Java 8 New Features

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

Java 8 - March 18th 2014

Java 9 - September 22nd 2016

Java 10 - 2018

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

Java 8 New Features:

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

   Etc…..

Lambda (λ) Expression

- Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly this concepts started using in programming world. "LISP" is the first programming which uses Lambda Expression.
- The other languages which uses lambda expressions are:
  - C#.Net
  - C Objective
  - C
  - C++
  - Python
  - Ruby etc.
  - and finally in java also.

☀ The Main Objective of λLambda Expression is to bring benefits of functional programming into java.

What is Lambda Expression (λ):

Lambda Expression is just an anonymous(nameless) function. That means the function which doesn't have the name,return type and access modifiers.

Lambda Expression also known as anonymous functions or closures.

Ex: 1

```
                                 () ➔ {
public void m1() {   ⎫                 sop("hello");
     sop("hello");   ⎬              }
}                    ⎭           () → { sop("hello"); }
                                 ()→ sop("hello");
```

Ex:2

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

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

Ex: 3

```
public String str(String str) {  ⎫  (String str) → return str;
       return str;               ⎬          ⬇
}                                ⎭      (str) → str;
```

Conclusions:

1) A lambda expression can have zero or more number of parameters(arguments).
   Ex:
   () → sop("hello");
   (int a) → sop(a);
   (inta, int b) → return a+b;

2) Usually we can specify type of parameter.If the compiler expect the type based on the context then we can remove type. i.e., programmer is not required.
   Ex:
   (inta, int b) → sop(a+b);
   (a,b) → sop(a+b);

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

4) If zero number of parameters available then we have to use empty parameter [like ()].
   Ex:
   () → sop("hello");

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

Ex:

```
(int a) → sop(a);
        ⬇
(a)→ sop(a);
    ⬇
A → sop(a);
```

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

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

Functional Interfaces:

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

Ex:

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

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

Ex:

```
interface Interf {
        public abstract void m1();
        default void m2() {
                System.out.println (â€œhelloâ€  );
        }
}
```

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

Ex:

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

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

Ex:

```
@FunctionalInterface {
          public void m1();
          public void m2();
}
```

this code gives compilation error.

Inside FunctionalInterface 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

FunctionalInterface with respect to Inheritance:

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

Ex:

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

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

Ex:

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

No Compile Time Error

In the child interface we can't define any new abstract methods otherwise child interface won't be FunctionalInterface and if we are trying to use @FunctionalInterface annotation then compiler gives an error message.

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

Compiletime Error

Ex:

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

No compile time error

this's Normal interface so that code compiles without error

In the above example in both parent & child interface we can write any number of default methods and there are no restrictions.Restrictions are applicable only for abstract methods.

FunctionalInterface Vs Lambda Expressions:

Once we write Lambda expressions to invoke it's functionality, then FunctionalInterface is required. We can use FunctionalInterface reference to refer Lambda Expression.
Where ever FunctionalInterface concept is applicable there we can use Lambda Expressions

Ex:1

Without Lambda Expression

```
interface  Interf {
        public void methodOne() {}
        public class Demo implements Interface {
             public void methodOne() {
                    System.out.println( â€œ method one execution â€œ);
             }
             public class Test {
                  public static void main(String[] args) {
                         Interfi = new Demo();
                         i.methodOne();
                  }
}
```

Above code With Lambda expression

```
interface Interf {
        public void methodOne() {}
        class Test {
              public static void main(String[] args) {
                   Interfi = ()  → System.out.println("MethodOne Execution");
                   i.methodOne();
               }
}
```

Without Lambda Expression

```
interface Interf {
          public void sum(inta,int b);
}
class Demo implements Interf {
          public void sum(inta,int b) {
                 System.out.println("The sum:"  +(a+b));
          }
}
public class Test {
          public static void main(String[] args) {
                   Interfi = new Demo();
                   i.sum(20,5);
           }
}
```

Above code With Lambda Expression

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

Without Lambda Expressions

```
interface Interf {
         publicint square(int x);
}
class Demo implements Interf {
```

```
            public int square(int x) {
                return x*x;  OR  (int x) → x*x
            }
}
class Test {
        public static void main(String[] args) {
            Interfi = new Demo();
            System.out.println("The Square of 7 is: " +i.square(7));
        }
}
```

Above code with Lambda Expression

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

Without Lambda expression

```
class MyRunnable implements Runnable {
        public void main() {
            for(int i=0; i<10; i++) {
                System.out.println("Child  Thread");
            }
        }
}
class ThreadDemo {
        public static void main(String[] args) {
            Runnable r = new myRunnable();
            Thread t = new Thread(r);
            t.start();
            for(int i=0; i<10; i++) {
                System.out.println("Main Thread")
            }
        }
}
```

With Lambda expression

```
class ThreadDemo {
        public static void main(String[] args) {
                Runnable r = () → {
                        for(int i=0; i<10; i++) {
                                System.out.println("Child  Thread");
                        }
                };
                Thread t = new Thread(r);
                t.start();
                for(i=0; i<10; i++) {
                        System.out.println("Main Thread");
                }
        }
}
```

<u>Anonymous inner classes   vs Lambda Expressions</u>

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.

<u>Ex:</u> With anonymous inner class

```
class Test {
        public static void main(String[] args) {
                Thread t = new Thread(new Runnable() {
                        public void run() {
                                for(int i=0; i<10; i++) {
                                        System.out.println("Child  Thread");
                                }
                        }
                });
                t.start();
               for(int i=0; i<10; i++)
                    System.out.println("Main  thread");

        }
}
```

<u>With Lambda expression</u>

```
class Test {
          public static void main(String[] args) {
                  Thread t = new Thread(() → {
                                                        for(int i=0; i<10; i++) {
                                                              System.out.println("Child  Thread");
                                                        }
                 });
                 t.start();
                 for(int i=0; i<10; i++) {
                        System.out.println("Main  Thread");
                    }
            }
}
```

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(FunctionalInterface).
- Hence if anonymous inner class implements functionalinterface 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:

```
interface Interf {
            public void m1();
}
class Test {
          int x = 777;
          public void m2() {
                   Interfi = ()→ {
                             int x = 888;
                             System.out.println(x);  888
                             System.out.println(this.x);  777
            };
            i.m1();
        }
        public static void main(String[] args) {
            Test t = new Test();
            t.m2();
        }
}
```

☀ 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:

```
interface Interf {
        public void m1();
}
class Test {
        int x = 10;
        public void m2() {
                int y = 20;
                Interfi = () → {
                        System.out.println(x);  10
                        System.out.println(y); 20
                        x = 888;
                        y = 999; //CE
                };
                i.m1();
                y = 777;
        }
        public static void main(String[] args) {
                Test t = new Test();
                t.m2();
        }
}
```

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 concreateclasses	lambda expression can't extend Abstract and concreate 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 (FunctionalInterface)
Inside anonymous inner class we can Declare instance variables.	Inside lambda expression we can't Declare instance variables,whater the variables declare 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 (FuntionalInterface).
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 convert 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

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

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:__

```
interface Interf {
        default void m1() {
                System.out.println("Default Method");
        }
}
class Test implements Interf {
         public static void main(String[] args) {
                 Test t = new Test();
                 t.m1();
         }
}
```

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

__Note:__
We can't override object class methods as default methods inside interface otherwise we get compiletime error.

__Ex:__

```
interface Interf {
       default inthashCode() {
             return 10;
       }
}
```

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 compiletime error.

```
Eg 1:
interface Left {
        default void m1() {
                System.out.println("Left Default Method");
        }
}

Eg 2:
interface Right {
        default void m1() {
                System.out.println("Right Default Method");
        }
}

Eg 3:
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:

```
class Test implements Left, Right {
        public void m1() {
                System.out.println("Test Class Method"); OR Left.super.m1();
        }
        public static void main(String[] args) {
                Test t = new Test();
                t.m1();
        }
}
```

Differences between interface with default methods and abstract class

Eventhough we can add concrete methods in the form of default methods to the interface , it wont 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 1.8version onwards in addition to default methods we can write static methods also inside interface to define utility functions.
Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static methods.we should call interface static methods by using interface name.

Ex:

```
interface Interf {
        public static void sum(int a, int b) {
                System.out.println("The Sum:"+(a+b));
        }
}
class Test implements Interf {
         public static void main(String[] args) {
                Test t = new Test();
                t.sum(10, 20); //CE
                Test.sum(10, 20); //CE
                Interf.sum(10, 20);
        }
}
```

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

