**Java 8 Features**

**Lambda Expression in Java**

In Java, Lambda expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface). Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value. Lambda Expressions are recently included in Java SE 8.

**Functionalities of Lambda Expression in Java**

Lambda Expressions implement the only abstract function and therefore implement functional interfaces lambda expressions are added in Java 8 and provide the below functionalities.

* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.

**Java Lambda Expression Example**

Java

*// Java program to demonstrate lambda expressions*

*// to implement a user defined functional interface.*

*// A sample functional interface (An interface with*

*// single abstract method*

**interface** **FuncInterface**

{

*// An abstract function*

void abstractFun(int x);

*// A non-abstract (or default) function*

**default** void normalFun()

{

System.out.println("Hello");

}

}

**class** **Test**

{

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

{

*// lambda expression to implement above*

*// functional interface. This interface*

*// by default implements abstractFun()*

FuncInterface fobj = (int x)->System.out.println(2\*x);

*// This calls above lambda expression and prints 10.*

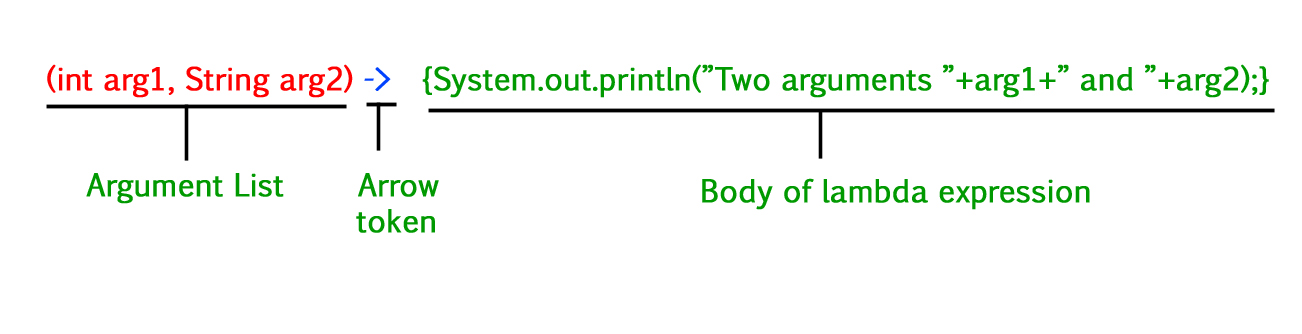
fobj.abstractFun(5);

}

}

**Output**

10

**Lambda Expression Syntax**

lambda operator -> body

**Lambda Expression Parameters**

There are three Lambda Expression Parameters are mentioned below:

1. Zero Parameter
2. Single Parameter
3. Multiple Parameters

**1. Lambda Expression with Zero parameter**

() -> System.out.println("Zero parameter lambda");

**2. Lambda Expression with Single parameter**

(p) -> System.out.println("One parameter: " + p);

It is not mandatory to use parentheses if the type of that variable can be inferred from the context

Java

*// A Java program to demonstrate simple lambda expressions*

**import** **java.util.ArrayList**;

**class** **Test** {

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

{

*// Creating an ArrayList with elements*

*// {1, 2, 3, 4}*

ArrayList<Integer> arrL = **new** ArrayList<Integer>();

arrL.add(1);

arrL.add(2);

arrL.add(3);

arrL.add(4);

*// Using lambda expression to print all elements*

*// of arrL*

arrL.forEach(n -> System.out.println(n));

*// Using lambda expression to print even elements*

*// of arrL*

arrL.forEach(n -> {

**if** (n % 2 == 0)

System.out.println(n);

});

}

}

**Output**

1

2

3

4

2

4

***Note:****that lambda expressions can only be used to implement functional interfaces. In the above example also, the lambda expression implements*[*Consumer*](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)*Functional Interface.*

**3. Lambda Expression with Multiple parameters**

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

A Java program to demonstrate the working of a lambda expression with two arguments.

Java

*// Java program to demonstrate working of lambda expressions*

**public** **class** **Test** {

*// operation is implemented using lambda expressions*

**interface** **FuncInter1** {

int operation(int a, int b);

}

*// sayMessage() is implemented using lambda expressions*

*// above*

**interface** **FuncInter2** {

void sayMessage(String message);

}

*// Performs FuncInter1's operation on 'a' and 'b'*

**private** int operate(int a, int b, FuncInter1 fobj)

{

**return** fobj.operation(a, b);

}

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

{

*// lambda expression for addition for two parameters*

*// data type for x and y is optional.*

*// This expression implements 'FuncInter1' interface*

FuncInter1 add = (int x, int y) -> x + y;

*// lambda expression multiplication for two*

*// parameters This expression also implements*

*// 'FuncInter1' interface*

FuncInter1 multiply = (int x, int y) -> x \* y;

*// Creating an object of Test to call operate using*

*// different implementations using lambda*

*// Expressions*

Test tobj = **new** Test();

*// Add two numbers using lambda expression*

System.out.println("Addition is "

+ tobj.operate(6, 3, add));

*// Multiply two numbers using lambda expression*

System.out.println("Multiplication is "

+ tobj.operate(6, 3, multiply));

*// lambda expression for single parameter*

*// This expression implements 'FuncInter2' interface*

FuncInter2 fobj = message

-> System.out.println("Hello " + message);

fobj.sayMessage("Geek");

}

}

**Output**

Addition is 9

Multiplication is 18

Hello Geek

***Note:****Lambda expressions are just like functions and they accept parameters just like functions.*

**Conclusion**

Some Important points intake from this article is mentioned below:

* The body of a lambda expression can contain zero, one, or more statements.
* When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there is more than one statement, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

**FAQs in Lambda Expression**

**Q1. What type of lambda expression Java?**

**Answer:**

*Java Lambda Expressions are the short block of code that accepts input as parameters and returns a resultant value.*

**Q2. Is it good to use lambda expressions in Java?**

**Answer:**

*Yes, using lambda expressions makes it easier to use and support other APIs.*

**Q3. What are the drawbacks of Java lambda?**

**Answer:**

*Java lambda functions can be only used with functional interfaces.*

**Q4. Based on the syntax rules just shown, which of the following is/are NOT valid lambda expressions?**

1. **() -> {}**
2. **() -> “geeksforgeeks”**
3. **() -> { return “geeksforgeeks”;}**
4. **(Integer i) -> return “geeksforgeeks” + i;**
5. **(String s) -> {“geeksforgeeks”;}**

**Answer:**

*4 and 5 are invalid lambdas, the rest are valid. Details:*

1. *This lambda has no parameters and returns void. It’s similar to a method with an empty body: public void run() { }.*
2. *This lambda has no parameters and returns a String as an expression.*
3. *This lambda has no parameters and returns a String (using an explicit return statement, within a block).*
4. *return is a control-flow statement. To make this lambda valid, curly braces are required as follows: (Integer i) -> { return “geeksforgeeks” + i; }.*
5. *“geeks for geeks” is an expression, not a statement. To make this lambda valid, you can remove the curly braces and semicolon as follows: (String s) -> “geeks for geeks”. Or if you prefer, you can use an explicit return statement as follows: (String s) -> { return “geeks for geeks”; }.*

**Functional Interfaces in Java**

Java has forever remained an Object-Oriented Programming language. By object-oriented programming language, we can declare that everything present in the Java programming language rotates throughout the Objects, except for some of the primitive data types and primitive methods for integrity and simplicity. There are no solely functions present in a programming language called Java. Functions in the Java programming language are part of a class, and if someone wants to use them, they have to use the class or object of the class to call any function.

**Java Functional Interfaces**

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/) can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***,*and****Comparable*** are some of the examples of functional interfaces.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called *@FunctionalInterface***. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only. We can also call Lambda expressions as the instance of functional interface.

**Java Functional Interfaces Example**

**Example 1:**

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

* Java

|  |
| --- |
| // Java program to demonstrate functional interface    **class** Test {  **public** **static** **void** main(String args[])      {          // create anonymous inner class object  **new** Thread(**new** Runnable() {              @Override **public** **void** run()              {                  System.out.println("New thread created");              }          }).start();      }  } |

**Output**

New thread created

**Example 2:**

Java 8 onwards, we can assign [lambda expression](https://www.geeksforgeeks.org/lambda-expressions-java-8/) to its functional interface object like this:

* Java

|  |
| --- |
| // Java program to demonstrate Implementation of  // functional interface using lambda expressions    **class** Test {  **public** **static** **void** main(String args[])      {            // lambda expression to create the object  **new** Thread(() -> {              System.out.println("New thread created");          }).start();      }  } |

**Output**

New thread created

**@FunctionalInterface Annotation**

@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message. However, it is not mandatory to use this annotation.

**Below is the implementation of the above topic:**

* Java

|  |
| --- |
| // Java program to demonstrate lambda expressions to  // implement a user defined functional interface.    @FunctionalInterface    **interface** Square {  **int** calculate(**int** x);  }    **class** Test {  **public** **static** **void** main(String args[])      {  **int** a = 5;            // lambda expression to define the calculate method          Square s = (**int** x) -> x \* x;            // parameter passed and return type must be          // same as defined in the prototype  **int** ans = s.calculate(a);          System.out.println(ans);      }  } |

**Output**

25

**Some Built-in Java Functional Interfaces**

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interfaces. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

* **Runnable –>** This interface only contains the run() method.
* **Comparable –>** This interface only contains the compareTo() method.
* **ActionListener –>** This interface only contains the actionPerformed() method.
* **Callable –>** This interface only contains the call() method.

**Java SE 8 included four main kinds of functional interfaces**which can be applied in multiple situations as mentioned below:

1. **Consumer**
2. **Predicate**
3. **Function**
4. **Supplier**

Amidst the previous four interfaces, the first three interfaces,i.e., Consumer, Predicate, and Function, likewise have additions that are provided beneath –

1. Consumer -> Bi-Consumer
2. Predicate -> Bi-Predicate
3. Function -> Bi-Function, Unary Operator, Binary Operator

**1. Consumer**

The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument. The consumer interface has no return value. It returns nothing. There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments.

Other than these variants, there is also one more variant of the Consumer interface known as Bi-Consumer.

**Bi-Consumer –**Bi-Consumer is the most exciting variant of the Consumer interface. The consumer interface takes only one argument, but on the other side, the Bi-Consumer interface takes two arguments. Both, Consumer and Bi-Consumer have no return value. It also returns nothing just like the Consumer interface. It is used in iterating through the entries of the map.

**Syntax / Prototype of Consumer Functional Interface –**

Consumer<Integer> consumer = (value) -> System.out.println(value);

This implementation of the Java Consumer functional interface prints the value passed as a parameter to the print statement. This implementation uses the Lambda function of Java.

**2. Predicate**

In scientific logic, a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate. Similarly, in the Java programming language, a predicate functional interface of Java is a type of function that accepts a single value or argument and does some sort of processing on it, and returns a boolean (True/ False) answer. The implementation of the Predicate functional interface also encapsulates the logic of filtering (a process that is used to filter stream components on the base of a provided predicate) in Java.

Just like the Consumer functional interface, Predicate functional interface also has some extensions. These are IntPredicate, DoublePredicate, and LongPredicate. These types of predicate functional interfaces accept only primitive data types or values as arguments.

**Bi-Predicate –**Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, takes two arguments, does some processing, and returns the boolean value.

**Syntax of Predicate Functional Interface –**

public interface Predicate<T> {

   boolean test(T t);

}

The predicate functional interface can also be implemented using a class. The syntax for the implementation of predicate functional interface using a class is given below –

public class CheckForNull implements Predicate {

   @Override

   public boolean test(Object o) {

       return o != null;

   }

}

The Java predicate functional interface can also be implemented using Lambda expressions. An example of the implementation of the Predicate functional interface is given below –

Predicate predicate = (value) -> value != null;

This implementation of functional interfaces in Java using Java Lambda expressions is more manageable and effective than the one implemented using a class as both the implementations are doing the same work, i.e., returning the same output.

**3. Function**

A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing. There are many versions of Function interfaces because a primitive type can’t imply a general type argument, so we need these versions of function interfaces. Many different versions of the function interfaces are instrumental and are commonly used in primitive types like double, int, long. The different sequences of these primitive types are also used in the argument.

These versions are:

**Bi-Function**

The Bi-Function is substantially related to a Function. Besides, it takes two arguments, whereas Function accepts one argument.

**The prototype and syntax of Bi-Function is given below –**

@FunctionalInterface

public interface BiFunction<T, U, R>

{

  R apply(T t, U u);

   .......

}

In the above code of interface, T and U are the inputs, and there is only one output which is R.

**Unary Operator and Binary Operator**

There are also two other functional interfaces which are named Unary Operator and Binary Operator. They both extend the Function and Bi-Function, respectively. In simple words, Unary Operator extends Function, and Binary Operator extends Bi-Function.

**The prototype of the Unary Operator and Binary Operator is mentioned below :**

**i. Unary Operator**

@FunctionalInterface

public interface UnaryOperator<T> extends Function<T, U>

{

   ……...

}

**ii. Binary Operator**

@FunctionalInterface

public interface BinaryOperator<T> extends BiFunction<T, U, R>

{

   ……...

}

We can understand front the above example that the Unary Operator accepts only one argument and returns a single argument only. Still, in Unary Operator both the input and output values must be identical and of the same type.

On the other way, Binary Operator takes two values and returns one value comparable to Bi- Function but similar to a Unary Operator, the input and output value types must be identical and of the same type.

**4. Supplier**

The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output. This type of functional interface is generally used in the lazy generation of values. Supplier functional interfaces are also used for defining the logic for the generation of any sequence. For example – The logic behind the Fibonacci Series can be generated with the help of the Stream. generate method, which is implemented by the Supplier functional Interface.

The different extensions of the Supplier functional interface hold many other suppliers functions like BooleanSupplier, DoubleSupplier, LongSupplier, and IntSupplier. The return type of all these further specializations is their corresponding primitives only.

**Syntax / Prototype of Supplier Functional Interface is –**

@FunctionalInterface

public interface Supplier<T>{

// gets a result

………….

// returns the specific result

…………

T.get();

}

**Below is the implementation of the above topic:**

* Java

|  |
| --- |
| // A simple program to demonstrate the use  // of predicate interface  **import** java.util.\*;  **import** java.util.function.Predicate;    **class** Test {  **public** **static** **void** main(String args[])      {          // create a list of strings          List<String> names = Arrays.asList(              "Geek", "GeeksQuiz", "g1", "QA", "Geek2");            // declare the predicate type as string and use          // lambda expression to create object          Predicate<String> p = (s) -> s.startsWith("G");            // Iterate through the list  **for** (String st : names) {              // call the test method  **if** (p.test(st))                  System.out.println(st);          }      }  } |

**Output**

Geek

GeeksQuiz

Geek2

**Important Points/Observations:**

*Here are some significant points regarding Functional interfaces in Java:*

1. *In functional interfaces, there is only one abstract method supported. If the annotation of a functional interface, i.e., @FunctionalInterface is not implemented or written with a function interface, more than one abstract method can be declared inside it. However, in this situation with more than one functions, that interface will not be called a functional interface. It is called a non-functional interface.*
2. *There is no such need for the @FunctionalInterface annotation as it is voluntary only. This is written because it helps in checking the compiler level. Besides this, it is optional.*
3. *An infinite number of methods (whether static or default) can be added to the functional interface. In simple words, there is no limit to a functional interface containing static and default methods.*
4. *Overriding methods from the parent class do not violate the rules of a functional interface in Java.*
5. *The****java.util.function****package contains many built-in functional interfaces in Java 8.*

**Method References in Java with examples**

[Functional Interfaces in Java](https://www.geeksforgeeks.org/functional-interfaces-java/) and [Lambda Function](https://www.geeksforgeeks.org/lambda-expressions-java-8/) are prerequisites required in order to grasp grip over method references in Java. As we all know that a [method](https://www.geeksforgeeks.org/methods-in-java/) is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In this article, we will see how to use methods as value.

