

To being the benefits of Functional Programming lambda Expression came to Java

It is a anonymous function (name less)

Without return type

Without modifiers

Ex : 1

Public void m1(){ system.out.println(“Hello “);}

~~Public void m1~~()->{ system.out.println(“Hello “);}

()->{ system.out.println(“Hello “);}

If the Body contains only one line , then remove the Curly Braces

()-> system.out.println(“Hello “);

Ex 2 :

Public void m1( int a , int b) {system.out.println(a+b);}

~~Public void m1~~( int a , int b) {system.out.println(a+b);}

( int a , int b) -> system.out.println(a+b);

Sometimes if the compiler can Pick the data Type Automatically so we can remove that

(a,b) -> system.out.println(a+b);

Ex-3

Public int SquareIt(int n){ return n\*n ;}// if you want to have the return key word , then compulsorily we need Curly braces

(int n) -> n\*n // if it’s only one line , we can remove the Braces and also we can remove the rerun keyword as the compiler can assume the return keyword automatically

(~~int~~ n) -> n\*n // if the compiler can guse the type automatically we can remove the in

~~(~~n~~)~~ > n\*n // if there is only one param , then we can remove the () as well

n->n\*n

public void m1(String s) { return s.lenght();}

s->s.lemgth();

Now, How to call this Methods?

Ans: FI => Functional Interfaces

What is the Functional Interfaces?

An Inter faces that contain only one static Abstract classes is FI

(ex)

Runnable => run()

Comparable => compareTo()

ActionalListener => actionPerformed();

Callable => call()

All these are Interfafces also FI (SAM)

If you want to invoke the Lambda Expression , then we need FI

Comparator .Compare (@FunctionalInterface)

To Sort based on the ascendingOrder of number( we can implement to customize )

Collection.sort( al,(e1,e2)->(e1.eno<e2.eno)?-1: (e1.eno>e2.eno)?1: 0);

Camparable.compareTo(@FunctionalInterface)

Default Natural sorting order is internally implemented in Camparable.compareTo ( Alphabetical order of String)

Collection.sort(al,(e1,e2)->e1.name.compareTo(e2.name));

**Define Trhead ()**

Class MyRunnable implements Runnable {

Public void run (){

Child Thread;

}

}

In Main

//Normal

MyRunnable r = new MyRunnable();

Thread t = new Thread(r);

r.start();

//Lambda

Thread t = new Thread( ()->sysout(“ChildThread”) );

r.start();

**Anonymous Inner Class**

The Class without Name , Can Extends a Class or Implements Interface

Thread t = new Thread(); // Normal

InnerClass

Thread t = new Thread {

-----

-----

};

This means , there is a Anonymus class that Extends Thread Class ( this Anonymous Class is the Child of Thread Class ,and Child Reference is created in Thread t object)

Class MyRunnable implements Runnable {

Public void run (){

For ( int i=0; i<10 ; i++ ) {

Sysout(“Child Thread”);

}

}

}

Other way

Runnable r1 = ()->{

**for** ( **int** i=0;i<10;i++) {

System.***out***.println("Child-1 Thread") ;

}

};

Thread t1 = **new** Thread(r1);

t1.start();

**Interface :**

Variable : Every variable inside the Interface is Public Static Final

Methods : 1.7 - Every Method Present inside the Interface is Public and abstract

Public abstract void m1();

1.8 – **Default** and **static** methods are also allowed

1.9 – Private are also allowed

**Default** Methods / Virtual Extension Method /Defender Method

Method with Default Implementation

Without affecting the Implementation classes if we want to add a method to Interface , then we can go for default Method

**Static** - If All the Methods are Static , then it’s better Go for Interface ,

so no dealing with object , which is a Costly operation

For Interface we are dealing with runtime ( reference ) as it was static method ,

To Define a General Utility Method inside Interface with Static

Sum ( int a , int b)

