contain multiple statements. If more than one statements present then we have to enclose inside within curly braces. If one statement present then curly braces are optional.

7) Once we write lambda expression we can call that expression just like a method, for this functional interfaces are required.

Functional Interfaces

If an interface contain only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

1) Runnable → It contains only run() method
2) Comparable → It contains only compareTo() method
3) ActionListener → It contains only actionPerformed()
4) Callable → It contains only call() method

Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

Ex:

```
1)  interface Interf {
2)          public abstract void m1();
3)          default void m2() {
4)                  System.out.println ("hello");
5)          }
6)  }
```

In Java 8, Sun Micro System introduced @Functional Interface annotation to specify that the interface is Functional Interface.

Ex:

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

This code compiles without any compilation errors.

Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raise an error message that is called we will get compilation error.

Ex:

```
@Functional Interface {
        public void m1();
        public void m2();
}
```

This code gives compilation error.

Inside Functional Interface we have to take exactly only one abstract method.If we are not declaring that abstract method then compiler gives an error message.

Ex:

```
@Functional Interface {
        interface Interface {
}
```

compilation error

Functional Interface with respect to Inheritance:

If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface

Ex:

```
1)  @Functional Interface
2)  interface A {
3)          public void methodOne();
4)  }
5)  @Functional Interface
6)  Interface B extends A {
7)  }
```

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

Ex:

```
1)  @Functional Interface
2)  interface A {
3)          public void methodOne();
4)  }
5)  @Functional Interface
6)  interface B extends A {
7)          public void methodOne();
8)  }
```

No Compile Time Error

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.

```
1)  @Functional Interface {
2)  interface A {
3)        public void methodOne();
4)  }
5)  @Functional Interface
6)  interface B extends A {
7)        public void methodTwo();
8)  }
```

Compiletime Error

Ex:

```
@Functional Interface
interface A {
      public void methodOne();
}
interface B extends A {
       public void methodTwo();
}
```

No compile time error

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.

Functional Interface Vs Lambda Expressions:

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**

```
1)  interface  Interf {
2)        public void methodOne() {}
3)        public class Demo implements Interface {
4)            public void methodOne() {
5)                  System.out.println("method one execution");
6)            }
7)            public class Test {
8)                public static void main(String[] args) {
9)                      Interfi = new Demo();
10)                     i.methodOne();
11)               }
12) }
```

Above code With Lambda expression

```
1)  interface Interf {
2)        public void methodOne() {}
3)        class Test {
4)             public static void main(String[] args) {
5)                  Interfi = () → System.out.println("MethodOne Execution");
6)                  i.methodOne();
7)              }
8)  }
```

Without Lambda Expression

```
1)  interface Interf {
2)         public void sum(inta,int b);
3)  }
4)  class Demo implements Interf {
5)          public void sum(inta,int b) {
6)                System.out.println("The sum:"+(a+b));
7)          }
8)  }
9)  public class Test {
10)         public static void main(String[] args) {
11)                Interfi = new Demo();
12)                i.sum(20,5);
13)         }
14) }
```

Above code With Lambda Expression

```
1)  interface Interf {
2)       public void sum(inta, int b);
3)  }
4)  class Test {
5)        public static void main(String[] args) {
6)             Interfi = (a,b) → System.out.println("The Sum:" +(a+b));
7)             i.sum(5,10);
8)        }
9)  }
```

Without Lambda Expressions

```
1)  interface Interf {
2)          publicint square(int x);
3)  }
4)  class Demo implements Interf {
5)              public int square(int x) {
6)                      return x*x;  OR  (int x) → x*x
7)              }
8)  }
9)  class Test {
10)         public static void main(String[] args) {
11)             Interfi = new Demo();
12)             System.out.println("The Square of 7 is: " +i.square(7));
13)         }
14) }
```

Above code with Lambda Expression

```
1)  interface Interf {
2)          public int square(int x);
3)  }
4)  class Test {
5)          public static void main(String[] args) {
6)                  Interfi = x → x*x;
7)                  System.out.println("The Square of 5 is:"+i.square(5));
8)          }
9)  }
```

Without Lambda expression

```
1)  class MyRunnable implements Runnable {
2)              public void main() {
3)                      for(int i=0; i<10; i++) {
4)                              System.out.println("Child Thread");
5)                      }
6)              }
7)  }
8)  class ThreadDemo {
9)          public static void main(String[] args) {
10)                     Runnable r = new myRunnable();
11)                     Thread t = new Thread(r);
12)                     t.start();
13)                     for(int i=0; i<10; i++) {
14)                             System.out.println("Main Thread")
15)                     }
16)         }
```

```
17) }
```

With Lambda expression

```
1)  class ThreadDemo {
2)        public static void main(String[] args) {
3)              Runnable r = () → {
4)                    for(int i=0; i<10; i++) {
5)                          System.out.println("Child Thread");
6)                    }
7)              };
8)              Thread t = new Thread(r);
9)              t.start();
10)             for(i=0; i<10; i++) {
11)                   System.out.println("Main Thread");
12)             }
13)       }
14) }
```

Anonymous inner classes vs Lambda Expressions

Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.

Ex: With anonymous inner class

```
1)  class Test {
2)      public static void main(String[] args) {
3)          Thread t = new Thread(new Runnable() {
4)                  public void run() {
5)                      for(int i=0; i<10; i++) {
6)                          System.out.println("Child Thread");
7)                      }
8)                  }
9)          });
10)         t.start();
11)         for(int i=0; i<10; i++)
12)             System.out.println("Main thread");
13)     }
14) }
```

With Lambda expression

```
1)  class Test {
2)        public static void main(String[] args) {
3)                Thread t = new Thread(() → {
4)                                                for(int i=0; i<10; i++) {
5)                                                    System.out.println("Child Thread");
6)                                                }
7)                });
8)                t.start();
9)                for(int i=0; i<10; i++) {
10)                   System.out.println("Main Thread");
11)               }
12)       }
13) }
```

What are the advantages of Lambda expression?

- We can reduce length of the code so that readability of the code will be improved.
- We can resolve complexity of anonymous inner classes.
- We can provide Lambda expression in the place of object.
- We can pass lambda expression as argument to methods.

Note:

- Anonymous inner class can extend concrete class, can extend abstract class, can implement interface with any number of methods but
- Lambda expression can implement an interface with only single abstract method (Functional Interface).
- Hence if anonymous inner class implements Functional Interface in that particular case only we can replace with lambda expressions. Hence wherever anonymous inner class concept is there, it may not possible to replace with Lambda expressions.
- Anonymous inner class! = Lambda Expression

- Inside anonymous inner class we can declare instance variables.
- Inside anonymous inner class "this" always refers current inner class object(anonymous inner class) but not related outer class object

Ex:

- Inside lambda expression we can't declare instance variables.
- Whatever the variables declare inside lambda expression are simply acts as local variables
- Within lambda expression 'this" keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)

Ex:

```
1)  interface Interf {
2)          public void m1();
3)  }
4)  class Test {
5)          int x = 777;
6)          public void m2() {
7)                  Interfi = ()→ {
8)                          int x = 888;
9)                          System.out.println(x); 888
10)                         System.out.println(this.x); 777
11)                 };
12)                 i.m1();
13)         }
14)         public static void main(String[] args) {
15)                 Test t = new Test();
16)                 t.m2();
17)         }
18) }
```

- From lambda expression we can access enclosing class variables and enclosing method variables directly.
- The local variables referenced from lambda expression are implicitly final and hence we can't perform re-assignment for those local variables otherwise we get compile time error

Ex:

```
1)  interface Interf {
2)          public void m1();
3)  }
4)  class Test {
5)          int x = 10;
6)          public void m2() {
7)                  int y = 20;
8)                  Interfi = () → {
9)                          System.out.println(x); 10
10)                         System.out.println(y); 20
11)                         x = 888;
12)                         y = 999; //CE
13)                 };
14)                 i.m1();
15)                 y = 777;
16)         }
17)         public static void main(String[] args) {
18)                 Test t = new Test();
19)                 t.m2();
```

```
20)         }
21) }
```

Differences between anonymous inner classes and Lambda expression

Anonymous Inner class	Lambda Expression
It's a class without name	It's a method 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 implement 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 variables.	Inside lambda expression we can't Declare instance variables, whatever the variables declared are simply acts as local variables.
Anonymous inner classes can be Instantiated	lambda expressions can't 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. That is enclosing class object.
Anonymous inner class is the best choice If we want to handle multiple methods.	Lambda expression is the best Choice if we want to handle interface With single abstract method (Functional Interface).
In the case of anonymous inner class At the time of compilation a separate Dot class file will be generated (outerclass$1.class)	At the time of compilation no dot Class file will be generated for Lambda expression. It simply converts in to private method outer class.
Memory allocated on demand Whenever we are creating an object	Reside in permanent memory of JVM (Method Area).