In Java 8 we can use the method as if they were objects or primitive values, and we can treat them as a variable. The example shows the function as a variable in java:

// This square function is a variable getSquare.

Function<Integer, Integer> getSquare = i -> i \* i;

SomeFunction(a, b, getSquare);

// Pass function as a argument to other function easily

Sometimes, a lambda expression only calls an existing method. In those cases, it looks clear to refer to the existing method by name. The method references can do this, they are compact, easy-to-read as compared to lambda expressions. A method reference is the shorthand syntax for a lambda expression that contains just one method call. Here’s the general syntax of a

**Generic syntax:**Method reference

**A.** To refer to a method in an object

Object :: methodName

**B.** To print all elements in a list

Following is an illustration of a lambda expression that just calls a single method in its entire execution:

list.forEach(s -> System.out.println(s));

**C.** Shorthand to print all elements in a list

To make the code clear and compact, In the above example, one can turn lambda expression into a method reference:

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

The method references can only be used to replace a single method of the lambda expression. A code is more clear and short if one uses a lambda expression rather than using an anonymous class and one can use method reference rather than using a single function lambda expression to achieve the same. In general, one doesn’t have to pass arguments to method references.

The following example is about performing some operations on elements in the list and adding them. The operation to be performed on elements is a function argument and the caller can pass accordingly.

**Illustration:**

public int transformAndAdd(List<Integer> l,

Function<Integer, Integer> ops) {

int result = 0;

for (Integer s : l)

result += f.apply(s);

return results;

}

// Operations utility class

class OpsUtil {

// Method 1

// To half the variable

public static Integer doHalf(Integer x) {

return x / 2;

}

// Method 2

// Square up the integer number

public static Integer doSquare(Integer x) {

return x \* x;

}

... many more operations...

}

Following are the ways to call the above method as depicted below as follows:

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

// Add some element to list

...

**// Using an anonymous class**

transformAndAdd(list, new Function<Integer, Integer>() {

public Integer apply(Integer i) {

// The method

return OpsUtil.doHalf(i);

}

});

**// Using a lambda expression**

tranformAndAdd(list, i -> OpsUtil.doHalf(i));

**// Using a method reference**

tranformAndAdd(list, OpsUtil::doHalf);

**Types of Method References**

There are four type method references that are as follows:

1. Static Method Reference.
2. Instance Method Reference of a particular object.
3. Instance Method Reference of an arbitrary object of a particular type.
4. Constructor Reference.

To look into all these types we will consider a common example of sorting with a comparator which is as follows:

**Type 1: Reference to a static method**

If a Lambda expression is like:

*// If a lambda expression just call a static method of a class   
(args) -> Class.staticMethod(args)*

Then method reference is like:

*// Shorthand if a lambda expression just call a static method of a class   
Class::staticMethod*

**Example:**

* Java

|  |
| --- |
| // Java Program to Illustrate How One can use  // Static method reference  // To Sort with Custom Comparator    // Importing required classes  **import** java.io.\*;  **import** java.util.\*;    // Class 1  // Helper class  // Object need to be sorted  **class** Person {    **private** String name;  **private** Integer age;        // Constructor  **public** Person(String name, **int** age)      {          // This keyword refers to current instance itself  **this**.name = name;  **this**.age = age;      }        // Getter-setters  **public** Integer getAge() { **return** age; }  **public** String getName() { **return** name; }  }    // Class 2  // Main class  **public** **class** GFG {        // Method 1      // Static method to compare with name  **public** **static** **int** compareByName(Person a, Person b)      {  **return** a.getName().compareTo(b.getName());      }        // Method 2      // Static method to compare with age  **public** **static** **int** compareByAge(Person a, Person b)      {  **return** a.getAge().compareTo(b.getAge());      }        // Method 3      // Main driver method  **public** **static** **void** main(String[] args)      {            // Creating an empty ArrayList of user-defined type          // List of person          List<Person> personList = **new** ArrayList<>();            // Adding elements to above List          // using add() method          personList.add(**new** Person("vicky", 24));          personList.add(**new** Person("poonam", 25));          personList.add(**new** Person("sachin", 19));            // Using static method reference to          // sort array by name          Collections.sort(personList, GFG::compareByName);            // Display message only          System.out.println("Sort by name :");            // Using streams over above object of Person type          personList.stream()              .map(x -> x.getName())              .forEach(System.out::println);            // Now using static method reference          // to sort array by age          Collections.sort(personList, GFG::compareByAge);            // Display message only          System.out.println("Sort by age :");            // Using streams over above object of Person type          personList.stream()              .map(x -> x.getName())              .forEach(System.out::println);      }  } |

**Output:**

Sort by name :

poonam

sachin

vicky

Sort by age :

sachin

vicky

poonam

**Type 2: Reference to an instance method of a particular object**

If a Lambda expression is like:

*// If a lambda expression just call a default method of an object*

*(args) -> obj.instanceMethod(args)*

Then method reference is like:

*// Shorthand if a lambda expression just call a default method of an object*

*obj::instanceMethod*

**Example:**

* Java

|  |
| --- |
| // Java Program to Illustrate How One can use  // Static method reference  // To Sort with Custom Comparator  // But using object method reference    // Importing required classes  **import** java.io.\*;  **import** java.util.\*;    // Class 1  // Helper class  // Object need to be sorted  **class** Person {        // Attributes of a person  **private** String name;  **private** Integer age;        // Constructor  **public** Person(String name, **int** age)      {          // This keyword refers to current object itself  **this**.name = name;  **this**.age = age;      }        // Getter-setter methods  **public** Integer getAge() { **return** age; }  **public** String getName() { **return** name; }  }    // Class 2  // Helper class  // Comparator class  **class** ComparisonProvider {        // Method 1      // To compare with name  **public** **int** compareByName(Person a, Person b)      {  **return** a.getName().compareTo(b.getName());      }        // Method 2      // To compare with age  **public** **int** compareByAge(Person a, Person b)      {  **return** a.getAge().compareTo(b.getAge());      }  }    // Class 3  // Main class  **public** **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {          // Creating an empty ArrayList of user-defined type          // List of person          List<Person> personList = **new** ArrayList<>();            // Adding elements to above object          // using add() method          personList.add(**new** Person("vicky", 24));          personList.add(**new** Person("poonam", 25));          personList.add(**new** Person("sachin", 19));            // A comparator class with multiple          // comparator methods          ComparisonProvider comparator              = **new** ComparisonProvider();            // Now using instance method reference          // to sort array by name          Collections.sort(personList,                           comparator::compareByName);            // Display message only          System.out.println("Sort by name :");            // Using streams          personList.stream()              .map(x -> x.getName())              .forEach(System.out::println);            // Using instance method reference          // to sort array by age          Collections.sort(personList,                           comparator::compareByAge);            // Display message only          System.out.println("Sort by age :");            personList.stream()              .map(x -> x.getName())              .forEach(System.out::println);      }  } |

**Output:**

Sort by name :

poonam

sachin

vicky

Sort by age :

sachin

vicky

poonam

**Type 3: Reference to an instance method of an arbitrary object of a particular type**

If a Lambda expression is like:

*// If a lambda expression just call an instance method of a  ObjectType*

*(obj, args) -> obj.instanceMethod(args)*

Then method reference is like:

*// Shorthand if a lambda expression just call an instance method of a ObjectType*

*ObjectType::instanceMethod*

**Example:**

* Java

|  |
| --- |
| // Java Program to Illustrate how One can use  // Instance type method reference to  // sort with custom comparator    // Importing required classes  **import** java.io.\*;  **import** java.util.\*;    // Main class  **public** **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {          // Creating an empty ArrayList of user defined type          // List of person          List<String> personList = **new** ArrayList<>();            // Adding elements to above object of List          // using add() method          personList.add("vicky");          personList.add("poonam");          personList.add("sachin");            // Method reference to String type          Collections.sort(personList,                           String::compareToIgnoreCase);            // Printing the elements(names) on console          personList.forEach(System.out::println);      }  } |

**Output:**

poonam

sachin

vicky

**Type 4: Constructor method reference**

If a Lambda expression is like:

*// If a lambda expression just create an object   
(args) -> new ClassName(args)*

Then method reference is like:

*// Shorthand if a lambda expression just create an object   
ClassName::new*

**Example:**

* Java

|  |
| --- |
| // Java Program to Illustrate How We can Use  // constructor method reference    // Importing required classes  **import** java.io.\*;  **import** java.nio.charset.Charset;  **import** java.util.\*;  **import** java.util.function.\*;    // Object need to be sorted  **class** Person {  **private** String name;  **private** Integer age;        // Constructor  **public** Person()      {          Random ran = **new** Random();            // Assigning a random value          // to name  **this**.name              = ran                    .ints(97, 122 + 1)                    .limit(7)                    .collect(StringBuilder::**new**,                             StringBuilder::appendCodePoint,                             StringBuilder::append)                    .toString();      }    **public** Integer getAge()      {  **return** age;      }  **public** String getName()      {  **return** name;      }  }    **public** **class** GFG {        // Get List of objects of given      // length and Supplier  **public** **static** <T> List<T>      getObjectList(**int** length,                    Supplier<T> objectSupply)      {          List<T> list = **new** ArrayList<>();    **for** (**int** i = 0; i < length; i++)              list.add(objectSupply.get());  **return** list;      }    **public** **static** **void** main(String[] args)      {            // Get 10 person by supplying          // person supplier, Supplier is          // created by person constructor          // reference          List<Person> personList              = getObjectList(5, Person::**new**);            // Print names of personList          personList.stream()              .map(x -> x.getName())              .forEach(System.out::println);      }  } |

**Output:**

vzskgmu

iupltfx

kocsipj

lyvhxsp

hbdphyv

**Java 8 Stream**

**Java 8** introduces Stream, which is a new abstract layer, and some new additional packages in Java 8 called java.util.stream. A Stream is a sequence of components that can be processed sequentially. These packages include classes, interfaces, and enum to allow functional-style operations on the elements.

The stream can be used by importing java.util.stream package. Stream API is used to process collections of objects. Streams are designed to be efficient and can support improving your program’s performance by allowing you to avoid unnecessary loops and iterations. Streams can be used for filtering, collecting, printing, and converting from one data structure to another, etc.

This Java 8 Stream Tutorial will cover all the basic to advanced concepts of **Java 8 stream** like Java 8 filter and collect operations, and real-life examples of Java 8 streams.

**Prerequisites for Java Stream**

Before proceeding to [Java 8](https://www.geeksforgeeks.org/java-8-features/), it’s recommended to have a basic knowledge of Java 8 and its important concepts such as lambda expression, Optional, method references, etc.

***Note:***

* *If we want to represent a group of objects as a single entity then we should go for*[***collection***](https://www.geeksforgeeks.org/collections-in-java-2)*.*
* *But if we want to process objects from the collection then we should go for streams.*

If we want to use the concept of streams then stream() is the method to be used. Stream is available as an interface.

**Syntax:**

Stream s = c.stream();

In the above pre-tag, ‘c’ refers to the collection. So on the collection, we are calling the ***stream() method***and at the same time, we are storing it as the Stream object. Henceforth, this way we are getting the Stream object.

***Note:****Streams are present in java’s utility package named****java.util.stream***

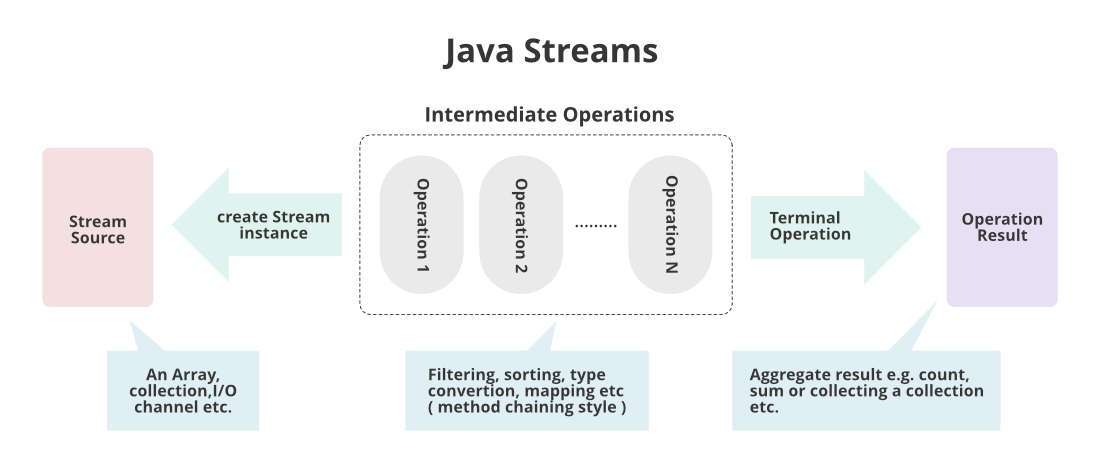
Let us now start with the basic components involved in streams. They as listed as follows:

* Sequence of Elements
* Source
* Aggregate Operations
* Pipelining
* Internal iteration

**Features of Java Stream**

* A stream is not a data structure instead it takes input from the [Collections](https://www.geeksforgeeks.org/java-collection-tutorial/), [Arrays](https://www.geeksforgeeks.org/arrays-in-java/)**,** or [I/O channels](https://www.geeksforgeeks.org/i-o-channels-and-its-types/).
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Before moving ahead in the concept consider an example in which we are having ArrayList of integers, and we suppose we apply a filter to get only even numbers from the object inserted.



**How does Stream Work Internally?**

*In streams,*

* To filter out from the objects we do have a function named ***filter()***
* To impose a condition we do have a logic of predicate which is nothing but a functional interface. Here function interface can be replaced by a random expression. Hence, we can directly impose the condition check-in our predicate.
* To collect elements we will be using ***Collectors.toList()*** to collect all the required elements.
* Lastly, we will store these elements in a List and display the outputs on the console.

**Example**

Java

*// Java Program to illustrate FILTER*

*// & COLLECT Operations*

**import** **java.io.\***;

**import** **java.util.\***;

**import** **java.util.stream.\***;

*// Main class*

**public** **class** **GFG** {

*// Main driver method*

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

{

*// Creating an ArrayList object of integer type*

ArrayList<Integer> al = **new** ArrayList<Integer>();

*// Inserting elements to ArrayList class object*

*// Custom input integer numbers*

al.add(2);

al.add(6);

al.add(9);

al.add(4);

al.add(20);

*// First lets print the collection*

System.out.println("Printing the collection : "

+ al);

*// Printing new line for better output readability*

System.out.println();

*// Stream operations*

*// 1. Getting the stream from this collection*

*// 2. Filtering out only even elements*

*// 3. Collecting the required elements to List*

List<Integer> ls

= al.stream()

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

.collect(Collectors.toList());

*// Print the collection after stream operation*

*// as stored in List object*

System.out.println(

"Printing the List after stream operation : "

+ ls);

}

}

**Output**

Printing the collection : [2, 6, 9, 4, 20]

Printing the List after stream operation : [2, 6, 4, 20]

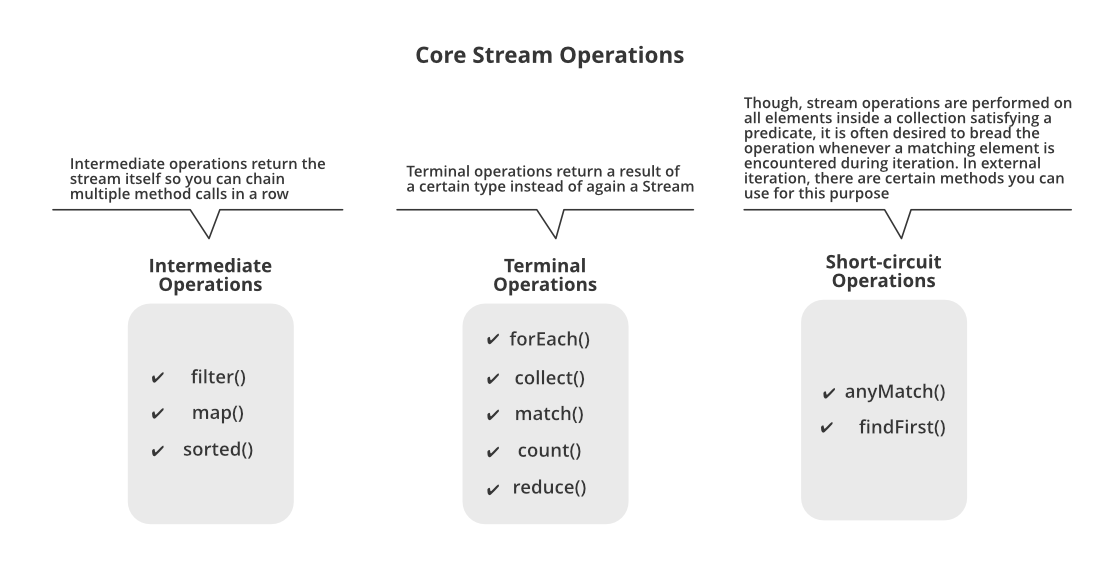
**Explanation of the above program:**

In our collection object, we were having elements entered using the add() operation. After processing the object in which they were stored through streams we impose a condition in the predicate of streams to get only even elements, we get elements in the object as per our requirement.  Hence, streams helped us this way in processing over-processed collection objects.