```
interface Interf {
        public static void m1() {}
}
class Test implements Interf {
        public static void m1() {}
}
```

It's valid but not overriding

Ex:2

```
interface Interf {
        public static void m1() {}
}
class Test implements Interf {
        public void m1() {}
}
```

This's valid but not overriding

Ex3:

```
class P {
        private void m1() {}
}
class C extends P {
        public void m1() {}
}
```

This's valid but not overriding

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

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

At the command prompt:
```
javac Interf.java
javaInterf
```

Predicates

A predicate is a function with a single argument and returns boolean value.
To implement predicate functions injava,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:

```
import java.util.function;
class Test {
        public static void main(String[] args) {
            predicate<Integer> p = I → (i>10);
            System.out.println(p.test(100));
            System.out.println(p.test(7));
            System.out.println(p.test(true));  //CE
        }
}
```

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

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

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

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

Predicate joining

It's possible to join predicates into a single predicate by using the following methods.

and()
or()
negate()

these are exactly same as logical AND ,OR complement operators

Ex:

```
import java.util.function.*;
class test {
        public static void main(string[] args) {
            int[] x = {0, 5, 10, 15, 20, 25, 30};
            predicate<integer> p1 = i->i>10;
            predicate<integer> p2=i -> i%2==0;
            System.out.println("The Numbers Greater Than 10:");
            m1(p1, x);
            System.out.println("The Even Numbers Are:");
            m1(p2, x);
            System.out.println("The Numbers Not Greater Than 10:");
            m1(p1.negate(), x);
            System.out.println("The  Numbers Greater Than 10 And Even Are:â€  );
            m1(p1.and(p2), x);
            System.out.println("The Numbers Greater Than 10 OR Even:â€  );
            m1(p1.or(p2), x);
        }
        public static void m1(predicate<integer>p, int[] x) {
            for(int x1:x) {
                if(p.test(x1))
                    System.out.println(x1);
                }
            }
 }
```

Function

Functions are exactly same as predicates except that functions can return any type of result but function should(can)return only one value and that value can be any type as per our requirement.
To implement functions oracle people introduced Function interface in 1.8version.
Function interface present in java.util.function package.
Functional interface contains only one method i.e., apply()

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

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

Ex:

```
import java.util.function.*;
class Test {
        public static void main(String[] args) {
                Function<String, Integer> f = s ->s.length();
                System.out.println(f.apply("Durga"));
                System.out.println(f.apply("Soft"));
        }
}
```

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

Differenece between predicate and function

Predicate	Function
To implement conditional checks We should go for predicate	To perform certain operation And to return some result we Should go for function.
Predicate can take one type Parameter which represents Input argument type. Predicate<T>	Function can take 2 type Parameters.first one represent Input argument type and Second one represent return Type. Function<T,R>
Predicate interface defines only one method called test()	Function interface defines only one Method called apply().
publicboolean test(T t)	public R apply(T t)
Predicate can return only boolean value.	Function can return any type of value

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

functionalInterface 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.
FunctionalInterface method and our specified method should have same argument types ,except this the remaining things like
returntype,methodname,modifiersetc are not required to match.

Syntax:
if our specified method is static method

Classname::methodName

if the method is instance method

Objref::methodName

FunctionalInterface can refer lambda expression and FunctionalInterface 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

```
class Test {
    public static void main(String[] args) {
            Runnable r = () → {
                            for(int i=0; i<=10; i++) {
                                    System.out.println("Child Thread");
                            }
                        };
            Thread t = new Thread(r);
            t.start();
            for(int i=0; i<=10; i++) {
                System.out.println("Main Thread");
            }
    }
}
```

With Method Reference

```
class Test {
     public static void m1() {
          for(int i=0; i<=10; i++) {
               System.out.println("Child Thread");
          }
 }
 public static void main(String[] args) {
         Runnable r = Test:: m1;
         Thread t = new Thread(r);
         t.start();
         for(int i=0; i<=10; i++) {
              System.out.println("Main Thread");
         }
 }
```

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

Ex:

```
interface Interf {
    public void m1(int i);
 }
class Test {
     public void m2(int i) {
            System.out.println("From Method Reference:"+i);
        }
        public static void main(String[] args) {
            Interf  f = I ->sop("From Lambda Expression:"+i);
            f.m1(10);
            Test t = new Test();
            Interf i1 = t::m2;
            i1.m1(20);
        }
}
```

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:

```
class Sample {
        private String s;
        Sample(String s) {
                this.s = s;
                System.out.println("Constructor  Executed:"+s);
        }
}
interface Interf {
        public Sample get(String s);
}
class Test {
        public static void main(String[] args) {
                Interf f = s -> new Sample(s);
                f.get("From Lambda Expression");
                Interf f1 = Sample :: new;
                f1.get("From Constructor Reference");
        }
}
```

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

Streams

To process objects of the collection, in 1.8 version Streams concept introduced.

What is the differences between java.util.streams and java.io streams?

java.util streams meant for processing objects from the collection. Ie, it represents a stream of objects from the collection but java.io streams meant for processing binary and character data with respect to file. i.e it represents stream of binary data or character data from the file .hence java.io streams and java.util streams both are different.

What is the difference between collection and stream?

if we want to represent a group of individual objects as a single entity then we should go for collection.
if we want to process a group of objects from the collection then we should go for streams.

we can create a stream object to the collection by using stream()method of Collection interface.
stream() method is a default method added to the Collection in 1.8 version.

default Stream stream()

Ex:
Stream s = c.stream();

Stream is an interface present in java.util.stream.
once we got the stream, by using that we can process objects of that collection.

we can process the objects in the following two phases

1.configuration
2.processing

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 booleancondition we should go for filter()method.

__Mapping:__
If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map () method of Stream interface.

public Stream map (Function f);

It can be lambda expression also

__Ex:__
Stream s = c.stream();
Stream s1 = s.map(i-> i+10);
Once we performed configuration we can process objects by using several methods.

2.Processing

processing by collect() method
Processing by count()method
Processing by sorted()method
Processing by min() and max() methods
forEach() method
toArray() method
Stream.of()method

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

__Ex:1__
To collect only even numbers from the array list

__Approach-1:__ without Streams

```
import java.util.*;
class Test {
        public static void main(String[] args) {
                ArrayList<Integer> l1 = new ArrayList<Integer>();
                for(int i=0; i<=10; i++) {
                    l1.add(i);
                }
                System.out.println(l1);
                ArrayList<Integer> l2 = new ArrayList<Integer>();
                for(Integer i:l1) {
                        if(i%2 == 0)
                                l2.add(i);
                }
                System.out.println(l2);
        }
}
```

<u>Approach-2:</u> With Streams

```
import java.util.*;
import java.util.stream.*;
class Test {
        public static void main(String[] args) {
                ArrayList<Integer> l1 = new ArrayList<Integer>();
                for(inti=0; i<=10; i++) {
                        l1.add(i);
                }
                System.out.println(l1);
                List<Integer> l2 = l1.stream().filter(i -> i%2==0).collect(Collectors.toList());
                System.out.println(l2);
        }
}
```

<u>Ex:</u> Program for map() and collect() Method

```
import java.util.*;
import java.util.stream.*;
class Test {
        public static void main(String[] args) {
                ArrayList<String> l = new ArrayList<String>();
                l.add("rvk"); l.add("rk"); l.add("rkv"); l.add("rvki"); l.add("rvkir");
                System.out.println(l);
                List<String> l2 = l.Stream().map(s ->s.toUpperCase()).collect(Collectors.toList());
                System.out.println(l2);
        }
}
```

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:

```
import java.util.*;
import java.util.stream.*;
class Test1 {
            public static void main(String[] args) {
                        ArrayList<Integer> l1 = new ArrayaList<Integer>();
                        l1.add(0); l1.add(15); l1.add(10); l1.add(5); l1.add(30); l1.add(25); l1.add(20);
                        System.out.println(l1);
                        ArrayList<Integer> l2=l1.stream().map(i-> i+10).collect(Collectors.toList());
                        System.out.println(l2);
                        long count = l1.stream().filter(i->i%2==0).count();
                        System.out.println(count);
                        List<Integer> l3=l1.stream().sorted().collect(Collectors.toList());
                        System.out.println(l3);
                        Comparator<Integer>  comp=(i1,i2)->i1.compareTo(i2);
                        List<Integer> l4=l1.stream().sorted(comp).collect(Collectors.toList());
                        System.out.println(l4);
                        Integer min=l1.stream().min(comp).get();
                        System.out.println(min);
                        Integer max=l1.stream().max(comp).get();
                        System.out.println(max);
                        l3.stream().forEach(i->sop(i));
                        l3.stream().forEach(System.out::  println);

         }
}
```

<u>VI. toArray() method</u>

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);
}
```

<u>VII. Stream.of()method</u>

we can also apply a stream for group of values and for arrays.