Default Methods

- 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

```
1)  default void m1(){
2)  System.out.println ("Default Method");
3)  }
```

- Interface default methods are by-default available to all implementation classes. Based on requirement implementation class can use these default methods directly or can override.

Ex:

```
1)  interface Interf {
2)          default void m1() {
3)                  System.out.println("Default Method");
4)          }
5)  }
6)  class Test implements Interf {
7)          public static void main(String[] args) {
8)                  Test t = new Test();
9)                  t.m1();
10)         }
11) }
```

- Default methods also known as defender methods or virtual extension methods.
- The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

Note: We can't override object class methods as default methods inside interface otherwise we get compile time error.

Ex:

```
1)  interface Interf {
2)        default inthashCode() {
3)              return 10;
4)        }
5)  }
```

CompileTimeError

Reason: **Object class methods are by-default available to every Java class hence it's not required to bring through default methods.**

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.

```
1)  Eg 1:
2)  interface Left {
3)        default void m1() {
4)              System.out.println("Left Default Method");
5)        }
6)  }
7)
8)  Eg 2:
9)  interface Right {
10)       default void m1() {
11)             System.out.println("Right Default Method");
12)       }
13) }
14)
15) Eg 3:
16) class Test implements Left, Right {}
```

How to override default method in the implementation class?

In the implementation class we can provide complete new implementation or we can call any interface method as follows.
interfacename.super.m1();

Ex:

```
1)  class Test implements Left, Right {
2)          public void m1() {
3)                System.out.println("Test Class Method"); OR Left.super.m1();
4)          }
5)          public static void main(String[] args) {
6)                  Test t = new Test();
7)                  t.m1();
8)          }
9)  }
```

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.

Ex:

```
1)  interface Interf {
2)          public static void sum(int a, int b) {
3)                  System.out.println("The Sum:"+(a+b));
4)          }
5)  }
6)  class Test implements Interf {
7)           public static void main(String[] args) {
8)                  Test t = new Test();
9)                  t.sum(10, 20); //CE
10)                 Test.sum(10, 20); //CE
11)                 Interf.sum(10, 20);
12)         }
13) }
```

- As interface static methods by default not available to the implementation class, overriding concept is not applicable.
- Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

Ex:1

```
1)  interface Interf {
2)          public static void m1() {}
3)  }
4)  class Test implements Interf {
5)           public static void m1() {}
6)  }
```

It's valid but not overriding

Ex:2

```
1)  interface Interf {
2)        public static void m1() {}
3)  }
4)  class Test implements Interf {
5)        public void m1() {}
6)  }
```

This's valid but not overriding

Ex3:

```
1)  class P {
2)       private void m1() {}
3)  }
4)  class C extends P {
5)        public void m1() {}
6)  }
```

This's valid but not overriding

From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

Ex:

```
1)  interface Interf {
2)        public static void main(String[] args) {
3)              System.out.println("Interface Main Method");
4)        }
5)  }
```

At the command prompt:
Javac Interf.Java
JavaInterf

Predicates

- 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 method i.e., test()

Ex:

```
interface Predicate<T> {
    public 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);

predicate<Integer> p = I →(I >10);
System.out.println (p.test(100)); true
System.out.println (p.test(7)); false

Program:

```
1)  import Java.util.function;
2)  class Test {
3)        public static void main(String[] args) {
4)            predicate<Integer> p = I → (i>10);
5)            System.out.println(p.test(100));
6)            System.out.println(p.test(7));
7)            System.out.println(p.test(true)); //CE
8)        }
9)  }
```

1 Write a predicate to check the length of given string is greater than 3 or not.
Predicate<String> p = s → (s.length() > 3);
System.out.println (p.test("rvkb")); true
System.out.println (p.test("rk")); false

#-2 write a predicate to check whether the given collection is empty or not.
Predicate<collection> p = c → c.isEmpty();

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

Ex:

```
1)  import Java.util.function.*;
2)  class test {
3)        public static void main(string[] args) {
4)            int[] x = {0, 5, 10, 15, 20, 25, 30};
5)            predicate<integer> p1 = i->i>10;
6)            predicate<integer> p2=i -> i%2==0;
7)            System.out.println("The Numbers Greater Than 10:");
8)            m1(p1, x);
9)            System.out.println("The Even Numbers Are:");
```

```
10)             m1(p2, x);
11)             System.out.println("The Numbers Not Greater Than 10:");
12)             m1(p1.negate(), x);
13)             System.out.println("The  Numbers Greater Than 10 And Even Are:â€");
14)             m1(p1.and(p2), x);
15)             System.out.println("The Numbers Greater Than 10 OR Even:â€");
16)             m1(p1.or(p2), x);
17)         }
18)          public static void m1(predicate<integer>p, int[] x) {
19)                 for(int x1:x) {
20)                     if(p.test(x1))
21)                         System.out.println(x1);
22)                     }
23)                 }
24) }
```

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.**

Ex:

```
1)  import Java.util.function.*;
2)  class Test {
3)          public static void main(String[] args) {
4)                  Function<String, Integer> f = s ->s.length();
5)                  System.out.println(f.apply("Durga"));
6)                  System.out.println(f.apply("Soft"));
7)          }
8)  }
```

Note: **Function is a functional interface and hence it can refer lambda expression.**

Differences between predicate and function

Predicate	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<T>	Function can take 2 type Parameters. First one represent Input argument type and Second one represent return Type. Function<T,R>
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

Note: Predicate is a boolean valued function and(), or(), negate() are default methods present inside Predicate interface.

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: With Lambda Expression

```
1)  class Test {
2)      public static void main(String[] args) {
3)              Runnable r = () → {
4)                                  for(int i=0; i<=10; i++) {
5)                                                  System.out.println("Child Thread");
6)                                  }
7)                                 };
8)              Thread t = new Thread(r);
9)              t.start();
10)             for(int i=0; i<=10; i++) {
11)                     System.out.println("Main Thread");
12)             }
13)     }
14) }
```

With Method Reference

```
1)  class Test {
2)      public static void m1() {
3)          for(int i=0; i<=10; i++) {
4)              System.out.println("Child Thread");
5)          }
```

```
6)   }
7)   public static void main(String[] args) {
8)              Runnable r = Test:: m1;
9)              Thread t = new Thread(r);
10)             t.start();
11)             for(int i=0; i<=10; i++) {
12)                   System.out.println("Main Thread");
13)             }
14) }
```

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

Ex:

```
1)   interface Interf {
2)         public void m1(int i);
3)    }
4)   class Test {
5)          public void m2(int i) {
6)                System.out.println("From Method Reference:"+i);
7)          }
8)          public static void main(String[] args) {
9)               Interf  f = I ->sop("From Lambda Expression:"+i);
10)              f.m1(10);
11)              Test t = new Test();
12)              Interf i1 = t::m2;
13)              i1.m1(20);
14)        }
15) }
```

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).

Constructor References

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

Syntax: classname :: new

Ex:

Interf f = sample :: new;
functional interface f referring sample class constructor

Ex:

```
1)  class Sample {
2)          private String s;
3)          Sample(String s) {
4)                  this.s = s;
5)                  System.out.println("Constructor Executed:"+s);
6)           }
7)  }
8)  interface Interf {
9)          public Sample get(String s);
10) }
11) class Test {
12)         public static void main(String[] args) {
13)                 Interf f = s -> new Sample(s);
14)                 f.get("From Lambda Expression");
15)                 Interf f1 = Sample :: new;
16)                 f1.get("From Constructor Reference");
17)         }
18) }
```

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

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:

We can configure a filter to filter elements from the collection based on some boolean condition 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:

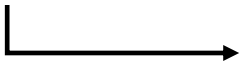
```
Stream s = c.stream();
Stream s1 = s.filter(i -> i%2==0);
```

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 our requirement then we should go for map() method of Stream interface.

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



It can be lambda expression also

Ex:

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

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

2) 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

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

```
1)  import Java.util.*;
2)  class Test {
3)        public static void main(String[] args) {
4)              ArrayList<Integer> l1 = new ArrayList<Integer>();
5)              for(int i=0; i<=10; i++) {
6)                  l1.add(i);
7)              }
8)              System.out.println(l1);
9)              ArrayList<Integer> l2 = new ArrayList<Integer>();
10)             for(Integer i:l1) {
11)                   if(i%2 == 0)
12)                       l2.add(i);
13)             }
14)             System.out.println(l2);
15)       }
16) }
```

Approach-2: With Streams

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test {
4)        public static void main(String[] args) {
5)              ArrayList<Integer> l1 = new ArrayList<Integer>();
6)              for(inti=0; i<=10; i++) {
7)                    l1.add(i);
8)              }
9)              System.out.println(l1);
10)             List<Integer> l2 = l1.stream().filter(i -> i%2==0).collect(Collectors.toList());
11)             System.out.println(l2);
12)       }
13) }
```

