**Difference between map() and flatMap() in Java 8 Stream:** when you use a **[map()](http://java67.blogspot.com/2015/01/java-8-map-function-examples.html" \t "https://javarevisited.blogspot.com/2016/03/_blank)**, it applies a function on each element of stream and stores the value returned by the function into a new Stream. This way one stream is transformed into another e.g. a Stream of String is transformed into a Stream of Integer where each element is the length of the corresponding Stream.

The Stream.map() function performs map functional operation i.e. it take a Stream and transform it to another Stream. It applies a function on [each element of Stream](http://javarevisited.blogspot.com/2014/03/2-examples-of-streams-with-Java8-collections.html" \t "https://javarevisited.blogspot.com/2016/03/_blank) and store return value into new Stream. This way you can transform a Stream of String into a Stream of Integer where Integer could be [length of String](http://java67.blogspot.com/2014/09/right-way-to-check-if-string-is-empty.html" \t "https://javarevisited.blogspot.com/2016/03/_blank) if you supply the length() function. This is a very powerful function which is very helpful while dealing with collection in Java.  
Here is an example of Stream.map() in Java 8:

List listOfIntegers = Stream.of("1", "2", "3", "4")

.map(Integer::valueOf)

.collect(Collectors.toList());

The **Stream.flatMap()**function, as the name suggests, is the combination of a map and a flat operation. This means you first apply map function and then flattens the result.

To understand what flattening a stream consists in, consider a structure like [ [1,2,3],[4,5,6],[7,8,9] ] which has "two levels". It's basically a big List containing three more List.  Flattening this means transforming it in a "one level" structure e.g. [ 1,2,3,4,5,6,7,8,9 ] i.e. just one list.  
  
In short,  
Before flattening - Stream of List of Integer  
After flattening - Stream of Integer  
  
List evens = Arrays.asList(2, 4, 6);

List odds = Arrays.asList(3, 5, 7);

List primes = Arrays.asList(2, 3, 5, 7, 11);

List numbers = Stream.of(evens, odds, primes)

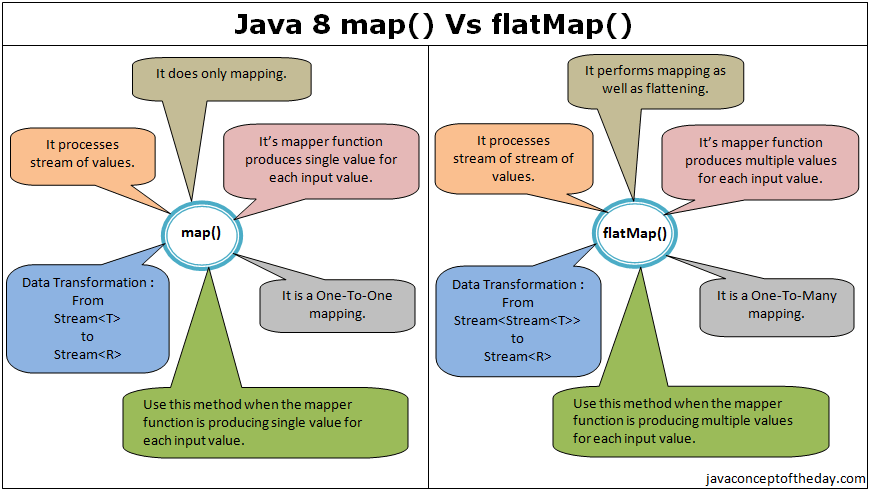
.flatMap(list -> list.stream())