<u>Ex:</u>

```
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 upto 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.

```
import java.time.*;
public class DateTime {
    public static void main(String[] args) {
        LocalDate date = LocalDate.now();
        System.out.println(date);
        LocalTime time=LocalTime.now();
        System.out.println(time);
    }
}
```

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:

```
import java.time.*;
class Test {
        public static void main(String[] args) {
                LocalDate date = LocalDate.now();
                System.out.println(date);
                int dd = date.getDayOfMonth();
                int mm = date.getMonthValue();
                int yy = date.getYear();
                System.out.println(dd+"..."+mm+"..."+yy);
                System.out.printf("\n%d-%d-%d",dd,mm,yy);
    }
}
```

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

```
importjava.time.*;
class Test {
    public static void main(String[] args)   {
        LocalTime time = LocalTime.now();
        int h = time.getHour();
        int m = time.getMinute();
        int s = time.getSecond();
        int n = time.getNano();
        System.out.printf("\n%d:%d:%d:%d",h,m,s,n);
    }
}
```

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

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

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

We can represent a particular Date and Time by using LocalDateTime object as follows.
Ex:
LocalDateTime dt1=LocalDateTime.of(1995,Month.APRIL,28,12,45);
sop(dt1);

Ex:
LocalDateTime dt1=LocalDateTime.of(1995,04,28,12,45);
sop(dt1);

Sop("After six months:"+dt.plusMonths(6));
Sop("Before six months:"+dt.minusMonths(6));

<u>To Represent Zone:</u>
ZoneId object can be used to represent Zone.

Ex:

```
import java.time.*;
class ProgramOne {
        public static void main(String[] args) {
                ZoneId zone = ZoneId.systemDefault();
                System.out.println(zone);
        }
}
```

We can create ZoneId for a particular zone as follows
Ex:
ZoneId la = ZoneId.of("America/Los_Angeles");
ZonedDateTimezt = ZonedDateTime.now(la);
System.out.println(zt);

Period Object:

Period object can be used to represent quantity of time

Ex:

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1989,06,15);
Period p = Period.between(birthday,today);
System.out.printf("age is %d year %d months %d days",p.getYears(),p.getMonths(),p.getDays());
```

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

```
import java.time.*;
public class Leapyear {
    int n = Integer.parseInt(args[0]);
    Year y = Year.of(n);
    if(y.isLeap())
            System.out.printf("%d is Leap year",n);
    else
            System.out.printf("%d is not Leap year",n);
}
```

Java 8 Features

Oracle released a new version of Java as Java 8 in March 18, 2014. It was a revolutionary release of the Java for software development platform. It includes various upgrades to the Java programming, JVM, Tools and libraries.

Features

- **Lambda expression** – Adds functional processing capability to Java.
- Method reference**s** – Referencing functions by their names instead of invoking them directly. Using functions as parameter.
- **Default method** – Interface to have default method implementation.
- **New tools** – New compiler tools and utilities are added like 'jdeps' to figure out dependencies.
- **Stream API** – New stream API to facilitate pipeline processing.
- **Date Time API** – Improved date time API.
- **Optional** – Emphasis on best practices to handle null values properly.
- **Nashorn, JavaScript Engine** – A Java-based engine to execute JavaScript code.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an anotation @*FunctionalInterface*, which is used to declare an interface as functional interface.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java Lambda Expression Syntax

```
(argument-list) -> {body}
```

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Without Lambda Expression

```java
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}
```

Test it Now

Output:

```
Drawing 10
```

Java Lambda Expression Example

Now, we are implementing the above example with the help of lambda expression.

```java
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
    };
    d2.draw();
  }
}
```

Stream API:

Collection->If we want to represent a group of objects as a single entity then we should go for collection.
Stream->If we want to process objects from the collection-stream.
Even Number:

```
package collection;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
public class Test {

	public static void main(String[] args) {
		ArrayList<Integer> list=new ArrayList<Integer>();

		list.add(0);
		list.add(5);
		list.add(10);
		list.add(15);
		list.add(20);
		list.add(25);

	List<Integer> l1=list.stream().filter(i->i%2==0).collect(Collectors.toList());
		System.out.println(l1);

	}

}
Output:
[0, 10, 20]
package collection;

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

public class Test {

	public static void main(String[] args) {
		ArrayList<Integer> marks=new ArrayList<Integer>();

		marks.add(0);
		marks.add(5);
		marks.add(10);
		marks.add(15);
		marks.add(20);
		marks.add(25);

List<Integer> updatedMarks=marks.stream().map(i->i+5).collect(Collectors.toList());
```

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

	}

}
Output:
[5, 10, 15, 20, 25, 30]
```

Where we need to use map?
For every object if we want new object by performing some operation then we should use map(mapping)
Filter(Predicate): To perform some conditional operation.(Boolean condition)
Map(Function): To perform some operation and generate some new value.(Can return any value)
Filter:
Input:10 element
Output: <=10 element
Map:
Input:10 element
Output: 10 element
stream()-> To get stream of collection objects.
count()-:

```
public class Test {

	public static void main(String[] args) {
		ArrayList<Integer> marks=new ArrayList<Integer>();

		marks.add(0);
		marks.add(5);
		marks.add(10);
		marks.add(15);
		marks.add(20);
		marks.add(25);

		long failedStudent=marks.stream().filter(i->i<15).count();
		System.out.println(failedStudent);

	}

}
```

Output:3
sorted();

```
package collection;

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

public class Test {

	public static void main(String[] args) {
		ArrayList<Integer> marks=new ArrayList<Integer>();

		marks.add(0);
		marks.add(20);
		marks.add(15);
```

```
        marks.add(10);
        marks.add(5);
        marks.add(25);

    List<Integer> list=marks.stream().sorted().collect(Collectors.toList());
        System.out.println(list);

    }

}
```

Output:

```
[0, 5, 10, 15, 20, 25]
```

Custom Sorting order:

```
public class Test {

    public static void main(String[] args) {
        ArrayList<Integer> marks=new ArrayList<Integer>();

        marks.add(0);
        marks.add(20);
        marks.add(15);
        marks.add(10);
        marks.add(5);
        marks.add(25);

List<Integer> list=marks.stream().sorted((i1,i2)->(i1<i2)?1:(i1>i2)?-
1:0).collect(Collectors.toList());
        System.out.println(list);

    }

}
```

Output: [25, 20, 15, 10, 5, 0]

Comparable→CompareTo→ default natural sorting order.
sorted() internally use compareTo
sorted ((i1,i2)->i1.compareTo(i2))→natural sorting order
sorted ((i1,i2)->-i1.compareTo(i2))→reverse sorting order
Comparator→compare

String sorting:

```
public class Test {

    public static void main(String[] args) {
        ArrayList<String> marks=new ArrayList<String>();

        marks.add("Sunny");
        marks.add("Deepak");
        marks.add("Ram");
        marks.add("Anurag");


        List<String> list=marks.stream().sorted().collect(Collectors.toList());
        System.out.println(list);
```

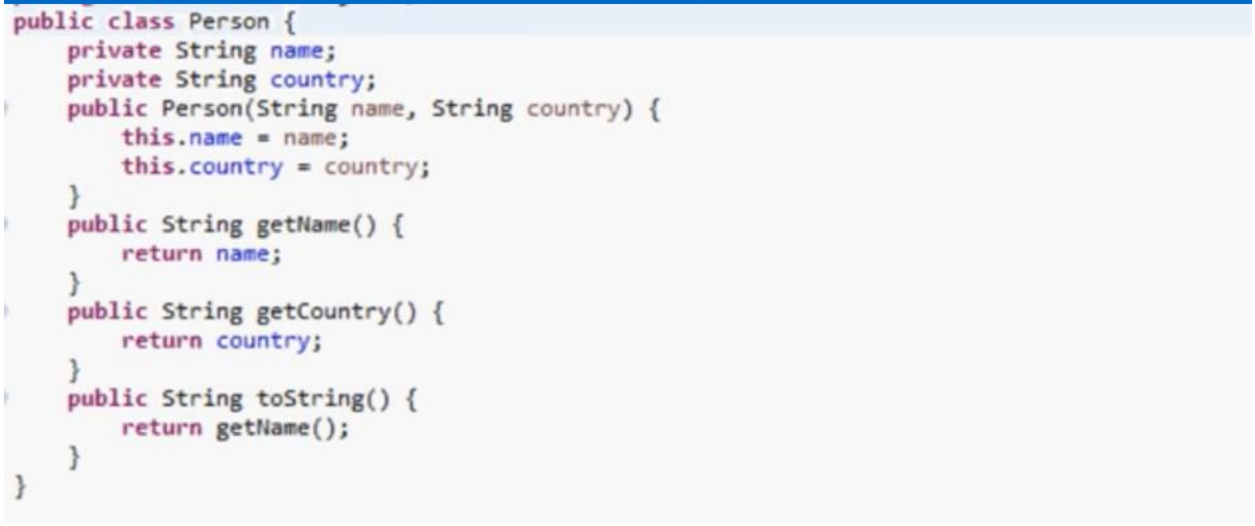
```
	}