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

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test {
4)        public static void main(String[] args) {
5)                ArrayList<String> l = new ArrayList<String>();
6)                l.add("rvk"); l.add("rk"); l.add("rkv"); l.add("rvki"); l.add("rvkir");
7)                System.out.println(l);
8)                List<String> l2 = l.Stream().map(s ->s.toUpperCase()).collect(Collectors.toList());
9)                System.out.println(l2);
10)       }
11) }
```

II.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);
```

III.Processing by sorted()method

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

IV.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);
```

V.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:

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test1 {
4)          public static void main(String[] args) {
5)                  ArrayList<Integer> l1 = new ArrayaList<Integer>();
6)                  l1.add(0); l1.add(15); l1.add(10); l1.add(5); l1.add(30); l1.add(25); l1.add(20);
7)                  System.out.println(l1);
8)                  ArrayList<Integer> l2=l1.stream().map(i-> i+10).collect(Collectors.toList());
9)                  System.out.println(l2);
10)                 long count = l1.stream().filter(i->i%2==0).count();
11)                 System.out.println(count);
12)                 List<Integer> l3=l1.stream().sorted().collect(Collectors.toList());
13)                 System.out.println(l3);
14)                 Comparator<Integer> comp=(i1,i2)->i1.compareTo(i2);
15)                 List<Integer> l4=l1.stream().sorted(comp).collect(Collectors.toList());
16)                 System.out.println(l4);
17)                 Integer min=l1.stream().min(comp).get();
18)                 System.out.println(min);
19)                 Integer max=l1.stream().max(comp).get();
20)                 System.out.println(max);
```

```
21)                 l3.stream().forEach(i->sop(i));
22)                 l3.stream().forEach(System.out:: println);
23)
24)         }
25) }
```

VI.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);
}
```

VII.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);
```

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.

```
1)  import Java.time.*;
2)  public class DateTime {
3)     public static void main(String[] args) {
4)        LocalDate date = LocalDate.now();
5)        System.out.println(date);
6)        LocalTime time=LocalTime.now();
7)        System.out.println(time);
8)     }
9)  }
```

O/p:
2015-11-23
12:39:26:587

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

Ex:

```
1)  import Java.time.*;
2)  class Test {
3)        public static void main(String[] args) {
4)                LocalDate date = LocalDate.now();
5)                System.out.println(date);
6)                int dd = date.getDayOfMonth();
7)                int mm = date.getMonthValue();
8)                int yy = date.getYear();
9)                System.out.println(dd+"..."+mm+"..."+yy);
10)               System.out.printf("\n%d-%d-%d",dd,mm,yy);
11)   }
12) }
```

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

Ex:

```
1)  importJava.time.*;
2)  class Test {
3)     public static void main(String[] args)  {
4)        LocalTime time = LocalTime.now();
5)        int h = time.getHour();
6)        int m = time.getMinute();
7)        int s = time.getSecond();
8)        int n = time.getNano();
9)        System.out.printf("\n%d:%d:%d:%d",h,m,s,n);
10)    }
11) }
```

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

```
LocalDateTimedt = LocalDateTime.now();
System.out.println(dt);
```

O/p: 2015-11-23T12:57:24.531

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));
```

To Represent Zone:

ZoneId object can be used to represent Zone.

Ex:

```
1)  import Java.time.*;
2)  class ProgramOne {
3)        public static void main(String[] args) {
4)              ZoneId zone = ZoneId.systemDefault();
5)              System.out.println(zone);
6)        }
7)  }
```

We can create ZoneId for a particular zone as follows

Ex:

```
ZoneId la = ZoneId.of("America/Los_Angeles");
ZonedDateTimezt = ZonedDateTime.now(la);
System.out.println(zt);
```

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());
```

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

```
1)  import Java.time.*;
2)  public class Leapyear {
3)     int n = Integer.parseInt(args[0]);
4)     Year y = Year.of(n);
5)     if(y.isLeap())
6)           System.out.printf("%d is Leap year",n);
7)     else
8)           System.out.printf("%d is not Leap year",n);
9)  }
```

Lambda Expressions with Collections

Collection is nothing but a group of objects represented as a single entity.
The important Collection types are:

1. List(I)
2. Set(I)
3. Map(I)

1. 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 shoud 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:

```
1)  import java.util.ArrayList;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        ArrayList<String> l = new ArrayList<String>();
7)        l.add("Sunny");
8)        l.add("Bunny");
9)        l.add("Chinny");
10)       l.add("Sunny");
11)       System.out.println(l);
12)    }
13) }
```

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

2. Set(I):

If we want to represent a group of individual 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. Duplicates 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:

```
1)  import java.util.HashSet;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        HashSet<String> l = new HashSet<String>();
7)        l.add("Sunny");
8)        l.add("Bunny");
9)        l.add("Chinny");
10)       l.add("Sunny");
11)       System.out.println(l);
12)    }
13) }
```

Output: **[Chinny, Bunny, Sunny]**

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

3. Map(I):

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

Eg:
Rollno-->Name
mobilenumber-->address

The important implementation classes of Map are:
1. HashMap 2. TreeMap etc

Demo Program for Map:

```
1)  import java.util.HashMap;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        HashMap<String,String> m= new HashMap<String,String>();
7)        m.put("A","Apple");
8)        m.put("Z","Zebra");
9)        m.put("Durga","Java");
10)       m.put("B","Boy");
11)       m.put("T","Tiger");
12)       System.out.println(m);
13)    }
14) }
```

Output: {A=Apple, B=Boy, T=Tiger, Z=Zebra, Durga=Java}

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

Demo Program to Sort elements of ArrayList according to Defaut Natural Sorting Order(Ascending Order):

```
1)  import java.util.ArrayList;
2)  import java.util.Collections;
3)  class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        ArrayList<Integer> l = new ArrayList<Integer>();
8)        l.add(10);
9)        l.add(0);
10)       l.add(15);
11)       l.add(5);
12)       l.add(20);
13)       System.out.println("Before Sorting:"+l);
14)       Collections.sort(l);
15)       System.out.println("After Sorting:"+l);
16)    }
17) }
```

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):

```
1)  import java.util.TreeSet;
2)  import java.util.Comparator;
3)  class MyComparator implements Comparator<Integer>
4)  {
5)     public int compare(Integer I1,Integer I2)
6)     {
7)        if(I1<I2)
8)        {
9)           return +1;
10)       }
11)       else if(I1>I2)
12)       {
13)          return -1;
14)       }
15)       else
16)       {
17)          return 0;
18)       }
19)    }
20) }
21) class Test
```

```
22) {
23)    public static void main(String[] args)
24)    {
25)       TreeSet<Integer> l = new TreeSet<Integer>(new MyComparator());
26)       l.add(10);
27)       l.add(0);
28)       l.add(15);
29)       l.add(5);
30)       l.add(20);
31)       System.out.println(l);
32)    }
33) }
```

//Descending order Comparator

Output: [20, 15, 10, 5, 0]

Shortcut way:

```
1)  import java.util.ArrayList;
2)  import java.util.Comparator;
3)  import java.util.Collections;
4)  class MyComparator implements Comparator<Integer>
5)  {
6)     public int compare(Integer I1,Integer I2)
7)     {
8)        return (I1>I2)?-1:(I1<I2)?1:0;
9)     }
10) }
11) class Test
12) {
13)    public static void main(String[] args)
14)    {
15)       ArrayList<Integer> l = new ArrayList<Integer>();
16)       l.add(10);
17)       l.add(0);
18)       l.add(15);
19)       l.add(5);
20)       l.add(20);
21)       System.out.println("Before Sorting:"+l);
22)       Collections.sort(l,new MyComparator());
23)       System.out.println("After Sorting:"+l);
24)    }
25) }
```

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):

```
1)  import java.util.ArrayList;
2)  import java.util.Collections;
3)  class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        ArrayList<Integer> l= new ArrayList<Integer>();
8)        l.add(10);
9)        l.add(0);
10)       l.add(15);
11)       l.add(5);
12)       l.add(20);
13)       System.out.println("Before Sorting:"+l);
14)       Collections.sort(l,(I1,I2)->(I1<I2)?1:(I1>I2)?-1:0);
15)       System.out.println("After Sorting:"+l);
16)    }
17) }
```

Output:

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

```
1)  import java.util.TreeSet;
2)  class Test
3)  {
4)     public static void main(String[] args)
```

```
5)    {
6)       TreeSet<Integer> t = new TreeSet<Integer>();
7)       t.add(10);
8)       t.add(0);
9)       t.add(15);
10)      t.add(5);
11)      t.add(20);
12)      System.out.println(t);
13)   }
14) }
```

Output: [0, 5, 10, 15, 20]

