Java 8 New Features

```
java 7 - July 28th 2011
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. i.e, it doesn't mean Java is functional oriented programming language.

AGENDA :

Lambda Expression

History:

- 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:

- 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"");
}

lambda ex:

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



Ex:2
public void add(int a, int b) {
sop(a+b);
}

lambda ex:
(int a, 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;
}

Lambda ex:
(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);
(int a, int b) -> return a+b;
```

6. 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:
(int a, int b) -> sop(a+b);
(a,b) -> sop(a+b);
```

10. If multiple parameters present then these parameters should be separated with comma (,).
11. If zero number of parameters available then we have to use empty parameter [like ()].

```
Ex: () -> sop("hello");
```

13. 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);
```

18. 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.
19. 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 Interface {
2) public abstract void m1();
3) default void m2() {
4) System.out.println ("hello");
5) }
6) }
```

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

```
Ex:
@FunctionalInterface
interface Interf { //This code compiles without any compilation errors.
public void m1();
}
```

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:
@FunctionalInterface
interface Interf {
public void m1(); //This code gives compilation error.
public void m2();
}
```

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:
@FunctionalInterface
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) @FunctionalInterface
2) interface A {
3) public void methodOne();
4) }

5) @FunctionalInterface
6) interface B extends A {  //No Compile Time Error
7) }
```

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

```
Ex:
1) @FunctionalInterface
2) interface A {
```

```
3) public void methodOne();
4) }

5) @FunctionalInterface
6) interface B extends A {
7) public void methodOne();  //No Compile Time Error
8) }
```

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 @FunctionalInterface annotation then compiler gives an error message.

```
1) @FunctionalInterface {
2) interface A {
3) public void methodOne();
4) }
5) @FunctionalInterface
6) interface B extends A {
7) public void methodTwo();  //Compile Time Error
8) }
Ex:
@FunctionalInterface
interface A {
public void methodOne(); //No Compile Time Error
}
interface B extends A {
public void methodTwo(); //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 Interf {
```

```
4) public void methodOne() {
5) System.out.println("method one execution");
6) } }
7) public class Test {
8) public static void main(String[] args) {
9) Interf i = 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) Interf i = () -> System.out.println("method one execution");
6) i.methodOne();
7) }
8) }
```

Ex 2: 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) Interf i = 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) Interf i = (a,b) -> System.out.println("The Sum:" +(a+b));
7) i.sum(5,10);
8) }
9) }
```

Ex 3: Without Lambda Expressions

```
1) interface Interf {
2) public int 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) Interf i = 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) Interf i = x -> x*x;
7) System.out.println("The Square of 5 is:"+i.square(5));
8) }
9) }
```

Ex 4: Without Lambda expression

```
1) class MyRunnable implements Runnable {
2) public void run() {
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) Interf i = ()-> {
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) Interf i = () -> {
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 int hashCode() {
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
java Interf
```

Predicate

- 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 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:");
14) m1(p1.and(p2), x);
15) System.out.println("The Numbers Greater Than 10 OR Even:");
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.
- Function 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("Times"));
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, 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**

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.stream and java.io streams ?

- java.util streams meant for processing objects from the collection. i.e, 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

I. 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) import java.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.

```
LocalDateTime dt = 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");
ZonedDateTime zt = 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) }
```

Consumer

The **Consumer 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 takes in one argument and produces a result. However these kind of functions don't return any value. The **Consumer 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 takes in one argument and produces a result. However these kind of functions don't return any value.

Hence this functional interface which takes in one generic namely:-

- **T**: denotes the type of the input argument to the operation

The lambda expression assigned to an object of Consumer type is used to define its **accept()** which eventually applies the given operation on its argument. Consumers are useful when it not needed to return any value as they are expected to operate via side-effects.

Functions in Consumer Interface

The Consumer interface consists of the following two functions:

1. accept(): This method accepts one value and performs the operation on the given argument.

Syntax:
void accept(T t)

Parameters: This method takes in one parameter:

- **t**– the input argument

Returns: This method does not return any value.
Below is the code to illustrate accept() method

Ex.

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

```
            // Consumer to display a number
            Consumer<Integer> display = a -> System.out.println(a);

            // Implement display using accept()
            display.accept(10);

            // Consumer to multiply 2 to every integer of a list
            Consumer<List<Integer> > modify = list ->
            {
                  for (int i = 0; i < list.size(); i++)
                        list.set(i, 2 * list.get(i));
            };

            // Consumer to display a list of numbers
            Consumer<List<Integer> >
dispList = list -> list.stream().forEach(a -> System.out.print(a + " "));

            List<Integer> list = new ArrayList<Integer>();
            list.add(2);
            list.add(1);
            list.add(3);

            // Implement modify using accept()
            modify.accept(list);

            // Implement dispList using accept()
            dispList.accept(list);
      }
}
```

2. andThen(): It returns a composed Comsumer wherein the parameterized Consumer will be executed after the first one. If evaluation of either function throws an error, it is relayed to the caller of the composed operation.

Note: The function being passed as the argument should be of type Consumer.

Syntax:

default Consumer <T>

andThen(Consumer<? super T> after)

Parameters: This method accepts a parameter **after** which is the Consumer to be applied after the current one.

Return Value: This method returns a composed Consumer that first applies the current Consumer first and then the after operation.

Exception: This method throws **NullPointerException** if the after operation is null.

Below is the code to illustrate andThen() method:

Ex.

```
import java.util.*;

public class Main {
	public static void main(String args[]){
		// Consumer to multiply 2 to every integer of a list
		Consumer<List<Integer> > modify = list ->
		{
			for (int i = 0; i < list.size(); i++)
				list.set(i, 2 * list.get(i));
		};

		// Consumer to display a list of integers
		Consumer<List<Integer> >
			dispList = list -> list.stream().forEach(a ->
System.out.print(a + " "));

		List<Integer> list = new ArrayList<Integer>();
		list.add(2);
		list.add(1);
		list.add(3);

		// using addThen()
		modify.andThen(dispList).accept(list);
		;
	}
}
```

Ex: To demonstrate when NullPointerException is returned.

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

		// Consumer to multiply 2 to every integer of a list
		Consumer<List<Integer> > modify = list ->
		{
			for (int i = 0; i < list.size(); i++)
				list.set(i, 2 * list.get(i));
		};

		// Consumer to display a list of integers
		Consumer<List<Integer> >
			dispList = list -> list.stream().forEach(a ->
System.out.print(a + " "));

		List<Integer> list = new ArrayList<Integer>();
		list.add(2);
		list.add(1);
		list.add(3);
		try {
			// using addThen()
			modify.andThen(null).accept(list);
		}
```

```
            catch (Exception e) {
                  System.out.println("Exception: " + e);
            }
      }
}
```

Ex: To demonstrate how an Exception in the after function is returned and handled.

```
import java.util.*;
public class Main {
      public static void main(String args[]){
            // Consumer to multiply 2 to every integer of a list
            Consumer<List<Integer> > modify = list ->
            {
                  for (int i = 0; i <= list.size(); i++)
                        list.set(i, 2 * list.get(i));
            };

            // Consumer to display a list of integers
            Consumer<List<Integer> >
                  dispList = list -> list.stream().forEach(a ->
System.out.print(a + " "));
            System.out.println();

            List<Integer> list = new ArrayList<Integer>();
            list.add(2);
            list.add(1);
            list.add(3);

            // using addThen()
            try {
                  dispList.andThen(modify).accept(list);
                  ;
            }
            catch (Exception e) {
                  System.out.println("Exception: " + e);
            }
      }
}
```

Supplier

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.

Syntax:T get()

Returns: This method returns a **value** of type T.
Below is the code to illustrate get() method:

Ex.

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

public class Main {
      public static void main(String args[])
      {
                  // This function returns a random value.
            Supplier<Double> randomValue = () -> Math.random();

            // Print the random value using get()
            System.out.println(randomValue.get());
      }
}
Output: 0.5685808855697841
```

Type Annotations

Before Java 8, we can use annotations at many places but only on declarations e.g. on classes, interfaces, fields, method declaration etc. In Java 8 we can generally use annotations on any use of a type, , e.g. with generics, with method return types, with object initiation using 'new', with 'implements' and 'extends' clauses, with type casts and with throws clauses. etc.

Annotations are used to provide supplement information about a program.

New Target ElementType in Java 8

The enum constants provided by java.lang.annotation.ElementType are used in java.lang.annotation.Target meta-annotations to specify where it is legal to write annotations of a given type

Java 8 introduced two new Element types

- ElementType.TYPE_USE: allows an annotation to be applied at any type use.
- ElementType.TYPE_PARAMETER: allows an annotation to be applied at type variables (e.g. T in MyClass<T>).