}
Output: [Anurag, Deepak, Ram, Sunny]
public class Test {

	public static void main(String[] args) {
		ArrayList<String> marks=new ArrayList<String>();

		marks.add("Sunny");
		marks.add("Deepak");
		marks.add("Ram");
		marks.add("Anurag");


		List<String> list=marks.stream().sorted((s1,s2)-
>s2.compareTo(s1)).collect(Collectors.toList());
		System.out.println(list);

	}

}
Output: [Sunny, Ram, Deepak, Anurag]
package collection;

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

public class Test {

	public static void main(String[] args) {
		ArrayList<String> marks=new ArrayList<String>();

		marks.add("AAAAA");
		marks.add("AA");
		marks.add("A");
		marks.add("BBBB");
		marks.add("BBBBB");
		marks.add("BB");
		marks.add("B");
		marks.add("AAAA");


		List<String> list=marks.stream().sorted((s1,s2)->
		{
			int l1=s1.length();
			int l2=s2.length();
			if(l1<l2)
				return -1;
			else if(l1>l2)
			return 1;
			else
			return s1.compareTo(s2);
```

```
                    }).collect(Collectors.toList());
              System.out.println(list);

/* Comparator<String> c=(s1,s2)->
              {
                     int l1=s1.length();
                     int l2=s2.length();
                     if(l1<l2)
                            return -1;
                     else if(l1>l2)
                     return 1;
                     else
                     return s1.compareTo(s2);
                     };

              List<String> list=marks.stream().sorted(c).collect(Collectors.toList());
              System.out.println(list); */

       }

}
Output:
[A, B, AA, BB, AAAA, BBBB, AAAAA, BBBBB]


Min and Max value:
public class Test {

       public static void main(String[] args) {
              ArrayList<Integer> marks=new ArrayList<Integer>();

              marks.add(50);
              marks.add(35);
              marks.add(95);
              marks.add(88);
              marks.add(55);

       Integer max=marks.stream().max((i1,i2)->i1.compareTo(i2)).get();
       System.out.println(max);
       Integer min=marks.stream().min((i1,i2)->i1.compareTo(i2)).get();
       System.out.println(min);
       }

}
Output:
95
35
```

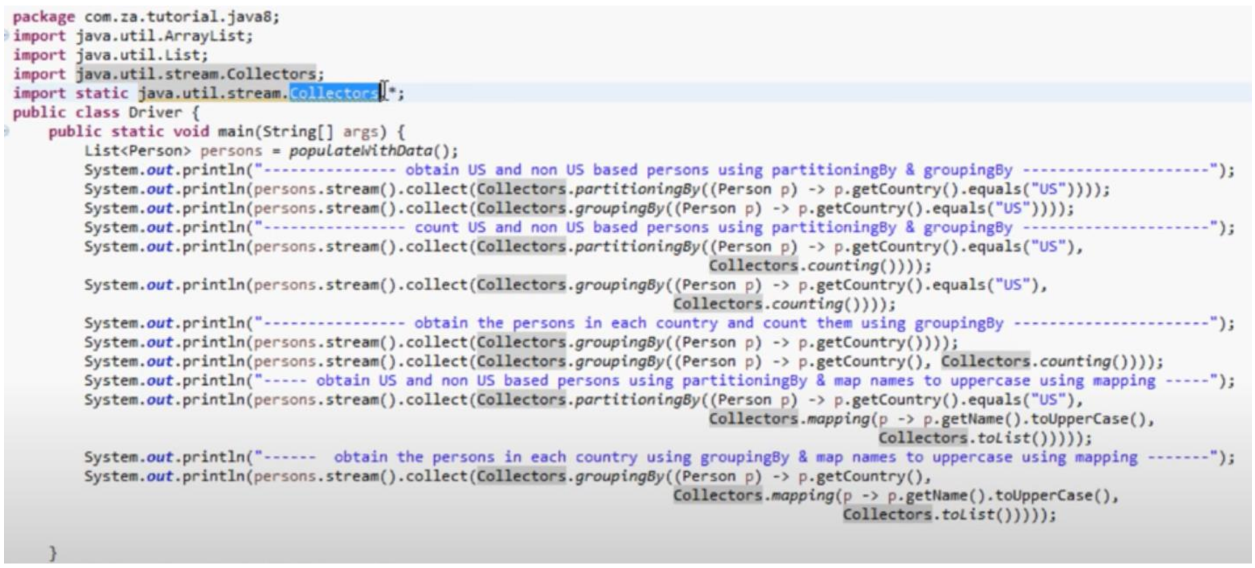
The complete program for sorting map by value in reverse order :

```
public class SortMapByValueExample {
    public static Map<String, Integer> sortByValue(final Map<String, Integer> wordCounts)
{

        return wordCounts.entrySet()
                .stream()
                .sorted((Map.Entry.<String, Integer>comparingByValue().reversed()))
                .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue, (e1,
e2) -> e1, LinkedHashMap::new));
    }

    public static void main(String[] args) {
        final Map<String, Integer> wordCounts = new HashMap<>();
        wordCounts.put("USA", 100);
        wordCounts.put("jobs", 200);
        wordCounts.put("software", 50);
        wordCounts.put("technology", 70);
        wordCounts.put("opportunity", 200);

        final Map<String, Integer> sortedByCount = sortByValue(wordCounts);

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

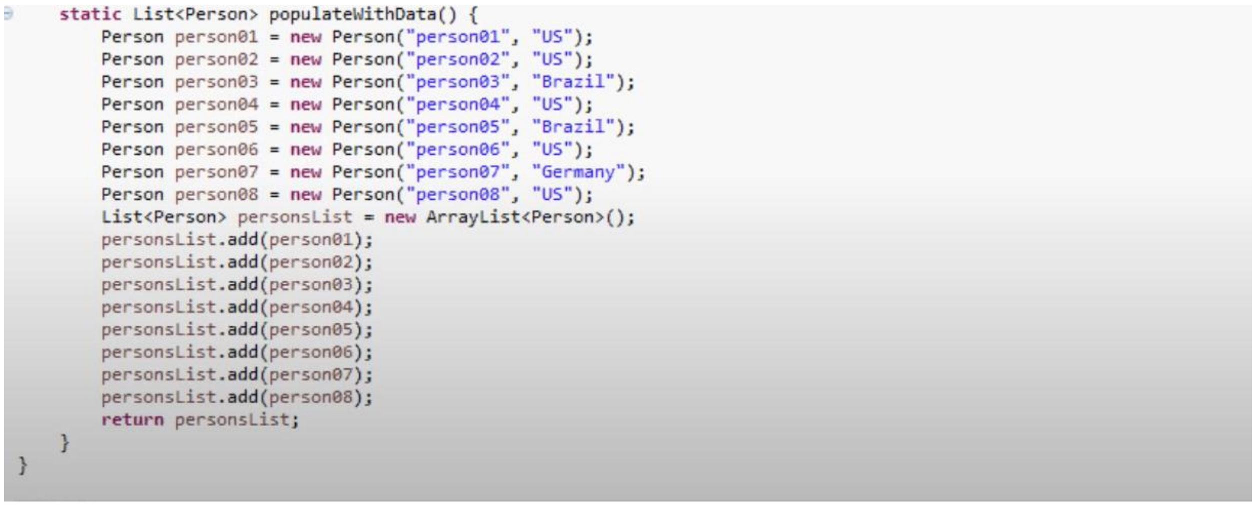
Java 8 Lambda - collect Stream method + groupingBy, partitioningBy, counting, & mapping Collectors methods



```
public class Person {
    private String name;
    private String country;
    public Person(String name, String country) {
        this.name = name;
        this.country = country;
    }
    public String getName() {
        return name;
    }
    public String getCountry() {
        return country;
    }
    public String toString() {
        return getName();
    }
}
```