Demo Program for Customized Sorting Order(Descending Order):

```
1)  import java.util.TreeSet;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        TreeSet<Integer> t = new TreeSet<Integer>((I1,I2)->(I1>I2)?-1:(I1<I2)?1:0);
7)        t.add(10);
8)        t.add(0);
9)        t.add(15);
10)       t.add(25);
11)       t.add(5);
12)       t.add(20);
13)       System.out.println(t);
14)    }
15) }
```

Output: [25, 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):

```
1)  import java.util.TreeMap;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        TreeMap<Integer,String> m = new TreeMap<Integer,String>();
7)        m.put(100,"Durga");
8)        m.put(600,"Sunny");
9)        m.put(300,"Bunny");
10)       m.put(200,"Chinny");
11)       m.put(700,"Vinny");
12)       m.put(400,"Pinny");
13)       System.out.println(m);
14)    }
15) }
```

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

Demo Program for Customized Sorting Order(Descending Order):

```
1)  import java.util.TreeMap;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        TreeMap<Integer,String> m = new TreeMap<Integer,String>((I1,I2)->(I1<I2)?1:(I1>I2)?-
    1:0);
7)        m.put(100,"Durga");
8)        m.put(600,"Sunny");
9)        m.put(300,"Bunny");
10)       m.put(200,"Chinny");
11)       m.put(700,"Vinny");
12)       m.put(400,"Pinny");
13)       System.out.println(m);
14)    }
15) }
```

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

Sorting for Customized class objects by using Lambda Expressions:

```
1)  import java.util.ArrayList;
2)  import java.util.Collections;
3)  class Employee
4)  {
5)     int eno;
6)     String ename;
```

```
7)     Employee(int eno,String ename)
8)     {
9)        this.eno=eno;
10)       this.ename=ename;
11)    }
12)    public String toString()
13)    {
14)       return eno+":"+ename;
15)    }
16) }
17) class Test
18) {
19)    public static void main(String[] args)
20)    {
21)       ArrayList<Employee> l= new ArrayList<Employee>();
22)       l.add(new Employee(100,"Katrina"));
23)       l.add(new Employee(600,"Kareena"));
24)       l.add(new Employee(200,"Deepika"));
25)       l.add(new Employee(400,"Sunny"));
26)       l.add(new Employee(500,"Alia"));
27)       l.add(new Employee(300,"Mallika"));
28)       System.out.println("Before Sorting:");
29)       System.out.println(l);
30)       Collections.sort(l,(e1,e2)->(e1.eno<e2.eno)?-1:(e1.eno>e2.eno)?1:0);
31)       System.out.println("After Sorting:");
32)       System.out.println(l);
33)    }
34) }
```

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]
```

Predefined Functional Interface

Predicate

Study Material

Predicates

- 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 method i.e., test()

Ex:

```
interface Predicate<T> {
    public 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);
```

```
predicate<Integer> p = I →(I >10);
System.out.println (p.test(100)); true
System.out.println (p.test(7)); false
```

Program:

```
1)  import Java.util.function;
2)  class Test {
3)        public static void main(String[] args) {
4)            predicate<Integer> p = I → (i>10);
5)            System.out.println(p.test(100));
6)            System.out.println(p.test(7));
7)            System.out.println(p.test(true)); //CE
8)        }
9)  }
```

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

Predicate<String> p = s → (s.length() > 3);
System.out.println (p.test("rvkb")); true
System.out.println (p.test("rk")); false

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

Predicate<collection> p = c → c.isEmpty();

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

Ex:

```
1)  import Java.util.function.*;
2)  class test {
3)        public static void main(string[] args) {
4)            int[] x = {0, 5, 10, 15, 20, 25, 30};
5)            predicate<integer> p1 = i->i>10;
6)            predicate<integer> p2=i -> i%2==0;
7)            System.out.println("The Numbers Greater Than 10:");
8)            m1(p1, x);
9)            System.out.println("The Even Numbers Are:");
10)           m1(p2, x);
11)           System.out.println("The Numbers Not Greater Than 10:");
12)           m1(p1.negate(), x);
13)           System.out.println("The  Numbers Greater Than 10 And Even Are:â€⍰);
14)           m1(p1.and(p2), x);
15)           System.out.println("The Numbers Greater Than 10 OR Even:â€⍰);
16)           m1(p1.or(p2), x);
17)       }
```

```
18)     public static void m1(predicate<integer>p, int[] x) {
19)         for(int x1:x) {
20)             if(p.test(x1))
21)                 System.out.println(x1);
22)             }
23)         }
24) }
```

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

```
1)  import java.util.function.Predicate;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        String[] names={"Sunny","Kajal","Mallika","Katrina","Kareena"};
7)        Predicate<String> startsWithK=s->s.charAt(0)=='K';
8)        System.out.println("The Names starts with K are:");
9)        for(String s: names)
10)       {
11)          if(startsWithK.test(s))
12)          {
13)             System.out.println(s);
14)          }
15)       }
16)    }
17) }
```

Output:

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

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

```
1)  import java.util.ArrayList;
2)  import java.util.function.Predicate;
3)  class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        String[] names={"Durga","",null,"Ravi","","Shiva",null};
8)        Predicate<String> p=s-> s != null && s.length()!=0;
9)        ArrayList<String> list=new ArrayList<String>();
10)       for(String s : names)
11)       {
12)          if(p.test(s))
13)          {
14)             list.add(s);
15)          }
16)       }
17)       System.out.println("The List of valid Names:");
```

```
18)       System.out.println(list);
19)    }
20) }
```

Output:

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

Program for User Authentication by using Predicate:

```
1)  import java.util.function.Predicate;
2)  import java.util.Scanner;
3)  class User
4)  {
5)     String username;
6)     String pwd;
7)     User(String username,String pwd)
8)     {
9)        this.username=username;
10)       this.pwd=pwd;
11)    }
12) }
13) class Test
14) {
15)    public static void main(String[] args)
16)    {
17)       Predicate<User> p = u->u.username.equals("durga")&& u.pwd.equals("java");
18)       Scanner sc= new Scanner(System.in);
19)       System.out.println("Enter User Name:");
20)       String username=sc.next();
21)       System.out.println("Enter Password:");
22)       String pwd=sc.next();
23)       User user=new User(username,pwd);
24)       if(p.test(user))
25)       {
26)          System.out.println("Valid user and can avail all services");
27)       }
28)       else
29)       {
30)          System.out.println("invalid user you cannot avail services");
31)       }
32)    }
33) }
```

```
D:\durgaclasses>java Test
Enter User Name:
durga
```

```
Enter Password:
java
Valid user and can avail all services

D:\durgaclasses>java Test
Enter User Name:
ravi
Enter Password:
java
invalid user you cannot avail services
```

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

```
1)  import java.util.function.Predicate;
2)  class SoftwareEngineer
3)  {
4)     String name;
5)     int age;
6)     boolean isHavingGf;
7)     SoftwareEngineer(String name,int age,boolean isHavingGf)
8)     {
9)        this.name=name;
10)       this.age=age;
11)       this.isHavingGf=isHavingGf;
12)    }
13)    public String toString()
14)    {
15)       return name;
16)    }
17) }
18) class Test
19) {
20)    public static void main(String[] args)
21)    {
22)       SoftwareEngineer[] list={ new SoftwareEngineer("Durga",60,false),
23)                                 new SoftwareEngineer("Sunil",25,true),
24)                                 new SoftwareEngineer("Sayan",26,true),
25)                                 new SoftwareEngineer("Subbu",28,false),
26)                                 new SoftwareEngineer("Ravi",17,true)
27)                                 };
28)       Predicate<SoftwareEngineer> allowed= se -> se.age>= 18 && se.isHavingGf;
29)       System.out.println("The Allowed Members into Pub are:");
30)       for(SoftwareEngineer se : list)
31)       {
32)          if(allowed.test(se))
33)          {
```

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

Output:

```
The allowed members into Pub are:
Sunil
Sayan
```

Employee Management Application:

```
import java.util.function.Predicate;
import java.util.ArrayList;
class Employee
{
   String name;
   String designation;
   double salary;
   String city;
   Employee(String name,String designation,double salary,String city)
   {
      this.name=name;
      this.designation=designation;
      this.salary=salary;
      this.city=city;
   }
   public String toString()
   {
      String s=String.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;
      }
   }
}
```