**Various Core Operations Over Streams**

There are broadly 3 types of operations that are carried over streams namely as follows as depicted from the image shown above:

1. Intermediate operations
2. Terminal operations
3. Short-circuit operations



Let us do discuss out intermediate operations here only in streams to a certain depth with the help of an example in order to figure out other operations via theoretical means.

**1. Intermediate Operations:**

Intermediate operations transform a stream into another stream. Some common intermediate operations include:

1. **filter():**Filters elements based on a specified condition.
2. **map():**Transforms each element in a stream to another value.
3. **sorted():**Sorts the elements of a stream.

All three of them are discussed below as they go hand in hand in nearly most of the scenarios and to provide better understanding by using them later by implementing in our clean Java programs below. As we already have studied in the above example of which we are trying to filter processed objects can be interpreted as filter() operation operated over streams.

**2. Terminal Operations**

Terminal Operations are the operations that on execution return a final result as an absolute value.

1. **collect():**It is used to return the result of the intermediate operations performed on the stream.
2. **forEach():**It iterates all the elements in a stream.
3. **reduce():**It is used to reduce the elements of a stream to a single value.

**3. Short Circuit Operations**

Short-circuit operations provide performance benefits by avoiding unnecessary computations when the desired result can be obtained early. They are particularly useful when working with large or infinite streams.

1. **anyMatch():**it checks the stream if it satisfies the given condition.
2. **findFirst():**it checks the element that matches a given condition and stops processing when it finds it.

***Note:****They are lazy, meaning they are not executed until a terminal operation is invoked.*

Later on from that processed filtered elements of objects, we are collecting the elements back to List using Collectors for which we have imported a specific package named ***java.util.stream***with the help of Collectors.toList() method. This is referred to as collect() operation in streams so here again we won’t be taking an example to discuss them out separately.

**Example:**

Java

*// Java program to illustrate Intermediate Operations*

*// in Streams*

*// Importing required classes*

**import** **java.io.\***;

**import** **java.util.\***;

**import** **java.util.stream.\***;

*// Main class*

**class** **Test** {

*// Main driver method*

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

{

*// Creating an integer Arraylist to store marks*

ArrayList<Integer> marks = **new** ArrayList<Integer>();

*// These are marks of the students*

*// Considering 5 students so input entries*

marks.add(30);

marks.add(78);

marks.add(26);

marks.add(96);

marks.add(79);

*// Printing the marks of the students before grace*

System.out.println(

"Marks of students before grace : " + marks);

*// Now we want to grace marks by 6*

*// using the streams to process over processing*

*// collection*

*// Using stream, we map every object and later*

*// collect to List*

*// and store them*

List<Integer> updatedMarks

= marks.stream()

.map(i -> i + 6)

.collect(Collectors.toList());

*// Printing the marks of the students after grace*

System.out.println(

"Marks of students after grace : "

+ updatedMarks);

}

}

**Output**

Marks of students before grace : [30, 78, 26, 96, 79]

Marks of students after grace : [36, 84, 32, 102, 85]

***Note:****For every object if there is urgency to do some operations be it square, double or any other than only we need to use map() function  operation else try to use filter() function operation.*

**Lazy Evaluation**

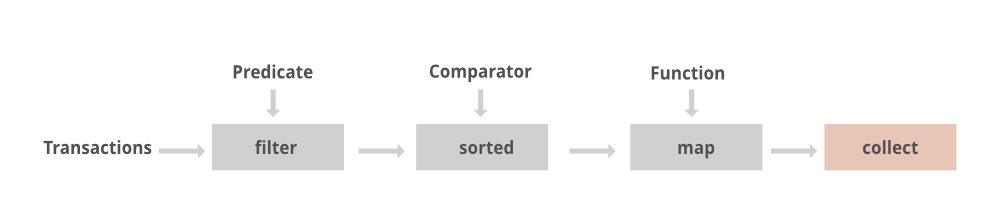
Lazy Evaluation is the concept in Java Streams where computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed. It is called lazy because intermediate operations are not evaluated unless a terminal operation is invoked.

Now geeks you are well aware of ‘why’ streams were introduced, but you should be wondering ‘where’ to use them. The answer is very simple as we do use them too often in our day-to-day life. Hence, the geek in simpler words we say directly lands p on wherever the concept of the collection is applicable, stream concept can be applied there.

**Java Stream: Real-life Examples**

**Example 1:**

In general, daily world, whenever the data is fetched from the database, it is more likely we will be using collection so there stream concept is must apply to deal with processed data.



Now we will be discussing real-time examples to interrelate streams in our life. Here we will be taking the most widely used namely as follows:

1. Streams in a Grocery store
2. Streams in mobile networking

**1. Streams in a Grocery store**



The above pictorial image has been provided is implemented in streams which are as follows:

List<Integer> transactionsIds =   
 transactions.stream()  
 .filter(t -> t.getType() == Transaction.GROCERY)  
 .sorted(comparing(Transaction::getValue).reversed())  
 .map(Transaction::getId)  
 .collect(toList());

**2. Streams in mobile networking**

Similarly, we can go for another widely used concept which is our dealing with our mobile numbers. Here we will not be proposing listings, simply will be demonstrating how the stream concept is invoked in mobile networking by various service providers across the globe.

*Collection can hold any number of object so let ‘mobileNumber’ be a collection and let it be holding various mobile numbers say it be holding 100+ numbers as objects. Suppose now the only carrier named ‘Airtel’ whom with which we are supposed to send a message if there is any migration between states in a country. So here streams concept is applied as if while dealing with all mobile numbers we will look out for this carrier using the filter() method operation of streams. In this way, we are able to deliver the messages without looking out for all mobile numbers and then delivering the message which senses impractical if done so as by now we are already too late to deliver. In this way these intermediate operations namely filter(), collect(), map() help out in the real world. Processing becomes super simpler which is the necessity of today’s digital world.*

Hope by now you the users come to realize the power of streams in Java as if we have to do the same task we do need to map corresponding to every object, increasing in code length, and decreasing the optimality of our code. With the usage of streams, we are able to in a single line irrespective of elements contained in the object as with the concept of streams we are dealing with the object itself.

***Note:****filter, sorted, and map, which can be connected together to form a pipeline.*

**What is a Pipeline?**

A Stream Pipeline is a concept of chaining operations together Terminal Operations and Intermediate Operations. A Pipeline contains a stream source, which is further followed by zero or more intermediate operations, and a terminal operation.

**Java Stream Operations**

**Method Types and Pipelines**

Methods are of two types in Stream as mentioned below:

1. **Terminal Operations**
2. **Intermediate Operations**

**Terminal Operations**

These are the operations that after consumed can’t further be used. There are few operations mentioned below:

**1.**[forEach](https://www.geeksforgeeks.org/stream-foreach-method-java-examples)

forEach performs an action for each element of the stream. Stream forEach is a *terminal operation*.

**Syntax**

void forEach(Consumer<? super T> action)

**2.**[toArray](https://www.geeksforgeeks.org/stream-toarray-java-examples)

Stream toArray() returns an array containing the elements of this stream. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used.

**Syntax**

Object[] toArray()

**3.**[min](https://www.geeksforgeeks.org/stream-min-method-in-java-with-examples)**and**[max](https://www.geeksforgeeks.org/stream-max-method-java-examples)

min and max return the min and max elements from the stream.

**Syntax**

Optional<**T**> min(Comparator<**?** super **T**> comparator)  
Optional<**T**> max(Comparator<**?** super **T**> comparator)

Where, Optional is a container object which may or may not contain a non-null value, and **T** is the type of object that may be compared by this comparator.

**Intermediate Operations**

It returns a new stream that can be further processed. There are certain operations mentioned below:

**1.**[filter](https://www.geeksforgeeks.org/stream-filter-java-examples)

Stream filter returns a stream consisting of the elements of this stream that match the given predicate.

**Syntax**

Stream<T> filter(Predicate<? super T> predicate)

**2.**[distinct](https://www.geeksforgeeks.org/stream-distinct-java)

**distinct()** returns a stream consisting of distinct elements in a stream. distinct() is the method of **Stream** interface.

**Syntax**

Stream<T> distinct()

Where Stream is an interface and the function returns a stream consisting of distinct elements.

**3.**[Sorted](https://www.geeksforgeeks.org/stream-sorted-in-java)

Stream sorted() returns a stream consisting of the elements of this stream, sorted according to natural order. For ordered streams, the sort method is stable but for unordered streams, no stability is guaranteed.

**Syntax**

Stream<**T**> sorted()

where Stream is an interface and **T** is the type of stream elements.

**Comparison-Based Stream Operations**

Comparison Based Stream Operations are the used for comparing, sorting, and ordering elements within a stream. There are certain examples of Comparison Based Stream Operations mentioned below:

1. Sorted
2. min and max
3. distinct

**Java Stream Specializations**

As there are primitive data types or specializations like int, long and double. Similarly, streams have IntStream, LongStream, and DoubleStream. These are convenient for making performing transactions with numerical primitives.

**1. Specialized Operations**

Specialized streams provide additional operations as compared to the standard *Stream* – which are quite convenient when dealing with numbers.

**2. Reduction Operations**

Reduce Operation applies a binary operator, it takes a sequence of input elements and combines them to a single summary result. It is all done where first argument to the operator is the return value of the previous application and second argument is the current stream element.

**Parallel Streams**

Parallel Streams are the type of streams that can perform operations concurrently on multiple threads. These Streams are meant to make use of multiple processors or cores available to speed us the processing speed. There are two methods to create parallel streams are mentioned below:

1. Using the parallel() method on a stream
2. Using parallelStream() on a Collection

To know more about Parallel Streams refer to [this link](https://www.geeksforgeeks.org/what-is-java-parallel-streams).

**Infinite Streams**

Infinite Streams are the type of Streams that can produce unbounded(Infnite) number of elements. These Streams are useful when you need to work with the data sources that are not finite.

**Java Stream: File Operation**

In this section, we see how to utilize Java stream in file I/O operation.

**1. File Read Operation**

Let’s understand file read operation through the given example

Java

*// Java Program to demonstrate*

*// File Read Operation*

**import** **java.io.IOException**;

**import** **java.nio.file.Files**;

**import** **java.nio.file.Paths**;

**import** **java.util.List**;

**import** **java.util.stream.Collectors**;

**import** **java.util.stream.Stream**;

**class** **GFG** {

*// Method to filter strings of a given length and*

*// convert them to uppercase*

**private** **static** List<String>

filterAndConvertToUpper(Stream<String> stream,

int length)

{

**return** stream.filter(s -> s.length() == length)

.map(String::toUpperCase)

.collect(Collectors.toList());

}

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

{

*// Replace with the*

*// actual file path*

String fileName = "path/to/your/file.txt";

*// Step 1: Create a Stream of lines from the*

*// file*

**try** (Stream<String> lines

= Files.lines(Paths.get(fileName))) {

List<String> filteredStrings

= filterAndConvertToUpper(lines, 5);

System.out.println(

"Filtered strings with length 5 (converted to uppercase): "

+ filteredStrings);

}

**catch** (IOException e) {

e.printStackTrace();

}

}

}

**Input:**

Geeks  
gfg  
geeks  
geeksforgeeks  
Coder  
Guys

**Output:**

Filtered strings with length 5 (converted to uppercase): [GEEKS, GEEKS, CODER]

**2. File Write Operation**

Let’s understand file write operation through the given example

Java

*// Java Program to demonstrate*

*// File Write Operation*

**import** **java.io.\***;

**import** **java.nio.file.Files**;

**import** **java.nio.file.Paths**;

**import** **java.util.stream.Stream**;

*// Driver Class*

**class** **GFG** {

*// main function*

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

{

String[] words

= { "Geeks", "for", "Geeks", "Hello", "World" };

*// Replace with the*

*// actual file path*

String fileName = "path/to/your/file.txt";

*// Step 1: Create a PrintWriter to write to the*

*// file*

**try** (PrintWriter pw

= **new** PrintWriter(Files.newBufferedWriter(

Paths.get(fileName)))) {

*// Step 2: Use Stream to write each word to the*

*// file*

Stream.of(words).forEach(pw::println);

*// Step 3: Print success message to the console*

System.out.println(

"Words written to the file successfully.");

}

**catch** (IOException e) {

*// Step 4: Handle any IO exception that occurs*

*// during the file writing process*

e.printStackTrace();

}

}

}

**Output:**

Words written to the file successfully.

**Java Streams Improvements in Java 9**

*As we know, Java 8 introduced a Java stream to the world which provided a powerful way to process collections of data. However, the following version of the language also contributed to the feature. Thereafter, some improvements and features were added to the Java stream in JAVA 9. In this section, we will provide an overview of the advancements introduced by Java 9 to the Streams API like takeWhile, dropWhile, iterate, ofNullable, etc.*

**Conclusion**

Java Streams offer a powerful way to handle data in Java. They allow developers to write more readable and concise code for processing collections. By using Java Streams, you can easily filter, map, and reduce data with simple and expressive methods. This not only makes your code easier to maintain but also helps improve performance by taking advantage of parallel processing. If you’re looking to make your data operations more efficient and straightforward, learning Java Streams is definitely worth the effort.