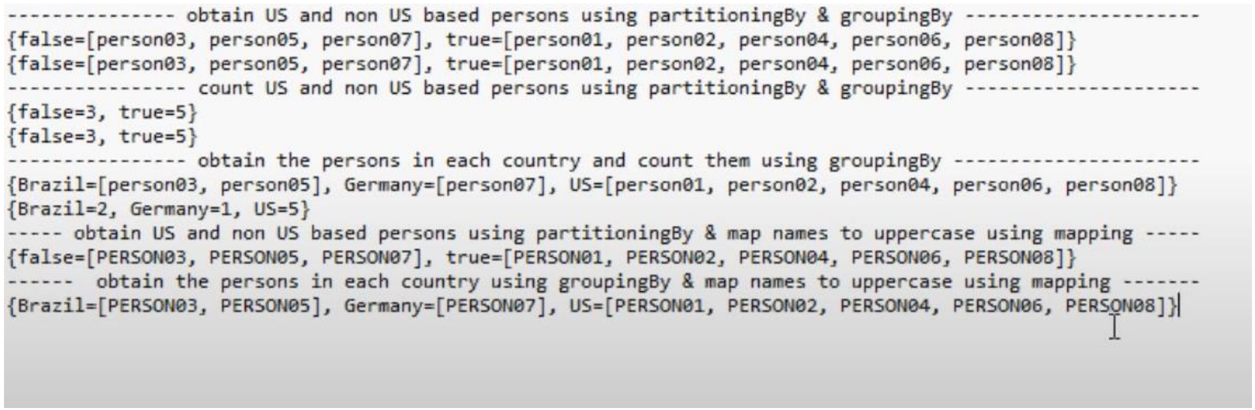


```java
package com.za.tutorial.java8;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import static java.util.stream.Collectors.*;
public class Driver {
    public static void main(String[] args) {
        List<Person> persons = populateWithData();
        System.out.println("--------------- obtain US and non US based persons using partitioningBy & groupingBy ---------------------");
        System.out.println(persons.stream().collect(Collectors.partitioningBy((Person p) -> p.getCountry().equals("US"))));
        System.out.println(persons.stream().collect(Collectors.groupingBy((Person p) -> p.getCountry().equals("US"))));
        System.out.println("---------------- count US and non US based persons using partitioningBy & groupingBy ---------------------");
        System.out.println(persons.stream().collect(Collectors.partitioningBy((Person p) -> p.getCountry().equals("US"),
                                                                              Collectors.counting())));
        System.out.println(persons.stream().collect(Collectors.groupingBy((Person p) -> p.getCountry().equals("US"),
                                                                          Collectors.counting())));
        System.out.println("---------------- obtain the persons in each country and count them using groupingBy ---------------------");
        System.out.println(persons.stream().collect(Collectors.groupingBy((Person p) -> p.getCountry())));
        System.out.println(persons.stream().collect(Collectors.groupingBy((Person p) -> p.getCountry(), Collectors.counting())));
        System.out.println("----- obtain US and non US based persons using partitioningBy & map names to uppercase using mapping -----");
        System.out.println(persons.stream().collect(Collectors.partitioningBy((Person p) -> p.getCountry().equals("US"),
                                                                              Collectors.mapping(p -> p.getName().toUpperCase(),
                                                                                                 Collectors.toList()))));
        System.out.println("------  obtain the persons in each country using groupingBy & map names to uppercase using mapping -------");
        System.out.println(persons.stream().collect(Collectors.groupingBy((Person p) -> p.getCountry(),
                                                                          Collectors.mapping(p -> p.getName().toUpperCase(),
                                                                                             Collectors.toList()))));

    }
```



```java
    static List<Person> populateWithData() {
        Person person01 = new Person("person01", "US");
        Person person02 = new Person("person02", "US");
        Person person03 = new Person("person03", "Brazil");
        Person person04 = new Person("person04", "US");
        Person person05 = new Person("person05", "Brazil");
        Person person06 = new Person("person06", "US");
        Person person07 = new Person("person07", "Germany");
        Person person08 = new Person("person08", "US");
        List<Person> personsList = new ArrayList<Person>();
        personsList.add(person01);
        personsList.add(person02);
        personsList.add(person03);
        personsList.add(person04);
        personsList.add(person05);
        personsList.add(person06);
        personsList.add(person07);
        personsList.add(person08);
        return personsList;
    }
}
```

OUTPUT:



```
--------------- obtain US and non US based persons using partitioningBy & groupingBy ---------------------
{false=[person03, person05, person07], true=[person01, person02, person04, person06, person08]}
{false=[person03, person05, person07], true=[person01, person02, person04, person06, person08]}
---------------- count US and non US based persons using partitioningBy & groupingBy ---------------------
{false=3, true=5}
{false=3, true=5}
---------------- obtain the persons in each country and count them using groupingBy ----------------------
{Brazil=[person03, person05], Germany=[person07], US=[person01, person02, person04, person06, person08]}
{Brazil=2, Germany=1, US=5}
----- obtain US and non US based persons using partitioningBy & map names to uppercase using mapping -----
{false=[PERSON03, PERSON05, PERSON07], true=[PERSON01, PERSON02, PERSON04, PERSON06, PERSON08]}
------  obtain the persons in each country using groupingBy & map names to uppercase using mapping -------
{Brazil=[PERSON03, PERSON05], Germany=[PERSON07], US=[PERSON01, PERSON02, PERSON04, PERSON06, PERSON08]}
```

Employee details by address city name using Java8:

```
package com;

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

class Address {

	private String name;
	private int pin;
	public String getName() {
		return name;
	}
	public int getPin() {
		return pin;
	}
	public Address(String name, int pin) {
		super();
		this.name = name;
		this.pin = pin;
	}
	@Override
	public String toString() {
		return "Address [name=" + name + ", pin=" + pin + "]";
	}

}

public class Employee {
	private String name;
	private  String department;
	private int empId;
	private Address address;

	public Employee(String name, String department, int empId, Address address) {
		super();
		this.name = name;
		this.department = department;
		this.empId = empId;
		this.address = address;
	}

	@Override
	public String toString() {
		return "Employee [name=" + name + ", department=" + department + ", empId="
+ empId + ", address=" + address
				+ "]";
	}

	public static void main(String[] args) {

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

		listEmployee.add(new Employee("JOHN", "dev", 12345, new
Address("UK",20001)));
		listEmployee.add(new Employee("RAJ", "dev", 12345, new
Address("USA",20002)));
             listEmployee.add(new Employee("RAM", "dev", 12345, new
Address("UK",20001)));
             listEmployee.add(new Employee("KUMAR", "dev", 12345, new
Address("USA",20002)));

             List<Employee> obj = listEmployee.stream().filter(a->
(a.address.getName().equals("UK"))).collect(Collectors.toList());
             System.out.println(obj);

      }

}
```

OUTPUT:

```
[Employee [name=JOHN, department=dev, empId=12345, address=Address [name=UK, pin=20001]],
Employee [name=RAM, department=dev, empId=12345, address=Address [name=UK, pin=20001]]]
```

Most Frequently Asked Java 8 Interview Questions

Q #1) List down the new features introduced in Java 8?

Answer: New features that are introduced in Java 8 are enlisted below:

- Lambda Expressions
- Method References
- Optional Class
- Functional Interface
- Default methods
- Nashorn, JavaScript Engine
- Stream API
- Date API

Q #2) What are Functional Interfaces?

Answer: Functional Interface is an interface that has only one abstract method. The implementation of these interfaces is provided using a Lambda Expression which means that to use the Lambda Expression, you need to create a new functional interface or you can use the predefined functional interface of Java 8.
The annotation used for creating a new Functional Interface is "**@FunctionalInterface**".

Q #3) What is an optional class?

Answer: 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 #4) What are the default methods?

Answer: Default methods are the methods of the Interface which has a body. These methods, as the name suggests, use the default keywords. The use of these default methods is "Backward Compatibility" which means if JDK modifies any Interface (without default method) then the classes which implement this Interface will break.
On the other hand, if you add the default method in an Interface then you will be able to provide the default implementation. This won't affect the implementing classes.

Syntax:

```
public interface questions{

        default void print() {
```

```
System.out.println("www.softwaretestinghelp.com");
              }
     }
```

Q #5) What are the main characteristics of the Lambda Function?