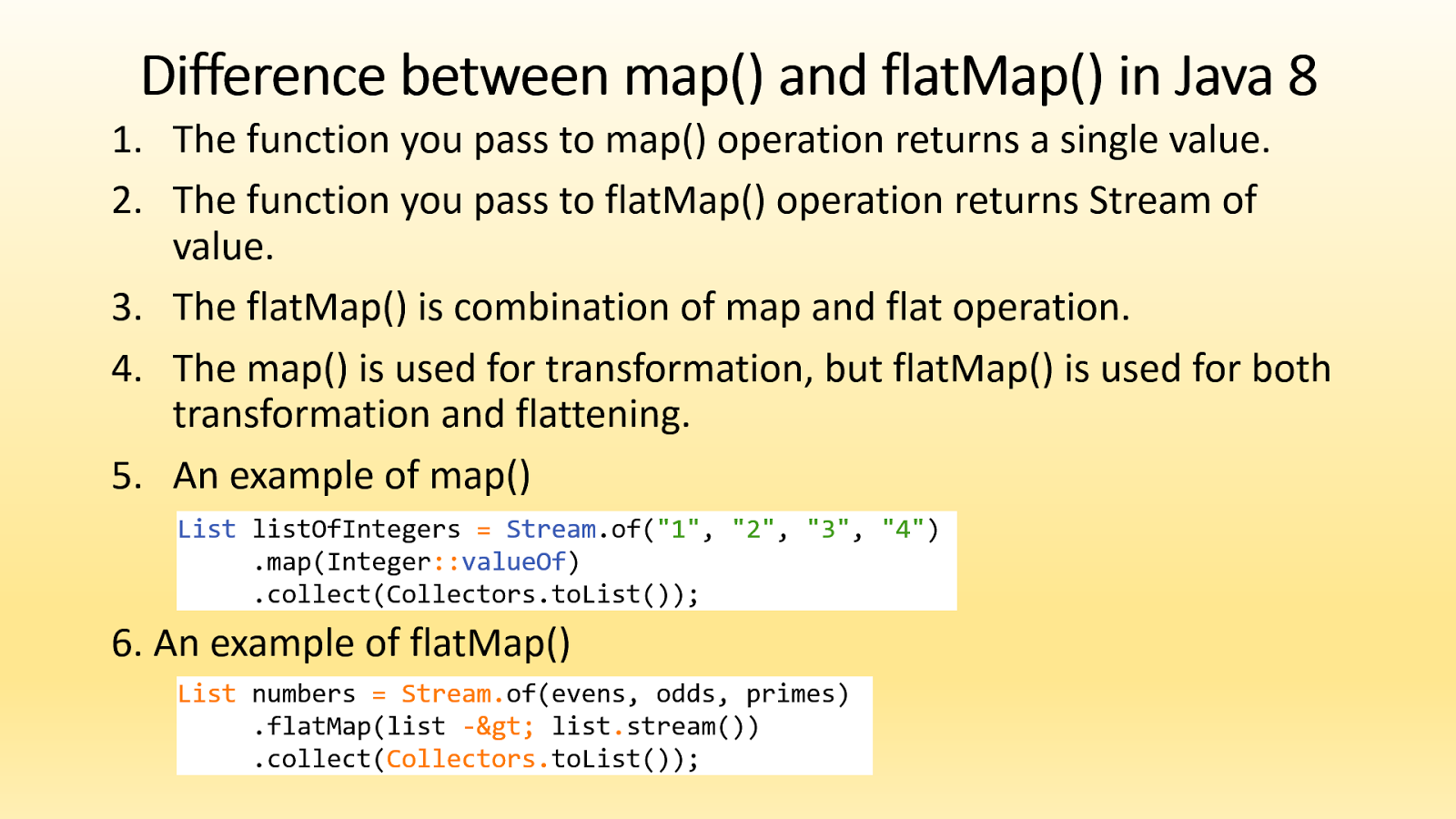
.collect(Collectors.toList());

System.out.println("flattend list: " + numbers); Output: flattend list: [2, 4, 6, 3, 5, 7, 2, 3, 5, 7, 11]

**Stream.map() vs Stream.flatMap() in Java 8**

* The function you pass to map() operation returns a single value.
* The function you pass to flatMap() operation returns a Stream of value.
* flatMap() is a combination of map and flat operation.
* map() is used for transformation only, but flatMap() is used for both transformation and flattening.





**Stream.peek()** method is very useful for debugging and understating the stream related code in Java.

1. The peek() method of Stream class is an intermediate method, hence you can call other stream methods after this.  
     
   2) It returns a new Stream, which is basically the stream it got.  
     
   3) It accepts an object of functional interface Consumer to perform non-interfering action e.g. printing values.  
     
   4) For parallel stream pipelines, the action may be called at whatever time and whatever thread the element is made available by the upstream operation.  
     
   You can use the peek() method for debugging. It allows you to see the elements as they flow past a certain point in the pipeline. By using this you can check whether your filter() method is working properly or not. You can see exactly which elements are got filtered by using peek() in Java 8.

The **Stream.peek()** method is mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline.

The Stream.peek() method returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

For parallel stream pipelines, the action may be called at whatever time and in whatever thread the element is made available by the upstream operation. If the action modifies shared state, it is responsible for providing the required synchronization.

**public** **class** StreamPeekEx {

**public** **static** **void** main(String a[]) {

        Stream.of("bus", "car", "bycle", "flight", "train")

                .filter(e -> e.length() > 3)

                .peek(e -> System.out.println("Filtered value: " + e))

                .map(String::toUpperCase)

                .peek(e -> System.out.println("Mapped value: " + e))

                .collect(Collectors.toList());

    }

}

**How Java 8 Streams work:** Streams support functional-style operations on streams of elements,

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines.