For example we can now create a new annotation like:

```
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface MyAnnotation {
}
```

Usage Examples

extends clause:

```
public class MyClass extends @MyAnnotation OtherClass { }
```

implements clause:

```
public class MyClass implements @MyAnnotation someInterface { }
```

Generics:

```
public class MyClass<@MyAnnotation T> {}

List<@MyAnnotation String> myList;
List<@MyAnnotation T> myList;
List<@MyAnnotation ? extends OtherClass> myList;
List<@MyAnnotation ? super T> myList
List<@MyAnnotation ?> myList;
```

Simple Field Types

```
private @MyAnnotation String str;
private @MyAnnotation List<String> mylist
```

On arrays

```
private @MyAnnotation char[] chars; //on char
private @MyAnnotation char[][] chars2;//on char
private char @MyAnnotation [] chars3;//on char array, char[]
private char[] @MyAnnotation [] chars4; //on char array, char[] which is
component of char[][]
```

Nested type

```
MyClass.@MyAnnotation NestedClass nestedClass;//NestedClass is nested class
of MyClass
```

Creating objects

```
List<String> list = new @MyAnnotation ArrayList<>();
MyClass myClass = new @MyAnnotation MyClass();
```

Creating non-static nested class object

```
MyClass myClass = new @MyAnnotation MyClass();
MyClass.NestedClass nestedClass = myClass.new @MyAnnotation NestedClass();
```

Type casting

```
MyClass myClass = (@MyAnnotation MyClass) otherClassInstance;
```

Instanceof

```
boolean b = myObject instanceof @MyAnnotation OtherClass;
```

Method return type

```
public @MyAnnotation String getString(){.....}
```

Method parameters

```
public void showString(@MyAnnotation String str){.....}
```

Calling generic method

```
myObject.<@MyAnnotation String>myMethod(.....)
```

Local variables

```
public void myMethod(){
        @MyAnnotation List<String> str = new ArrayList<>();
}
```

Java 8 method references

```
Function<MyClass, String> f = @MyAnnotation MyClass::myMethod;
```

throws clause

```
void myMethod() throws @MyAnnotation Exception{}
```

catch clause

```
try {
     myObject.myMethod();
  } catch (@MyAnnotation Exception e) {
            .....
 }
```

Nashorn JavaScript Engine

Nashorn: Nashorn is a JavaScript engine which is introduced in JDK 8. With the help of Nashorn, we can execute JavaScript code at Java Virtual Machine. Nashorn is introduced in JDK 8 to replace existing JavaScript engine i.e. Rhino. Nashorn is far better than Rhino in term of performance.

The uses of invoking dynamic feature, conversion of JavaScript code into the bytecode directly into the memory etc makes the Nashorn more famous in JDK 8. We can execute JavaScript code by using the command-line tool and by embedding the JavaScript code into Java source code.

Executing JavaScript code by using console: For Nashorn engine, Java 8 introduced one new command-line tool i.e.jjl. We have to follow the below steps to execute JavaScript code through the console:

•Create one file named with index.js.

Open index.js and write following code into the file and save it.

```
var x= function(){
      print("Welcome to Java World!!!");
};
x();
```

Open CMD, write **jjl index.js** and press enter. It will generate the below output:
Welcome to Java World!!!

Executing JavaScript file by embedding JavaScript file into Java code: We can execute JavaScript file by embedding JavaScript file into Java code with the help of **ScriptEngine** class. ScriptEngine class is introduced in JDK 6. By the help of the ScriptEngine class, we can create a JavaScript engine and with the JavaScript engine, we can execute the javaScript file.
Ex. Program to illustrate embedding of JavaScript file into Java code

```
import javax.script.*;
import java.io.*;

public class Example {
      public static void main(String[] args)throws Exception{
            // Here we are generating Nashorn JavaScript Engine
            ScriptEngine ee = new ScriptEngineManager()
                                          .getEngineByName("Nashorn");

            // Reading JavaScript file create in first approach
            ee.eval(new FileReader("index.js"));
      }
}
```

Apart from above, with the help of Nashorn JavaScript Engine, we can perform multiple operations like:

1. Providing JavaScript variable from Java Code: Suppose we have one JavaScript file name with geeks.js and geeks.js requires one variable during execution. With the help of Nashorn, we can pass the variable to JavaScript file from java code.
   Ex: index.js file, which needs name variable to get executed

```
// JavaScript file name with geeks.js
print("Welcome to Java World!!! "+name);
```

Ex: Java code providing name variable to the JS file

```
import javax.script.*;
```

```
import java.io.*;

public class Geeksforgeeks {
      public static void main(String[] args)
            throws Exception
      {
            ScriptEngine ee
                  = new ScriptEngineManager()
                        .getEngineByName("Nashorn");
            Bindings bind
                  = ee.getBindings(
                        ScriptContext.ENGINE_SCOPE);
            bind.put("name", "Welcome to Java World!!");
            ee.eval(new FileReader("index.js"));
      }
}
```

2. **Calling JavaScript function from Java code:** We can call JavaScript function from Java code with the help of Nashorn. Suppose we create one file name with geeks.js and the file contains two functions like below:

```
// JavaScript file name with index.js

var func1 = function(){
      print("Simple JavaScript function!!!");
}

var func2 = function(reader){
      print("Hello "+reader);
}
```

Ex. Program to illustrate calling of JavaScript function from Java code

```
import javax.script.*;
import java.io.*;

public class Example {
      public static void main(String[] args)
            throws Exception
      {
            ScriptEngine ee
                  = new ScriptEngineManager()
                        .getEngineByName("Nashorn");
            ee.eval(new FileReader("index.js"));
            Invocable invocable = (Invocable)ee;

            // Here we are calling func1
            invocable.invokeFunction("func1");

            // Here we are calling func2
            // as well as passing argument
            invocable.invokeFunction("func2","Welcome To JAVA World");
```

```
        }
}
```

Concurrent Accumulators

One or more variables that together maintain a running long value updated using a supplied function. When updates (method accumulate(long)) are contended across threads, the set of variables may grow dynamically to reduce contention. Method get() (or, equivalently, longValue()) returns the current value across the variables maintaining updates.

This class is usually preferable to AtomicLong when multiple threads update a common value that is used for purposes such as collecting statistics, not for fine-grained synchronization control. Under low update contention, the two classes have similar characteristics. But under high contention, expected throughput of this class is significantly higher, at the expense of higher space consumption.

The order of accumulation within or across threads is not guaranteed and cannot be depended upon, so this class is only applicable to functions for which the order of accumulation does not matter. The supplied accumulator function should be side-effect-free, since it may be re-applied when attempted updates fail due to contention among threads. The function is applied with the current value as its first argument, and the given update as the second argument. For example, to maintain a running maximum value, you could supply Long::max along with Long.MIN_VALUE as the identity.

Class LongAdder provides analogs of the functionality of this class for the common special case of maintaining counts and sums. The call new LongAdder() is equivalent to new LongAccumulator((x, y) -> x + y, 0L.

This class extends Number, but does *not* define methods such
as equals, hashCode and compareTo because instances are expected to be mutated,
and so are not useful as collection keys.

we'll be looking at two constructs from
the **java.util.concurrent** package: **LongAdder** and **LongAccumulator**.

Both are created to be very efficient in the multi-threaded environment and both leverage very clever tactics to be **lock-free and still remain thread-safe.**

1)LongAdder

Let's consider some logic that's incrementing some values very often, where using an AtomicLong can be a bottleneck. This uses a compare-and-swap operation, which – under heavy contention – can lead to a lot of wasted CPU cycles.

LongAdder, on the other hand, uses a very clever trick to reduce contention between threads, when these are incrementing it.

When we want to increment an instance of the LongAdder, we need to call the increment() method. That implementation **keeps an array of counters that can grow on demand**.

And so, when more threads are calling increment(), the array will be longer. Each record in the array can be updated separately – reducing the contention. Due to that fact, the LongAdder is a very efficient way to increment a counter from multiple threads.

Let's create an instance of the LongAdder class and update it from multiple threads:

```
LongAdder counter = new LongAdder();
ExecutorService executorService = Executors.newFixedThreadPool(8);

int numberOfThreads = 4;
int numberOfIncrements = 100;

Runnable incrementAction = () -> IntStream
  .range(0, numberOfIncrements)
  .forEach(i -> counter.increment());

for (int i = 0; i < numberOfThreads; i++) {
    executorService.execute(incrementAction);
}
```

The result of the counter in the LongAdder is not available until we call the sum() method. That method will iterate over all values of the underneath array, and sum those values returning the proper value. We need to be careful though because the call to the sum() method can be very costly:

```
assertEquals(counter.sum(), numberOfIncrements * numberOfThreads);
```

Sometimes, after we call sum(), we want to clear all state that is associated with the instance of the LongAdder and start counting from the beginning. We can use the sumThenReset() method to achieve that:

```
assertEquals(counter.sumThenReset(), numberOfIncrements * numberOfThreads);
assertEquals(counter.sum(), 0);
```

Note that the subsequent call to the sum() method returns zero meaning that the state was successfully reset.

Moreover, Java also provides DoubleAdder to maintain a summation of double values with a similar API to LongAdder.

2. LongAccumulator

LongAccumulator is also a very interesting class – which allows us to implement a lock-free algorithm in a number of scenarios. For example, it can be used to accumulate results according to the supplied LongBinaryOperator – this works similarly to the *reduce()* operation from Stream API.

The instance of the LongAccumulator can be created by supplying the LongBinaryOperator and the initial value to its constructor. The important thing to remember that LongAccumulator will work correctly if we supply it with a commutative function where the order of accumulation does not matter.

```
LongAccumulator accumulator = new LongAccumulator(Long::sum, 0L);
```

We're creating a LongAccumulator whi*ch* will add a new value to the value that was already in the accumulator. We are setting the initial value of the LongAccumulator to zero, so in the first call of the accumulate() method, the previousValue will have a zero value.

Let's invoke the accumulate() method from multiple threads:

```
int numberOfThreads = 4;
int numberOfIncrements = 100;

Runnable accumulateAction = () -> IntStream
  .rangeClosed(0, numberOfIncrements)
  .forEach(accumulator::accumulate);

for (int i = 0; i < numberOfThreads; i++) {
    executorService.execute(accumulateAction);
}
```

Notice how we're passing a number as an argument to the *accumulate()* method. That method will invoke our *sum()* function.