```
34) class Test
35) {
36)    public static void main(String[] args)
37)    {
38)       ArrayList<Employee> list= new ArrayList<Employee>();
39)       populate(list);
40)
41)       Predicate<Employee> p1=emp->emp.designation.equals("Manager");
42)       System.out.println("Managers Information:");
43)       display(p1,list);
44)
45)       Predicate<Employee> p2=emp->emp.city.equals("Bangalore");
46)       System.out.println("Bangalore Employees Information:");
47)       display(p2,list);
48)
49)       Predicate<Employee> p3=emp->emp.salary<20000;
50)       System.out.println("Employees whose slaray <20000 To Give Increment:");
51)       display(p3,list);
52)
53)       System.out.println("All Managers from Bangalore city for Pink Slip:");
54)       display(p1.and(p2),list);
55)
56)       System.out.println("Employees Information who are either Managers or salary <2000
    0");
57)       display(p1.or(p3),list);
58)
59)       System.out.println("All Employees Information who are not managers:");
60)       display(p1.negate(),list);
61)
62)       Predicate<Employee> isCEO=Predicate.isEqual(new Employee("Durga","CEO",30000,"
    Hyderabad"));
63)
64)       Employee e1=new Employee("Durga","CEO",30000,"Hyderabad");
65)       Employee e2=new Employee("Sunny","Manager",20000,"Hyderabad");
66)       System.out.println(isCEO.test(e1));//true
67)       System.out.println(isCEO.test(e2));//false
68)
69)    }
70)    public static void populate(ArrayList<Employee> list)
71)    {
72)       list.add(new Employee("Durga","CEO",30000,"Hyderabad"));
73)       list.add(new Employee("Sunny","Manager",20000,"Hyderabad"));
74)       list.add(new Employee("Mallika","Manager",20000,"Bangalore"));
75)       list.add(new Employee("Kareena","Lead",15000,"Hyderabad"));
76)       list.add(new Employee("Katrina","Lead",15000,"Bangalore"));
77)       list.add(new Employee("Anushka","Developer",10000,"Hyderabad"));
78)       list.add(new Employee("Kanushka","Developer",10000,"Hyderabad"));
79)       list.add(new Employee("Sowmya","Developer",10000,"Bangalore"));
```

```
80)      list.add(new Employee("Ramya","Developer",10000,"Bangalore"));
81)   }
82)   public static void display(Predicate<Employee> p,ArrayList<Employee> list)
83)   {
84)      for (Employee e: list )
85)      {
86)         if(p.test(e))
87)         {
88)            System.out.println(e);
89)         }
90)      }
91)      System.out.println("*************************************************");
92)   }
93) }
```

Output:

```
Managers Information:
[Sunny,Manager,20000.00,Hyderabad]
[Mallika,Manager,20000.00,Bangalore]
************************************************
Bangalore Employees Information:
[Mallika,Manager,20000.00,Bangalore]
[Katrina,Lead,15000.00,Bangalore]
[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]
************************************************
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:
[Durga,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
```

Predicate Practice Bits

Q1. Which of the following abstract method present in Predicate interface?

A. test()
B. apply()
C. get()
D. accept()

Answer: A

Explanation: **Predicate functional interface contains only one abstract method: test()**

Q2. Which of the following is the static method present in Predicate interface?

A. test()
B. and()
C. or()
D. isEqual()

Answer: D
Explanation: **Predicate functional interface contains only one static method: isEqual()**

Q3. Which of the following default methods present in Predicate interface?

A. and()
B. or()
C. negate()
D. All the above

Answer: D

Explanation: **Predicate Functional interface contains the following 3 default methods: and(),or(),not()**

Q4. Which of the following is Predicate interface declaration?

A.

```
1)  interface Predicate<T>
2)  {
3)     public boolean test(T t);
4)  }
```

B.

```
1)  interface Predicate<T>
2)  {
3)     public boolean apply(T t);
4)  }
```

C.

```
1)  interface Predicate<T,R>
2)  {
3)     public R test(T t);
4)  }
```

D.

```
1)  interface Predicate<T,R>
2)  {
3)     public R apply(T t);
4)  }
```

Answer: A

Explanation:

```
1)  interface Predicate<T>
2)  {
3)     public boolean test(T t);
4)  }
```

Predicate interface can take only one Type parameter which represents only input type. We are not required to specify return type because return type is always boolean type.

Q5. Which of the following is valid Predicate to check whether the given Integer is divisible by 10 or not?

A. Predicate<Integer> p = i -> i%10 == 10;
B. Predicate<Integer,Boolean> p =i->i%10==0;
C. Predicate<Boolean,Integer> p =i->i%10==0;
D. None of the above

Answer: A

Explanation:

```
1)  interface Predicate<T>
2)  {
3)     public boolean test(T t);
4)  }
```

Predicate interface can take only one Type parameter which represents only input type. We are not required to specify return type because return type is always boolean type.

Q6. Which of the following is valid regarding Predicate functional interface?

A. Predicate Functional interface present in java.util.function package
B. It is introduced in java 1.8 version
C. We can use Predicate to implement conditional checks
D. It is possible to join 2 predicates into a single predicate also.
E. All the above

Answer: E

Q7. Which of the following is valid Predicate to check whether the given user is admin or not?

A. Predicate<User> p=user->user.getRole().equals("Admin");
B. Predicate<Boolean> p=user->user.getRole().equals("Admin");
C. Predicate<User> p=(user,s="admin")->user.getRole().equals(s);
D. None of the above

Answer: A

Explanation:

```
1)  interface Predicate<T>
2)  {
3)     public boolean test(T t);
4)  }
```

Predicate interface can take only one Type parameter which represents only input type. We are not required to specify return type because return type is always boolean type.

Q8. Consider the following Predicates

Predicate<Integer> p1=i->i%2==0;
Predicate<Integer> p1=i->i>10;

Which of the following are invalid ?

A. p1.and(p2)
B. p1.or(p2)
C. p1.negate(p2)
D. p1.negate()

Answer: C

Explanation: **negate() method won't take any argument**

Predefined Functional Interface

Predicate

Study Material

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()

```
1)  interface function(T,R) {
2)          public R apply(T t);
3)  }
```

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

Ex:

```
1)  import Java.util.function.*;
2)  class Test {
3)          public static void main(String[] args) {
4)                  Function<String, Integer> f = s ->s.length();
5)                  System.out.println(f.apply("Durga"));
6)                  System.out.println(f.apply("Soft"));
7)          }
8)  }
```

Note: Function is a functional interface and hence it can refer lambda expression.

Differences between predicate and function

Predicate	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<T>	Function can take 2 type Parameters. First one represent Input argument type and Second one represent return Type. Function<T,R>
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

Note: Predicate is a boolean valued function and(), or(), negate() are default methods present inside Predicate interface.

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        String s="durga  software  solutions hyderabad";
7)        Function<String,String> f= s1->s1.replaceAll(" ","");
8)        System.out.println(f.apply(s));
9)     }
10) }
```

Output: durgasoftwaresolutionshyderabad

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        String s="durga  software  solutions  hyderabad ";
7)        Function<String,Integer> f= s1->s1.length() - s1.replaceAll(" ","").length();
8)        System.out.println(f.apply(s));
9)     }
10) }
```

Output: 3

Program to find Student Grade by using Function:

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Student
4)  {
5)     String name;
6)     int marks;
7)     Student(String name,int marks)
8)     {
9)        this.name=name;
10)       this.marks=marks;
11)    }
12) }
```

```
13) class Test
14) {
15)    public static void main(String[] args)
16)    {
17)       ArrayList<Student> l= new ArrayList<Student>();
18)       populate(l);
19)       Function<Student,String> f=s->{
20)          int marks=s.marks;
21)          if(marks>=80)
22)          {
23)             return "A[Dictinction]";
24)          }
25)          else if(marks>=60)
26)          {
27)             return "B[First Class]";
28)          }
29)          else if(marks>=50)
30)          {
31)             return "C[Second Class]";
32)          }
33)          else if(marks>=35)
34)          {
35)             return "D[Third Class]";
36)          }
37)          else
38)          {
39)             return "E[Failed]";
40)          }
41)       };
42)       for(Student s : l)
43)       {
44)          System.out.println("Student Name:"+s.name);
45)          System.out.println("Student Marks:"+s.marks);
46)          System.out.println("Student Grade:"+f.apply(s));
47)          System.out.println();
48)       }
49)    }
50)    public static void populate(ArrayList<Student> l)
51)    {
52)       l.add(new Student("Sunny",100));
53)       l.add(new Student("Bunny",65));
54)       l.add(new Student("Chinny",55));
55)       l.add(new Student("Vinny",45));
56)       l.add(new Student("Pinny",25));
57)    }
58) }
```