**Java 8 Stream – FAQs**

**1. How to learn streams in Java 8?**

*To learn java streams effectivily you need to get you theory concepets as well as practical skills strong. for that first you need to make your concepts strong then make it in pratice.*

1. ***Grasp the fundamental concept of streams.***
   * *What are streams?*
   * *Why are they useful?*
   * *How do they work?*
2. ***Familiarize yourself with creating streams from diverse sources.***
   * *How can you create a stream from an array, a collection, or a file?*
3. ***Dive into intermediate operations.***
   * *What are intermediate operations?*
   * *How do they transform data?*
4. ***Understand terminal operations.***
   * *What are terminal operations?*
   * *How do they produce final results or trigger actions on the stream?*
5. ***Embrace the concept of lazy evaluation.***
   * *How does lazy evaluation work? How does it optimize resource usage?*
6. ***Practice chaining operations to create intricate data pipelines.***
   * *How can you chain multiple operations together to create complex data pipelines?*
7. ***Explore parallel processing.***
   * *How can you use parallel processing to improve the performance of your streams?*
8. ***Apply streams to real-world scenarios.***
   * *Find situations where streams can simplify code and improve efficiency.*
9. ***Utilize online resources, tutorials, and exercises.***
10. ***Consider in-depth tutorials.****If you want a comprehensive learning experience, consider taking an in-depth tutorial on streams.*

**2. Why we use stream in Java 8?**

*Java streams offer a range of functionalities that significantly enhance their importance.*

1. ***Efficient Data Processing:****They don’t store the data themselves, but instead act as a way to process data from various*[*DS*](https://www.geeksforgeeks.org/data-structures)*.*
2. ***Functional and Non-Destructive****: Streams follow a functional programming approach.*
3. ***Lazy Evaluation****: which means they perform computations only when needed*
4. ***Single Pass Processing****: The elements in a stream are processed only once during its lifetime*

**3. What is the best practice of Java streams?**

*Java Stream API is a powerful and flexible tool that can significantly simplify code for data processing tasks and here are some best pratices for using java streams.*

1. *Use primitive streams for better performance*
2. *Avoid nesting streams*
3. *use stream with immutable objects*
4. *Use filter() before map() to avoid unnecessary processing*
5. *Prefer method references over lambda expressions*
6. *Use distinct() to remove duplicates*
7. *Use sorted() with caution*
8. *Use lazy evaluation for better performance*

**Difference between stream and collection**

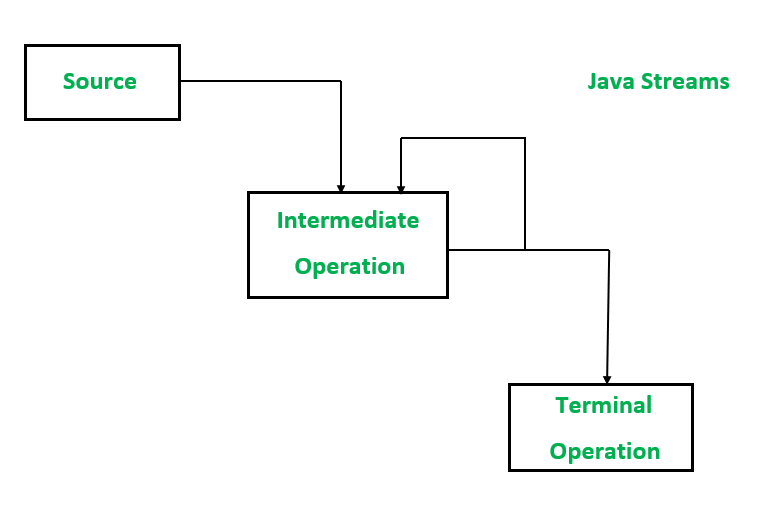
| STREAMS | COLLECTIONS |
| --- | --- |
| It doesn’t store data, it operates on the source data structure i.e collection. | It stores/holds all the data that the data structure currently has in a particular data structure like Set, List or Map, |
| They use functional interfaces like lambda which makes it a good fit for programming language. | Don’t use functional interfaces. |
| Java Streams are consumable i.e; to traverse the stream, it needs to be created every time. | Non-consumable i.e; can be traversable multiple times without creating it again. |
| Java streams support both sequential and parallel processing. | Supports parallel processing and parallel processing can be very helpful in achieving high performance. |
| All the Java stream API interfaces and classes are in java.util.stream package. | Specific classes for primitive types such as IntStream, LongStream, and DoubleStream are used in collections since primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time. |
| Streams are not modifiable i.e one can’t add or remove elements from streams. | These are modifiable i.e one can easily add to or remove elements from collections. |
| Streams are iterated internally by just mentioning the operations. | Collections are iterated externally using loops. |

**Difference Between map() and flatmap()**

| **map()** | **flatMap()** |
| --- | --- |
| The function passed to map() operation returns a single value for a single input. | The function you pass to flatmap() operation returns an arbitrary number of values as the output. |
| One-to-one mapping occurs in map(). | One-to-many mapping occurs in flatMap(). |
| Only perform the mapping. | Perform mapping as well as flattening. |
| Produce a stream of value. | Produce a stream of stream value. |
| map() is used only for transformation. | flatMap() is used both for transformation and mapping. |

**Functional Programming in Java 8+ using the Stream API with Example**

API is an acronym for Application Programming Interface, which is software and the java streams work on a data source. Consider a stream like a flow of water in a small canal. Let’s take a real-life example. Each time a user uses an application that is popular these days like WhatsApp in order to communicate via delivering text messages or calls to other users. Both users are using an API.



Java streams work on three operations which as mentioned below

1. Data Source
2. Intermediate operation
3. Terminal operation

**Methods:**Streams can be created in three ways

1. Using an object of any class from the collection framework
2. Using an array of the reference data type
3. Using the interface defined in the ‘java.util.stream’ package

**Method 1:** Data Source

The data source can be widely varied such as an array, List, etc

**Syntax:**

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

Integer[] numbers = {1,2,3};

**Example 1:**Using an object as a data source

* Java

|  |
| --- |
| // Java Program showcasing data source  // using an object as a data source    // Importing input output classes  **import** java.io.\*;    // Class  **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {          // Data Source            // Creating an arrayList object          // Declaring object of String type          ArrayList<String> gfgNames = **new** ArrayList<>();            // Custom input elements to above object          gfgNames.add("Dean");          gfgNames.add("castee");          gfgNames.add("robert");            // Creating object of Stream where Stream is created          // using arrayList and object as data source          Stream<String> streamOfNames = gfgNames.stream();            // Print and display element          System.out.print(streamOfNames);      }  } |

**Example 2:**Using an array as a data source

// Data Source

Integer[] numbers = {1,2,3,4,5};

// Stream using an array

Stream<Integer> streamOfNumbers = Arrays.stream(numbers);

// using predefined Instream interface

integerStream = IntStream.range(1,100); // a stream from 1 to 99;

* Java

|  |
| --- |
| // Java Program showcasing data source  // using an array as a data source    // Importing java input output class  **import** java.io.\*;    // Importing all classes from  // java.util package  **import** java.util.\*;    // Importing class for additional operations,  // additionls and pipelines  **import** java.util.stream.IntStream;    // Class  **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {          // Creating a predefined stream using range method          // Custom inputs for range as parameters          var stream = IntStream.range(1, 100);            // Stream is created          var max = stream.filter(number -> number % 4 == 0)                        .count();            // Print and display the maximum number          // from the stream          System.out.println(max);      }  } |

24

**Method 2:** Intermediate Operation

Intermediate operations are some methods that one can apply on a stream.

***Note:****It can be of any number*

filter()

**Example:**

* Java

|  |
| --- |
| **import** java.io.\*;    **class** GFG {  **public** **static** **void** main(String[] args)      {          // Data Source          Integer[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 }            // Stream          Stream<Integer> streamOfNumbers              = Arrays.stream(numbers);            // filter all the even numbers          Stream<Integer> evenNumbersStream              = streamOfNumbers.filter(                  number -> number % 2 == 0)      }  } |

**Method 3:**Terminal operation

Terminal operation methods that we can apply on a stream that will cause a stream to be “closed”.

**Concept:**

Some terminal operations can be used to iterate on the elements of the stream.

min(),max(),count()

forEach(),noneMatch()

**Example 1:**Explaining stream API

* Java

|  |
| --- |
| // Importing input output classes  **import** java.io.\*;  // Importing all classes from  // java.util package  **import** java.util.\*;    // Class  **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {          // Data source          // Custom integer inputs          Integer[] numbers              = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };            // Stream          var streamOfNumbers = Arrays.stream(numbers);            // Using filter method          var evenNumbersStream              = streamOfNumbers                    .filter(number -> number % 3 == 0)                    .count();            // Print and display          System.out.println(evenNumbersStream);      }  } |

**Output**

4

**Example 2:**

* Java

|  |
| --- |
| // Importing all classes from java.util package  **import** java.io.\*;  **import** java.util.\*;    // Class  **class** GFG {      // Main driver method  **public** **static** **void** main(String[] args)      {        // Creating an ArrayList of Integer type          ArrayList<Integer> list = **new** ArrayList<>();          // Adding elements to above object of Arraylist        // Custom inputs          list.add(20);          list.add(4);          list.add(76);          list.add(21);          list.add(3);          list.add(80);            var stream = list.stream();            var numbers              = stream.filter(number -> number % 2 == 0)                    .filter(number -> number > 20);          // Print all the elements of the stream on the console          numbers.forEach(System.out::println);      }  } |

**Output**

76

80

***Note:****One can pass lambda also number -> System.out.println(number + ” “)*

**Java 8 Optional Class**

Every Java Programmer is familiar with [NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/). It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in **java.util package**. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

Java

*// Java program without Optional Class*

**public** **class** **OptionalDemo** {

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

{

String[] words = **new** String[10];

String word = words[5].toLowerCase();

System.out.print(word);

}

}

**Output:**

Exception in thread "main" java.lang.NullPointerException

To avoid abnormal termination, we use the Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

**The above program using Optional Class**

Java

*// Java program with Optional Class*

**import** **java.util.Optional**;

*// Driver Class*

**public** **class** **OptionalDemo** {

*// Main Method*

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

{

String[] words = **new** String[10];

Optional<String> checkNull = Optional.ofNullable(words[5]);

**if** (checkNull.isPresent()) {

String word = words[5].toLowerCase();

System.out.print(word);

}

**else**

System.out.println("word is null");

}

}

**Output**

word is null

Optional is a container object which may or may not contain a non-null value. You must import ***java.util package*** to use this class. If a value is present, **isPresent()** will return true and **get()** will return the value. Additional methods that depend on the presence or absence of a contained value are provided, such as **orElse()** which returns a default value if the value is not present, and **ifPresent()** which executes a block of code if the value is present. This is a ***value-based*** class, i.e their instances are :

* Final and immutable (though may contain references to mutable objects).
* Considered equal solely based on equals(), not based on reference equality(==).
* Do not have accessible constructors.

**Static Methods:**Static methods are the methods in Java that can be called without creating an object of the class. They are referenced by the class name itself or reference to the object of that class.

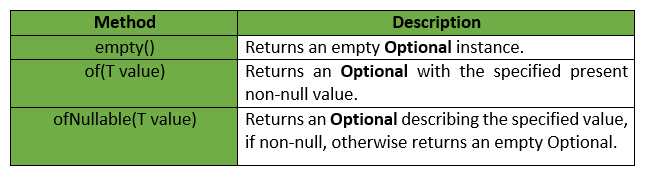
**Syntax :**

public static void geek(String name)  
{  
 // code to be executed....  
}  
  
// Must have static modifier in their declaration.  
// Return type can be int, float, String or user-defined data type.

**Important Points:** Since Static methods belong to the class, they can be called to without creating the object of the class. Below given are some important points regarding Static Methods :

* Static method(s) are associated with the class in which they reside i.e. they can be called even without creating an instance of the class.
* They are designed with the aim to be shared among all objects created from the same class.
* Static methods can not be overridden. But can be overloaded since they are resolved using static binding by the compiler at compile time.

The following table shows the list of Static Methods provided by Optional Class :



**Instance Methods:**Instance methods are methods that require an object of its class to be created before it can be called. To invoke an instance method, we have to create an Object of the class within which it is defined.

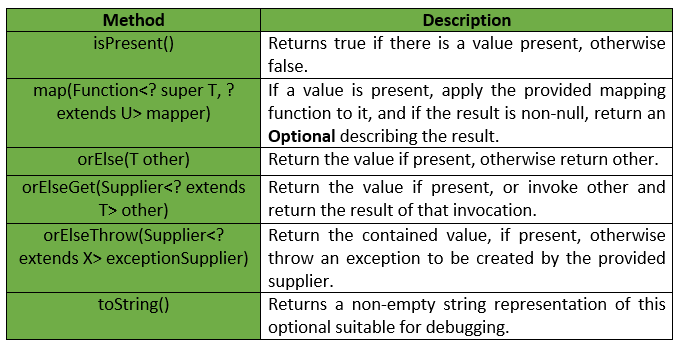
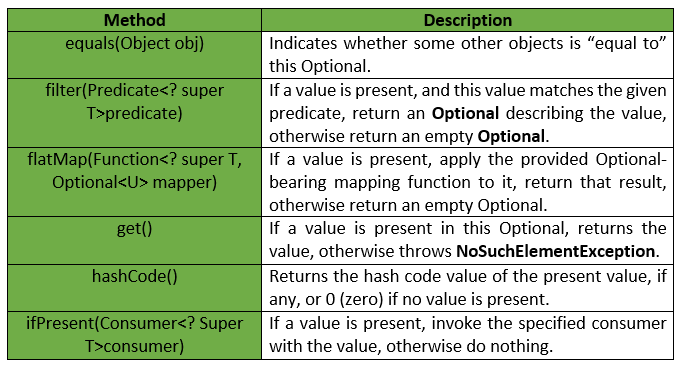
**Syntax :**

public void geek(String name)  
{  
 // code to be executed....  
}  
// Return type can be int, float String or user defined data type.

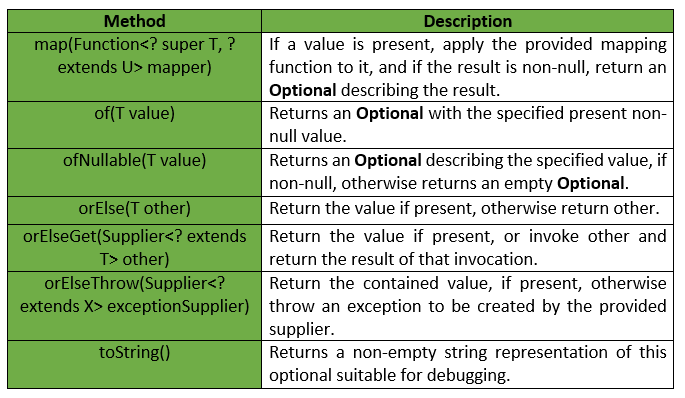
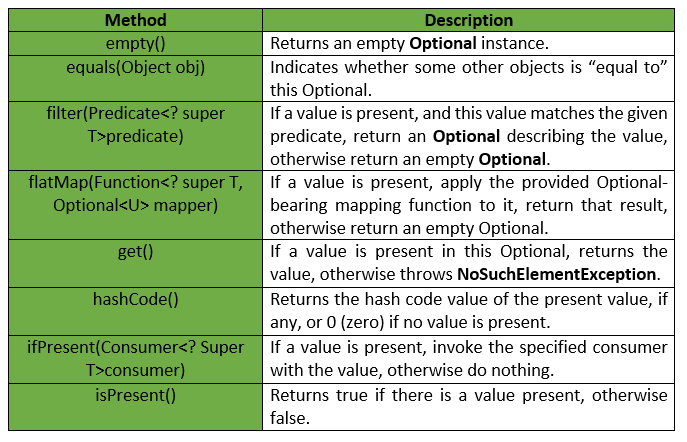
**Important Points:**Instance Methods can be called within the same class in which they reside or from the different classes defined either in the same package or other packages depending on the access type provided to the desired instance method. Below given are some important points regarding Instance Methods :

* Instance method(s) belong to the Object of the class, not to the class i.e. they can be called after creating the Object of the class.
* Every individual object created from the class has its own copy of the instance method(s) of that class.
* They can be overridden since they are resolved using dynamic binding at run time.