The *LongAccumulator* is using the compare-and-swap implementation – which leads to these interesting semantics.

Firstly, it executes an action defined as a *LongBinaryOperator*, and then it checks if the *previousValue* changed. If it was changed, the action is executed again with the new value. If not, it succeeds in changing the value that is stored in the accumulator.

We can now assert that the sum of all values from all iterations was *20200*:

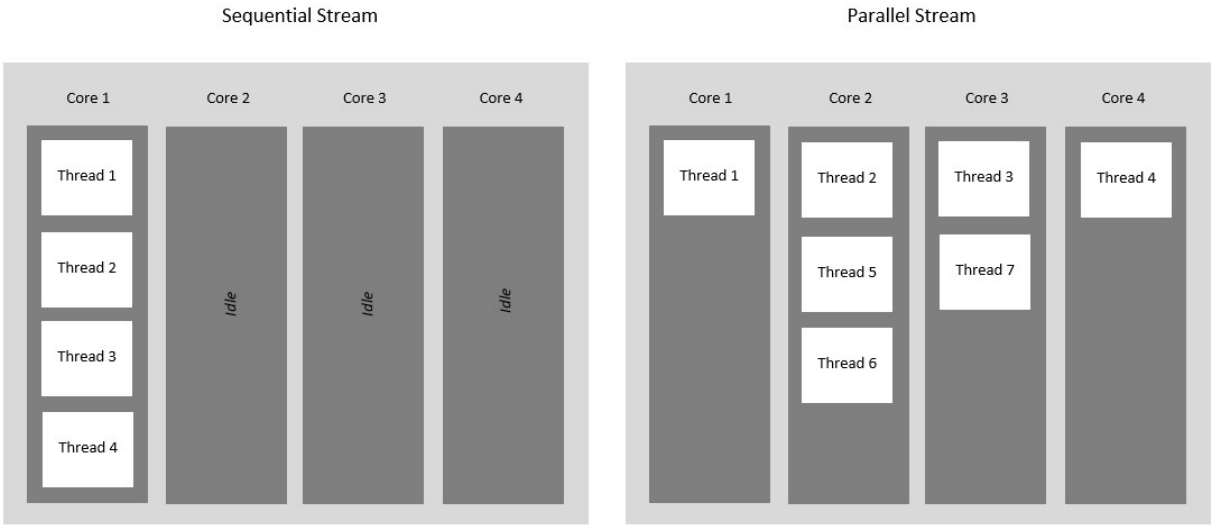
```
assertEquals(accumulator.get(), 20200);
```

Interestingly, Java also provides DoubleAccumulator with the same purpose and API but for *double* values.

Parallel operations

Java Parallel Streams is a feature of Java 8 and higher, meant for utilizing multiple cores of the processor. Normally any java code has one stream of processing, where it is executed sequentially. Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes. The order of execution, however, is not under our control.

Therefore, it is advisable to use parallel streams in cases where no matter what is the order of execution, the result is unaffected and the state of one element does not affect the other as well as the source of the data also remains unaffected.



There are two ways we can create parallel streams in Java:

1. Using parallel() method on a stream

The **parallel() method** of the **BaseStream interface** returns an equivalent parallel stream. Let us explain how it would work with the help of an example.

In the code given below, we create a file object which points to a pre-existent txt file in the system. Then we create a Stream that reads from the text file one line at a time. Then we use the **parallel() method** to print the read file on the console. The order of execution is different for each run, you can observe this in the output. The two outputs given below have different orders of execution.

Ex. An example to understand the parallel() method

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.stream.Stream;

public class ParallelStreamTest {

	public static void main(String[] args) throws IOException {

		// Create a File object
		File fileName = new File("M:\\Documents\\Textfile.txt");

		// Create a Stream of String type
		// Using the lines() method to read one line at a time
		// from the text file
		Stream<String> text = Files.lines(fileName.toPath());

		// Use StreamObject.parallel() to create parallel streams
		// Use forEach() to print the lines on the console
		text.parallel().forEach(System.out::println);

		// Close the Stream
		text.close();
	}
}
```

2. Using parallelStream() on a Collection

The **parallelStream() method** of the Collection interface returns a possible parallel stream with the collection as the source. Let us explain the working with the help of an example.

In the code given below, we are again using parallel streams but here we are using a List to read from the text file. Therefore, we need the **parallelStream() method**.

Ex. An example of parallelStream() method

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.*;

public class ParallelStreamsTest3 {

	public static void main(String[] args) throws IOException {

		// Create a File object
		File fileName = new File("M:\\Documents\\List_Textfile.txt");

		// Create a List
		// Using readAllLines() read the lines of the text file
		List<String> text = Files.readAllLines(fileName.toPath());

		// Using parallelStream() to create parallel streams
```

```
            text.parallelStream().forEach(System.out::println);
        }
}
```

Why Parallel Streams?

Parallel Streams were introduced to increase the performance of a program, but opting for parallel streams isn't always the best choice. There are certain instances in which we need the code to be executed in a certain order and in these cases, we better use sequential streams to perform our task at the cost of performance. The performance difference between the two kinds of streams is only of concern for large scale programs or complex projects. For small scale programs, it may not even be noticeable. Basically you should consider using Parallel Streams when the sequential stream behaves poorly.

****Do remember**, Parallel Streams must be used only with stateless, non-interfering, and associative operations i.e.

•A stateless operation is an operation in which the state of one element does not affect another element

•A non-interfering operation is an operation in which data source is not affected

•An associative operation is an operation in which the result is not affected by the order of operands

Parallel Streams Performance Implications

Parallel Stream has equal performance impacts as like its advantages.

Since each sub-stream is a single thread running and acting on the data, it has overhead compared to the sequential stream

Inter-thread communication is dangerous and takes time for coordination

When to use Parallel Streams?

They should be used when the output of the operation is not needed to be dependent on the order of elements present in source collection (i.e. on which the stream is created)

Parallel Streams can be used in case of aggregate functions

Parallel Streams quickly iterate over the large-sized collections

Parallel Streams can be used if developers have performance implications with the Sequential Streams

If the environment is not multi-threaded, then Parallel Stream creates thread and can affect the new requests coming in

Now, open up the Eclipse Ide and I will explain further the parallel streams in Java8 programming.

PermGen Space Removed

In **Java 8**, **PermGen** space is completely removed and replace with **MetaSpace**. Most allocation for class metadata is now allocated in native memory and classes that describe class metadata have been removed now.

Now HotSpot JVM uses native memory for the representation of class metadata. MetaSpace memory limit is unbound i.e not mandatory to define the **upper** limit of memory usages. Now instead of reserved native memory, it will increase and decrease based on usages and you will not get OutOfMemoryError : PermGen error.

Note: You can define the max limit of MetaSpace but that can cause OutOfMemoryError: Metaspace. Try to define this limit cautiously to avoid memory waste.

After installation of JDK 8 and trying to run Eclipse with the configuration of **PermSize** and **MaxPerSize**. You will get a warning message:

```
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512m; support was removed in 8.0
```

Main difference between old PermGen and new MetaSpace is, you don't have to mandatory define upper limit of memory usage. You can keep MetaSpace space limit unbounded. Thus when memory usage increases you will not get OutOfMemoryError error. Instead the reserved native memory is increased to full-fill the increase memory usage.

You can define the max limit of space for MetaSpace, and then it will throw OutOfMemoryError : Metadata space. Thus it is important to define this limit cautiously, so that we can avoid memory waste.

Reason to remove PermGen from Java 8

There were lots of drawbacks of PermGen :

- **Fixed-size at startup:** difficult to tune.
- **Internal Hotspot types were Java objects:** Could move with full GC, opaque, not strongly typed and hard to debug, needed metadata.
- **Simplify full collections:** Added Special iterators for metadata for each collector.
- Instead of GC pause, now deallocate class data concurrently.
- Enable future improvements that were limited by PermGen.

Advantages of MetaSpace

There are lots of advantage of **MetaSpace** in terms of performance and memory management:

- **Take advantage of Java Specification property:** Classes and associated metadata lifetimes match class loaders.
- **Per loader storage area:** Metaspace
- Linear allocation only.
- No individual reclamation (except for Redefine Classes and class loading failure)
- No GC scan or compaction.
- No relocation for metaspace objects.

Metaspace Tuning

To set maximum metaspace size can use the **-XX:MaxMetaspaceSize** flag and the by default is **unlimited** based on your machine memory limit. If you don't specify this max limit flag, the Metaspace will dynamically re-size depending on your application demand at runtime.
This change enables other optimizations and features in the future

- Application class data sharing.
- Young collection optimizations, G1 class unloading.
- Metadata size reductions and internal JVM footprint projects.