A stream pipeline consists of a source followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

**Method References in Java 8 :** Method reference is a shorthand notation of a lambda expression to call a method. For example:

If your lambda expression is like this:

**str -> System.out.println(str)**

then you can replace it with a method reference like this:

**System.out::println**

The **::** operator is used in method reference to separate the class or object from the method name

**Method references** are a special form of the [lambda expression](https://www.gauravbytes.com/2017/01/java-8-method-references.html" \t "https://dzone.com/articles/_blank). Since your lambda expressions are doing nothing other than invoking existing behavior (methods), you can achieve the same result by referring to it by name.

* **::** is used to refer to a method.
* Method type arguments are inferred by JRE at runtime from the context it is defined.

**Four types of method references :**

* Static method reference
* Instance method reference of a particular object
* Instance method reference of an arbitrary object of a particular type
* Constructor reference

1. **Method reference to an instance method of an object –** object::instanceMethod

**MyInterface ref = obj::myMethod;**

2. **Method reference to a static method of a class – Class::staticMethod**

**BiFunction<Integer, Integer, Integer> product = Multiplication::multiply;**

**int pr = product.apply(11, 5);**

**System.out.println("Product of given number is: "+pr);**

3. **Method reference to an instance method of an arbitrary object of a** particular type – Class::instanceMethod

**Arrays.sort(stringArray, String::compareToIgnoreCase);**

4. **Method reference to a constructor – Class::new**

**MyInterface ref = Hello::new;**

**ref.display("Hello World!");**

**1)  What are new features which got introduced in Java 8?**

There are lots of new features which were added in Java 8.

* [Lambda Expression](https://java2blog.com/lambda-expressions-in-java-8/" \t "https://java2blog.com/java-8-interview-questions/_blank)
* [Stream API](https://java2blog.com/java-8-stream" \t "https://java2blog.com/java-8-interview-questions/_blank)
* [Default methods in the interface](https://java2blog.com/interface-default-methods-in-java-8/" \t "https://java2blog.com/java-8-interview-questions/_blank)
* [Functional Interface](https://java2blog.com/java-8-functional-interface-example/" \t "https://java2blog.com/java-8-interview-questions/_blank)
* [Optional](https://java2blog.com/java-8-optional/" \t "https://java2blog.com/java-8-interview-questions/_blank)
* [Method reference](https://java2blog.com/java-8-method-reference/" \t "https://java2blog.com/java-8-interview-questions/_blank)
* Date API
* Nashorn, JavaScript Engine

1. **What are main advantages of using Java 8?**

* More compact code
* Less boiler plate code
* More readable and reusable code
* More testable code
* Parallel operations

**3) What is lambda expression?**

Lambda expression is anonymous function which have set of parameters and a lambda (->) and a function body .You can call it function without name.

**Structure of Lambda Expressions :**

(Argument List) ->{expression;} or

(Argument List) ->{statements;}

Let see a simple example of thread execution:

public class ThreadSample {

public static void main(String[] args) {

**// old way**

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Thread is started");

}

}).start();

**// using lambda Expression**

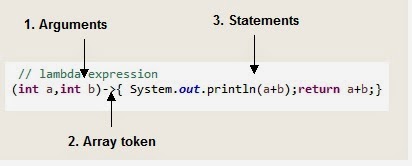
new Thread(()->System.out.println("Thread is started")).start();

}

}

**4) Can you explain the syntax of Lambda expression?**

we can divide structure of Lambda expression to three parts:



**1. Argument list or parameters**

* Lambda expression can have zero or more arguments.
* You can choose to not declare type of arguments as it can be inferred from context.
* you can not declare one argument’s type and do not declare type for other argument.
* When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

**2. Array token (->)**

**3. Body**

* Body can have expression or statements.
* If there is only one statement in body,curly brace is not needed and return type of the anonymous function is same as of  body expression
* If there are more than one statements, then it should be in curly braces and return type of anonymous function is same as value return from code block, void if nothing is returned.

()->{System.out.println("Hello")}; //**Without argument, will print hello**

(int a)->{System.out.println(a)} //; **One argument, will print value of a**

(int a,int b)-> {a+b};//**two argument, will return sum of these two integer**s

(a,b)->{a+b};//**two argument, will return sum of these two numbers**

(int a,b)->{a+b};//**Compilation error**

a->{System.out.println(a)}; // **Will print value of number a**

**5) What are functional interfaces?**

Functional interfaces are those interfaces which can have only one abstract method.It can have static method, default methods or can override Object’s class methods.

