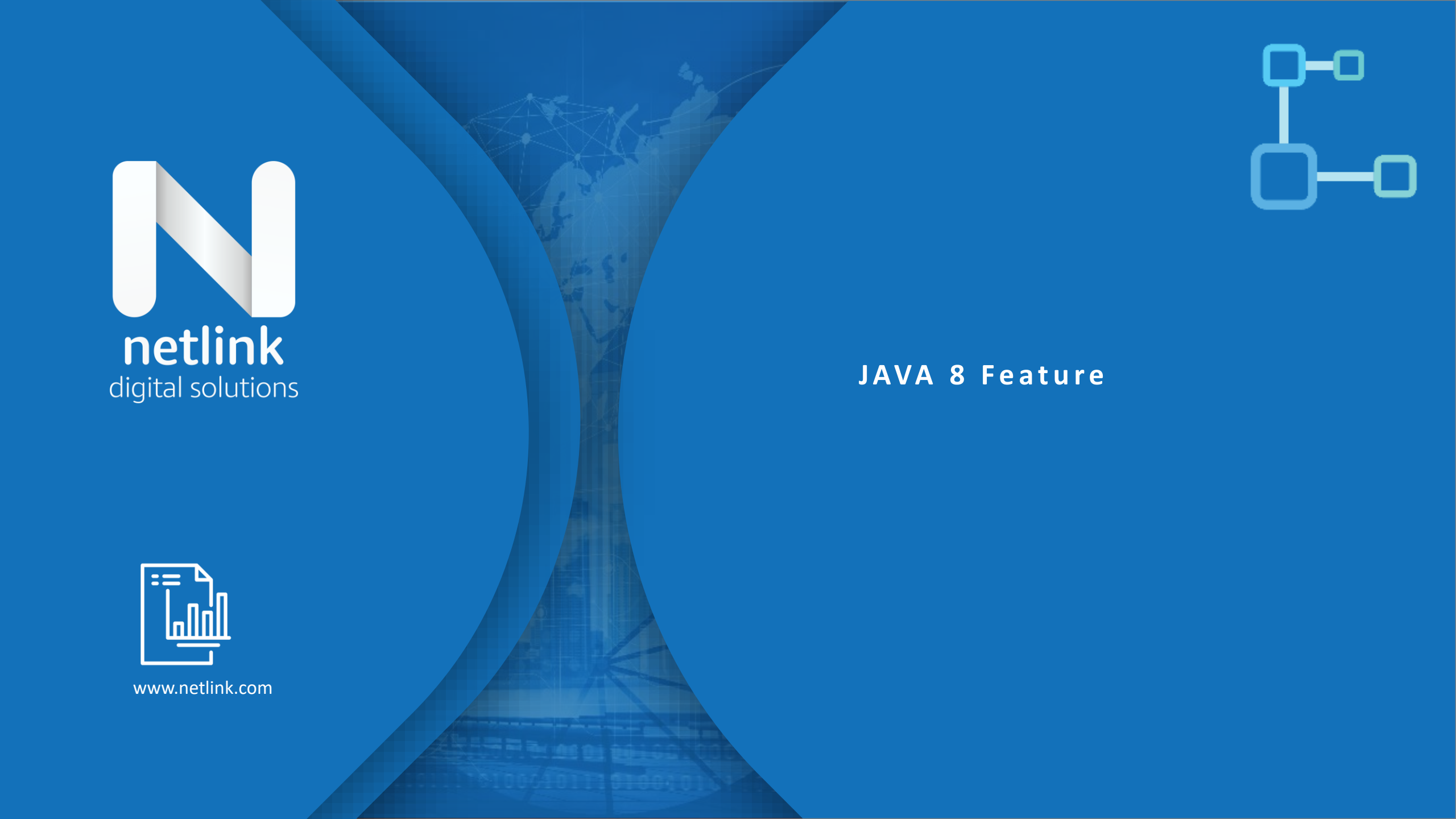

netlink
digital solutions

JAVA 8 Feature

www.netlink.com

Agenda

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

Lambda Expression

The Main Objective of λ 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);
```

Ex: 3

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

```
(String str) -> return str;

(str) -> str;
```

Lambda Expression

Conclusions:

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

Ex:

() -> sop(“hello”);

(int a) -> sop(a);

(int a, int b) -> return a+b;

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

Ex: (int a, 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”);

Lambda Expression

Conclusions:

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).**

1) Runnable → It contains only run() method
2) Comparable → It contains only compareTo() metho
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.