TLS SNI(Transport Level Security-Sever Name Indication)

Sever Name Indication

The SNI extension is a feature that extends the SSL/TLS protocols to indicate what server name the client is attempting to connect to during handshaking. Servers can use server name indication information to decide whether specific SSLSocket or SSLEngine instances should accept a connection. For example, when multiple virtual or name-based servers are hosted on a single underlying network address, the server application can use SNI information to determine whether this server is the exact server that the client wants to access. Instances of this class can be used by a server to verify the acceptable server names of a particular type, such as host names. For more information, see section 3 of TLS Extensions (RFC 6066). Developers of client applications can explicitly set the server name indication by using

the SSLParameters.setServerNames(List<SNIServerName>serverNames) method. The following example illustrates this function:

```
SSLSocketFactory factory = ...

SSLSocket sslSocket = factory.createSocket("172.16.10.6", 443);

// SSLEngine sslEngine = sslContext.createSSLEngine("172.16.10.6", 443);


SNIHostName serverName = new SNIHostName("www.example.com");

List<SNIServerName> serverNames = new ArrayList<>(1);

serverNames.add(serverName);


SSLParameters params = sslSocket.getSSLParameters();

params.setServerNames(serverNames);

sslSocket.setSSLParameters(params);

// sslEngine.setSSLParameters(params);
```

Developers of server applications can use the SNIMatcher class to decide how to recognize server name indication. The following two examples illustrate this function:

Ex.

```
SSLSocket sslSocket = sslServerSocket.accept();

SNIMatcher                                matcher                                =
SNIHostName.createSNIMatcher("www\\.example\\.(com|org)");

Collection<SNIMatcher> matchers = new ArrayList<>(1);

matchers.add(matcher);

SSLParameters params = sslSocket.getSSLParameters();

params.setSNIMatchers(matchers);

sslSocket.setSSLParameters(params);
```

Ex.

```
SSLServerSocket sslServerSocket = ...;

SNIMatcher                                matcher                                =
SNIHostName.createSNIMatcher("www\\.example\\.(com|org)");

Collection<SNIMatcher> matchers = new ArrayList<>(1);

matchers.add(matcher);

SSLParameters params = sslServerSocket.getSSLParameters();

params.setSNIMatchers(matchers);

sslServerSocket.setSSLParameters(params);

SSLSocket sslSocket = sslServerSocket.accept();
```

The following list provides examples for the behavior of the SNIMatcher class when receiving various server name indication requests in the ClientHello message:

- Matcher is configured to www\\.example\\.com:
  - If the requested host name is www.example.com, the request is accepted and a confirmation is sent in the ServerHello message.
  - If the requested host name is www.example.org, the request is rejected with an unrecognized_name unrecoverable error.
  - If there is no requested host name or it is empty, the request is accepted but no confirmation is sent in the ServerHello message.
- Matcher is configured to www\\.invalid\\.com:
  - If the requested host name is www.example.com, the request is rejected with an unrecognized_name unrecoverable error.
  - If the requested host name is www.example.org, the request is accepted and a confirmation is sent in the ServerHello message.
  - If there is no requested host name or it is empty, the request is accepted but no confirmation is sent in the ServerHello message.
- Matcher is not configured:
  - Any requested host name is accepted but no confirmation is sent in the ServerHello message.

Transport Level Security (TLS) is designed to encrypt conversations between two parties and ensure that others can neither read nor modify the conversation. When combined with Certificate Authorities, a proper level of trust is established: we know who is on the other end of the conversation and that conversation is protected from eavesdropping/modification.

Support for TLS 1.2 first appeared in JDK 7 (2011). For compatibility reasons, it is enabled by default on server sockets but disabled on clients. Since that time, the industry has made considerable improvements to address interoperability and backwards compatibility.

JDK 8 will use TLS 1.2 as default

We are setting JDK 8 to use TLS 1.2 as the default for two reasons:

1. TLS is backwards-compatible. After upgrading the default to 1.2, systems using 1.1 and 1.0 will continue to function*.
a. * Unless configured to use an algorithm that was removed for security reasons. Few systems are affected by this.
b. For a complete description of TLS 1.2, please see RFC 5246.
c. A quick summary of TLS/SSL differences is available from yaSSL.
2. It strengthens the protection of internet communications against eavesdropping.

For those testing JDK 8 early access, this first occurred in build 122.
TLS is transparent to most users and developers. For those that would like more details, we will cover:

- Threats and the role of encryption
- Compatibility with the JDK and other systems
- Understanding your TLS implementation
- Other considerations for TLS

Threats and the role of encryption

With a new well-motivated IETF working group for encryption as well as wide industry support for TLS 1.2, the time is right to update system defaults.
Qualys SSL Labs has done great research in depicting a threat model for TLS. Their best practices in dealing with the TLS threat model (specifically "2.2 use secure protocols") support this move.
Compatibility with the JDK and other systems

TLS 1.2 is designed to be backwards-compatible as described in the RFC Appendix E (above). If a 1.2 client connects to a server running a lower version, the client will adjust. If a lower client connects to a server running 1.2, the server will adjust. Because of backwards-compatibility, clients supporting TLS 1.2 will receive

improved communications and older clients will continue to function.

- We added support for TLS 1.2 in JDK 7 (July 2011) although it was not the default. JDK 8 (March 2014) will use TLS 1.2 as the default.
- OpenSSL added support for TLS 1.2 in version 1.0.1 (March 2012). Most Linux distributions and scripting languages use OpenSSL.
- Microsoft supported TLS 1.2 in Windows 7. Internet Explorer and .NET follow accordingly. TLS 1.2 was first enabled by default in Internet Explorer 11 (October 2013).
- Firefox turned TLS 1.2 on by default in version 27 (February 2014).
- Chrome supported TLS 1.2 in version 29 (August 2013).
- Etc.

Adoption statistics from the Trustworthy Internet's SSL Pulse show a sufficient number of internet-facing systems using TLS 1.2 and compatible ciphers.
Understanding your TLS implementation

Developers or System Administrators can test servers and clients through the Qualys SSL Labs (server or client) or a different How's My SSL website.
System Administrators can view their system's TLS implementation to monitor clients or disable specific TLS versions. For example some system administrators in highly sensitive businesses may want to disable older TLS versions from ever being used.

View your client's version through a GUI

1. Open the Java Control Panel
2. Navigate to the Advanced tab.
3. At the bottom, there is an "Advanced Security Settings."
4. Check or uncheck the "Use TLS X.Y" box.

On a server or without a GUI

To set this for everything:

1. Open the deployment.properties file, either user-level or system-level.
2. Set the appropriate property deployment.security.TLSvX.Y=false

1. To set for a specific application or script:

1. Use the startup flag -Ddeployment.security.TLSvX.Y=false

For System Administrators (or some Developers): Perfect Forward Secrecy can be used in Java TLS connections. Using Perfect Forward Secrecy protects past conversations: in the event that if keys are lost in the future, someone cannot decrypt past conversations. As is common with TLS implementations, Perfect Forward Secrecy is not enabled by default. Those that do want to use it can update their https.cipherSuites property. Common values for this property are:

- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- Anything on the Algorithm Standard Name list that start with TLS (Transport Level Security) followed by a type of DHE (Diffie-Hellman Exchange).

Java 8 Interview Questions

Q.1 What actual advantages does Java 8 bring?

- Code is more concise and readable.
- Code is more reusable.
- Code is more testable and maintainable.
- Code is now both highly concurrent and scalable.
- Users can write parallel code.
- Users can write database-like-operations.
- Applications now perform better.
- Code is far more productive.
- Less boiler plate code.
- It supports writing databases including promotions.

Q.2 What enhancements have been done to JDK?

- Until Java 7, JVM used an area called PermGen to store classes. It got removed in Java 8 and replaced by MetaSpace.
- Major advantage of MetaSpace over permGen: PermGen was fixed in term of maximum size and can not grow dynamically but Metaspace can grow dynamically and do not have any size constraint.

Q.3 What changes have been done to HashMap in Java8?

→ Buckets containing a large number of colliding keys will store their entries in a balanced tree instead of a linked list after certain threshold is reached.

Q.4 How lambda expression and functional interfaces are related.

→ Lambda expressions can only be applied to abstract method of functional interface.

Lambda expression is a type of function without a name. It may or may not have results and parameters. It is known as an anonymous function as it does not have type information by itself. It is executed on-demand. It is beneficial in iterating, filtering, and extracting data from a collection.

As lambda expressions are similar to anonymous functions, they can only be applied to the single abstract method of Functional Interface. It will infer the return type, type, and several arguments from the signature of the abstract method of functional interface.

Q.5 What is stream pipelining?

- Stream pipelining is the process of chaining different operations together. Pipelining accomplishes this function by dividing stream operations into two categories, intermediate operations, and terminal operations. When each intermediate operation runs, it returns an instance of the stream. Thus, a user can set up an arbitrary number of intermediate operations to process data, thereby forming a processing pipeline.
- At the end of the process, there must be a terminal operation to return a final value and terminate the pipeline.

Q.6 What are functional interfaces?

- Functional interfaces are those interfaces which can have only one abstract method. It can have static method, default methods or can override Object's class methods.
- There are many functional interfaces already present in Java such as Comparable, Runnable.

Q.7 Can you create your own functional interface?

→ Yes, you can create your own functional interface. Java can implicitly identify functional interface but you can also annotate it with @FunctionalInterface.

Example:
Create interface named "Printable" as below

```
public interface Printable {

 void print();

    default void printColor(){

     System.out.println("Printing Color copy");

    }

}
```

Create main class named "FunctionalIntefaceMain"

```
public class FunctionalIntefaceMain {

    public static void main(String[] args){

        FunctionalIntefaceMain pMain=new FunctionalIntefaceMain();

        pMain.printForm(() -> System.out.println("Printing form"));

    }

    public void printForm(Printable p){

        p.print();

    }

}
```

As you can see, since Printable has only one abstract method called print(), we were able to call it using lambda expression.

Q.8 What is method reference in Java 8?

→ Method reference is used refer method of functional interface. It is nothing but compact way of lambda expression. You can simply replace lambda expression with method reference.

Syntax: class::methodname

Q.9 **What is the default method, and why is it required?**

→ Default methods let you add new functionality to your libraries' interfaces and ensure binary compatibility with older coder written for the interfaces.