Output:

```
D:\durgaclasses>java Test
Student Name:Sunny
Student Marks:100
Student Grade:A[Dictinction]

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

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

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

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

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

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Student
4)  {
5)     String name;
6)     int marks;
7)     Student(String name,int marks)
8)     {
9)        this.name=name;
10)       this.marks=marks;
11)    }
12) }
13) class Test
14) {
15)    public static void main(String[] args)
16)    {
17)       ArrayList<Student> l= new ArrayList<Student>();
18)       populate(l);
19)       Function<Student,String> f=s->{
20)          int marks=s.marks;
21)          if(marks>=80)
```

```
22)         {
23)            return "A[Dictinction]";
24)         }
25)         else if(marks>=60)
26)         {
27)            return "B[First Class]";
28)         }
29)         else if(marks>=50)
30)         {
31)            return "C[Second Class]";
32)         }
33)         else if(marks>=35)
34)         {
35)            return "D[Third Class]";
36)         }
37)         else
38)         {
39)            return "E[Failed]";
40)         }
41)      };
42)      Predicate<Student> p=s->s.marks>=60;
43)
44)      for(Student s : l)
45)      {
46)         if(p.test(s))
47)         {
48)            System.out.println("Student Name:"+s.name);
49)            System.out.println("Student Marks:"+s.marks);
50)            System.out.println("Student Grade:"+f.apply(s));
51)            System.out.println();
52)         }
53)      }
54)   }
55)   public static void populate(ArrayList<Student> l)
56)   {
57)      l.add(new Student("Sunny",100));
58)      l.add(new Student("Bunny",65));
59)      l.add(new Student("Chinny",55));
60)      l.add(new Student("Vinny",45));
61)      l.add(new Student("Pinny",25));
62)   }
63) }
```

Output:

```
D:\durgaclasses>java Test

Student Name:Sunny
```

Student Marks:100
Student Grade:A[Dictinction]

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

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

```
1)  import java.util.*;
2)  import java.util.function.*;
3)  class Employee
4)  {
5)     String name;
6)     double salary;
7)     Employee(String name,double salary)
8)     {
9)        this.name=name;
10)       this.salary=salary;
11)    }
12)    public String toString()
13)    {
14)       return name+":"+salary;
15)    }
16) }
17) class Test
18) {
19)    public static void main(String[] args)
20)    {
21)       ArrayList<Employee> l= new ArrayList<Employee>();
22)       populate(l);
23)       System.out.println(l);
24)       Function<ArrayList<Employee>,Double> f= l1 ->{
25)          double total=0;
26)          for(Employee e: l1)
27)          {
28)             total=total+e.salary;
29)          }
30)          return total;
31)       };
32)       System.out.println("The total salary of this month:"+f.apply(l));
33)    }
34)
35)    public static void populate(ArrayList<Employee> l)
36)    {
37)       l.add(new Employee("Sunny",1000));
```

```
38)       l.add(new Employee("Bunny",2000));
39)       l.add(new Employee("Chinny",3000));
40)       l.add(new Employee("Pinny",4000));
41)       l.add(new Employee("Vinny",5000));
42)    }
43) }
```

Output:

```
D:\durgaclasses>java Test
[Sunny:1000.0, Bunny:2000.0, Chinny:3000.0, Pinny:4000.0, Vinny:5000.0]
The total salary of this month:15000.0
```

Progarm to perform Salary Increment for Employees by using Predicate & Function:

```
1)  import java.util.*;
2)  import java.util.function.*;
3)  class Employee
4)  {
5)     String name;
6)     double salary;
7)     Employee(String name,double salary)
8)     {
9)        this.name=name;
10)       this.salary=salary;
11)    }
12)    public String toString()
13)    {
14)       return name+":"+salary;
15)    }
16) }
17) class Test
18) {
19)    public static void main(String[] args)
20)    {
21)       ArrayList<Employee> l= new ArrayList<Employee>();
22)       populate(l);
23)
24)       System.out.println("Before Increment:");
25)       System.out.println(l);
26)
27)       Predicate<Employee> p=e->e.salary<3500;
28)       Function<Employee,Employee> f=e->{
29)          e.salary=e.salary+477;
30)          return e;
31)       };
32)
```

```
33)     System.out.println("After Increment:");
34)     ArrayList<Employee> l2= new ArrayList<Employee>();
35)     for(Employee e: l)
36)     {
37)        if(p.test(e))
38)        {
39)           l2.add(f.apply(e));
40)        }
41)     }
42)     System.out.println(l);
43)     System.out.println("Employees with  incremented salary:");
44)     System.out.println(l2);
45)  }
46)  public static void populate(ArrayList<Employee> l)
47)  {
48)     l.add(new Employee("Sunny",1000));
49)     l.add(new Employee("Bunny",2000));
50)     l.add(new Employee("Chinny",3000));
51)     l.add(new Employee("Pinny",4000));
52)     l.add(new Employee("Vinny",5000));
53)     l.add(new Employee("Durga",10000));
54)  }
55) }
```

Output:

Before Increment:
[Sunny:1000.0, Bunny:2000.0, Chinny:3000.0, Pinny:4000.0, Vinny:5000.0, Durga:10000.0]
After Increment:
[Sunny:1477.0, Bunny:2477.0, Chinny:3477.0, Pinny:4000.0, Vinny:5000.0, Durga: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.

Demo Program-1 for Function Chaining:

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)
7)        Function<String,String> f1=s->s.toUpperCase();
8)        Function<String,String> f2= s->s.substring(0,9);
9)
10)       System.out.println("The Result of f1:"+f1.apply("AishwaryaAbhi"));
11)       System.out.println("The Result of f2:"+f2.apply("AishwaryaAbhi"));
12)       System.out.println("The Result of f1.andThen(f2):"+f1.andThen(f2).apply("Aishwarya
    Abhi"));
13)       System.out.println("The Result of f1.compose(f2):"+f1.compose(f2).apply("Aishwarya
    Abhi"));
14)    }
15) }
```

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
```

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        Function<Integer,Integer> f1= i->i+i;
7)        Function<Integer,Integer> f2= i->i*i*i;
8)        System.out.println(f1.andThen(f2).apply(2));
```

```
        System.out.println(f1.compose(f2).apply(2));
    }
}
```

Output:

```
64
16
```

Demo Program for Function Chaining:

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

      Function<String,String> f1=s->s.toLowerCase();
      Function<String,String> f2= s->s.substring(0,5);

      Scanner sc = new Scanner(System.in);
      System.out.println("Enter User Name:");
      String username=sc.next();

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

      if(f1.andThen(f2).apply(username).equals("durga") && pwd.equals("java"))
      {
         System.out.println("Valid User");
      }
      else
      {
         System.out.println("Invalid User");
      }
   }
}
```

Output:

```
D:\durgaclasses>java Test
Enter User Name:
durga
Enter Password:
java
Valid User

D:\durgaclasses>java Test
```

Enter User Name:
durgasoftwaresolutions
Enter Password:
java
Valid User

D:\durgaclasses>java Test
Enter User Name:
DURGATECHNOLOGIES
Enter Password:
java
Valid User

D:\durgaclasses>java Test
Enter User Name:
javajava
Enter Password:
java
Invalid User

D:\durgaclasses>java Test
Enter User Name:
durga
Enter Password:
Java
Invalid 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.

Eg:

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        Function<String,String> f1= Function.identity();
7)        String s2= f1.apply("durga");
8)        System.out.println(s2);
9)     }
10) }
```

Output: durga

Predefined Functional Interface

Consumer

Study Material

Consumer (I)

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.

```
1)  interface Consumer<T>
2)  {
3)    public void accept(T t);
4)  }
```

Demo Program-1 for Consumer:

```
1)  import java.util.function.Consumer;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        Consumer<String> c=s->System.out.println(s);
7)        c.accept("Hello");
8)        c.accept("DURGASOFT");
9)     }
10) }
```

Output:

```
Hello
DURGASOFT
```

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

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Movie
4)  {
5)     String name;
6)     String hero;
7)     String heroine;
8)     Movie(String name,String hero,String heroine)
```

```
9)     {
10)       this.name=name;
11)       this.hero=hero;
12)       this.heroine=heroine;
13)    }
14) }
15) class Test
16) {
17)    public static void main(String[] args)
18)    {
19)       ArrayList<Movie> l= new ArrayList<Movie>();
20)       populate(l);
21)       Consumer<Movie> c= m->{
22)          System.out.println("Movie Name:"+m.name);
23)          System.out.println("Movie Hero:"+m.hero);
24)          System.out.println("Movie Heroine:"+m.heroine);
25)          System.out.println();
26)       };
27)       for(Movie m : l)
28)       {
29)          c.accept(m);
30)       }
31)
32)    }
33)    public static void populate(ArrayList<Movie> l)
34)    {
35)       l.add(new Movie("Bahubali","Prabhas","Anushka"));
36)       l.add(new Movie("Rayees","Sharukh","Sunny"));
37)       l.add(new Movie("Dangal","Ameer","Ritu"));
38)       l.add(new Movie("Sultan","Salman","Anushka"));
39)    }
40)
41) }
```

```
D:\durgaclasses>java Tes
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

Demo Program-3 for Predicate, Function & Consumer:

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Student
4)  {
5)     String name;
6)     int marks;
7)     Student(String name,int marks)
8)     {
9)        this.name=name;
10)       this.marks=marks;
11)    }
12) }
13) class Test
14) {
15)    public static void main(String[] args)
16)    {
17)       ArrayList<Student> l= new ArrayList<Student>();
18)       populate(l);
19)       Predicate<Student> p= s->s.marks>=60;
20)       Function<Student,String> f=s->{
21)          int marks=s.marks;
22)          if(marks>=80)
23)          {
24)             return "A[Dictinction]";
25)          }
26)          else if(marks>=60)
27)          {
28)             return "B[First Class]";
29)          }
30)          else if(marks>=50)
31)          {
32)             return "C[Second Class]";
33)          }
34)          else if(marks>=35)
35)          {
36)             return "D[Third Class]";
37)          }
38)          else
39)          {
40)             return "E[Failed]";
41)          }
```

```
42)       };
43)       Consumer<Student> c=s->{
44)          System.out.println("Student Name:"+s.name);
45)          System.out.println("Student Marks:"+s.marks);
46)          System.out.println("Student Grade:"+f.apply(s));
47)          System.out.println();
48)       };
49)       for(Student s : l)
50)       {
51)          if(p.test(s))
52)          {
53)             c.accept(s);
54)          }
55)       }
56)    }
57)    public static void populate(ArrayList<Student> l)
58)    {
59)       l.add(new Student("Sunny",100));
60)       l.add(new Student("Bunny",65));
61)       l.add(new Student("Chinny",55));
62)       l.add(new Student("Vinny",45));
63)       l.add(new Student("Pinny",25));
64)    }
65) }
```