The following table shows the list of Instance Methods provided by the Optional Class :



**Concrete Methods:**A concrete method means, the method has a **complete definition** but can be overridden in the inherited class. If we make this method **final**, then it can not be overridden. Declaring method or class “final” means its implementation is complete. It is compulsory to override abstract methods. ***Concrete Methods can be overridden in the inherited classes if they are not final.*** The following table shows the list of Concrete Methods provided by the Optional Class :



Below given are some examples :

**Example 1 :**

Java

*// Java program to illustrate*

*// optional class methods*

**import** **java.util.Optional**;

**class** **GFG** {

*// Driver code*

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

{

*// creating a string array*

String[] str = **new** String[5];

*// Setting value for 2nd index*

str[2] = "Geeks Classes are coming soon";

*// It returns an empty instance of Optional class*

Optional<String> empty = Optional.empty();

System.out.println(empty);

*// It returns a non-empty Optional*

Optional<String> value = Optional.of(str[2]);

System.out.println(value);

}

}

**Output**

Optional.empty

Optional[Geeks Classes are coming soon]

**Example 2 :**

Java

*// Java program to illustrate*

*// optional class methods*

**import** **java.util.Optional**;

**class** **GFG** {

*// Driver code*

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

{

*// creating a string array*

String[] str = **new** String[5];

*// Setting value for 2nd index*

str[2] = "Geeks Classes are coming soon";

*// It returns a non-empty Optional*

Optional<String> value = Optional.of(str[2]);

*// It returns value of an Optional.*

*// If value is not present, it throws*

*// an NoSuchElementException*

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

*// It returns hashCode of the value*

System.out.println(value.hashCode());

*// It returns true if value is present,*

*// otherwise false*

System.out.println(value.isPresent());

}

}

**Output**

Geeks Classes are coming soon

1967487235

True

**Default Methods In Java 8**

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

|  |
| --- |
| // A simple program to Test Interface default  // methods in java  **interface** TestInterface  {      // abstract method  **public** **void** square(**int** a);        // default method  **default** **void** show()      {        System.out.println("Default Method Executed");      }  }    **class** TestClass **implements** TestInterface  {      // implementation of square abstract method  **public** **void** square(**int** a)      {          System.out.println(a\*a);      }    **public** **static** **void** main(String args[])      {          TestClass d = **new** TestClass();          d.square(4);            // default method executed          d.show();      }  } |

Output:

16

Default Method Executed

The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as **defender methods**or **virtual extension methods**.

**Static Methods:**  
The interfaces can have static methods as well which is similar to static method of classes.

|  |
| --- |
| // A simple Java program to TestClassnstrate static  // methods in java  **interface** TestInterface  {      // abstract method  **public** **void** square (**int** a);        // static method  **static** **void** show()      {          System.out.println("Static Method Executed");      }  }    **class** TestClass **implements** TestInterface  {      // Implementation of square abstract method  **public** **void** square (**int** a)      {          System.out.println(a\*a);      }    **public** **static** **void** main(String args[])      {          TestClass d = **new** TestClass();          d.square(4);            // Static method executed          TestInterface.show();      }  } |

Output:

16

Static Method Executed

**Default Methods and Multiple Inheritance**  
In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

|  |
| --- |
| // A simple Java program to demonstrate multiple  // inheritance through default methods.  **interface** TestInterface1  {      // default method  **default** **void** show()      {          System.out.println("Default TestInterface1");      }  }    **interface** TestInterface2  {      // Default method  **default** **void** show()      {          System.out.println("Default TestInterface2");      }  }    // Implementation class code  **class** TestClass **implements** TestInterface1, TestInterface2  {      // Overriding default show method  **public** **void** show()      {          // use super keyword to call the show          // method of TestInterface1 interface          TestInterface1.**super**.show();            // use super keyword to call the show          // method of TestInterface2 interface          TestInterface2.**super**.show();      }    **public** **static** **void** main(String args[])      {          TestClass d = **new** TestClass();          d.show();      }  } |

Output:

Default TestInterface1

Default TestInterface2

**Important Points:**

1. Interfaces can have default methods with implementation in Java 8 on later.
2. Interfaces can have static methods as well, similar to static methods in classes.
3. Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.

**Can We Override Default Method in Java?**

[Default method](https://www.geeksforgeeks.org/default-methods-java/) in Java is a method in java which are defined inside the interface with the keyword default is known as the default method. It is a type of non-abstract method.

This method is capable of adding backward capability so that the old interface can grasp the lambda expression capability.

Java Interface Default method is also known as Defender Method or virtual extension method.

Interfaces could have only abstract methods before Java 8. The classes separately provide implementation to these methods. So, if a new method is to be added to an interface, then its implementation code has to be provided in the class implementing the same interface. For overcoming this issue, Java 8 introduced the concept of default methods that allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

**Can We Override Default Method in Java?**

It is not mandatory to override the default method in Java.

If we are using Only **one interface**in a Program then at a time we are using only a single default method and at that time Overriding is not required as shown in the below program:

* Java

|  |
| --- |
| // Creating Interface  **interface**  GfG{    **public** **default** **void** display() {        System.out.println("GEEKSFORGEEKS");     }  }    // Main Class With Implementation Of Interface  **public** **class** InterfaceExample **implements** GfG{  **public** **static** **void** main(String args[]) {        InterfaceExample obj = **new** InterfaceExample();          // Calling Interface        obj.display();     }  } |

**Output**

GEEKSFORGEEKS

But when**more than** **two Interfaces** are used and both act as parent class then at that time Overriding of the Default Method is required.If we are using more than one interface and in both interfaces, if both interfaces have the same name and same structure. So at that time, one must override either one both the default method otherwise it will result in an error.

**Case 1: When Two Interfaces are not overridden**

* Java

|  |
| --- |
| // Java program to demonstrate the case when  // two interfaces are not overridden    // Creating Interface One  **interface** GfG{  **public** **default** **void** display() {        System.out.println("GEEKSFORGEEKS");     }  }    // Creating Interface Two  **interface** gfg{    **public** **default** **void** display() {        System.out.println("geeksforgeeks");     }  }    // Interfaces are not Overridden  **public** **class** InterfaceExample **implements** GfG,gfg {  **public** **static** **void** main(String args[])     {        InterfaceExample obj = **new** InterfaceExample();        obj.display();     }  } |

**Output:**

InterfaceExample.java:18: error: types GfG and gfg are incompatible;

public class InterfaceExample implements GfG,gfg {

^

class InterfaceExample inherits unrelated defaults for display() from types GfG and gfg

1 error

**Case 2: When Two Interfaces are Overridden**

* Java

|  |
| --- |
| // Java program to demonstrate the case  // when two interfaces are overridden    // Creating Interface One  **interface** GfG{  **public** **default** **void** display()     {        System.out.println("GEEKSFORGEEKS");     }  }    // Creating Interface Two  **interface** gfg{    **public** **default** **void** display()     {        System.out.println("geeksforgeeks");     }  }    **public** **class** InterfaceExample **implements** GfG,gfg {    // Interfaces are Overrided  **public** **void** display() {          GfG.**super**.display();          gfg.**super**.display();     }    **public** **static** **void** main(String args[]) {        InterfaceExample obj = **new** InterfaceExample();        obj.display();     }  } |

**Output**

GEEKSFORGEEKS

geeksforgeeks

**Static method in Interface in Java**

**Static Methods** in **Interface**are those methods, which are defined in the interface with the keyword static. Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.  
Similar to [Default Method in Interface](https://www.geeksforgeeks.org/default-methods-java/), the static method in an interface can be defined in the interface, but cannot be overridden in Implementation Classes. To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.  
Below programs illustrate static methods in interfaces:  
**Program 1:** To demonstrate use of Static method in Interface.  
In this program, a simple static method is defined and declared in an interface which is being called in the main() method of the Implementation Class InterfaceDemo. Unlike the default method, the static method defines in Interface hello(), cannot be overridden in implementing the class.

* Java

|  |
| --- |
| // Java program to demonstrate  // static method in Interface.    **interface** NewInterface {        // static method  **static** **void** hello()      {          System.out.println("Hello, New Static Method Here");      }        // Public and abstract method of Interface  **void** overrideMethod(String str);  }    // Implementation Class  **public** **class** InterfaceDemo **implements** NewInterface {    **public** **static** **void** main(String[] args)      {          InterfaceDemo interfaceDemo = **new** InterfaceDemo();            // Calling the static method of interface          NewInterface.hello();            // Calling the abstract method of interface          interfaceDemo.overrideMethod("Hello, Override Method here");      }        // Implementing interface method        @Override  **public** **void** overrideMethod(String str)      {          System.out.println(str);      }  } |

**Output:**

Hello, New Static Method Here

Hello, Override Method here

**Program 2:** To demonstrate Scope of Static method.  
In this program, the scope of the static method definition is within the interface only. If same name method is implemented in the implementation class then that method becomes a static member of that respective class.

* Java

|  |
| --- |
| // Java program to demonstrate scope  // of static method in Interface.    **interface** PrintDemo {        // Static Method  **static** **void** hello()      {          System.out.println("Called from Interface PrintDemo");      }  }    **public** **class** InterfaceDemo **implements** PrintDemo {    **public** **static** **void** main(String[] args)      {            // Call Interface method as Interface          // name is preceding with method          PrintDemo.hello();            // Call Class static method          hello();      }        // Class Static method is defined  **static** **void** hello()      {          System.out.println("Called from Class");      }  } |

**Output:**

Called from Interface PrintDemo

Called from Class

**Java 8 Interview Questions and Answers**

**Java 8** introduced a host of powerful features that have significantly enhanced the **Java programming language**. Introducing new **features** such as**Lambda Expressions**, **Stream API**, **Functional Interfaces**, the **new Date and Time API**, and more.

As a result, **Java 8** skills are highly sought after by employers in the tech industry. To help you prepare for your next interview, we have compiled a list of commonly asked **Java 8 Interview Questions**and provided detailed answers. Whether you are a seasoned **Java developer** or just starting, this article will help you showcase your knowledge and expertise in Java 8.

But before going towards the interview questions for Java 8 let’s have a quick brief in Java 8.

**What is Java 8?**

Oracle Corporation announced Java 8 in March 2014, as a major update to the [Java programming language](https://www.geeksforgeeks.org/java) and platform. It introduced several key innovations and enhancements, such as [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8), the [Stream API](https://www.geeksforgeeks.org/java-stream-api-filters), Functional Interfaces, the**java.time package** for date and time manipulation, and the **Optional class** for handling potentially null values. Java 8 was designed to improve developer **productivity**, **code readability**, and **performance,** making it a watershed moment in the Java language’s evolution.

**Java 8 Interview Questions and Answers**

In the upcoming section, resources are crafted for Freshers and Experienced Professionals. These **Interview Questions for Java 8** offer you valuable insight and solutions. So, let’s dive into the **world of Java 8** and uncover key concepts that will sharpen your skills and ace your interviews!

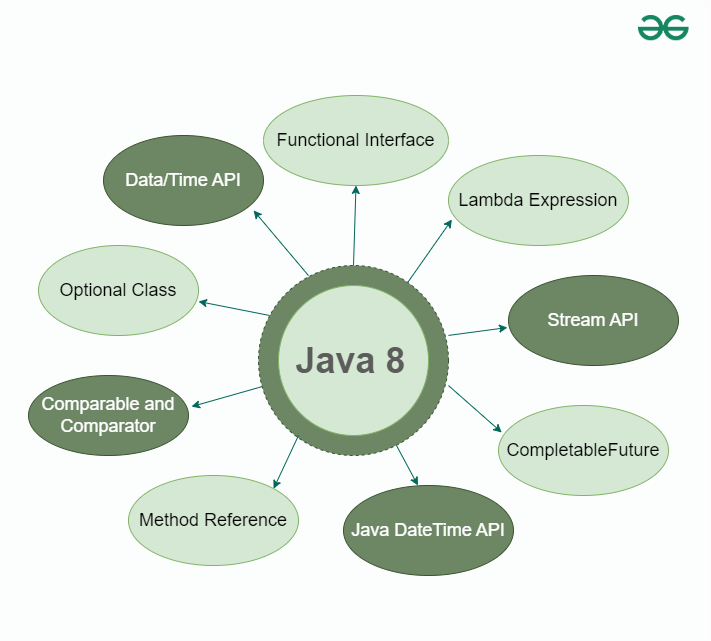
**Java 8 Interview Questions for Freshers**

Here in this section we have compiled some Java 8 basic questions.

**1. What features do you know or use in Java 8?**

Here you can list down all the key features of Java 8 like,

* Functional Interface
* Lambda Expression
* Stream API
* CompletableFuture
* Java DateTime API
* Method Reference
* Comparable and Comparator
* Optional Class
* Date/Time API



**2. What is Lambda Expression?**

Lambda Expression basically shows an instance of functional interface in other words you could say that it provides a clear and concise way to represent a method of performing functional interface using an expression Lambda Expressions have been added in Java 8 and provide the functionality below.

* This enables to treat any functionality as a method argument, and code as data.
* A Function that can be created independently of any class.
* Lambda expression can be moved around like an object and it can be executed on demand.

**3. What is Stream API in Java 8?**

Stream API is introduced in Java 8 and it is used to process collections of objects with the functional style of coding using the lambda expression. So to understand what is stream API you must have knowledge of both lambda and functional interface.

**4. What is Functional Interface in Java 8?**

An interface with only one abstract method is known as a functional interface but there is no restriction, in a functional interface you can have n number of default methods and static methods.

**5. What is Stream in Java 8?**

A stream is a sequence of objects that helps different methods that can be pipelined to produce the desired outcome. The features of Java Stream are:

* Stream is not a data structure rather it takes input from Collections, Arrays, I/O channels.
* Stream doesn’t change the original data structure they only provide the result as per the pipeline methods.

**6. When to use map and flatMap?**

* **map():** It is used where we have to map the elements of a particular collection to a specific function, and then we need to return the stream that contains the updated results.
  + **Example:** Multiply all the elements of a list by 3 and return the updated list.
* **flatMap():** It is used where we have to transform or flatten the string, as we can’t flatten our string using map().
  + **Example:**Get the first Character of all the String present in a List of Strings and return the result in form of a stream.

**7. Can we extend a functional interface from another functional interface?**

Yes, we can extend but if you extend that your functional interface will not act as a functional interface because it will find multiple abstract methods inside that. You may observe the thing by demonstrating a sample code in your local Java IDE.

**8. What are the advantages of Lambda Expression?**

* Avoid writing anonymous implementation
* Saves a lot of code
* Code is directly readable without interpretation

**9. Differentiate Between Comparable and Comparator in Java.**

Java provides two interfaces for configuring objects using class data members:

* Comparable
* Comparator

**Comparable Interface:**

Comparable object can be compared to another objects. To compare its instances, the class itself must implement the java.lang.Comparable interface. Consider a Movie class that has members like rating, name, and year. Suppose we want to sort the list of Movies by release year. We can use the Comparable interface with the Movie class, and override the compareTo() method of the Comparable interface.

**Comparator Interface:**

Unlike Comparable, Comparator is external to the element type we are comparing. There is a special category. We create several different classes (using Comparator) to compare different members. The Collections class has a second sort() method and accepts a Comparator. The sort() method calls compare() to sort the objects.

**10. Tell a few functional interfaces which are already there before Java 8?**

To answer this question you can tell the below interfaces

* Runnable
* Callable
* Comparator

**11. What are all functional interfaces introduced in Java 8?**

* Function
* Predicate
* Consumer
* Supplier

**12. Tell a few stream methods you used in your project?**

* filter
* forEach
* sorted
* map
* flatMap
* reduce
* groupingBy
* collect

***Note****: The interviewer might ask you to explain some methods in detail.*

**13. What are the disadvantages of Lambda expression?**

* Hard to use without an IDE
* Complex to debug

**14. What is Optional Class in Java 8?**

In Java 8, Optional Class is a container object.

* The Optional class used to represent a value that may be present or may not be.
* This class helps in avoiding null pointer exceptions by providing methods to check the presence of a value before accessing it.
* This helps null values handling more effectively.

**Example:**

Optional<String> optionalName = Optional.ofNullable("John");  
  
// Check if value is present  
if (optionalName.isPresent()) {  
 System.out.println("Name is present: " + optionalName.get());  
} else {  
 System.out.println("Name is not present");  
}

**15. Provide Some Optional Methods in Java 8.**

Some Optional methods are described below.