A method in the interface that has a predefined body is known as the default method. It uses the keyword default. default methods were introduced in Java 8 to have 'Backward Compatibility in case JDK modifies any interfaces. In case a new abstract method is added to the interface, all classes implementing the interface will break and will have to implement the new method. With default methods, there will not be any impact on the interface implementing classes. default methods can be overridden if needed in the implementation. Also, it does not qualify as synchronized or final.

```
@FunctionalInterface // Annotation is optional

public interface Foo() {

// Default Method - Optional can be 0 or more

public default String HelloWorld() {

return "Hello World";

}

// Single Abstract Method

public void bar();

}
```

Q.10 What's the difference between findFirst() and findAny()?

→findFirst() returns the first element meeting the criterion, while findAny() returns any element meeting the standard, a feature that is very useful when working with a parallel stream.

Q.11 Describe the newly added features in Java 8?

→Here are the newly added features of Java 8:

Feature Name	Description
Lambda expression	A function that can be shared or referred to as an object.
Functional Interfaces	Single abstract method interface.
Method References	Uses function as a parameter to invoke a method.
Default method	It provides an implementation of methods within interfaces enabling 'Interface evolution' facilities.
Stream API	Abstract layer that provides pipeline processing of the data.
Date Time API	New improved joda-time inspired APIs to overcome the drawbacks in previous versions
Optional	Wrapper class to check the null values and helps in further processing based on the value.
Nashorn, JavaScript Engine	An improvised version of JavaScript Engine that enables JavaScript executions in Java, to replace Rhino.

Q.12 In which programming paradigm Java 8 falls?

•Object-oriented programming language.

•Functional programming language.

•Procedural programming language.

•Logic programming language

Q.13 What is MetaSpace? How does it differ from PermGen?

PremGen: MetaData information of classes was stored in PremGen (PermanentGeneration) memory type before Java 8. PremGen is fixed in size and cannot be dynmically resized. It was a contiguous Java Heap Memory.

MetaSpace: Java 8 stores the MetaData of classes in native memory called 'MetaSpace'. It is not a contiguous Heap Memory and hence can be grown dynamically which helps to overcome the size constraints. This improves the garbage collection, auto-tuning, and de-allocation of metadata.

Q.14 What are functional or SAM interfaces?

→ Functional Interfaces are an interface with only one abstract method. Due to which it is also known as the Single Abstract Method (SAM) interface. It is known as a functional interface because it wraps a function as an interface or in other words a function is represented by a single abstract method of the interface.

Functional interfaces can have any number of default, static, and overridden methods. For declaring Functional Interfaces @FunctionalInterface annotation is optional to use. If this annotation is used for interfaces with more than one abstract method, it will generate a compiler error.

```
@FunctionalInterface // Annotation is optional

public interface Foo() {

// Default Method - Optional can be 0 or more

public default String HelloWorld() {

return "Hello World";

}

// Static Method - Optional can be 0 or more

public static String CustomMessage(String msg) {

return msg;
```

```
}
// Single Abstract Method
public void bar();
}


public class FooImplementation implements Foo {
// Default Method - Optional to Override
@Override
public default String HelloWorld() {
return "Hello Java 8";
}
// Method Override
@Override
public void bar() {
                System.out.println(“Hello World”);
}
}
public static void main(String[] args) {
FooImplementation fi = new FooImplementation();
System.out.println(fi.HelloWorld());
System.out.println(fi.CustomMessage(“Hi”));
fi.bar();
}
```

Q.15 Can a functional interface extend/inherit another interface?

→ A functional interface cannot extend another interface with abstract methods as it will void the rule of one abstract method per functional interface. E.g:

```
interface Parent {
public int parentMethod();
```

```
}

@FunctionalInterface // This cannot be FunctionalInterface

interface Child extends Parent {

public int childMethod();

// It will also extend the abstract method of the Parent Interface

// Hence it will have more than one abstract method

// And will give a compiler error

}
```

It can extend other interfaces which do not have any abstract method and only have the default, static, another class is overridden, and normal methods. For eg:

```
interface Parent {

public void parentMethod(){

System.out.println("Hello");

}

}

@FunctionalInterface

interface Child extends Parent {

public int childMethod();

}
```

Q.16 What are static methods in Interfaces?

→ Static methods, which contains method implementation is owned by the interface and is invoked using the name of the interface, it is suitable for defining the utility methods and cannot be overridden.

Q.17 What are some standard Java pre-defined functional interfaces?

Some of the famous pre-defined functional interfaces from previous Java versions are Runnable, Callable, Comparator, and Comparable. While Java 8 introduces functional interfaces like Supplier, Consumer, Predicate, etc. Please refer to the java.util.function doc for other predefined functional interfaces and its description introduced in Java 8.

Runnable: use to execute the instances of a class over another thread with no arguments and no return value.

Callable: use to execute the instances of a class over another thread with no arguments and it either returns a value or throws an exception.

Comparator: use to sort different objects in a user-defined order

Comparable: use to sort objects in the natural sort order

Q.18 What are the various categories of pre-defined function interfaces?

Function: To transform arguments in returnable value.

Predicate: To perform a test and return a Boolean value.

Consumer: Accept arguments but do not return any values.

Supplier: Do not accept any arguments but return a value.

Operator: Perform a reduction type operation that accepts the same input types.

Q.19 What is the basic structure/syntax of a lambda expression?

```
FunctionalInterface fi = (String name) -> {

System.out.println("Hello "+name);

return "Hello "+name;

}
```

Lambda expression can be divided into three distinct parts as below:

1. List of Arguments/Params:

(String name)

A list of params is passed in () round brackets. It can have zero or more params. Declaring the type of parameter is optional and can be inferred for the context.

2. Arrow Token:

->
Arrow token is known as the lambda arrow operator. It is used to separate the

parameters from the body, or it points the list of arguments to the body. 3. Expression/Body:

```
{

System.out.println("Hello "+name);

return "Hello "+name;

}
```

A body can have expressions or statements. {} curly braces are only required when there is more than one line. In one statement, the return type is the same as the return type of the statement. In other cases, the return type is either inferred by the return keyword or void if nothing is returned.

Q.20 What are the features of a lambda expression?

→Below are the two significant features of the methods that are defined as the lambda expressions:

- Lambda expressions can be passed as a parameter to another method.
- Lambda expressions can be standalone without belonging to any class.

Q.21 What is a type interface?

→ Type interface is available even in earlier versions of Java. It is used to infer the type of argument by the compiler at the compile time by looking at method invocation and corresponding declaration.

Q.22 What are the types and common ways to use lambda expressions?

→A lambda expression does not have any specific type by itself. A lambda expression receives type once it is assigned to a functional interface. That same lambda expression can be assigned to different functional interface types and can have a different type.

For eg consider expression s -> s.isEmpty() :

```
Predicate<String> stringPredicate = s -> s.isEmpty();
Predicate<List> listPredicate = s -> s.isEmpty();
```

```
Function<String, Boolean> func = s -> s.isEmpty();
Consumer<String> stringConsumer = s -> s.isEmpty();
```

Common ways to use the expression

Assignment to a functional Interface —> Predicate<String> stringPredicate = s -> s.isEmpty();
Can be passed as a parameter that has a functional type —> stream.filter(s -> s.isEmpty())
Returning it from a function —> return s -> s.isEmpty()
Casting it to a functional type —> (Predicate<String>) s -> s.isEmpty()

Q.23 What does the String::ValueOf expression mean?

It is a static method reference to method Valueof() of class String. It will return the string representation of the argument passed.

Q.24 What is an Optional class?

Optional is a container type which may or may not contain value i.e. zero(null) or one(not-null) value. It is part of java.util package. There are pre-defined methods like isPresent(), which returns true if the value is present or else false and the method get(), which will return the value if it is present.

```
static Optional<String> changeCase(String word) {

if (name != null && word.startsWith("A")) {

   return Optional.of(word.toUpperCase());

  }

else {

return Optional.ofNullable(word); // someString can be null

}

}
```

Q.25 What are the advantages of using the Optional class?

→Below are the main advantage of using the Optional class:

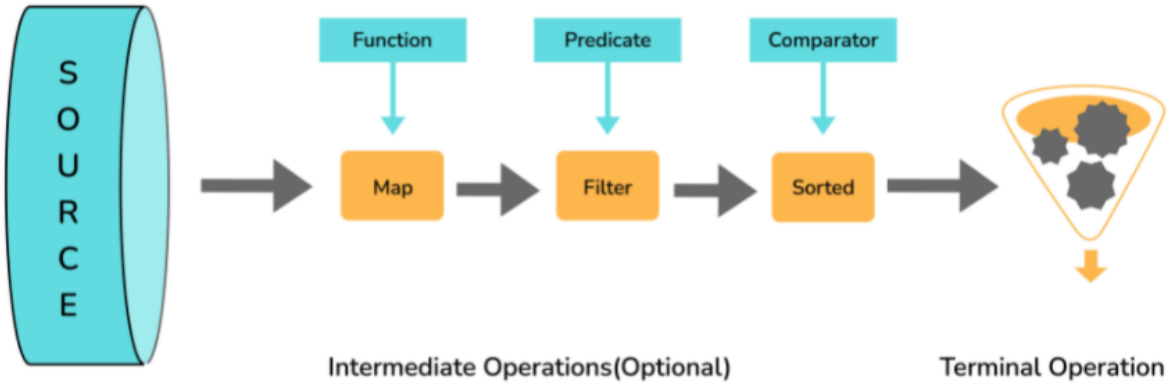
It encapsulates optional values, i.e., null or not-null values, which helps in avoiding null checks, which results in better, readable, and robust code It acts as a wrapper

around the object and returns an object instead of a value, which can be used to avoid run-time NullPointerExceptions.