**PreDefined Functional interface :**

**Predicate**

**Function**

**Consumer**

**Supplier**

**In out Code where ever the condition check is required , Predicated Function with lambda can be used**

**Java.utils.functioin.\***

**Predicate<T> = Return Boolean**

**Predicate<Integer> p = I->I%2==0;**

**Sysout(p.test());// true /false**

**Predicate Chaining**

**P1.and(p2).test(z);**

**P1.or(p2).test(z);**

**P1.negate.test**

**Function @FunctionalInterface**

**If the Return type of the lambda Expression is other then Boolean , then we can Function**

**Function< T , R > = return R**

**Function < Integer,Integer > f = a->a\*a ;**

sysout**(f.apply(5));**

**Function Chaining**

**F1.andThen(f2).apply(5);// First f1 will Execute and that output is given as input to f2**

**F1.compose(f2).apply(5);// First f2 will Execute and that output is given as input to f1**

**Consumer @FunctionalInterface**

Consumer goes to consumes to the value

Consumer<T> = void

Consumer <Integer> c = n->sysout(n\*n);

c.accept(5);

Consumer Chaining

Consumer<Integer> cc = c1.andthen(c2).andThen(c3);

Supplier<> = Returns <R>

Supplier does not take any Params but only return the values

Supplier<Date> dt = ()-> new Date();

Sysout(Dt.get());

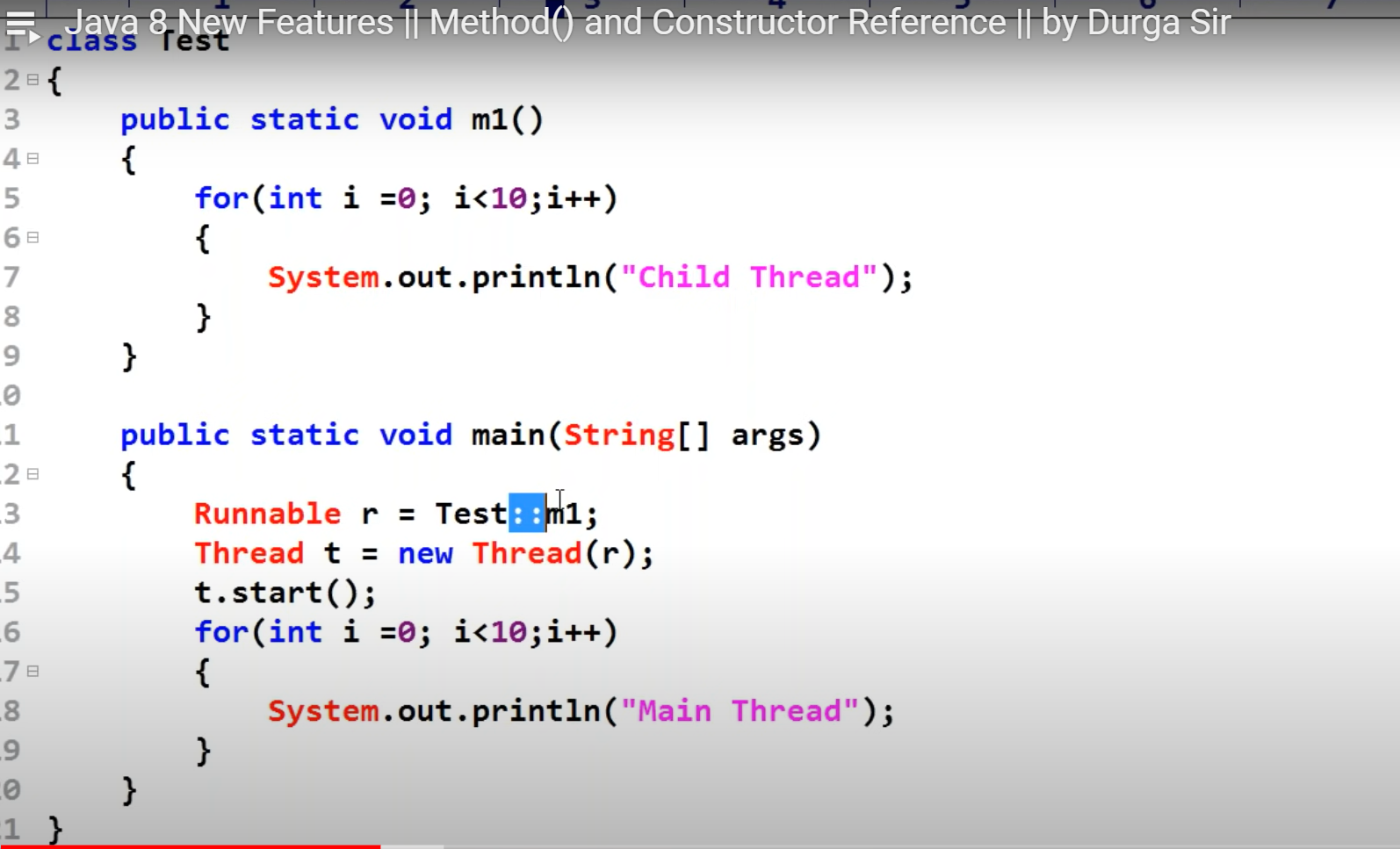
Supplier<String> otp = ()-> { “ Generate OTP and return” };