Output:

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

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.

Demo Program for Consumer Chaining:

```
1)  import java.util.function.*;
2)  class Movie
3)  {
4)     String name;
5)     String result;
6)     Movie(String name,String result)
7)     {
8)        this.name=name;
9)        this.result=result;
10)    }
11) }
12) class Test
13) {
14)    public static void main(String[] args)
15)    {
16)       Consumer<Movie> c1=m-
    >System.out.println("Movie:"+m.name+" is ready to release");
17)
18)       Consumer<Movie> c2=m-
    >System.out.println("Movie:"+m.name+" is just Released and it is:"+m.result);
19)
20)       Consumer<Movie> c3=m-
    >System.out.println("Movie:"+m.name+" information storing in the database");
21)
22)       Consumer<Movie> chainedC = c1.andThen(c2).andThen(c3);
23)
24)       Movie m1= new Movie("Bahubali","Hit");
25)       chainedC.accept(m1);
26)
27)       Movie m2= new Movie("Spider","Flop");
28)       chainedC.accept(m2);
29)    }
30) }
```

Output:

```
Movie:Bahubali is ready to release
Movie:Bahubali is just Released and it is:Hit
Movie:Bahubali information storing in the database
Movie:Spider is ready to release
Movie:Spider is just Released and it is:Flop
Movie:Spider information storing in the database
```

Predefined Functional Interface

Supplier

Study Material

Supplier (I)

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().

```
1)  interface Supplier<R>
2)  {
3)    public R get();
4)  }
```

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

Demo Program-1 For Supplier to generate Random Name:

```
1)  import java.util.function.Supplier;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        Supplier<String> s =()->{
7)           String[] s1={"Sunny","Bunny","Chinny","Pinny"};
8)           int x =(int)(Math.random()*4);
9)           return s1[x];
10)       };
11)       System.out.println(s.get());
12)       System.out.println(s.get());
13)       System.out.println(s.get());
14)    }
15) }
```

Output:

```
D:\durgaclasses>java Test
Chinny
Bunny
Pinny

D:\durgaclasses>java Test
Bunny
Pinny
Bunny
```

Demo Program-2 For Supplier to supply System Date:

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        Supplier<Date> s=()->new Date();
8)        System.out.println(s.get());
9)        System.out.println(s.get());
10)       System.out.println(s.get());
11)    }
12) }
```

```
D:\durgaclasses>java Test
Sun Aug 05 21:34:06 IST 2018
Sun Aug 05 21:34:06 IST 2018
Sun Aug 05 21:34:06 IST 2018
```

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

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        Supplier<String> otps=()->{
8)           String otp="";
9)           for(int i =1;i<=6;i++)
10)          {
11)             otp=otp+(int)(Math.random()*10);
12)          }
13)          return otp;
```

```
14)       };
15)       System.out.println(otps.get());
16)       System.out.println(otps.get());
17)       System.out.println(otps.get());
18)       System.out.println(otps.get());
19)    }
20) }
```

Output:

```
492499
055284
531389
925530
```

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,@,#,$

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        Supplier<String> s=()->
8)        {
9)           String symbols="ABCDEFGHIJKLMNOPQRSTUVWXYZ#$@";
10)          Supplier<Integer> d=()->(int)(Math.random()*10);
11)          Supplier<Character> c=()->symbols.charAt((int)(Math.random()*29));
12)          String pwd="";
13)          for(int i =1;i<=8;i++)
14)          {
15)             if(i%2==0)
16)             {
17)                pwd=pwd+d.get();
18)             }
19)             else
20)             {
21)                pwd=pwd+c.get();
22)             }
23)          }
24)          return pwd;
25)       };
26)       System.out.println(s.get());
```

```
27)       System.out.println(s.get());
28)       System.out.println(s.get());
29)       System.out.println(s.get());
30)       System.out.println(s.get());
31)    }
32) }
```

Output:

```
G2W3Y8W5
W7M8$0L3
T5T5N2F3
O9N2L0V2
A4I1$1P6
```

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 <T> { :::::::::::::: }	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	and(), or(), negate()	andThen(), compose()	andThen()	-
5) Static Method	isEqual()	identify()	-	-

Two-Argument (Bi) Functional Interfaces

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

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.

```
1)  interface BiPredicate<T1,T2>
2)  {
3)     public boolean test(T1 t1,T2 t2);
4)     //remaining default methods: and(), or() , negate()
5)  }
```

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        BiPredicate<Integer,Integer> p=(a,b)->(a+b) %2==0;
7)        System.out.println(p.test(10,20));
8)        System.out.println(p.test(15,20));
9)     }
10) }
```

Output:
true
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.

```
1)  interface BiFunction<T,U,R>
2)  {
3)     public R apply(T t,U u);
4)     //default method andThen()
5)   }
```

To find product of 2 given integers by using BiFunction:

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        BiFunction<Integer,Integer,Integer> f=(a,b)->a*b;
7)        System.out.println(f.apply(10,20));
8)        System.out.println(f.apply(100,200));
9)     }
10) }
```

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

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Student
4)  {
5)     String name;
6)     int rollno;
7)     Student(String name,int rollno)
8)     {
9)        this.name=name;
10)       this.rollno=rollno;
11)    }
12) }
13) class Test
14) {
```

```
15)   public static void main(String[] args)
16)   {
17)      ArrayList<Student> l = new ArrayList<Student>();
18)      BiFunction<String,Integer,Student> f=(name,rollno)->new Student(name,rollno);
19)
20)      l.add(f.apply("Durga",100));
21)      l.add(f.apply("Ravi",200));
22)      l.add(f.apply("Shiva",300));
23)      l.add(f.apply("Pavan",400));
24)      for(Student s : l)
25)      {
26)         System.out.println("Student Name:"+s.name);
27)         System.out.println("Student Rollno:"+s.rollno);
28)         System.out.println();
29)      }
30)   }
31) }
```

Output:

```
Student Name:Durga
Student Rollno:100

Student Name:Ravi
Student Rollno:200

Student Name:Shiva
Student Rollno:300

Student Name:Pavan
Student Rollno:400
```

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

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Employee
4)  {
5)     int eno;
6)     String name;
7)     double dailyWage;
8)     Employee(int eno,String name,double dailyWage)
9)     {
10)       this.eno=eno;
11)       this.name=name;
12)       this.dailyWage=dailyWage;
13)    }
```

```
14) }
15) class TimeSheet
16) {
17)    int eno;
18)    int days;
19)    TimeSheet(int eno,int days)
20)    {
21)       this.eno=eno;
22)       this.days=days;
23)    }
24) }
25) class Test
26) {
27)    public static void main(String[] args)
28)    {
29)       BiFunction<Employee,TimeSheet,Double> f=(e,t)->e.dailyWage*t.days;
30)       Employee e= new Employee(101,"Durga",1500);
31)       TimeSheet t= new TimeSheet(101,25);
32)       System.out.println("Employee Monthly Salary:"+f.apply(e,t));
33)    }
34) }
```

Output: Employee 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.