* **of**: It creates an Optional with a non-null value.
* **ofNullable**: It creates an Optional with a given nullable value.
* **empty**: It creates an empty Optional.
* **isPresent**: This checks whether the Optional contains a non-null value.
* **get**: It gets the value if present, otherwise it throws an exception i.e. NoSuchElementException.
* **orElse**: It returns the value if present, otherwise returns the specified default value.
* **orElseGet**: It returns the value if present, otherwise it returns the result of invoking the supplier function.
* **orElseThrow**: It returns the value if present, otherwise it throws an exception produced by the provided supplier.
* **map**: It applies a function to the value if present and return a new Optional with the result, or return an empty Optional if no value is present.
* **filter**: It applies a predicate to the value if present and return an Optional with the value if it matches the predicate, otherwise return an empty Optional.

**16. What is Date-Time API in Java 8?**

The Date-Time API in Java 8 provides a set of classes for date-time conversions, including timelines and advanced programming.

* It imports the **java.time** package, and this package contains **LocalDate, LocalTime, LocalDateTime, ZonedDateTime,** and other classes.
* This API provides better robustness, consistency and thread safety compared to legacy Date and Calendar classes.

**17. What is Optional equals() method in Java?**

In Java, the **equals()** method of the Optional class is used to compare two Optional objects for equality.

* It returns true if both the Optional objects contains the same value.
* And, it returns false if both does not contain the same value.

**Illustration**:

import java.util.Optional;  
  
public class Main   
{  
 public static void main(String args[])   
{  
 // Creating Optional objects  
 Optional<String> opt1 = Optional.of("Sweta");  
 Optional<String> opt2 = Optional.of("Sweta");  
 Optional<String> opt3 = Optional.of("Dash");  
  
 // Comparing Optional objects  
 System.out.println(opt1.equals(opt2)); // true  
 System.out.println(opt1.equals(opt3)); // false  
 }  
}

**18. What is Default Methods In Java 8?**

In Java 8, Default methods allows interfaces to have method implementations. This means that interfaces can contain concrete methods along with the abstract methods. The Default methods are defined using the **default** keyword.

**Illustration:**

interface Vehicle   
{  
 // Abstract method  
 void start();  
  
 // Default method  
 default void stop()   
{  
 System.out.println("Vehicle stopped");  
 }  
}  
  
class Car implements Vehicle   
{  
 @Override  
 public void start()   
{  
 System.out.println("Car started");  
 }  
}  
  
public class Main   
{  
 public static void main(String args[])   
{  
 Car car = new Car();  
 car.start(); // Output: Car started  
 car.stop(); // Output: Vehicle stopped  
 }  
}

**19. How are functional interfaces and Lambda Expressions related?**

Functional interfaces in Java are interfaces that only contains one abstract method.

* Lambda expressions provide a simple way to implement functional interfaces.
* Lambda expressions can be used wherever functional interfaces are needed.
* This allows us to write expressive and concise code.

**Illustration:**

// Functional interface  
interface MyFunctionalInterface {  
 void myMethod();  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Lambda expression for implemention of the functional interface  
 MyFunctionalInterface myLambda = () -> System.out.println("Hello Lambda!");  
   
 // calling method, using lambda expression  
 myLambda.myMethod();  
 }  
}

**20. What is ArrayList forEach() method in Java?**

In Java, the forEach() method is used to iterate over each ArrayList element.

* It performs specified operation for each element.
* It simplifies iteration and shortens the code.
* It takes a Consumer as a parameter, which represents the action to be performed on each element.

ArrayList<Integer> numbers = new ArrayList<>();  
numbers.add(1);  
numbers.add(2);  
numbers.add(3);  
  
numbers.forEach(num -> System.out.println(num));  
  
**Output:**  
1  
2  
3

**Java 8 Interview Questions for Experienced**

Once you have gained confidence after solving the basic questions, let’s increase the level of questions. Here in this section, we have listed more complex Java 8 questions.

**21. How to find duplicate elements in a Stream in Java?**

Java

*// Java program to find the duplicate*

*// elements in a Stream using Set*

**import** **java.util.\***;

**import** **java.util.stream.\***;

**public** **class** **GfG**

{

*// Function to find the*

*// duplicates in a Stream*

**public** **static** <T> Set<T>

findDuplicateInStream(Stream<T> stream)

{

*// Set for storing the duplicate elements*

Set<T> items = **new** HashSet<>();

*// Returning the set of duplicate elements*

**return** stream

*// Set.add() returns false*

*// if the element was*

*// already present in the set.*

*// Hence filter such elements*

.filter(n -> !items.add(n))

*// Collect duplicate elements*

*// in the set*

.collect(Collectors.toSet());

}

*// Driver code*

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

{

*// Initial stream*

Stream<Integer> stream

= Stream.of(2, 17, 5,

20, 17, 30,

4, 23, 59, 23);

*// Print the found duplicate elements*

System.out.println(

findDuplicateInStream(stream));

}

}

**Output**

[17, 23]

**22. Count occurrence of a given character in a string using Stream API in Java.**

Java

*// Java program to count occurrences*

*// of a character using Stream*

**import** **java.util.stream.\***;

**class** **GFG**

{

**public** **static** long count(String s, char ch)

{

*// converting the string to an IntStream of the character codes,*

*// filter by the character code of the specified character,*

*// and count the occurrences*

**return** s.chars()

.filter(c -> c == ch)

.count();

}

*// Main method to test count method*

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

{

String str = "geeksforgeeks";

char c = 'g';

System.out.println(count(str, c));

}

}

**Output**

2

**23. How to get Slice of a Stream in Java?**

Java

*// Java program to get slice of a stream using*

*// Stream skip() and limit()*

**import** **java.util.\***;

**import** **java.util.stream.Stream**;

**class** **GFG**

{

*// Method to get a slice of a stream from startIndex to endIndex*

**public** **static** <T> Stream<T>

getSliceOfStream(Stream<T> stream, int startIndex,

int endIndex)

{

**return** stream

*// Skip elements until the startIndex*

.skip(startIndex)

*// Limit the stream to elements between startIndex and endIndex*

.limit(endIndex - startIndex + 1);

}

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

{

*// Create a list of integers*

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

**for** (int i = 10; i <= 19; i++)

list.add(i);

*// Get a stream from the list*

Stream<Integer> intStream = list.stream();

*// Print the original list*

System.out.println("List: " + list);

*// Get a slice of the stream from index 3 to 7*

Stream<Integer>

sliceOfIntStream = getSliceOfStream(intStream, 3, 7);

*// Print the slice of the stream*

System.out.println("\nSlice of Stream:");

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

}

}

**Output**

List: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Slice of Stream:

13

14

15

16

17

**24. How to Reverse elements of a Parallel Stream in Java?**

Java

*// Java program to reverse elements*

*// of a parallel Stream*

**import** **java.util.\***;

**import** **java.util.stream.\***;

**class** **GFG**

{

*// Generic function to reverse*

*// the elements of the parallel stream*

**public** **static** <T> Collector<T, ?, Stream<T> > reverseStream()

{

**return** Collectors

.collectingAndThen(Collectors.toList(),

list -> {

Collections.reverse(list);

**return** list.stream();

});

}

*// Driver code*

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

{

*// Get the parallel stream*

List<Integer> lists = Arrays.asList(217, 317, 417, 517);

Stream<Integer> stream = lists.parallelStream();

*// Reverse and print the elements*

stream.collect(reverseStream())

.forEach(System.out::println);

}

}

**Output**

517

417

317

217

**25. Write a Program to Iterate over a Stream with Indices in Java 8.**

Java

*// Java program demonstration to iterate over Stream with Indices*

**import** **java.util.stream.IntStream**;

**class** **GFG** {

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

*// Array of Strings*

String[] array = { "G", "E", "E", "k" };

*// Iterating over the indices of an array*

IntStream

*// Generate indices from 0 to array length*

.range(0, array.length)

*// Map each index to its corresponding string representation*

.mapToObj(index -> String.format("%d -> %s", index, array[index]))

*// print each and every element of the stream*

.forEach(System.out::println);

}

}

**Output**

0 -> H

1 -> E

2 -> L

3 -> L

4 -> O

**26. What is CompletableFuture?**

CompletableFuture is just an extension of the future object introduced in JDK5.

* In Java, CompletableFuture is used for asynchronous programming.
* Asynchronous programming is a method of writing non-blocking code by executing a task on a separate thread than the main application thread.
* And, notifies the main thread of its progress, completion, or failure.

**27. Why CompletableFuture why not Future?**

* Future cannot be manually completed.
* Multiple Futures can’t be chained together.
* We can’t combine multiple Futures together.
* No exception handling.

**28. What is method reference in Java 8?**

Method reference is a concise way to use a lambda expression for calling a method directly. It simplifies the code by providing a shorthand notation. are four types of method references that are listed below:

* Static Method Reference
* Instance Method Reference of a particular object
* Referencing an instance method of an unspecified object belonging to a specific class.
* Constructor Reference.

**Example:**

numList.stream().filter(n -> n > 5).sorted().forEach(System.out::println);

**29. What is MetaSpace in Java 8?**

In Java 8, Metaspace stores class metadata in native memory, separate from the heap. It can dynamically expand, overcoming size limitations, and enhances garbage collection efficiency, auto-tuning, and metadata distribution.

* It is used by the JVM to store metadata about loaded classes and methods.
* It replaces the PermGen space, offering dynamic allocation, separate memory management from the heap, and improved garbage collection, thereby mitigating PermGen space errors.

**30. What is Java class dependency analyser in Java 8?**

The Java Class Dependency Analyzer in Java 8 is a tool for analyzing dependencies between classes in a Java application.

* It helps in understanding the structure and interactions within a codebase.
* Useful for analyzing dependencies and managing code complexity.
* It can provide insights into potential refactoring or optimizations.
* Typically visualized through diagrams or dependency graphs for easier comprehension.

**Java 8 MCQs**

Once you have explored all the Java 8 interview questions, try this section to upscale you Java 8 skill more. Here we have listed some important Java 8 MCQ to boost your Java knowledge.

**Q. What is a key feature introduced in Java 8?**

A. Lambda Expressions

B. Annotations

C. Generics

D. Checked Exceptions

*Answer: A) Lambda Expressions*

**Q. Which interface in Java 8 is used to represent a sequence of elements?**

A. List

B. Queue

C. Stream

D. Set

*Answer: C. Stream*

**Q. Which functional interface is used to represent a function that accepts two arguments and produces a result?**

A. Function

B. BiConsumer

C. BiFunction

D. Predicate

*Answer: C. BiFunction*

**Q. What does the forEach method do in Java 8 Stream API?**

A. Filters the stream elements

B. Maps the stream elements

C. Applies an action to each element of the stream

D. Terminates the stream

*Answer: C. Applies an action to each element of the stream*

**Q. Which method is used to merge two streams in Java 8 Stream API?**

A. concat()

B. merge()

C. join()

D. combine()

*Answer: A. concat()*

**Q. What does the peek method do in Java 8 Stream API?**

A. Removes elements from the stream

B. Modifies elements in the stream

C. Applies an action to each element of the stream without changing its contents

D. Terminates the stream

*Answer: C. Applies an action to each element of the stream without changing its contents*

**Q. Which collector is used to collect elements into an immutable list in Java 8 Stream API?**

A. toList()

B. toSet()

C. toMap()

D. toImmutableList()

*Answer: D. toImmutableList()*

**Q. What is the purpose of the Optional class introduced in Java 8?**

A. To handle checked exceptions

B. To represent an object that may or may not contain a value

C. To perform mathematical operations

D. To represent an immutable list

*Answer: B. To represent an object that may or may not contain a value*

**Q. Which interface is used to sort objects in Java 8?**

A. Comparator

B. Sortable

C. Sorter

D. Comparable

*Answer: A. Comparator*

**Q. What does the reduce method do in Java 8 Stream API?**

A. Filters the stream elements based on a predicate

B. Terminates the stream and produces a single result by iteratively applying a binary operation

C. Maps each element of the stream to a new value

D. Concatenates the stream elements into a single string

*Answer: B. Terminates the stream and produces a single result by iteratively applying a binary operation*

**Conclusion**

In closing, preparing for **Java 8 Interview Questions** is critical for any Java developer. **Java 8** introduced fantastic features like **Lambda Expressions** and the **Stream API**, which make code more efficient and expressive. You’ll be well-prepared to demonstrate your abilities in any interview after reviewing the important questions and answers we discussed.

**1Given a list of integers, find out all the even numbers that exist in the list using Stream functions?**

**import java.util.\*;**

|  |
| --- |
| **import java.util.stream.\*;  public class EvenNumber{  public static void main(String args[]) {  List<Integer> list = Arrays.asList(10,15,8,49,25,98,32);  list.stream()  .filter(n -> n%2 == 0)  .forEach(System.out::println);  }  }** |

**Output:   
10, 8, 98, 32**

**2. Given a list of integers, find out all the numbers starting with 1 using Stream functions?**

|  |
| --- |
| **import java.util.\*; import java.util.stream.\*;  public class NumberStartingWithOne{  public static void main(String args[]) {  List<Integer> myList = Arrays.asList(10,15,8,49,25,98,32);  myList.stream()  .map(s -> s + "") // Convert integer to String  .filter(s -> s.startsWith("1"))  .forEach(System.out::println);  } }** |

**Output:  
10, 15**

**3. How to find duplicate elements in a given integers list in java using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
  
public class DuplicateElements {  
 public static void main(String args[]) {  
 List<Integer> myList = Arrays.asList(10,15,8,49,25,98,98,32,15);  
 Set<Integer> set = new HashSet();  
 myList.stream()  
 .filter(n -> !set.add(n))  
 .forEach(System.out::println);  
 }  
}  
  
Output:  
98, 15  
  
// Or you can also try using distinct() keyword  
  
public static void getDataWithoutDuplicates() {  
 List<Integer> myList = Arrays.asList(1, 1, 85, 6, 2, 3, 65, 6, 45, 45, 5662, 2582, 2, 2, 266, 666, 656);  
 myList.stream().distinct().forEach(noDuplicateData -> System.out.println(noDuplicateData));  
 }  
  
Output : 1 85 6 2 3 65 45 5662 2582 266 666 656  
  
  
  
//Or you can also use below   
  
public static void getDataWithoutDuplicates() {  
 List<Integer> myList = Arrays.asList(1, 1, 85, 6, 2, 3, 65, 6, 45, 45, 5662, 2582, 2, 2, 266, 666, 656);  
 Set<Integer> set = new HashSet<>(myList);  
   
 // Convert the set back to a list if needed  
 List<Integer> uniqueData = set.stream().collect(Collectors.toList());  
   
 // Print the unique elements  
 uniqueData.forEach(System.out::println);  
 }  
  
Output : 1 65 2 3 6 266 45 656 85 2582 666 5662**

**4. Given the list of integers, find the first element of the list using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
  
public class FindFirstElement{  
 public static void main(String args[]) {  
 List<Integer> myList = Arrays.asList(10,15,8,49,25,98,98,32,15);  
 myList.stream()  
 .findFirst()  
 .ifPresent(System.out::println);  
 }  
}  
  
Output:  
10**

**5. Given a list of integers, find the total number of elements present in the list using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
  
public class FindTheTotalNumberOfElements{  
 public static void main(String args[]) {  
 List<Integer> myList = Arrays.asList(10,15,8,49,25,98,98,32,15);  
 long count = myList.stream()  
 .count();  
 System.out.println(count);   
 }  
}  
  
Output:  
9**

**6. Given a list of integers, find the maximum value element present in it using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
  
public class FindMaxElement{  
 public static void main(String args[]) {  
 List<Integer> myList = Arrays.asList(10,15,8,49,25,98,98,32,15);  
 int max = myList.stream()  
 .max(Integer::compare)  
 .get();  
 System.out.println(max);   
 }  
}  
  
Output:  
98**

**7. Given a String, find the first non-repeated character in it using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
import java.util.function.Function;  
  
public class FirstNonRepeated{  
 public static void main(String args[]) {  
 String input = "Java articles are Awesome";  
   