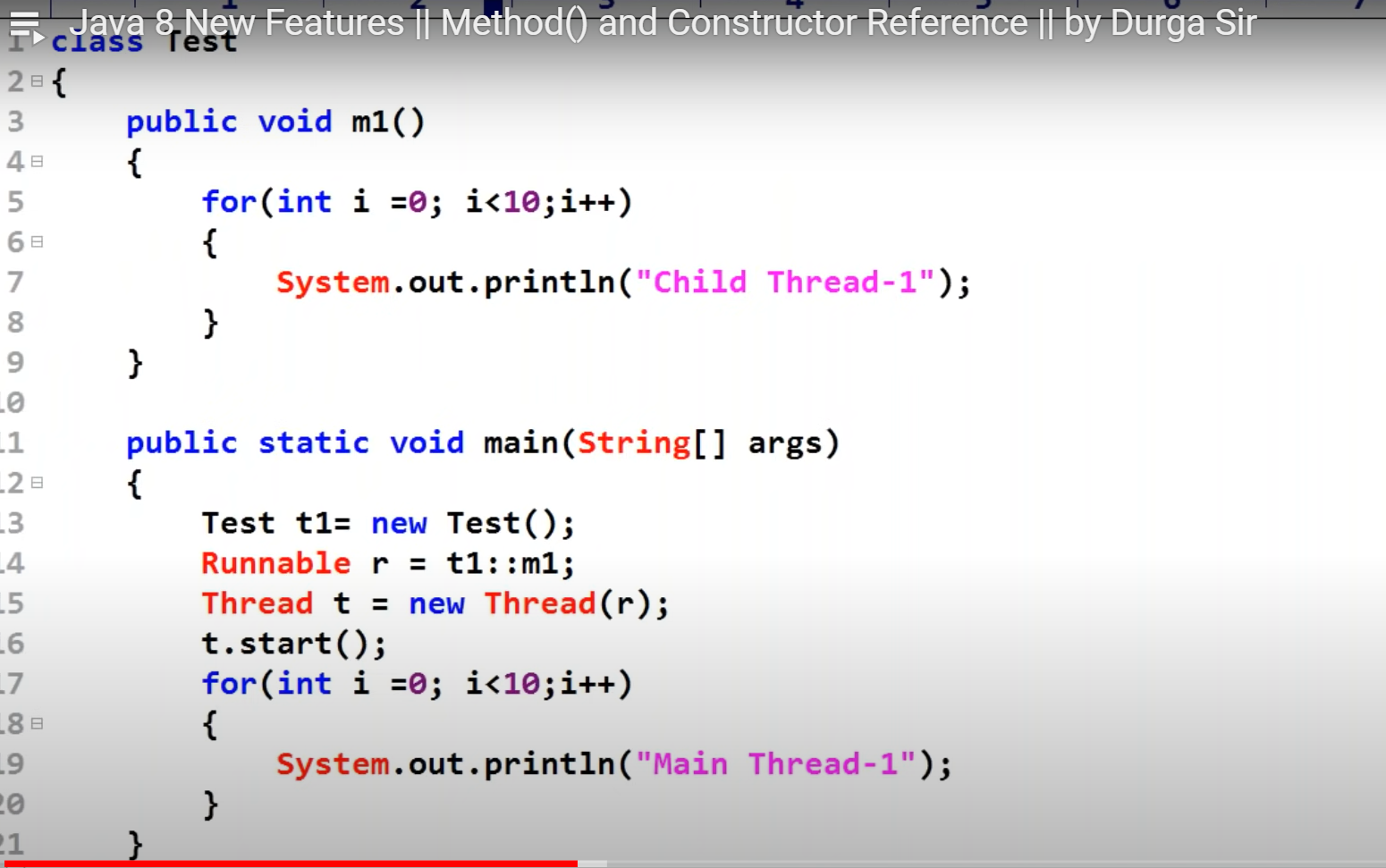
Sysout(otp.get());