Answer: Main characteristics of the Lambda Function are as follows:

- A method that is defined as Lambda Expression can be passed as a parameter to another method.
- A method can exist standalone without belonging to a class.
- There is no need to declare the parameter type because the compiler can fetch the type from the parameter's value.
- We can use parentheses when using multiple parameters but there is no need to have parenthesis when we use a single parameter.
- If the body of expression has a single statement then there is no need to include curly braces.

Q #6) What was wrong with the old date and time?

Answer: 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 #7) What is the difference between the Collection API and Stream API?

Answer: 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 action on 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 Collection object can be done multiple times.
It is used to compute data.	It is used to store data.

Q #8) How can you create a Functional Interface?

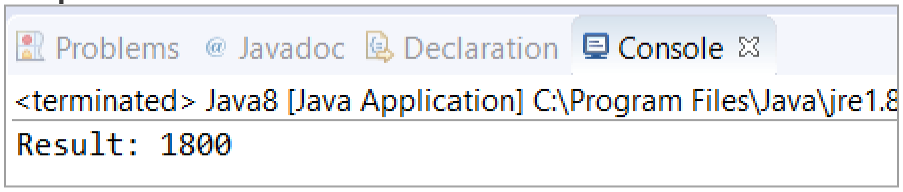
Answer: Although Java can identify a Functional Interface, you can define one with the annotation
@FunctionalInterface

Once you have defined the functional interface, you can have only one abstract method. Since you have only one abstract method, you can write multiple static methods and default methods.

Below is the programming example of FunctionalInterface written for multiplication of two numbers.

```
@FunctionalInterface // annotation for functional interface
interface FuncInterface {

    public int multiply(int a, int b);
}
public class Java8 {

   public static void main(String args[]) {
        FuncInterface Total = (a, b) -> a * b;
        // simple operation of multiplication of 'a' and 'b'
        System.out.println("Result: "+Total.multiply(30, 60));
    }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\Java\jre1.8
Result: 1800
```

Q #9) What is a SAM Interface?

Answer: Java 8 has introduced the concept of FunctionalInterface that can have only one abstract method. Since these Interfaces specify only one abstract method, they are sometimes called as SAM Interfaces. SAM stands for "Single Abstract Method".

Q #10) What is Method Reference?

Answer: In Java 8, a new feature was introduced known as Method Reference. This is used to refer to the method of functional interface. It can be used to replace Lambda Expression while referring to a method.

For Example: If the Lambda Expression looks like

num -> System.out.println(num)

Then the corresponding Method Reference would be,

System.out::println

where "::" is an operator that distinguishes class name from the method name.

Q #11) Explain the following Syntax

String:: Valueof Expression

Answer: It is a static method reference to the ***ValueOf*** method of the **String** class. System.out::println is a static method reference to println method of out object of System class. It returns the corresponding string representation of the argument that is passed. The argument can be Character, Integer, Boolean, and so on.

Q #12) What is a Predicate? State the difference between a Predicate and a Function?

Answer: Predicate is a pre-defined Functional Interface. It is under java.util.function.Predicate package. It accepts only a single argument which is in the form as shown below,
Predicate<T>

Predicate	Function
It has the return type as Boolean.	It has the return type as Object.
It is written in the form of **Predicate< T>** which accepts a single argument.	It is written in the form of **Function< T, R>** which also accepts a single argument.
It is a Functional Interface which is used to evaluate Lambda Expressions. This can be used as a target for a Method Reference.	It is also a Functional Interface which is used to evaluate Lambda Expressions. In Function< T, R>, T is for input type and R is for the result type. This can also be used as a target for a Lambda Expression and Method Reference.

Q #13) Is there anything wrong with the following code? Will it compile or give any specific error?

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

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

Answer: 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 #14) What is a Stream API? Why do we require the Stream API?

Answer: Stream API is a new feature added in Java 8. It is a special class that is used for processing objects from a source such as Collection.
We require the Stream API because,

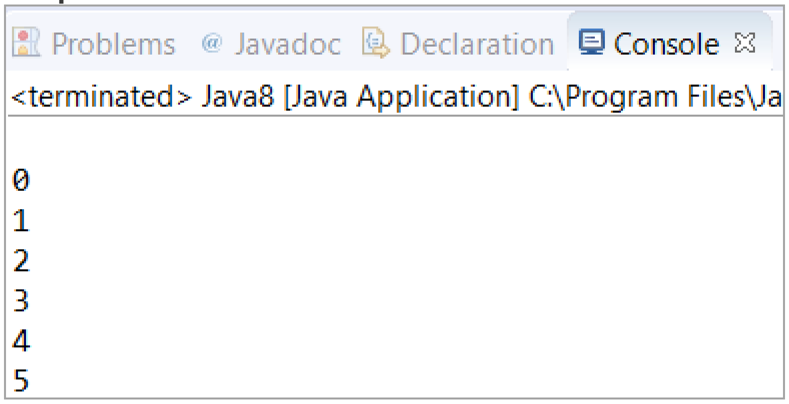
- It supports aggregate operations which makes the processing simple.
- It supports Functional-Style programming.
- It does faster processing. Hence, it is apt for better performance.
- It allows parallel operations.

Q #15) What is the difference between limit and skip?

Answer: 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));
    }
}
```

Output:



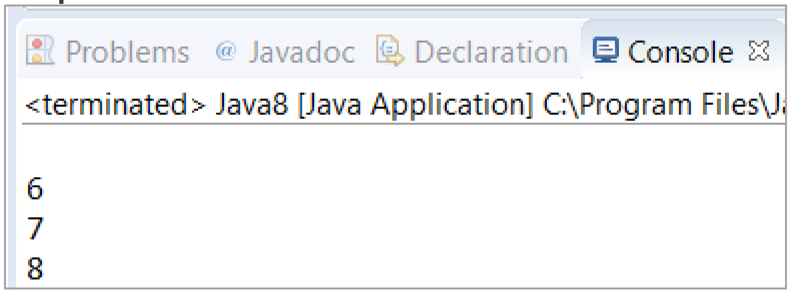
```
<terminated> Java8 [Java Application] C:\Program Files\Ja

0
1
2
3
4
5
```

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));
    }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\J

6
7
8
```

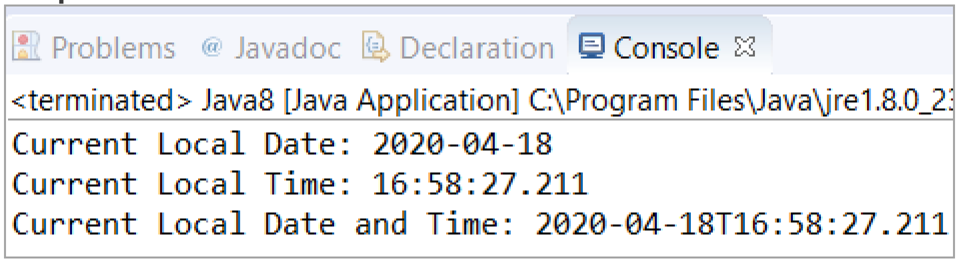
Q #16) How will you get the current date and time using Java 8 Date and Time API?

Answer: The below program is written with the help of the new API introduced in Java 8. We have made use of LocalDate, LocalTime, and LocalDateTime API to get the current date and time.

In the first and second print statement, we have retrieved the current date and time from the system clock with the time-zone set as default. In the third print statement, we have used LocalDateTime API which will print both date and time.

```
class Java8 {
    public static void main(String[] args) {
        System.out.println("Current Local Date: " + java.time.LocalDate.now());
        //Used LocalDate API to get the date
        System.out.println("Current Local Time: " + java.time.LocalTime.now());
        //Used LocalTime API to get the time
        System.out.println("Current Local Date and Time: " + java.time.LocalDateTime.now());
        //Used LocalDateTime API to get both date and time
    }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\Java\jre1.8.0_2
Current Local Date: 2020-04-18
Current Local Time: 16:58:27.211
Current Local Date and Time: 2020-04-18T16:58:27.211
```

Q #17) What is the purpose of the limit() method in Java 8?

Answer: 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)

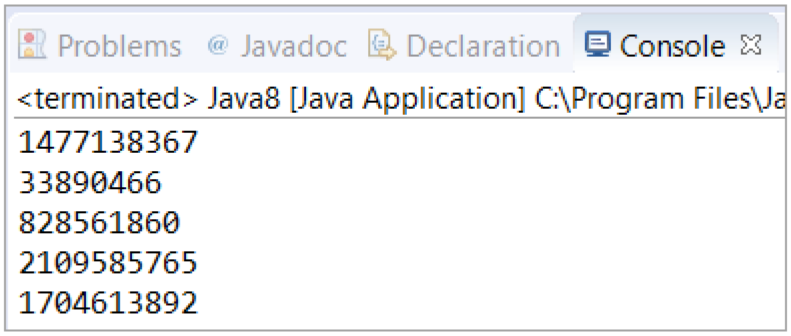
Where 'X' is the size of the element.