Q.26 What are the main components of a Stream?

→ Components of the stream are:

- A data source
- Set of Intermediate Operations to process the data source
- Single Terminal Operation that produces the result



Q.27 What are the sources of data objects a Stream can process?

→A Stream can process the following data:

- A collection of an Array.
- An I/O channel or an input device.
- A reactive source (e.g., comments in social media or tweets/re-tweets)
- A stream generator function or a static factory.

Q.28 What are Intermediate and Terminal operations?

Intermediate Operations:

- Process the stream elements.
- Typically transforms a stream into another stream.
- Are lazy, i.e., not executed till a terminal operation is invoked.
- Does internal iteration of all source elements.
- Any number of operations can be chained in the processing pipeline.
- Operations are applied as per the defined order.

- Intermediate operations are mostly lambda functions.

Terminal Operations:

- Kick-starts the Stream pipeline.
- used to collect the processed Stream data.

```
int count = Stream.of(1, 2, 3, 4, 5)

.filter(i -> i <4) // Intermediate Operation filter

.count(); // Terminal Operation count
```

Q.29 What are the most commonly used Intermediate operations?

Filter(Predicate<T>) - Allows selective processing of Stream elements. It returns elements that are satisfying the supplied condition by the predicate.

map(Funtion<T, R>) - Returns a new Stream, transforming each of the elements by applying the supplied mapper function.= sorted() - Sorts the input elements and then passes them to the next stage.

distinct() - Only pass on elements to the next stage, not passed yet.

limit(long maxsize) - Limit the stream size to maxsize.

skip(long start) - Skip the initial elements till the start.

peek(Consumer) - Apply a consumer without modification to the stream.

flatMap(mapper) - Transform each element to a stream of its constituent elements and flatten all the streams into a single stream.

Q.30 What is the stateful intermediate operation? Give some examples of stateful intermediate operations.

To complete some of the intermediate operations, some state is to be maintained, and such intermediate operations are called stateful intermediate operations. Parallel execution of these types of operations is complex.

For Eg: sorted() , distinct() , limit() , skip() etc.

Sending data elements to further steps in the pipeline stops till all the data is sorted for sorted() and stream data elements are stored in temporary data structures.

Q.31 What is the most common type of Terminal operations?

- collect() - Collects single result from all elements of the stream sequence.
- reduce() - Produces a single result from all elements of the stream sequence
- count() - Returns the number of elements on the stream.
- min() - Returns the min element from the stream.
- max() - Returns the max element from the stream.
- Search/Query operations
- anyMatch() , noneMatch() , allMatch() , ... - Short-circuiting operations.
- Takes a Predicate as input for the match condition.
- Stream processing will be stopped, as and when the result can be determined.
- Iterative operations
- forEach() - Useful to do something with each of the Stream elements. It accepts a consumer.
- forEachOrdered() - It is helpful to maintain order in parallel streams.

Q.32 How are Collections different from Stream?

Collections are the source for the Stream. Java 8 collection API is enhanced with the default methods returning Stream<T> from the collections.

Collections	Streams
Data structure holds all the data elements	No data is stored. Have the capacity to process an infinite number of elements on demand
External Iteration	Internal Iteration
Can be processed any number of times	Traversed only once
Elements are easy to access	No direct way of accessing specific elements
Is a data store	Is an API to process the data

Q.33 What is the feature of the new Date and Time API in Java 8?

- Immutable classes and Thread-safe
- Timezone support
- Fluent methods for object creation and arithmetic
- Addresses I18N issue for earlier APIs

- Influenced by popular joda-time package
- All packages are based on the ISO-8601 calendar system

Q.34 What are the important packages for the new Data and Time API?

- java.time
- dates
- times
- Instants
- durations
- time-zones
- periods
- Java.time.format
- Java.time.temporal
- java.time.zone

Q.35 Explain with example, LocalDate, LocalTime, and LocalDateTime APIs.

LocalDate

- Date with no time component
- Default format - yyyy-MM-dd (2020-02-20)
- LocalDate today = LocalDate.now(); // gives today's date
- LocalDate aDate = LocalDate.of(2011, 12, 30); //(year, month, date)

LocalTime

- Time with no date with nanosecond precision
- Default format - hh:mm:ss:zzz (12:06:03.015) nanosecond is optional
- LocalTime now = LocalTime.now(); // gives time now
- LocalTime aTime2 = LocalTime.of(18, 20, 30); // (hours, min, sec)

LocalDateTime

- Holds both Date and Time
- Default format - yyyy-MM-dd-HH-mm-ss.zzz (2020-02-20T12:06:03.015)
- LocalDateTime timestamp = LocalDateTime.now(); // gives timestamp now
- //(year, month, date, hours, min, sec)
- LocalDateTime dt1 = LocalDateTime.of(2011, 12, 30, 18, 20, 30);

Q.36 Define Nashorn in Java 8

Nashorn is a JavaScript processing engine that is bundled with Java 8. It provides better compliance with ECMA (European Computer Manufacturers Association) normalized JavaScript specifications and better performance at run-time than older versions.

Q.37 What is the use of JJS in Java 8?

As part of Java 8, JJS is a command-line tool that helps to execute the JavaScript code in the console. Below is the example of CLI commands:

```
JAVA>jjs
jjs> print("Hello, Java 8 - I am the new JJS!")
Hello, Java 8 - I am the new JJS!
jjs> quit()
>>
```

Q.38Are you aware of Date and Time API introduced in Java 8? What the issues with Old Date and time API?

→Issues with old Date and TIme API:

Thread Safety: You might be already aware that java.util.Date is mutable and not thread safe. Even java.text.SimpleDateFormat is also not Thread-Safe. New Java 8 date and time APIs are thread safe.

Performance: Java 8 ‘s new APIs are better in performance than old Java APIs.

More Readable: Old APIs such Calendar and Date are poorly designed and hard to understand. Java 8 Date and Time APIs are easy to understand and comply with ISO standards.

Q.39 Can you provide some APIs of Java 8 Date and TIme?

LocalDate, **LocalTime**, and **LocalDateTime** are the Core API classes for Java 8. As the name suggests, these classes are local to context of observer. It denotes current date and time in context of Observer.

Q.40 How will you get current date and time using Java 8 Date and TIme API?

You can simply use **now** method of LocalDate to get today's date.

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


System.out.println(currentDate);
```

You can use **now** method of LocalTime to get current time.

Q.41What is use of Optional in Java 8?

→ Optional class is a special wrapper class introduced in Java 8 which is used to avoid NullPointerExceptions. This final class is present under java.util package. NullPointerExceptions occurs when we fail to perform the Null checks.

Q.42 What is predicate function interface?

→Predicate is single argument function which returns true or false. It has test method which returns boolean.

Q.43 What is consumer function interface?

→Consumer is single argument functional interface which does not return any value. Consumer Functional Interface is also a single argument interface (like Predicate<T> and Function<T, R>). It comes under java.util.function.Consumer. This does not return any value.
In the below program, we have made use of the accept method to retrieve the value of the String object

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

public class Java8 {

public static void main(String[] args) {

Consumer<String> str = str1 -> System.out.println(str1);

str.accept("Saket");

/* We have used accept() method to get the

  value of the String Object

 */

}

}
```

Q.44What is supplier function interface?

→Supplier is function interface which does not take any parameter but returns the value using get method. Supplier Functional Interface does not accept input parameters. It comes under java.util.function.Supplier. This returns the value using the get method.

In the below program, we have made use of the get method to retrieve the value of the String object.

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


public class Java8 {

public static void main(String[] args){

Supplier<String> str = () -> "Saket";

 System.out.println(str.get());

       /* We have used get() method to retrieve the

        value of String object str.

       */

     }

}
```

Q.45 **What was wrong with the old date and time?**

→Enlisted below are the drawbacks of the old date and time:

- Java.util.Date is mutable and is not thread-safe whereas the new Java 8 Date and Time API are thread-safe.
- Java 8 Date and Time API meets the ISO standards whereas the old date and time were poorly designed.
- It has introduced several API classes for a date like LocalDate, LocalTime, LocalDateTime, etc.
- Talking about the performance between the two, Java 8 works faster than the old regime of date and time.

Q.46 **What is the difference between the Collection API and Stream API?**
→The difference between the Stream API and the Collection API can be understood from the below table**:**

Stream API	**Collection API**
It was introduced in Java 8 Standard Edition version.	It was introduced in Java version 1.2
There is no use of the Iterator and Spliterators.	With the help of forEach, we can use the Iterator and Spliterators to iterate the elements and perform an actio each item or the element.
An infinite number of features can be stored.	A countable number of elements can be stored.
Consumption and Iteration of elements from the Stream object can be done only once.	Consumption and Iteration of elements from the Collect object can be done multiple times.
It is used to compute data.	It is used to store data.

Q.47 **Is there anything wrong with the following code? Will it compile or give any specific error?**

```
@FunctionalInterface
public interface Test<A, B, C> {
    public C apply(A a, B b);
```

```
    default void printString() {
        System.out.println("softwaretestinghelp");
    }
}
```

→ Yes. The code will compile because it follows the functional interface specification of defining only a single abstract method. The second method, printString(), is a default method that does not count as an abstract method.

Q.48 **What is the difference between limit and skip?**

→ The limit() method is used to return the Stream of the specified size. **For Example,** If you have mentioned limit(5), then the number of output elements would be 5.

Let's consider the following example. The output here returns six elements as the limit is set to 'six'.