```
1)  interface BiConsumer<T,U>
2)  {
3)     public void accept(T t,U u);
4)     //default method andThen()
5)  }
```

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        BiConsumer<String,String> c=(s1,s2)->System.out.println(s1+s2);
7)        c.accept("durga","soft");
8)     }
9)  }
```

Output: durgasoft

Demo Program to increment employee Salary by using BiConsumer:

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Employee
4)  {
5)     String name;
6)     double salary;
7)     Employee(String name,double salary)
8)     {
9)        this.name=name;
10)       this.salary=salary;
11)    }
12) }
13) class Test
14) {
15)    public static void main(String[] args)
16)    {
17)       ArrayList<Employee> l= new ArrayList<Employee>();
18)       populate(l);
19)       BiConsumer<Employee,Double> c=(e,d)->e.salary=e.salary+d;
20)       for(Employee e:l)
21)       {
22)          c.accept(e,500.0);
23)       }
24)       for(Employee e:l)
25)       {
26)          System.out.println("Employee Name:"+e.name);
27)          System.out.println("Employee Salary:"+e.salary);
28)          System.out.println();
29)       }
30)
```

```
31)    }
32)    public static void populate(ArrayList<Employee> l)
33)    {
34)       l.add(new Employee("Durga",1000));
35)       l.add(new Employee("Sunny",2000));
36)       l.add(new Employee("Bunny",3000));
37)       l.add(new Employee("Chinny",4000));
38)    }
39) }
```

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()
	}

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

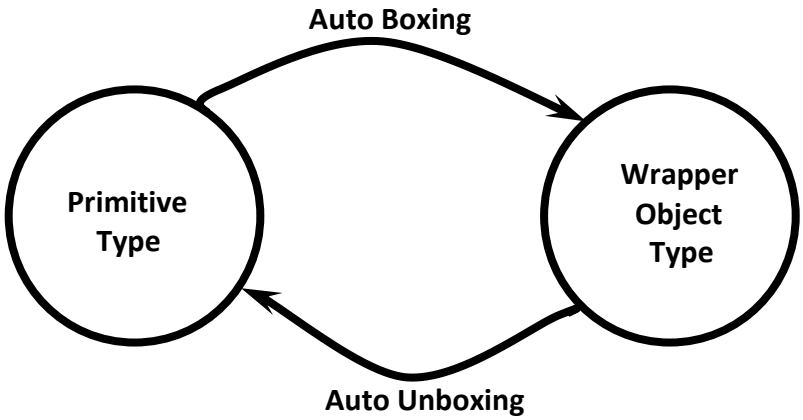
What is the Need of Primitive Type Functional Interfaces?

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. 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.

```
1)  import java.util.function.Predicate;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        int[] x ={0,5,10,15,20,25};
```

```
7)        Predicate<Integer> p=i->i%2==0;
8)        for (int x1 : x)
9)        {
10)         if(p.test(x1))
11)         {
12)           System.out.println(x1);
13)         }
14)       }
15)    }
16) }
```

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

1.

```
1)  interface IntPredicate
2)  {
3)    public boolean test(int i);
4)    //default methods: and(),or(),negate()
5)  }
```

2.

```
1)  interface LongPredicate
2)  {
3)    public boolean test(long l);
4)    //default methods: and(),or(),negate()
5)  }
```

3.

```
1)  interface DoublePredicate
2)  {
3)    public boolean test(double d);
4)    //default methods: and(),or(),negate()
5)  }
```

Demo Program for IntPredicate:

```
1)  import java.util.function.IntPredicate;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        int[] x ={0,5,10,15,20,25};
7)        IntPredicate p=i->i%2==0;
8)        for (int x1 : x)
9)        {
10)         if(p.test(x1))
11)         {
12)           System.out.println(x1);
13)         }
14)       }
15)    }
16) }
```

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: can take int type as input and return any type
public R apply(int i);

2. LongFunction: can take long type as input and return any type
public R apply(long i);

3. DoubleFunction: can take double type as input and return any type
public R apply(double d);

4. ToIntFunction: It can take any type as input but always returns int type
public int applyAsInt(T t)

5. ToLongFunction: It can take any type as input but always returns long type
public long applyAsLong(T t)

6. ToDoubleFunction: 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:return type must be int type but inputs can be anytype
public int applyAsInt(T t, U u)

14. ToLongBiFunction:return type must be long type but inputs can be anytype
public long applyAsLong(T t, U u)

15. ToDoubleBiFunction: return type must be double type but inputs can be anytype
public double applyAsDouble(T t, U u)

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        IntFunction<Integer,Integer> f=i->i*i;
7)        System.out.println(f.apply(4));
8)     }
9)  }
```

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        IntFunction<Integer> f=i->i*i;
7)        System.out.println(f.apply(5));
8)     }
9)  }
```

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        Function<String,Integer> f=s->s.length();
7)        System.out.println(f.apply("durga"));
8)     }
9)  }
```

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        ToIntFunction<String> f=s->s.length();
7)        System.out.println(f.applyAsInt("durga"));
8)     }
9)  }
```

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        Function<Integer,Double> f=i->Math.sqrt(i);
7)        System.out.println(f.apply(7));
8)     }
9)  }
```

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        IntToDoubleFunction f=i->Math.sqrt(i);
7)        System.out.println(f.applyAsDouble(9));
8)     }
9)  }
```

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)

Demo Program for IntConsumer:

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        IntConsumer c= i->System.out.println("The Square of i:"+(i*i));
7)        c.accept(10);
8)     }
9)  }
```

Output: The Square of i:100

Demo Program to increment employee Salary by using ObjDoubleConsumer:

```
1)  import java.util.function.*;
2)  import java.util.*;
3)  class Employee
4)  {
5)     String name;
6)     double salary;
7)     Employee(String name,double salary)
8)     {
9)        this.name=name;
```

```
10)       this.salary=salary;
11)    }
12) }
13) class Test
14) {
15)    public static void main(String[] args)
16)    {
17)       ArrayList<Employee> l= new ArrayList<Employee>();
18)       populate(l);
19)       ObjDoubleConsumer<Employee> c=(e,d)->e.salary=e.salary+d;
20)       for(Employee e:l)
21)       {
22)          c.accept(e,500.0);
23)       }
24)       for(Employee e:l)
25)       {
26)          System.out.println("Employee Name:"+e.name);
27)          System.out.println("Employee Salary:"+e.salary);
28)          System.out.println();
29)       }
30)
31)    }
32)    public static void populate(ArrayList<Employee> l)
33)    {
34)       l.add(new Employee("Durga",1000));
35)       l.add(new Employee("Sunny",2000));
36)       l.add(new Employee("Bunny",3000));
37)       l.add(new Employee("Chinny",4000));
38)    }
39) }
```

Output:

```
Employee Name:Durga
Employee Salary:1500.0

Employee Name:Sunny
Employee Salary:2500.0

Employee Name:Bunny
Employee Salary:3500.0

Employee Name:Chinny
Employee Salary:4500.0
```

Primitive Versions for Supplier:

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()

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

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        IntSupplier s=()->(int)(Math.random()*10);
7)        String otp="";
8)        for(int i =0;i<6;i++)
9)        {
10)          otp=otp+s.getAsInt();
11)       }
12)       System.out.println("The 6 digit OTP: "+otp);
13)    }
14) }
```

Output: The 6 digit OTP: 035716

UnaryOperator<T>:

- If input and output are same type then we should go for UnaryOperator
- It is child of Function<T,T>

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
```

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

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        UnaryOperator<Integer> f=i->i*i;
        System.out.println(f.apply(6));
    }
}
```

The primitive versions for UnaryOperator:

IntUnaryOperator:
public int applyAsInt(int)

LongUnaryOperator:
public long applyAsLong(long)

DoubleUnaryOperator:
public double applyAsDouble(double)

Demo Program-1 for IntUnaryOperator:

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        IntUnaryOperator f=i->i*i;
        System.out.println(f.applyAsInt(6));
    }
}
```

Demo Program-2 for IntUnaryOperator:

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

```
6)        IntUnaryOperator f1=i->i+1;
7)        System.out.println(f1.applyAsInt(4));
8)
9)        IntUnaryOperator f2=i->i*i;
10)       System.out.println(f2.applyAsInt(4));
11)
12)       System.out.println(f1.andThen(f2).applyAsInt(4));
13)
14)    }
15) }
```

Output:

```
5
16
25
```

BinaryOperator:

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

BinaryOperator<T>
public T apply(T,T)

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        BiFunction<String,String,String> f=(s1,s2)->s1+s2;
7)        System.out.println(f.apply("durga","software"));
8)     }
9)  }
```

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        BinaryOperator<String> b=(s1,s2)->s1+s2;
7)        System.out.println(b.apply("durga","software"));
8)     }
9)  }
```

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)

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        BinaryOperator<Integer> b=(i1,i2)->i1+i2;
7)        System.out.println(b.apply(10,20));
8)     }
9)  }
```

```
1)  import java.util.function.*;
2)  class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        IntBinaryOperator b=(i1,i2)->i1+i2;
7)        System.out.println(b.applyAsInt(10,20));
8)     }
9)  }
```