BiPredicate – Accept the two Argument as input

**Method Reference Alternate to Lambda Expression**

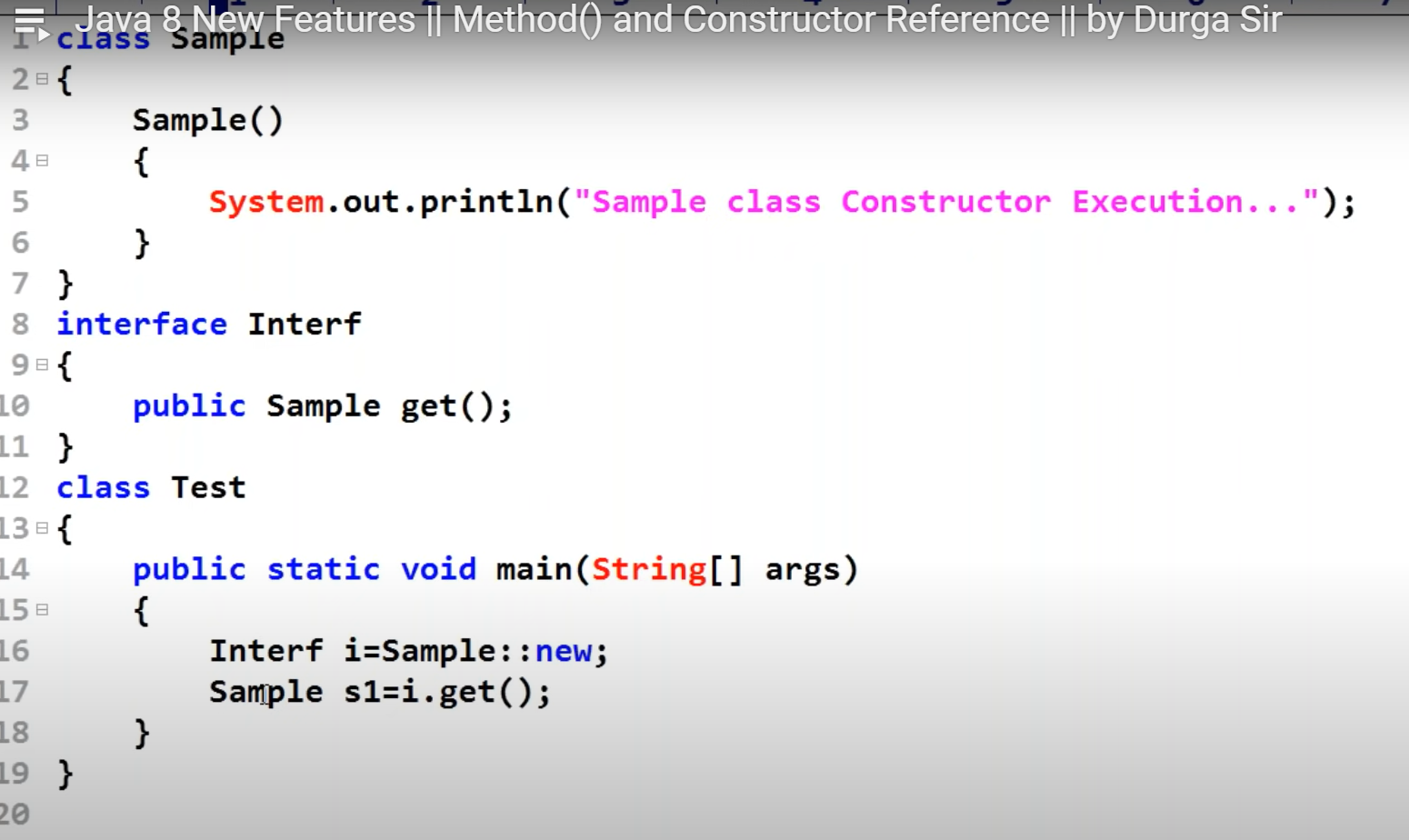
If the Implementation is already available better we can go for method reference