 Character result = input.chars() // Stream of String   
 .mapToObj(s -> Character.toLowerCase(Character.valueOf((char) s))) // First convert to Character object and then to lowercase   
 .collect(Collectors.groupingBy(Function.identity(), LinkedHashMap::new, Collectors.counting())) //Store the chars in map with count   
 .entrySet()  
 .stream()  
 .filter(entry -> entry.getValue() == 1L)  
 .map(entry -> entry.getKey())  
 .findFirst()  
 .get();  
 System.out.println(result);   
 }  
}  
  
Output:  
j**

**8. Given a String, find the first repeated character in it using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
import java.util.function.Function;  
  
public class FirstRepeated{  
 public static void main(String args[]) {  
 String input = "Java Articles are Awesome";  
  
 Character result = input.chars() // Stream of String   
 .mapToObj(s -> Character.toLowerCase(Character.valueOf((char) s))) // First convert to Character object and then to lowercase   
 .collect(Collectors.groupingBy(Function.identity(), LinkedHashMap::new, Collectors.counting())) //Store the chars in map with count   
 .entrySet()  
 .stream()  
 .filter(entry -> entry.getValue() > 1L)  
 .map(entry -> entry.getKey())  
 .findFirst()  
 .get();  
 System.out.println(result);   
 }  
}  
  
  
Output:  
a**

**9. Given a list of integers, sort all the values present in it using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
import java.util.function.Function;  
  
public class SortValues{  
 public static void main(String args[]) {  
 List<Integer> myList = Arrays.asList(10,15,8,49,25,98,98,32,15);  
  
 myList.stream()  
 .sorted()  
 .forEach(System.out::println);  
 }  
}  
  
Output:  
 8  
10  
15  
15  
25  
32  
49  
98  
98**

**10. Given a list of integers, sort all the values present in it in descending order using Stream functions?**

**import java.util.\*;  
import java.util.stream.\*;  
import java.util.function.Function;  
  
public class SortDescending{  
 public static void main(String args[]) {  
 List<Integer> myList = Arrays.asList(10,15,8,49,25,98,98,32,15);  
  
 myList.stream()  
 .sorted(Collections.reverseOrder())  
 .forEach(System.out::println);  
 }  
}  
  
Output:  
98  
98  
49  
32  
25  
15  
15  
10  
8**

**11. Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.**

**public boolean containsDuplicate(int[] nums) {  
 List<Integer> list = Arrays.stream(nums)  
 .boxed()  
 .collect(Collectors.toList());  
 Set<Integer> set = new HashSet<>(list);  
 if(set.size() == list.size()) {  
 return false;  
 }   
 return true;  
 }  
  
Input: nums = [1,2,3,1]  
Output: true  
  
Input: nums = [1,2,3,4]  
Output: false**

**12. How will you get the current date and time using Java 8 Date and Time API?**

**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  
 }  
}**

**13. Write a Java 8 program to concatenate two Streams?**

**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 -&gt; System.out.print(str + " "));  
   
 // Printed the Concatenated Stream  
   
 }  
}**

**14. Java 8 program to perform cube on list elements and filter numbers greater than 50.**

**import java.util.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 List<Integer> integerList = Arrays.asList(4,5,6,7,1,2,3);  
 integerList.stream()  
 .map(i -> i\*i\*i)  
 .filter(i -> i>50)  
 .forEach(System.out::println);  
 }  
}   
  
Output:  
64  
125  
216  
343**

**15. Write a Java 8 program to sort an array and then convert the sorted array into 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 \*/  
 }  
}**

**16. How to use map to convert object into Uppercase in Java 8?**

**public class Java8 {  
   
 public static void main(String[] args) {  
 List<String> nameLst = names.stream()  
 .map(String::toUpperCase)  
 .collect(Collectors.toList());  
 System.out.println(nameLst);  
 }  
}  
  
output:  
AA, BB, CC, DD**

**17. How to convert a List of objects into a Map by considering duplicated keys and store them in sorted order?**

**public class TestNotes {  
  
 public static void main(String[] args) {  
  
 List<Notes> noteLst = new ArrayList<>();  
 noteLst.add(new Notes(1, "note1", 11));  
 noteLst.add(new Notes(2, "note2", 22));  
 noteLst.add(new Notes(3, "note3", 33));  
 noteLst.add(new Notes(4, "note4", 44));  
 noteLst.add(new Notes(5, "note5", 55));  
  
 noteLst.add(new Notes(6, "note4", 66));  
  
  
 Map<String, Long> notesRecords = noteLst.stream()  
 .sorted(Comparator  
 .comparingLong(Notes::getTagId)  
 .reversed()) // sorting is based on TagId 55,44,33,22,11  
 .collect(Collectors.toMap  
 (Notes::getTagName, Notes::getTagId,  
 (oldValue, newValue) -> oldValue,LinkedHashMap::new));  
// consider old value 44 for dupilcate key  
// it keeps order  
 System.out.println("Notes : " + notesRecords);  
 }  
}**

**18. How to count each element/word from the String ArrayList in Java8?**

**public class TestNotes {  
  
 public static void main(String[] args) {  
 List<String> names = Arrays.asList("AA", "BB", "AA", "CC");  
 Map<String,Long> namesCount = names  
 .stream()  
 .collect(  
 Collectors.groupingBy(  
 Function.identity()  
 , Collectors.counting()  
 ));  
 System.out.println(namesCount);  
 }  
}  
  
Output:  
{CC=1, BB=1, AA=2}**

**19. How to find only duplicate elements with its count from the String ArrayList in Java8?**

**public class TestNotes {  
  
 public static void main(String[] args)   
 List<String> names = Arrays.asList("AA", "BB", "AA", "CC");  
 Map<String,Long> namesCount = names  
 .stream()  
 .filter(x->Collections.frequency(names, x)>1)  
 .collect(Collectors.groupingBy  
 (Function.identity(), Collectors.counting()));  
 System.out.println(namesCount);  
 }  
}  
  
Output:  
{AA=2}**

**20. How to check if list is empty in Java 8 using Optional, if not null iterate through the list and print the object?**

**Optional.ofNullable(noteLst)  
 .orElseGet(Collections::emptyList) // creates empty immutable list: [] in case noteLst is null  
 .stream().filter(Objects::nonNull) //loop throgh each object and consider non null objects  
 .map(note -> Notes::getTagName) // method reference, consider only tag name  
 .forEach(System.out::println); // it will print tag names**

**21. Write a Program to find the Maximum element in an array?**

**public static int findMaxElement(int[] arr) {  
 return Arrays.stream(arr).max().getAsInt();  
}  
  
Input: 12,19,20,88,00,9  
output: 88**

**22. Write a program to print the count of each character in a String?**

**public static void findCountOfChars(String s) {  
Map<String, Long> map = Arrays.stream(s.split(""))  
 .map(String::toLowerCase)  
 .collect(Collectors  
 .groupingBy(str -> str,   
 LinkedHashMap::new, Collectors.counting()));  
}  
  
Input: String s = "string data to count each character";  
Output: {s=1, t=5, r=3, i=1, n=2, g=1, =5, d=1, a=5, o=2, c=4, u=1, e=2, h=2}**

**1- Write a Java Program to find whether a number is prime or not.**

**import java.util.stream.IntStream;  
  
//Prime numbers are natural numbers that are divisible by only 1 and the number itself  
public class FindPrimeNumber {  
  
 public static void main(String[] args) {  
 System.out.println(" Is Prime No : " + isPrimeByJava8(7));  
 }  
 private static boolean isPrimeByJava8(int n) {  
 if (n == 0 || n == 1) {  
 return false;  
 }  
 if (n == 2) {  
 return true;  
 }  
 boolean isPrime = IntStream.rangeClosed(2, n / 2).noneMatch(x -> n % x == 0);  
 return isPrime;  
 }  
}**

**2-Write a Java Program to check Armstrong numbers.**

**//Armstrong number is the number in any given number base, which forms the total of the same number,  
//when each of its digits is raised to the power of the number of digits in the number.  
public class FindArmstrongNumber {  
  
 public static void main(String[] args) {  
 System.out.println("Is Armstrong Number : " + isArmstrongNumberByJava8(371));  
 }  
 private static boolean isArmstrongNumberByJava8(int n) {  
 int len = String.valueOf(n).length();  
 int sum = String.valueOf(n).chars().map(ch -> Character.digit(ch, 10))  
 .map(digit -> (int) Math.pow(digit, len))  
 .sum();  
 return sum == n;  
 }  
}**

**3- Write a program to print duplicate numbers.**

**public class FindDuplicateNumber {  
  
 public static void main(String[] args) {  
 int a[] = { 1, 4, 5, 2, 12, 34, 2, 11 };  
 System.out.println("Duplicate number by java 1.8 : " + findDuplicateNumberByJava1\_8(a));  
 }  
 private static int findDuplicateNumberByJava1\_8(int[] a) {  
 Map<Integer, Long> map = Arrays.stream(a).boxed()  
 .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));  
  
 int duplicate = map.keySet().stream().filter(x -> map.get(x) > 1).findFirst().orElse(0);  
  
 // If there have more then one duplicate then used this  
 map.keySet().stream().filter(x -> map.get(x) > 1).forEach(System.out::println);  
 return duplicate;  
 }  
}**

**4-Write a program to find whether a string or number is palindrome or not.**

**//A Palindromic number is a number (such as 16461) that   
// remains the same when its digits are reversed.  
public class FindPalindromeNumber {  
 public static void main(String[] args) {  
 System.out.println(" Is Palindrome No : " + isPalindromeNumberByJava8(16461));  
 }  
  
 private static boolean isPalindromeNumberByJava8(int n) {  
 String value = String.valueOf(n);  
 int len = value.length();  
 boolean isPalindromeNumber = IntStream.range(0, len / 2)  
 .anyMatch(index -> value.charAt(index) == value.charAt(len - index - 1));  
 return isPalindromeNumber;  
 }  
}**

**5- Write a program to print duplicate strings.**

**public class FindDuplicateInString {  
  
 public static void main(String[] args) {  
 String input = "JavaAPI";  
  
 List<Character> duplicateList= input.chars().mapToObj(x -> Character.toUpperCase((char) x))  
 .collect(Collectors.groupingBy(Function.identity(), LinkedHashMap::new, Collectors.counting()))  
 .entrySet().stream().filter(x -> x.getValue() > 1L).map(Entry::getKey).collect(Collectors.toList());  
  
 System.out.println(duplicateListj);  
 }  
}**

**6- Write a program to print the Fibonacci Series.**

**//The Fibonacci series is the sequence of numbers,   
// where every number is the sum of the preceding two numbers.  
public class FindFibonacciSeries {  
  
 public static void main(String[] args) {  
 findFibonacciSeriesByJava8();  
 }  
  
 private static void findFibonacciSeriesByJava8() {  
 Stream.iterate(new int[] { 0, 1 }, f -> new int[] { f[1], f[0] + f[1] }).limit(10).map(f -> f[0])  
 .forEach(System.out::println);  
 }  
}**

**7- Write a program to find min and max numbers in the array.**

**public class FindMinAndMaxInArray {  
  
 public static void main(String[] args) {  
  
 int a[] = { 2, 3, 1, 22, 11, 33, 5 };  
  
// Find the max number  
 int max = Arrays.stream(a).boxed().max(Integer::compareTo).get();  
  
 System.out.println("Max Value by java 8 : "+max);  
 findMaxValue(a);  
  
 // Find the min number  
 int min = Arrays.stream(a).boxed().max(Comparator.reverseOrder()).get();  
  
 System.out.println("Min Value by java 8 : "+min);  
 findMinValue(a);  
  
 }  
}**

**8-Write a program to find Min And Max In an Array Without using the max function**

**public class FindMinAndMaxInArrayWithoutMaxFunction {  
 public static void main(String[] args) {  
 int arr[] = { 2, 3, 1, 22, 11, 33, 5 };  
 int max = Arrays.stream(arr).boxed().reduce(Integer.MIN\_VALUE, (a, b) -> Integer.max(a, b)).intValue();  
 int min = Arrays.stream(arr).boxed().reduce(Integer.MAX\_VALUE, (a, b) -> Integer.min(a, b)).intValue();  
 System.out.println("Max : " + max + " Min : " + min);  
  
 System.out.println();  
 // Using the method reference;  
 int max1 = Arrays.stream(arr).boxed().reduce(Integer::max).get();  
 int min1 = Arrays.stream(arr).boxed().reduce(Integer::min).get();  
 System.out.println("Max : " + max1 + " Min : " + min1);  
 }  
}**

**9-Write a program to find the second-highest number in an array.**

**public class FindSecondHighehestNumber {  
  
 public static void main(String[] args) {  
 int a[] = { 3, 6, 32, 1, 8, 5, 31, 22 };  
  
 int secundMax = Arrays.stream(a).boxed().sorted(Comparator.reverseOrder()).skip(1).findAny().get();  
 System.out.println(secundMax);  
   
 }  
}**

**10- Write a program to Find the Second Lowest Number.**

**public class FindSecondLowestNumber {  
  
 public static void main(String[] args) {  
 int a[] = { 3, 6, 32, 1, 8, 5, 31, 22, 2 };  
 int secundMin = Arrays.stream(a).boxed().sorted().skip(1).findAny().get();  
 System.out.println(secundMin);  
 }  
}**

**11- Write a program to print the First Not Repeated Char in string.**

**public class FirstNotRepetedChar {  
  
 public static void main(String[] args) {  
 String input = "Java Stream API is very good concept";  
  
 char firstNotRepetedChar = input.chars().mapToObj(x -> Character.toUpperCase((char) x))//converting the object format  
 .collect(Collectors.groupingBy(Function.identity(), LinkedHashMap::new, Collectors.counting()))// find duplicate freq in linkedHashMap  
 .entrySet().stream().filter(x -> x.getValue() == 1L).map(x -> x.getKey()).findFirst().get();//filtering the freq which is not first time  
  
 System.out.println("First non repeated char : " + firstNotRepetedChar);  
  
 }  
}**

**12- Write a program to a Flattering list of objects.**

**public class FlatteringList {  
  
 public static void main(String[] args) {  
 List<Integer> oddList = Arrays.asList(1, 3, 5, 7, 9, 11);  
 List<Integer> evenList = Arrays.asList(2, 4, 6, 8, 10);  
 List<List<Integer>> listOfList = Arrays.asList(oddList, evenList);  
 System.out.println(listOfList);  
 List<Integer> flatList = listOfList.stream().flatMap(list -> list.stream()).collect(Collectors.toList());  
  
 System.out.println(flatList);  
 }  
}**

**13- Write a program to find the majority element in an array.**

**///Find the majority element in the array. A majority element in an array A[] of size n is   
// an element that appears more than n/2 times.  
public class MajorityElementInStream {  
  
 public static void main(String[] args) {  
 int majorityArray[] = { 5, 3, 2, 1, 2, 4, 3, 2, 2, 6, 2, 3, 2, 2, 2 };  
 int notMajorityArray[] = { 3, 6, 32, 1, 8, 5, 31, 22 };  
 printMajorityElementByJava8(majorityArray);  
 printMajorityElementByJava8(notMajorityArray);  
 }  
  
 private static void printMajorityElementByJava8(int a[]) {  
 int majorityElement = Arrays.stream(a).boxed()  
 .collect(Collectors.groupingBy(Function.identity(), HashMap::new, Collectors.counting())).entrySet()  
 .stream().filter(k -> k.getValue() >= a.length / 2).map(Entry::getKey).findFirst().orElse(0);  
 System.out.println(majorityElement);  
 }  
}**

**14- Write a program to Print Even and Odd Numbers.**

**public class PrintEvenOddNumber {  
  
 public static void main(String[] args) {  
  
 printEvenNumberByJava8();  
 printOddNumberByJava8();  
  
 }  
  
 private static void printEvenNumberByJava8() {  
  
 IntStream.rangeClosed(0, 10).filter(x -> x % 2 == 0).forEach(System.out::println);  
  