There are many functional interfaces already present in java such as **Comparable, Runnable**.

As we have only one method in Runnable, hence it is considered as functional interface.

**6) How lambda expression and functional interfaces are related?**

Lambda expressions can only be applied to abstract method of functional interface.  
**For example :** Runnable has only one abstract method called run, so it can be used as below:

**// Using lambda expression**

Thread t1=new Thread(

()->System.out.println("In Run method")

);

Here we are using Thread constructor which takes Runnable as parameter. As you can see we did not specify any function name here, as Runnable has only one abstract method, java will implicitly create anonymous Runnable and execute run method.

It will be as good as below code.

Thread t1=new Thread(new Runnable() {

@Override

public void run() {

System.out.println("In Run method");

}

});

**7) Can you create your own functional interface?**

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

**Example:** Create interface named "Printable" as below:

public interface Printable {

void print();

default void printColor() {

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

}

}

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

public class **FunctionalIntefaceMain** {

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

FunctionalIntefaceMain pMain=new FunctionalIntefaceMain();

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

}

public void printForm(Printable p) {

p.print();

}}

**8) What is method reference in java 8?**

[Method reference](https://java2blog.com/java-8-method-reference/" \t "https://java2blog.com/java-8-interview-questions/_blank) is used **refer method of functional interface.** It is nothing but compact way of lambda expression.You can simply replace lambda expression with method reference.  
Syntax:  
**class::methodname**

**9) What is Optional? Why and how can you use it?**

Java 8 has introduced new class Called Optional. This class is basically introduced to avoid NullPointerException in java.  
Optional class encapsulates optional value which is either present or not.  
It is a wrapper around object and can be use to avoid NullPointerExceptions.  
Let’s take a simple example

Do you see the problem, there is no non repeating character for getNonRepeatedCharacter("SASAS") hence it will return null and we are calling c.toString, so it will obviously throw NullPointerException.  
You can use Optional to avoid this NullPointerException.  
Let’s change the method to return Optional object rather than String.

public static Optional<Character> getNonRepeatedCharacterOpt(**String** str) {

          Map<Character, **Integer**> countCharacters = **new** LinkedHashMap<Character, **Integer**>();

**for** (**int** i = 0; i < str.length(); i++) {

           Character c = str.charAt(i);

**if** (!countCharacters.containsKey(c)) {

            countCharacters.put(c, 1);

           } **else** {

            countCharacters.put(c, countCharacters.get(c) + 1);

           }

          }

*// As LinkedHashMap maintains insertion order, first character with*

*// count 1 should return first non repeated character*

**for** (Entry<Character, **Integer**> e : countCharacters.entrySet()) {

**if** (e.getValue() == 1)

**return** Optional.of(e.getKey());

          }

**return** Optional.ofNullable(**null**);

         }

Optional<Character> opCh=getNonRepeatedCharacterOpt("SASAS");

**if(opCh.isPresent())**

System.out.println("Non repeated character is :"+opCh.toString());

else {

System.out.println("No non repeated character found in String");

}

If there is no value present in Optional, it will simply print "No non repeated character found in String".

**10) What are defaults methods?**

Default method are those methods in interface which have body and use default keywords.Default method are introduced in Java 8 mainly because of backward compatibility.

**11) What is the difference between Predicate and Function?**

Both are functional interfaces.  
**[Predicate](https://www.java2blog.com/java-8-predicate-examples/" \t "https://java2blog.com/java-8-interview-questions/_blank)<T>** is single argument function and either it returns true or false.This can be used as the assignment target for a lambda expression or method reference.

**Function<T,R>** is also single argument function but it returns an Object.Here T denotes type of input to the function and R denotes type of Result.

This can also be used as the assignment target for a lambda expression or method reference.

**12) Are you aware of Date and Time API introduced in Java 8? What the issues with Old Date and time API?**

Issues with old Date and TIme API:

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

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

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

**13) Can you provide some APIs of Java 8 Date and TIme?**

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

**14) How will you get current date and time using Java 8 Date and TIme API? :** You can simply use now method of LocalDate to get today’s date.