Q #18) Write a program to print 5 random numbers 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.

```
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
        */

    }
}
```

Output:



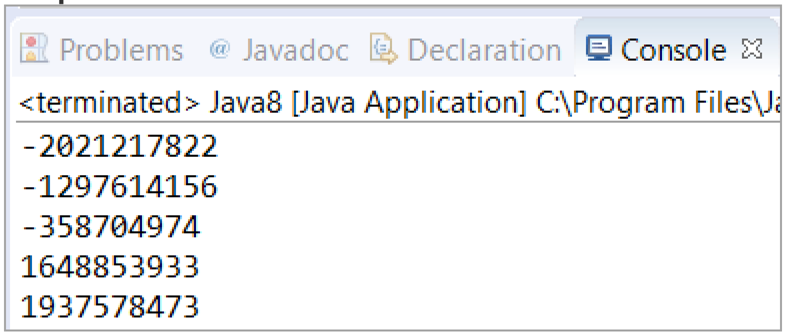
```
<terminated> Java8 [Java Application] C:\Program Files\Ja
1477138367
33890466
828561860
2109585765
1704613892
```

Q #19) 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
         */

    }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\Ja
-2021217822
-1297614156
-358704974
1648853933
1937578473
```

Q #20) What is the difference between Intermediate and Terminal Operations in Stream?

Answer: All Stream operations are either Terminal or Intermediate. Intermediate Operations are the operations that return the Stream so that some other operations can be carried out on that Stream. Intermediate operations do not process the Stream at the call site, hence they are called lazy.

These types of operations (Intermediate Operations) process data when there is a Terminal operation carried out.

Examples of **Intermediate operation are map and filter**.

Terminal Operations initiate Stream processing. During this call, the Stream undergoes all the Intermediate operations.

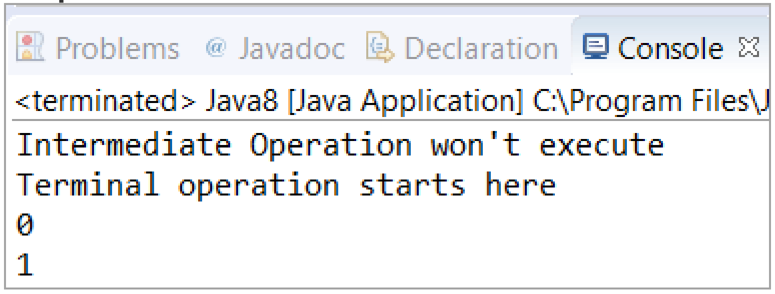
Examples of **Terminal Operation are sum, Collect, and forEach**.

In this program, we are first trying to execute Intermediate operation without Terminal operation. As you can see the first block of code won't execute because there is no Terminal operation supporting.

The second block successfully executed because of the Terminal operation sum().

```
import java.util.Arrays;

class Java8 {

    public static void main(String[] args) {
        System.out.println("Intermediate Operation won't execute");
        Arrays.stream(new int[] { 0, 1 }).map(i -> {
            System.out.println(i);
            return i;
            // No terminal operation so it won't execute
        });

        System.out.println("Terminal operation starts here");
        Arrays.stream(new int[] { 0, 1 }).map(i -> {
            System.out.println(i);
            return i;
            // This is followed by terminal operation sum()
        }).sum();
    }
}
```

Output:



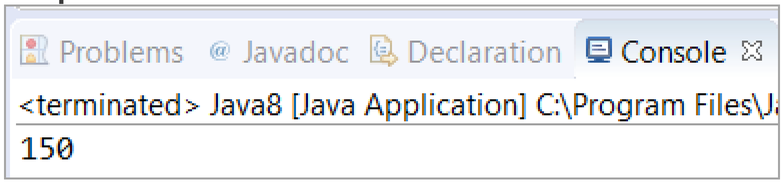
```
<terminated> Java8 [Java Application] C:\Program Files\J
Intermediate Operation won't execute
Terminal operation starts here
0
1
```

Q #21) Write a Java 8 program to get the sum of all numbers present in a list?

Answer: 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
    }
}
```

Output:



Problems @ Javadoc Declaration Console

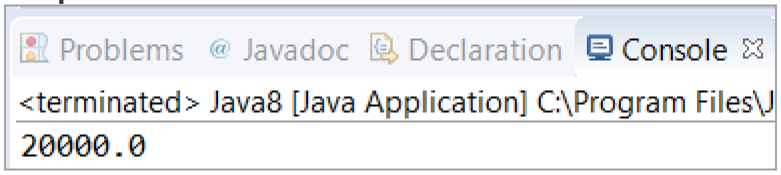
```
<terminated> Java8 [Java Application] C:\Program Files\J
150
```

Q #22) 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());
    }
}
```

Output:



Problems @ Javadoc Declaration Console

```
<terminated> Java8 [Java Application] C:\Program Files\J
20000.0
```

Q #23) What is the difference between Stream's findFirst() and findAny()?

Answer: As the name suggests, the findFirst() method is used to find the first element from the stream whereas the findAny() method is used to find any element from the stream.

The findFirst() is predestinarianism in nature whereas the findAny() is non-deterministic. In programming, Deterministic means the output is based on the input or initial state of the system.

Q #24) What is the difference between Iterator and Spliterator?

Answer: Below is the differences between Iterator and Spliterator.

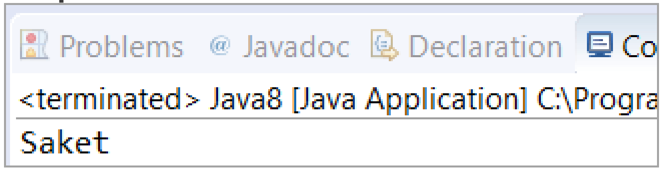
Iterator	Spliterator
It was introduced in Java version 1.2	It was introduced in Java SE 8
It is used for Collection API.	It is used for Stream API.
Some of the iterate methods are next() and hasNext() which are used to iterate elements.	Spliterator method is tryAdvance().
We need to call the iterator()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

Q #25) What is the Consumer Functional Interface?

Answer: 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
         */
    }
}
```

Output:



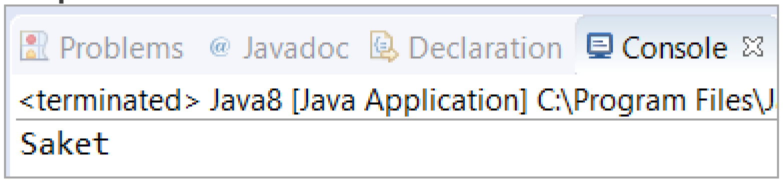
```
<terminated> Java8 [Java Application] C:\Progra
Saket
```

Q #26) What is the Supplier Functional Interface?

Answer: 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.
         */
    }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\J
Saket
```

Q #27) What is Nashorn in Java 8?

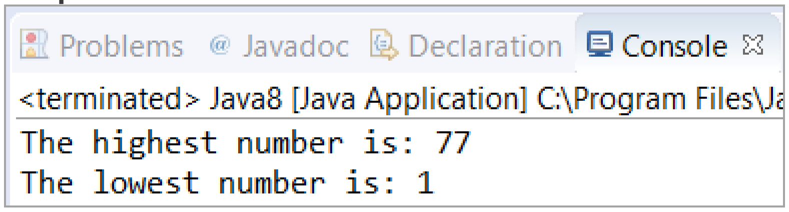
Answer: Nashorn in Java 8 is a Java-based engine for executing and evaluating JavaScript code.

Q #28) Write a Java 8 program to find the lowest and highest number of a Stream?

Answer: 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);
   }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\Ja
The highest number is: 77
The lowest number is: 1
```

Q #29) What is the Difference Between Map and flatMap Stream Operation?

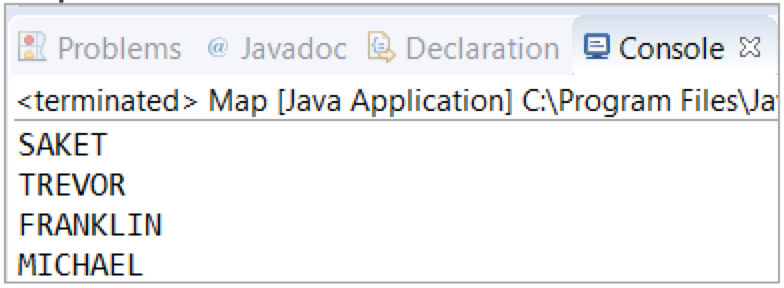
Answer: Map Stream operation gives one output value per input value whereas flatMap Stream operation gives zero or more output value per input value.