```
import java.util.stream.Stream;

public class Java8 {

 public static void main(String[] args) {

   Stream.of(0,1,2,3,4,5,6,7,8).limit(6)

 /*limit is set to 6, hence it will print the numbers starting from 0 to 5

*/

      .forEach(num->System.out.print("\n"+num));
    }
}
```

Whereas, the skip() method is used to skip the element.

Let's consider the following example. In the output, the elements are 6, 7, 8 which means it has skipped the elements till the 6th index (starting from 1).

```
import java.util.stream.Stream;

public class Java8 {

 public static void main(String[] args) {

 Stream.of(0,1,2,3,4,5,6,7,8).skip(6)

 /* It will skip till 6th index. Hence 7th, 8th and 9th index elements will be

printed */

 .forEach(num->System.out.print("\n"+num));
    }
}
```

Q.49 What is the purpose of the limit() method in Java 8?

→The Stream.limit() method specifies the limit of the elements. The size that you specify in the limit(X), it will return the Stream of the size of 'X'. It is a method of java.util.stream.Stream

Syntax:

limit(X)

Where 'X' is the size of the element.

Q.50**What is the difference between Iterator and Spliterator?**

→ Below is the differences between Iterator and Spliterator.

Iterator	**Spliterator**
It was introduced in Java version 1.2	It was introduced in Java SE 8
It is used for Collection API.	It is used for Stream API.
Some of the iterate methods are next() and hasNext() which are used to iterate elements.	Spliterator method is tryAdvance().
We need to call the iterator() method on Collection Object.	We need to call the spliterator() met Stream Object.
Iterates only in sequential order.	Iterates in Parallel and sequential or

Q.51**What is the Difference Between Map and flatMap Stream Operation?**

→Map Stream operation gives one output value per input value whereas flatMap Stream operation gives zero or more output value per input value.

Map Example – Map Stream operation is generally used for simple operation on Stream such as the one mentioned below.

In this program, we have changed the characters of "Names" into the upper case using map operation after storing them in a Stream and with the help of the forEach Terminal operation, we have printed each element.

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

```
public class Map {
    public static void main(String[] str) {
        List<String> Names = Arrays.asList("Saket", "Trevor", "Franklin",
"Michael");

        List<String> UpperCase =
Names.stream().map(String::toUpperCase).collect(Collectors.toList());
        // Changed the characters into upper case after converting it into
Stream
  UpperCase.forEach(System.out::println);
        // Printed using forEach Terminal Operation
    }
}
```

flatMap Example – flatMap Stream operation is used for more complex Stream operation.
Here we have carried out flatMap operation on "List of List of type String". We have given input names as list and then we have stored them in a Stream on which we have filtered out the names which start with 'S'.

Finally, with the help of the forEach Terminal operation, we have printed each element.

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

public class flatMap {
    public static void main(String[] str) {
        List<List<String>> Names = Arrays.asList(Arrays.asList("Saket",
"Trevor"), Arrays.asList("John", "Michael"),
                Arrays.asList("Shawn", "Franklin"), Arrays.asList("Johnty",
"Sean"));

        /* Created a "List of List of type String" i.e. List<List<String>>
           Stored names into the list
         */

        List<String> Start = Names.stream().flatMap(FirstName ->
FirstName.stream()).filter(s -> s.startsWith("S"))
                .collect(Collectors.toList());

        /* Converted it into Stream and filtered
            out the names which start with 'S'
         */

        Start.forEach(System.out::println);

        /*
         Printed the Start using forEach operation
         */
    }
}
```

Q.52**What is the difference between Java 8 Internal and External Iteration?**
→ The difference between Internal and External Iteration is enlisted below.

Internal Iteration	External Iteration
It was introduced in Java 8 (JDK-8).	It was introduced and practiced in the previous versio Java (JDK-7, JDK-6 and so on).
It iterates internally on the aggregated objects such as Collection.	It iterates externally on the aggregated objects.
It supports the Functional programming style.	It supports the OOPS programming style.
Internal Iterator is passive.	External Iterator is active.
It is less erroneous and requires less coding.	It requires little more coding and it is more error-pron

Q.53 **What is ChronoUnits in Java 8?**

→ ChronoUnits is the enum that is introduced to replace the Integer values that are used in the old API for representing the month, day, etc.

Q.54**Explain StringJoiner Class in Java 8? How can we achieve joining multiple Strings using StringJoiner Class?**

→ In Java 8, a new class was introduced in the package java.util which was known as StringJoiner. Through this class, we can join multiple strings separated by delimiters along with providing prefix and suffix to them.
In the below program, we will learn about joining multiple Strings using StringJoiner Class. Here, we have “,” as the delimiter between two different strings. Then we have joined five different strings by adding them with the help of the add() method. Finally, printed the String Joiner.

```
import java.util.StringJoiner;

public class Java8 {
    public static void main(String[] args) {

        StringJoiner stj = new StringJoiner(",");
        // Separated the elements with a comma in between.

        stj.add("Saket");
        stj.add("John");
        stj.add("Franklin");
        stj.add("Ricky");
```

```
        stj.add("Trevor");

        // Added elements into StringJoiner “stj”

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

Programs

Next 7 questions will be based on below class.

```
public class Employee {
    private String name;
    private int age;

    public Employee(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
        public String toString()
    {
        return "Employee Name: "+name+" age: "+age;
    }
}
```

Q.1Given the list of employees, count number of employees with age 25?

→You can use combination of filter and count to find this.

```
List<Employee> employeeList = createEmployeeList();


long count = employeeList.stream()


.filter(e->e.getAge()>25)


.count();


System.out.println("Number of employees with age 25 are : "+count);
```

Q.2Given the list of employees, find the employee with name “Mary”

→It is again very simple logic, change the main function in above class as following.

```
List<Employee> employeeList = createEmployeeList();

        Optional<Employee> e1 = employeeList.stream()

                  .filter(e->e.getName().equalsIgnoreCase("Mary")).findAny();

        if(e1.isPresent())

            System.out.println(e1.get());
```

Q.3Given a list of employee, find maximum age of employee?

It is again very simple logic, change the main function in above class as following.

```
List<Employee> employeeList = createEmployeeList();

        OptionalInt max = employeeList.stream().

                          mapToInt(Employee::getAge).max();

        if(max.isPresent())

            System.out.println("Maximum age of Employee: "+max.getAsInt());
```

Q.4 Given a list of employees, sort all the employee on the basis of age? Use java 8 APIs only

You can simply use sort method of list to sort the list of employees.

```
List<Employee> employeeList = createEmployeeList();

        employeeList.sort((e1,e2)->e1.getAge()-e2.getAge());

        employeeList.forEach(System.out::println);
```

Q.5 Join the all employee names with “,” using java 8?

```
List<Employee> employeeList = createEmployeeList();

        List<String> employeeNames = employeeList
```

```
                                        .stream()

                                        .map(Employee::getName)

                                        .collect(Collectors.toList());

        String employeeNamesStr = String.join(",", employeeNames);

    System.out.println("Employees are: "+employeeNamesStr);
```

Output: Employees are: John,Martin,Mary,Stephan,Gary

Q.6 Given the list of employee, group them by employee name?

You can use Collections.groupBy() to group list of employees by employee name.

```
import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;

public class MaximumUsingStreamMain {

    public static void main(String args[]){

        List<Employee> employeeList = createEmployeeList();

        Map<String, List<Employee>> map = employeeList.stream()


.collect(Collectors.groupingBy(Employee::getName));

        map.forEach((name,employeeListTemp)->System.out.println("Name:
"+name+" ==>"+employeeListTemp));

    }

    public static List<Employee> createEmployeeList() {

        List<Employee> employeeList=new ArrayList<>();

        Employee e1=new Employee("John",21);

        Employee e2=new Employee("Martin",19);

        Employee e3=new Employee("Mary",31);

        Employee e4=new Employee("Mary",18);

        Employee e5=new Employee("John",26);



        employeeList.add(e1);
```

```
        employeeList.add(e2);

        employeeList.add(e3);

        employeeList.add(e4);

        employeeList.add(e5);



        return employeeList;

    }

}

Output: Name: John ==>[Employee Name: John age: 21, Employee Name: John age:
26] Name: Martin ==>[Employee Name: Martin age: 19] Name: Mary ==>[Employee
Name: Mary age: 31, Employee Name: Mary age: 18]
```

Q.7 Given the list of numbers, remove the duplicate elements from the list.
You can simply use stream and then collect it to set using Collections.toSet() method.

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class RemoveDuplicatesFromListMain {
    public static void main(String[] args)
    {
        Integer[] arr=new Integer[]{1,2,3,4,3,2,4,2};
        List<Integer> listWithDuplicates = Arrays.asList(arr);

        Set<Integer> setWithoutDups = listWithDuplicates.stream().collect(Col
lectors.toSet());
        setWithoutDups.forEach((i)->System.out.print(" "+i));
    }
}
```

You can use distinct as well to avoid duplicates as following.
change main method of above program as below.

```
Integer[] arr=new Integer[]{1,2,3,4,3,2,4,2};
        List<Integer> listWithDuplicates = Arrays.asList(arr);
List<Integer> listWithoutDups = listWithDuplicates.stream().distinct().collec
t(Collectors.toList());
        listWithoutDups.forEach((i)->System.out.print(" "+i));
```

Q.8 Given a list of numbers, square them and filter the numbers which are greater 10000 and then find average of them.(Java 8 APIs only)

You can use the map function to square the number and then filter to avoid numbers w hich are less than 10000.We will use average as terminating function in this case.

```
import java.util.Arrays;
import java.util.List;
import java.util.OptionalDouble;

public class RemoveDuplicatesFromListMain {
    public static void main(String[] args)
    {
        Integer[] arr=new Integer[]{100,24,13,44,114,200,40,112};
        List<Integer> list = Arrays.asList(arr);
        OptionalDouble average = list.stream()
                                .mapToInt(n->n*n)
                                .filter(n->n>10000)
                                .average();

        if(average.isPresent())
            System.out.println(average.getAsDouble());

    }
}
Output: 21846.666666666668
```

Q.9 Given a list of employees, you need to filter all the employee whose age is greater than 20 and print the employee names.(Java 8 APIs only)

You can simply do it using below statement.

```
List<String> employeeFilteredList = employeeList.stream()
                  .filter(e->e.getAge()>20)
                  .map(Employee::getName)
                  .collect(Collectors.toList());
```

Complete main program for above logic.