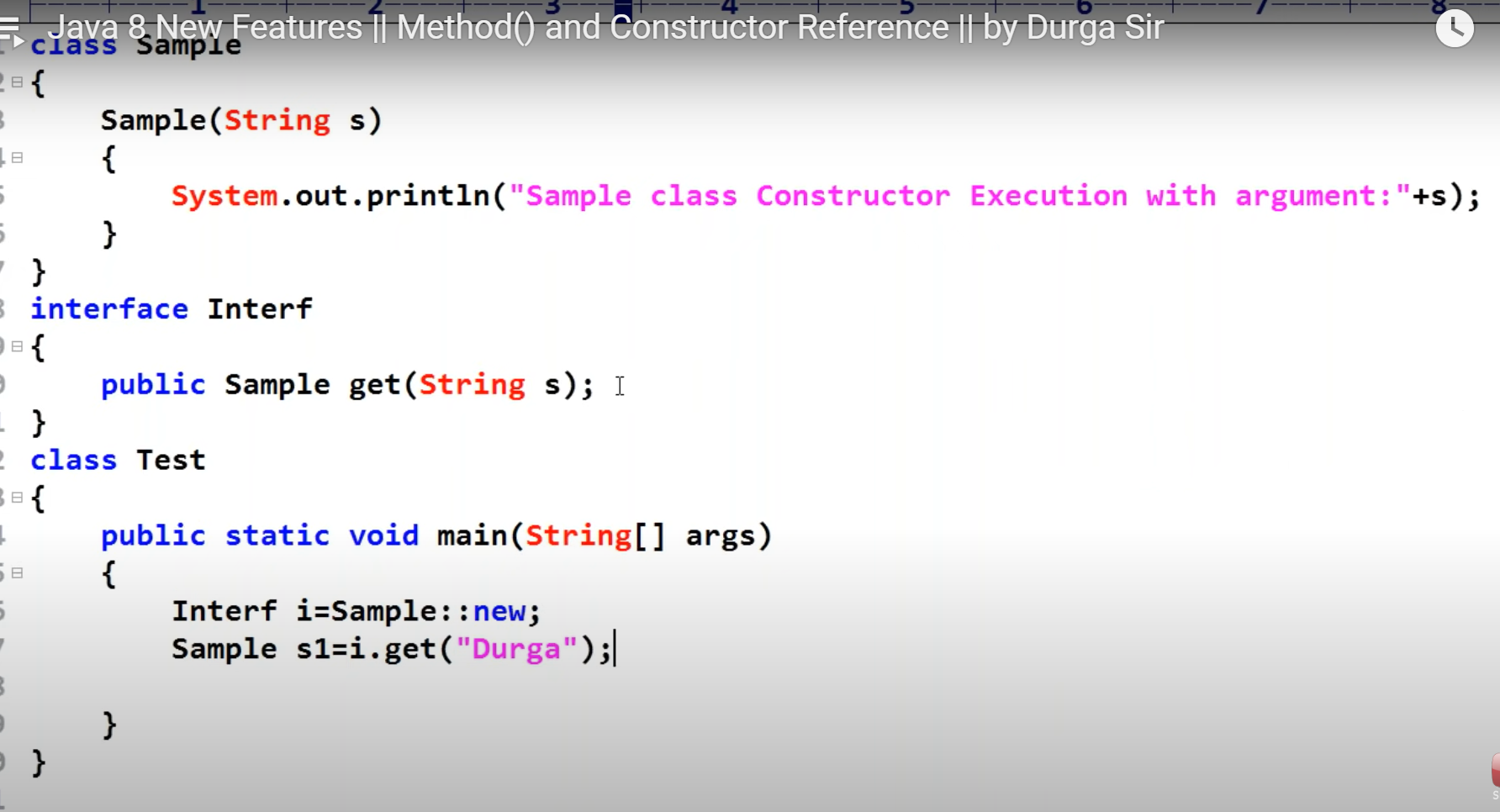




**Constructor Reference**

Test::new 