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
    }
}
```

Output:



```
<terminated> Map [Java Application] C:\Program Files\Ja
SAKET
TREVOR
FRANKLIN
MICHAEL
```

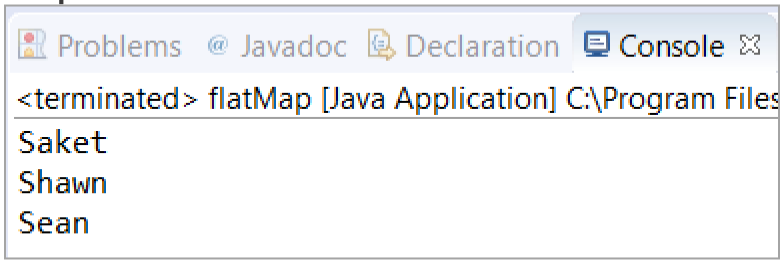
flatMap Example – flatMap Stream operation is used for more complex Stream operation.

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
         */
    }
}
```

Output:



```
<terminated> flatMap [Java Application] C:\Program Files
Saket
Shawn
Sean
```

Q #30) What is MetaSpace in Java 8?

Answer: In Java 8, a new feature was introduced to store classes. The area where all the classes that are stored in Java 8 are called MetaSpace. MetaSpace has replaced the PermGen. Till Java 7, PermGen was used by Java Virtual Machine to store the classes. Since MetaSpace is dynamic as it can grow dynamically and it does not have any size limitation, Java 8 replaced PermGen with MetaSpace.

Q #31) What is the difference between Java 8 Internal and External Iteration?

Answer: 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 version of 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-prone.

Q #32) What is JJS?

Answer: JJS is a command-line tool used to execute JavaScript code at the console. In Java 8, JJS is the new executable which is a JavaScript engine.

Q #33) What is ChronoUnits in Java 8?

Answer: 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 #34) Explain StringJoiner Class in Java 8? How can we achieve joining multiple Strings using StringJoiner Class?

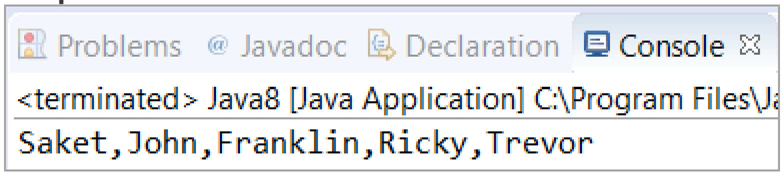
Answer: In Java 8, a new class was introduced in the package java.util which was known as StringJoiner. Through this class, we can join multiple strings separated by delimiters along with providing prefix and suffix to them.

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.

In the next question #35, you will learn about adding prefix and suffix to the string.

```
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);
    }
}
```

Output:



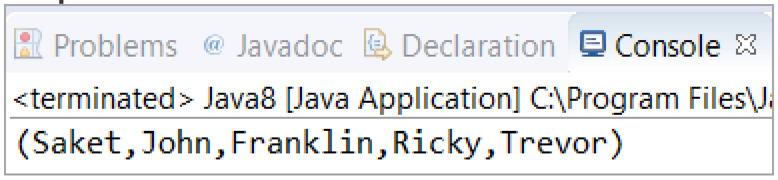
```
<terminated> Java8 [Java Application] C:\Program Files\J
Saket,John,Franklin,Ricky,Trevor
```

Q #35) Write a Java 8 program to add prefix and suffix to the String?

Answer: 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);
    }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\J
(Saket,John,Franklin,Ricky,Trevor)
```

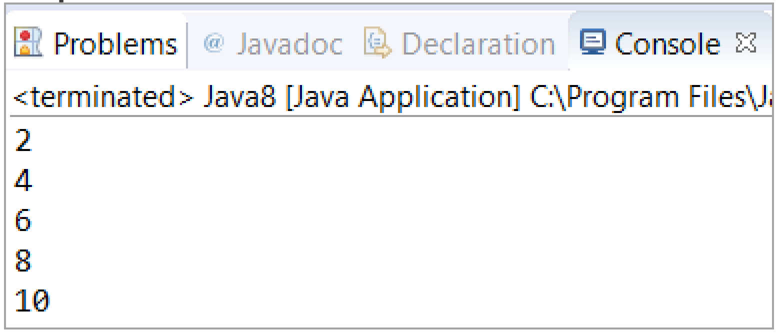
Q #36) Write a Java 8 program to iterate a Stream using the forEach method?

Answer: 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);
    }
}
```

Output:



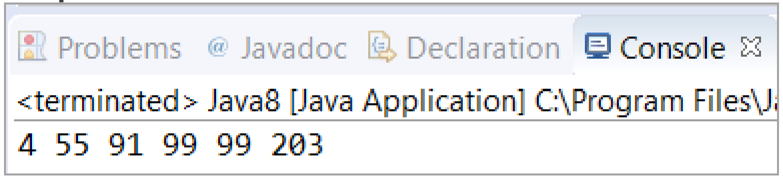
```
<terminated> Java8 [Java Application] C:\Program Files\J
2
4
6
8
10
```

Q #37) Write a Java 8 program to sort an array and then convert the sorted array into Stream?

Answer: 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 */
    }
}
```

Output:



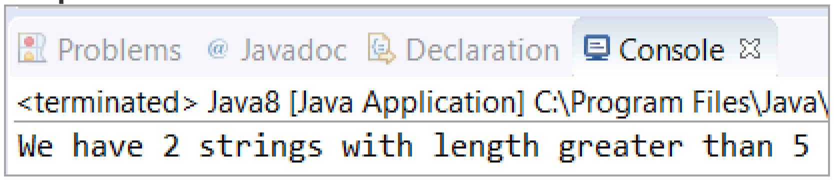
```
<terminated> Java8 [Java Application] C:\Program Files\J
4 55 91 99 99 203
```

Q #38) Write a Java 8 program to find the number of Strings in a list whose length is greater than 5?

Answer: 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");

    }
}
```

Output:



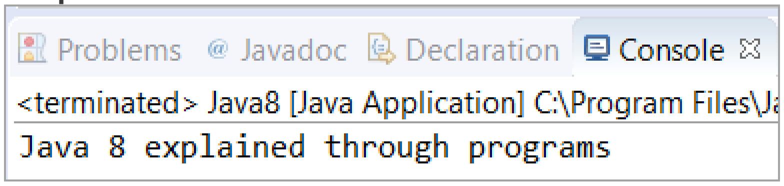
```
<terminated> Java8 [Java Application] C:\Program Files\Java\
We have 2 strings with length greater than 5
```

Q #39) Write a Java 8 program to concatenate two Streams?

Answer: 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

    }
}
```

Output:



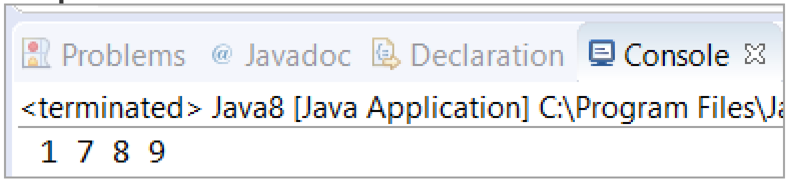
```
<terminated> Java8 [Java Application] C:\Program Files\J
Java 8 explained through programs
```

Q #40) Write a Java 8 program to remove the duplicate elements from the list?

Answer: In this program, we have stored the elements into an array and converted them into a list. Thereafter, we have used stream and collected it to “Set” with the help of the “Collectors.toSet()” method.

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

public class Java8 {
    public static void main(String[] args) {
        Integer[] arr1 = new Integer[] { 1, 9, 8, 7, 7, 8, 9 };
        List<Integer> listdup = Arrays.asList(arr1);

        // Converted the Array of type Integer into List

        Set<Integer> setNoDups = listdup.stream().collect(Collectors.toSet());

        // Converted the List into Stream and collected it to “Set”
        // Set won't allow any duplicates

        setNoDups.forEach((i) -> System.out.print(" " + i));


    }
}
```

Output:



```
<terminated> Java8 [Java Application] C:\Program Files\Ja
 1 7 8 9
```