```
interface Interf {
    public abstract void m1();
    default void m2() {
        System.out.println ("Hi... User");
        }
}
```

functional interfaces

In Java 8 ,SunMicroSystem introduced @FunctionalInterface annotation to specify that the interface is FunctionalInterface.

Ex:

```
@FunctionalInterface
Interface Interf {
public void m1();  //this code compiles without any compilation errors.
}
```

Inside FunctionalInterface 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:

```
@FunctionalInterface {
 public void m1(); this code gives compilation error.
 public void m2();
}
```

Functional Interface with respect to Inheritance:

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

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

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

@FunctionalInterface
Interface B extends A {
    public void methodOne();

    public void methodTwo();  // Compiletime Error
}
```

- **In the child interface we can't define any new abstract methods otherwise child interface won't FunctionalInterface and if we are trying to use @FunctionalInterface annotation then compiler gives a**

Ex:

```
@FunctionalInterface
interface A {
  public void methodOne();
}
interface B extends A {
  public void methodTwo();  //this's Normal interface so that code compiles without error
}
```

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.**
- **Wherever Functional Interface concept is applicable there we can use Lambda**

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.

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)  }
```

Lambda Expressions:

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)  }
```

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”

- 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

Static methods inside interface:

- From 1.8version 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.
- 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.

Predicates

- A predicate is a function with a single argument and returns boolean value.
- To implement predicate functions injava,oracle people introduced Predicate interface in 1.8 version(i.e.,Predicate).

- 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 {
public boolean test(T t);
}
```

Function

- ✓ Functions are exactly same as predicates except that functions **can return any type of result b**ut 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);
}
```

Consumer Interface

- ✓ It is a functional interface defined in **java.util.function package.**
- ✓ It contains an abstract accept() and a default andThen() method.
- ✓ It can be used as the assignment target for a lambda expression or method reference.
- ✓ The Consumer Interface accepts a single argument and does not return any result.

Method	Description
void accept(T t)	It performs this operation on the given argument.
default Consumer<T> andThen(Consumer<? super T> after)	It returns a composed Consumer that performs, in sequence, this operation followed by the after operation. If performing either operation throws an exception, it is relayed to the caller of the composed operation. If performing this operation throws an exception, the after operation will not be performed.

Supplier Interface

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

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

T: denotes the type of the result

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

•The Supplier interface consists of only one function:

1. get()

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

Differenece between predicate and function

Differenece 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().
publicboolean test(T t)	public R apply(T t)
Predicate can return only boolean value.	Function can return any type of value

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 return type, method name, modifiers etc are not required to match**

Syntax:

✓ if our specified method is static method

Classname::methodName

✓ if the method is instance method

Objref::methodName

✓ **Reference to an Instance Method of an Arbitrary Object of a Particular Type**

This type of Instance methods refers to a non-static method that are not bound to a receiver object. In this case we don't need to create an object of a particular type.

Classname :: methodName

Note : Functional Interface can refer to lambda expression and Functional Interface can also refer to method reference. Hence lambda expression can be replaced with method reference. hence method reference is an alternative syntax to lambda expression.

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

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

```
class Sample {
    private String s;

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

interface Interf2 {
    public Sample get(String s);
}

class ConstructorRefrenceEx1 {
    public static void main(String[] args) {

        Interf2 f = s -> new Sample(s);
        f.get("From Lambda Expression");

        Interf2 f1 = Sample::new;
        f1.get("From Constructor Reference");
    }
}
```

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?

1. java.util streams meant for **processing objects** from the collection.

Ie, it represents a stream of objects from the collection .

2. 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 .

Note :

java.io streams and java.util streams both are diferent.

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.

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 two phases **1.configuration**
  2.processing

Configuration/ Intermediate

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)

Ex:

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

Hence to filter elements of collection based on some boolean ondition 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.

Processing / Terminal

1. Processing by collect() method
2. Processing by count()method
3. Processing by sorted()method
4. Processing by

- min() and max() methods
- forEach() method
- toArray() method
- Stream.of()method

1. processing by collect() method

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

2. processing by count() method

this method returns number of elements present in the stream.

3. 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.

Processing

4. processing by min() and max() method

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

5. processing by .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);
```

Note : Here, you don't actually *need* the name s in order to invoke println for each of the elements. That's where the method reference is helpful - the :: operator denotes you will be invoking the println method with a parameter, which name you don't specify explicitly:


THANK
YOU!