🡺 Construction Reference





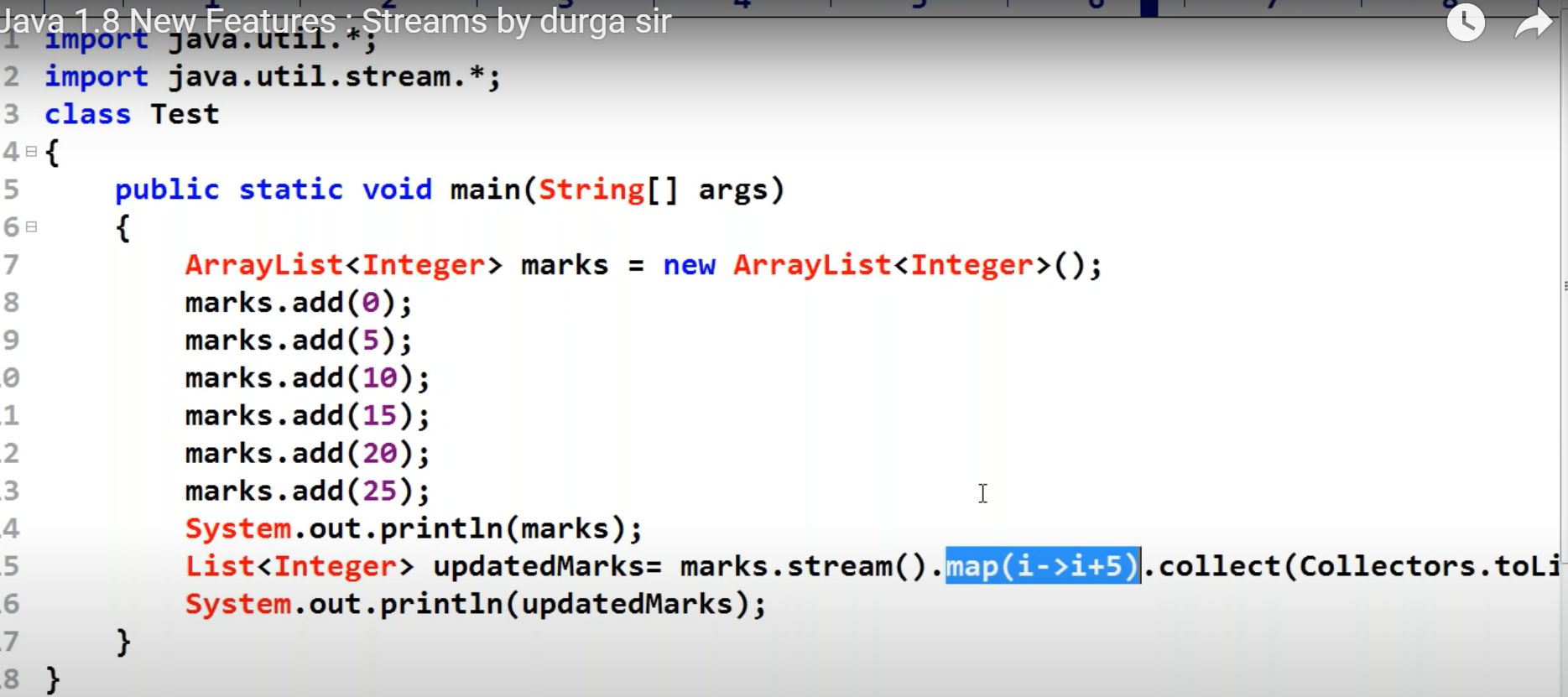
Stream :