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

public class MaximumUsingStreamMain {
    public static void main(String args[])
    {
        List<Employee> employeeList = createEmployeeList();
        List<String> employeeFilteredList = employeeList.stream()
                  .filter(e->e.getAge()>20)
                  .map(Employee::getName)
                  .collect(Collectors.toList());

        employeeFilteredList.forEach((name)-> System.out.println(name));

    }
```

```
    public static List<Employee> createEmployeeList()
    {
        List<Employee> employeeList=new ArrayList<>();

        Employee e1=new Employee("John",21);
        Employee e2=new Employee("Martin",19);
        Employee e3=new Employee("Mary",31);
        Employee e4=new Employee("Stephan",18);
        Employee e5=new Employee("Gary",26);

        employeeList.add(e1);
        employeeList.add(e2);
        employeeList.add(e3);
        employeeList.add(e4);
        employeeList.add(e5);

        return employeeList;
    }
}
```

Q.10 Write a program to print 5 random numbers using forEach in Java 8?

The below program generates 5 random numbers with the help of forEach in Java 8. Y ou can set the limit variable to any number depending on how many random numbers you want to generate.

```
import java.util.Random;

class Java8 {
    public static void main(String[] args) {

        Random random = new Random();
        random.ints().limit(5).forEach(System.out::println);
        /* limit is set to 5 which means only 5 numbers will be printed
        with the help of terminal operation forEach
        */

    }
}
```

Q.11 Write a program to print 5 random numbers in sorted order using forEach in Java 8?

Answer: The below program generates 5 random numbers with the help of forEach in Java 8. You can set the limit variable to any number depending on how many random numbers you want to generate. The only thing you need to add here is the sorted() method.

```
import java.util.Random;

class Java8 {

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

```
        Random random = new Random();

        random.ints().limit(5).sorted().forEach(System.out::println);

        /* sorted() method is used to sort the output after

         terminal operation forEach

         */

    }

}
```

Q.12 Write a Java 8 program to get the sum of all numbers present in a list?

→ In this program, we have used ArrayList to store the elements. Then, with the help of the sum() method, we have calculated the sum of all the elements present in the ArrayList. Then it is converted to Stream and added each element with the help of mapToInt() and sum() methods.

```
import java.util.*;

class Java8 {

    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<Integer>();

        list.add(10);

        list.add(20);

        list.add(30);

        list.add(40);

        list.add(50);

        // Added the numbers into Arraylist

        System.out.println(sum(list));

    }

    public static int sum(ArrayList<Integer> list) {

        return list.stream().mapToInt(i -> i).sum();

        // Found the total using sum() method after

        // converting it into Stream

    }
```

```
}
```

Q.13 Write a Java 8 program to square the list of numbers and then filter out the numbers greater than 100 and then find the average of the remaining numbers?

Answer: In this program, we have taken an Array of Integers and stored them in a list. Then with the help of mapToInt(), we have squared the elements and filtered out the numbers greater than 100. Finally, the average of the remaining number (greater than 100) is calculated.

```
import java.util.Arrays;

import java.util.List;

import java.util.OptionalDouble;

public class Java8 {

    public static void main(String[] args) {

        Integer[] arr = new Integer[] { 100, 100, 9, 8, 200 };

        List<Integer> list = Arrays.asList(arr);

        // Stored the array as list

        OptionalDouble avg = list.stream().mapToInt(n -> n * n).filter(n -> n
> 100).average();

        /* Converted it into Stream and filtered out the numbers

            which are greater than 100. Finally calculated the average

        */

        if (avg.isPresent())

            System.out.println(avg.getAsDouble());

    }

}
```

Q.14 Write a Java 8 program to find the lowest and highest number of a Stream?

→ In this program, we have used min() and max() methods to get the highest and lowest number of a Stream. First of all, we have initialized a Stream that has Integers and with the help of the Comparator.comparing() method, we have compared the elements of the Stream.

When this method is incorporated with max() and min(), it will give you the highest and lowest numbers. It will also work when comparing the Strings.

```
import java.util.Comparator;

import java.util.stream.*;

public class Java8{

   public static void main(String args[]) {

    Integer highest = Stream.of(1, 2, 3, 77, 6, 5)

                        .max(Comparator.comparing(Integer::valueOf))

                        .get();


    /* We have used max() method with Comparator.comparing() method

       to compare and find the highest number

    */

    Integer lowest = Stream.of(1, 2, 3, 77, 6, 5)

                        .min(Comparator.comparing(Integer::valueOf))

                        .get();

    /* We have used max() method with Comparator.comparing() method

       to compare and find the highest number

    */

    System.out.println("The highest number is: " + highest);

    System.out.println("The lowest number is: " + lowest);

   }

}
```

Q.15 Write a Java 8 program to add prefix and suffix to the String?

→ In this program, we have “,” as the delimiter between two different strings. Also, we have given “(” and “)” brackets as prefix and suffix. Then five different strings are joined by adding them with the help of the add() method. Finally, printed the String Joiner.

```
import java.util.StringJoiner;

public class Java8 {
```

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


        StringJoiner stj = new StringJoiner(",", "(", ")");


        // Separated the elements with a comma in between.

        //Added a prefix "(" and a suffix ")"


        stj.add("Saket");

        stj.add("John");

        stj.add("Franklin");

        stj.add("Ricky");

        stj.add("Trevor");


        // Added elements into StringJoiner "stj"


        System.out.println(stj);

    }

}
```

Q.16 Write a Java 8 program to iterate a Stream using the forEach method?

→ In this program, we are iterating a Stream starting from "number = 2", followed by the count variable incremented by "1" after each iteration.
Then, we are filtering the number whose remainder is not zero when divided by the number 2. Also, we have set the limit as ? 5 which means only 5 times it will iterate. Finally, we are printing each element using forEach.

```
import java.util.stream.*;

public class Java8 {

    public static void main(String[] args){

        Stream.iterate(2, count->count+1)
```

```
        // Counter Started from 2, incremented by 1

        .filter(number->number%2==0)

        // Filtered out the numbers whose remainder is zero

        // when divided by 2

        .limit(5)

        // Limit is set to 5, so only 5 numbers will be printed

        .forEach(System.out::println);

    }
}
```

Q.17 Write a Java 8 program to sort an array and then convert the sorted array into Stream?

→ In this program, we have used parallel sort to sort an array of Integers. Then converted the sorted array into Stream and with the help of forEach, we have printed each element of a Stream.

```
import java.util.Arrays;

public class Java8 {

    public static void main(String[] args) {

        int arr[] = { 99, 55, 203, 99, 4, 91 };

        Arrays.parallelSort(arr);

        // Sorted the Array using parallelSort()


        Arrays.stream(arr).forEach(n -> System.out.print(n + " "));

        /* Converted it into Stream and then

           printed using forEach */

    }

}
```

Q.18 Write a Java 8 program to find the number of Strings in a list whose length is greater than 5?

→ In this program, four Strings are added in the list using add() method, and then with the help of Stream and Lambda expression, we have counted the strings who has a length greater than 5.

```
import java.util.ArrayList;

import java.util.List;

public class Java8 {

    public static void main(String[] args) {

        List<String> list = new ArrayList<String>();

        list.add("Saket");

        list.add("Saurav");

        list.add("Softwaretestinghelp");

        list.add("Steve");


        // Added elements into the List


        long count = list.stream().filter(str -> str.length() > 5).count();


        /* Converted the list into Stream and filtering out

           the Strings whose length more than 5

           and counted the length

           */

        System.out.println("We have " + count + " strings with length greater
than 5");

    }

}
```

Q.19 Write a Java 8 program to concatenate two Streams?

→ In this program, we have created two Streams from the two already created lists and then concatenated them using a concat() method in which two lists are passed as an argument. Finally, printed the elements of the concatenated stream.

```
import java.util.Arrays;
```

```
import java.util.List;

import java.util.stream.Stream;

public class Java8 {

    public static void main(String[] args) {

        List<String> list1 = Arrays.asList("Java", "8");

        List<String> list2 = Arrays.asList("explained", "through",
"programs");

        Stream<String> concatStream = Stream.concat(list1.stream(),
list2.stream());



        // Concatenated the list1 and list2 by converting them into Stream

        concatStream.forEach(str -> System.out.print(str + " "));

        // Printed the Concatenated Stream

    }

}
```

Q.20 How will you print count of empty strings in java 8?

→The following code segment prints a count of empty strings using filter

```
List<String> strings=Arrays.asList(“abc”, “”,”bc”,”efg”,”abcd”,””,”jkl”);

//get count of empty string

Int count=strings.stream().filter(string -> string.empty()).count();
```

~THANK YOU!!!