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

System.out.println(currentDate);

It will give you output in below format: 2017-09-09

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

**LocalTime currentTime = LocalTime.now();**

System.out.println(currentTime);

It will give you output in below format: 23:17:51.817

1. **Do we have PermGen in Java 8? Are you aware of MetaSpace?**

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

**Questions will be based on below class :**

public class Employee {

private String name;

private int age;

// Constructor with two parameter

//Setter & Getter

public String toString() {

return "Employee Name: "+name+" age: "+age;

}

}

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

List<**String**> employeeFilteredList = employeeList.stream()

                  .filter(e->e.getAge()>20)

                  .map(Employee::getName)

                  .collect(Collectors.toList());

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

## **Given the list of employees, count number of employees with age 25?**

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

**long** count = employeeList.stream()

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

.count();

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

**18) Given the list of employees, find the employee with name “Mary”.**

Optional<Employee> e1 = employeeList.stream()

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

**if**(e1.isPresent())

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

1. **Given a list of employee, find maximum age of employee?**

OptionalInt max = employeeList.stream().

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

**if**(max.isPresent())

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

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

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

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

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

1. **Join the all employee names with “,” using java 8?**

List<**String**> employeeNames =

employeeList.stream()

             .map(Employee::getName)

             .collect(Collectors.toList());

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

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

**22) Given the list of employee, group them by employee name?**

Answer:  
You can use [Collections.groupBy()](https://java2blog.com/how-to-do-groupby-in-java/" \t "https://java2blog.com/java-8-interview-questions/_blank) to group list of employees by employee name.

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

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

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

1. **Difference between Intermediate and terminal operations in Stream?**

Java 8 Stream supports both intermediate and terminal operation.

**Intermediate** operations are lazy in nature and do not get executed immediately. **Terminal** operations are not lazy, they are executed as soon as encountered. Intermediate operation is memorized and is called when terminal operation is called.

All Intermediate operations return stream as it just transforms stream into another and terminal operation don’t.

**Example of Intermediate operations are:**

filter(Predicate)

map(Function)

flatmap(Function)

sorted(Comparator)

distinct()

limit(long n)

skip(long n)

**Example of terminal operations are :**

forEach

toArray

reduce

collect

min

max

count

anyMatch

allMatch

noneMatch

findFirst

findAny

1. **Given the list of numbers, remove the duplicate elements from the list.**

You can simply use stream and then collect it to set using Collections.toSet() method.

**Integer**[] arr=**new** **Integer**[]{1,2,3,4,3,2,4,2};

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

        Set<**Integer**> setWithoutDups = listWithDuplicates.stream()

.collect(Collectors.toSet());

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

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

**Integer**[] arr=**new** **Integer**[]{1,2,3,4,3,2,4,2};

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

List<**Integer**> listWithoutDups = listWithDuplicates.stream().distinct().collect(Collectors.toList());

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

**25) Difference between Stream’s findFirst() and findAny()?**

findFirst will always return the first element from the stream whereas findAny is allowed to choose any element from the stream.  
findFirst has deterministic behavior whereas findAny is nondeterministic behavior.

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

You can use the [map function](https://java2blog.com/java-8-stream-map/" \t "https://java2blog.com/java-8-interview-questions/_blank) to square the number and then filter to avoid numbers which are less than 10000.We will use average as terminating function in this case.

**Integer**[] arr=**new** **Integer**[]{100,24,13,44,114,200,40,112};

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

        OptionalDouble average = list.stream()

                                 .mapToInt(n->n\*n)

                                 .filter(n->n>10000)

                                 .average();

**if**(average.isPresent())

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

**27) What is use of Optional in Java 8?**

Java 8 optional can be used to avoid NullPointerException.

**28) What is predicate function interface?**

Predicate is single argument function which returns true or false. It has test method which returns boolean.  
When we are using filter in above example, we are actually passing Predicate functional interface to it.

**29) What is consumer function interface?**

Consumer is single argument functional interface which does not return any value.  
When we are using foreach in above example, we are actually passing Consumer functional interface to it.

**30) What is supplier function interface?**

Supplier is function interface which does not take any parameter but returns the value using get method.

**Java 8 Optional :** Java 8 has introduced a new class Optional in java.util package.It is used to represent a value is present or absent.It avoids any runtime NullPointerExceptions and supports us in developing clean and neat Java APIs or Applications.

**Advantages of Java 8 Optional:**

1. Null checks are not required.
2. No more NullPointerException at run-time.
3. We can develop clean and neat APIs.
4. No more Boiler plate code

## Where does Java Optional fits?

**Method Parameter:**

public void setResolution(**Optional<ScreenResolution>** resolution) {

this.resolution = resolution;

}

**Method Return Type:**

public **Optional<ScreenResolution>** getResolution() {

return resolution;

}

**Constructor Parameter :**

public DisplayFeatures(String size, **Optional<ScreenResolution>** resolution){

this.size = size;

this.resolution = resolution;

}

**Variable Declaration :**

private **Optional<ScreenResolution>** resolution;

**Class Level :**

public class B

public class A<T extends **Optional<B>**> { }