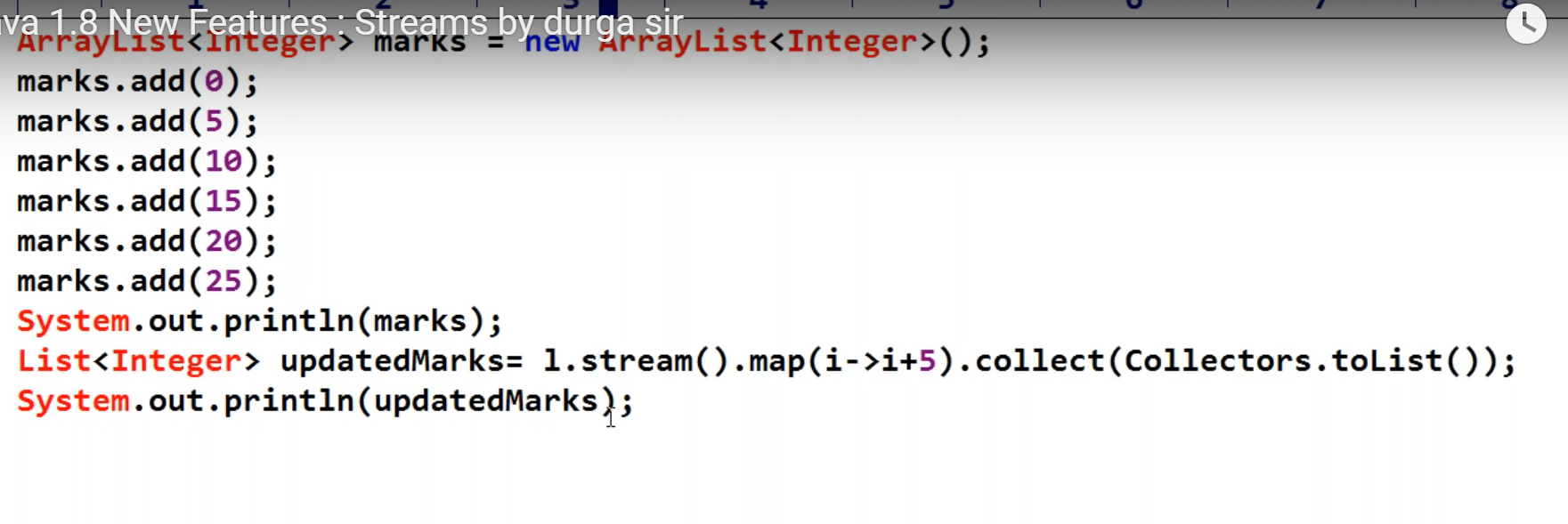
**IF WE WANT TO Represent a Group of object as a single entity, then we should go with Collection**

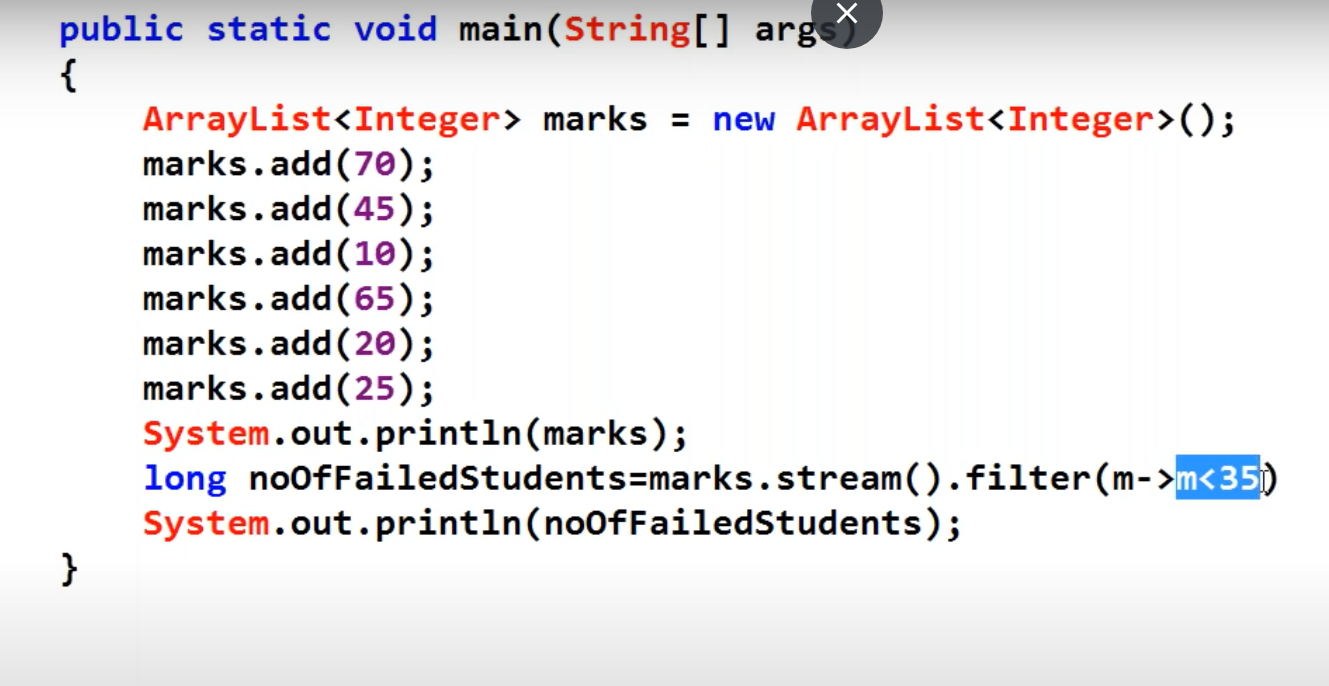
**To Process the object in the Collection, stream is used**

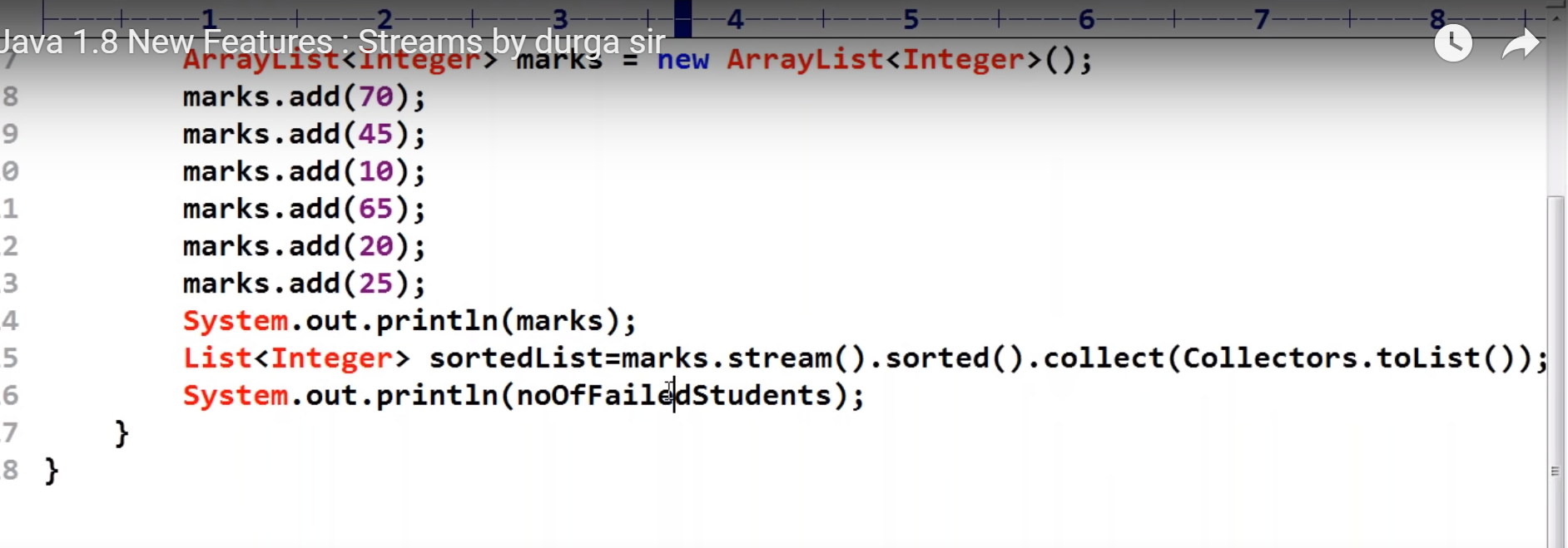
**Stream is the method**

**Stream s = c.Stream();**

****

****

****

****