 }  
  
 private static void printOddNumberByJava8() {  
  
 IntStream.rangeClosed(0, 10).filter(x -> x % 2 != 0).forEach(System.out::println);  
  
 }  
}**

**15- Write a program to sort the 2 arrays in ascending order.**

**public class Sort2ArrayInAssendingOrder {  
  
 public static void main(String[] args) {  
 int i[] = { 4, 5, 13, 22 };  
 int j[] = { 0, 9, 3, 7, 12, 11 };  
 List<Integer> sortedList= Stream.concat(Arrays.stream(i).boxed(), Arrays.stream(j).boxed()).sorted()  
 .collect(Collectors.toList());  
 System.out.println(sortedList);  
 }  
}**

**16-Write a program to sort the array.**

**public class SortArrayInReverseOrder {  
  
 public static void main(String[] args) {  
 int a[] = { 3, 6, 32, 1, 8, 5, 31, 22 };  
  
 List<Integer> newArra = Arrays.stream(a).boxed().sorted().collect(Collectors.toList());  
  
 System.out.println(newArra);  
  
 Arrays.sort(a);  
 System.out.println(Arrays.toString(a));  
 }  
}**

**17-Write a program to sum an array.**

**public class SumArray {  
  
 public static void main(String[] args) {  
 int a[] = { 3, 6, 32, 1, 8, 5, 31, 22 };  
 int sumValue = Arrays.stream(a).boxed().collect(Collectors.summingInt(Integer::intValue));  
 System.out.println(sumValue);  
 }  
}**

**18-Write a program to sum an array without using the sum method.**

**public class SumArray {  
  
 public static void main(String[] args) {  
 int a[] = { 3, 6, 32, 1, 8, 5, 31, 22 };  
 int sum = Arrays.stream(a).boxed().reduce(0, (x, y) -> x + y).intValue();  
 System.out.println(sum);  
 }  
}**

**19- Write a program to append char in char ex-input- {A, B, C} output->[A\_X, B\_Y, C\_Z].**

**public class AppendCharInChar {  
  
 public static void main(String[] args) {  
 Stream<Character> charStream = Stream.of('A', 'B', 'C');  
 charStream.forEach(ch -> {  
 char newChar = (char) (ch + 23);  
 System.out.println(ch + "\_" + newChar);  
 });  
 }  
}**

**20-Write a program to find the only duplicate count list in the List.**

**public class PrintOnlyDuplicateCountList {  
 public static void main(String[] args) {  
 List<String> names = Arrays.asList("Java", "Spring", "JPA", "Java", "Cloud", "JPA");  
 Map<String, Long> namesCount = names.stream().filter(x -> Collections.frequency(names, x) > 1)  
 .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));  
 System.out.println(namesCount);  
 }  
}**

**Solving Real Time Queries Using Java 8 Features -Employee Management System**

Let’s try to solve some of the real time queries faced in the [Employee Management System](https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/) using Java 8 features.

We will be using following *Employee* class and *employeeList* as example while solving the queries.

1) *Employee* Class :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | **class** Employee  {  **int** id;        String name;    **int** age;        String gender;        String department;    **int** yearOfJoining;    **double** salary;    **public** Employee(**int** id, String name, **int** age, String gender, String department, **int** yearOfJoining, **double** salary)      {  **this**.id = id;  **this**.name = name;  **this**.age = age;  **this**.gender = gender;  **this**.department = department;  **this**.yearOfJoining = yearOfJoining;  **this**.salary = salary;      }    **public** **int** getId()      {  **return** id;      }    **public** String getName()      {  **return** name;      }    **public** **int** getAge()      {  **return** age;      }    **public** String getGender()      {  **return** gender;      }    **public** String getDepartment()      {  **return** department;      }    **public** **int** getYearOfJoining()      {  **return** yearOfJoining;      }    **public** **double** getSalary()      {  **return** salary;      }        @Override  **public** String toString()      {  **return** "Id : "+id                  +", Name : "+name                  +", age : "+age                  +", Gender : "+gender                  +", Department : "+department                  +", Year Of Joining : "+yearOfJoining                  +", Salary : "+salary;      }  } |

2) List Of Employees : *employeeList*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | List<Employee> employeeList = **new** ArrayList<Employee>();    employeeList.add(**new** Employee(111, "Jiya Brein", 32, "Female", "HR", 2011, 25000.0));  employeeList.add(**new** Employee(122, "Paul Niksui", 25, "Male", "Sales And Marketing", 2015, 13500.0));  employeeList.add(**new** Employee(133, "Martin Theron", 29, "Male", "Infrastructure", 2012, 18000.0));  employeeList.add(**new** Employee(144, "Murali Gowda", 28, "Male", "Product Development", 2014, 32500.0));  employeeList.add(**new** Employee(155, "Nima Roy", 27, "Female", "HR", 2013, 22700.0));  employeeList.add(**new** Employee(166, "Iqbal Hussain", 43, "Male", "Security And Transport", 2016, 10500.0));  employeeList.add(**new** Employee(177, "Manu Sharma", 35, "Male", "Account And Finance", 2010, 27000.0));  employeeList.add(**new** Employee(188, "Wang Liu", 31, "Male", "Product Development", 2015, 34500.0));  employeeList.add(**new** Employee(199, "Amelia Zoe", 24, "Female", "Sales And Marketing", 2016, 11500.0));  employeeList.add(**new** Employee(200, "Jaden Dough", 38, "Male", "Security And Transport", 2015, 11000.5));  employeeList.add(**new** Employee(211, "Jasna Kaur", 27, "Female", "Infrastructure", 2014, 15700.0));  employeeList.add(**new** Employee(222, "Nitin Joshi", 25, "Male", "Product Development", 2016, 28200.0));  employeeList.add(**new** Employee(233, "Jyothi Reddy", 27, "Female", "Account And Finance", 2013, 21300.0));  employeeList.add(**new** Employee(244, "Nicolus Den", 24, "Male", "Sales And Marketing", 2017, 10700.5));  employeeList.add(**new** Employee(255, "Ali Baig", 23, "Male", "Infrastructure", 2018, 12700.0));  employeeList.add(**new** Employee(266, "Sanvi Pandey", 26, "Female", "Product Development", 2015, 28900.0));  employeeList.add(**new** Employee(277, "Anuj Chettiar", 31, "Male", "Product Development", 2012, 35700.0)); |

**Also Read :**[**Java 8 Lambda Expressions**](https://javaconceptoftheday.com/java-8-lambda-expressions/)

3) Real Time Queries On *employeeList*

**Query 3.1 : How many male and female employees are there in the organization?**

For queries such as above where you need to group the input elements, use the *Collectors.groupingBy()* method. In this query, we use *Collectors.groupingBy()* method which takes two arguments. We pass *Employee::getGender* as first argument which groups the input elements based on *gender* and *Collectors.counting()* as second argument which counts the number of entries in each group.

|  |  |
| --- | --- |
| 1  2  3  4 | Map<String, Long> noOfMaleAndFemaleEmployees=  employeeList.stream().collect(Collectors.groupingBy(Employee::getGender, Collectors.counting()));    System.out.println(noOfMaleAndFemaleEmployees); |

**Output :**

{Male=11, Female=6}

**Query 3.2 : Print the name of all departments in the organization?**

Use *distinct()* method after calling *map(Employee::getDepartment)* on the stream. It will return unique departments.

|  |  |
| --- | --- |
| 1  2  3  4 | employeeList.stream()              .map(Employee::getDepartment)              .distinct()              .forEach(System.out::println); |

**Output :**

HR  
Sales And Marketing  
Infrastructure  
Product Development  
Security And Transport  
Account And Finance

**Query 3.3 : What is the average age of male and female employees?**

Use same method as query 3.1 but pass *Collectors.averagingInt(Employee::getAge)* as the second argument to *Collectors.groupingBy()*.

|  |  |
| --- | --- |
| 1  2  3  4 | Map<String, Double> avgAgeOfMaleAndFemaleEmployees=  employeeList.stream().collect(Collectors.groupingBy(Employee::getGender, Collectors.averagingInt(Employee::getAge)));    System.out.println(avgAgeOfMaleAndFemaleEmployees); |

**Output :**

{Male=30.181818181818183, Female=27.166666666666668}

**Also Read :**[**Java 8 Collectors**](https://javaconceptoftheday.com/java-8-collectors-tutorial/)

**Query 3.4 : Get the details of highest paid employee in the organization?**

Use *Collectors.maxBy()* method which returns maximum element wrapped in an *Optional* object based on supplied *Comparator*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | Optional<Employee> highestPaidEmployeeWrapper=  employeeList.stream().collect(Collectors.maxBy(Comparator.comparingDouble(Employee::getSalary)));    Employee highestPaidEmployee = highestPaidEmployeeWrapper.get();    System.out.println("Details Of Highest Paid Employee : ");    System.out.println("==================================");    System.out.println("ID : "+highestPaidEmployee.getId());    System.out.println("Name : "+highestPaidEmployee.getName());    System.out.println("Age : "+highestPaidEmployee.getAge());    System.out.println("Gender : "+highestPaidEmployee.getGender());    System.out.println("Department : "+highestPaidEmployee.getDepartment());    System.out.println("Year Of Joining : "+highestPaidEmployee.getYearOfJoining());    System.out.println("Salary : "+highestPaidEmployee.getSalary()); |

**Output :**

Details Of Highest Paid Employee :  
==================================  
ID : 277  
Name : Anuj Chettiar  
Age : 31  
Gender : Male  
Department : Product Development  
Year Of Joining : 2012  
Salary : 35700.0

**Query 3.5 : Get the names of all employees who have joined after 2015?**

For such queries which require filtering of input elements, use *Stream.filter()* method which filters input elements according to supplied *Predicate*.

|  |  |
| --- | --- |
| 1  2  3  4 | employeeList.stream()              .filter(e -> e.getYearOfJoining() > 2015)              .map(Employee::getName)              .forEach(System.out::println); |

**Output :**

Iqbal Hussain  
Amelia Zoe  
Nitin Joshi  
Nicolus Den  
Ali Baig

**Query 3.6 : Count the number of employees in each department?**

This query is same as query 3.1 but here we are grouping the elements by *department*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Map<String, Long> employeeCountByDepartment=  employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));    Set<Entry<String, Long>> entrySet = employeeCountByDepartment.entrySet();    **for** (Entry<String, Long> entry : entrySet)  {      System.out.println(entry.getKey()+" : "+entry.getValue());  } |

**Output :**

Product Development : 5  
Security And Transport : 2  
Sales And Marketing : 3  
Infrastructure : 3  
HR : 2  
Account And Finance : 2

**Also Read :**[**Java 8 Streams**](https://javaconceptoftheday.com/java-8-streams-beginners-guide/)

**Query 3.7 : What is the average salary of each department?**

Use the same method as in the above query 3.6, but here pass *Collectors.averagingDouble(Employee::getSalary)* as second argument to *Collectors.groupingBy()* method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Map<String, Double> avgSalaryOfDepartments=  employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.averagingDouble(Employee::getSalary)));    Set<Entry<String, Double>> entrySet = avgSalaryOfDepartments.entrySet();    **for** (Entry<String, Double> entry : entrySet)  {      System.out.println(entry.getKey()+" : "+entry.getValue());  } |

**Output :**

Product Development : 31960.0  
Security And Transport : 10750.25  
Sales And Marketing : 11900.166666666666  
Infrastructure : 15466.666666666666  
HR : 23850.0  
Account And Finance : 24150.0

**Query 3.8 : Get the details of youngest male employee in the product development department?**

For this query, use *Stream.filter()* method to filter male employees in product development department and to find youngest among them, use *Stream.min()* method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Optional<Employee> youngestMaleEmployeeInProductDevelopmentWrapper=  employeeList.stream()              .filter(e -> e.getGender()=="Male" && e.getDepartment()=="Product Development")              .min(Comparator.comparingInt(Employee::getAge));    Employee youngestMaleEmployeeInProductDevelopment = youngestMaleEmployeeInProductDevelopmentWrapper.get();    System.out.println("Details Of Youngest Male Employee In Product Development");    System.out.println("----------------------------------------------");    System.out.println("ID : "+youngestMaleEmployeeInProductDevelopment.getId());    System.out.println("Name : "+youngestMaleEmployeeInProductDevelopment.getName());    System.out.println("Age : "+youngestMaleEmployeeInProductDevelopment.getAge());    System.out.println("Year Of Joinging : "+youngestMaleEmployeeInProductDevelopment.getYearOfJoining());    System.out.println("Salary : "+youngestMaleEmployeeInProductDevelopment.getSalary()); |

**Output :**

Details Of Youngest Male Employee In Product Development :  
———————————————-  
ID : 222  
Name : Nitin Joshi  
Age : 25  
Year Of Joinging : 2016  
Salary : 28200.0

**Query 3.9 : Who has the most working experience in the organization?**

For this query, sort *employeeList* by *yearOfJoining* in natural order and first employee will have most working experience in the organization. To solve this query, we will be using *sorted()* and *findFirst()* methods of *Stream*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | Optional<Employee> seniorMostEmployeeWrapper=  employeeList.stream().sorted(Comparator.comparingInt(Employee::getYearOfJoining)).findFirst();    Employee seniorMostEmployee = seniorMostEmployeeWrapper.get();    System.out.println("Senior Most Employee Details :");    System.out.println("----------------------------");    System.out.println("ID : "+seniorMostEmployee.getId());    System.out.println("Name : "+seniorMostEmployee.getName());    System.out.println("Age : "+seniorMostEmployee.getAge());    System.out.println("Gender : "+seniorMostEmployee.getGender());    System.out.println("Age : "+seniorMostEmployee.getDepartment());    System.out.println("Year Of Joinging : "+seniorMostEmployee.getYearOfJoining());    System.out.println("Salary : "+seniorMostEmployee.getSalary()); |

**Output :**

Senior Most Employee Details :  
—————————-  
ID : 177  
Name : Manu Sharma  
Age : 35  
Gender : Male  
Age : Account And Finance  
Year Of Joinging : 2010  
Salary : 27000.0

**Query 3.10 : How many male and female employees are there in the sales and marketing team?**

This query is same as query 3.1, but here use *filter()* method to filter sales and marketing employees.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Map<String, Long> countMaleFemaleEmployeesInSalesMarketing=  employeeList.stream()              .filter(e -> e.getDepartment()=="Sales And Marketing")              .collect(Collectors.groupingBy(Employee::getGender, Collectors.counting()));    System.out.println(countMaleFemaleEmployeesInSalesMarketing); |

**Output :**

{Female=1, Male=2}

**Query 3.11 : What is the average salary of male and female employees?**

This query is same as query 3.3 where you have found average age of male and female employees. Here, we will be finding average salary of male and female employees.

|  |  |
| --- | --- |
| 1  2  3  4 | Map<String, Double> avgSalaryOfMaleAndFemaleEmployees=  employeeList.stream().collect(Collectors.groupingBy(Employee::getGender, Collectors.averagingDouble(Employee::getSalary)));    System.out.println(avgSalaryOfMaleAndFemaleEmployees); |

**Output :**

{Male=21300.090909090908, Female=20850.0}

**Query 3.12 : List down the names of all employees in each department?**

For this query, we will be using *Collectors.groupingBy()* method by passing *Employee::getDepartment* as an argument.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Map<String, List<Employee>> employeeListByDepartment=  employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment));    Set<Entry<String, List<Employee>>> entrySet = employeeListByDepartment.entrySet();    **for** (Entry<String, List<Employee>> entry : entrySet)  {      System.out.println("--------------------------------------");        System.out.println("Employees In "+entry.getKey() + " : ");        System.out.println("--------------------------------------");        List<Employee> list = entry.getValue();    **for** (Employee e : list)      {          System.out.println(e.getName());      }  } |

**Output :**

————————————–  
Employees In Product Development :  
————————————–  
Murali Gowda  
Wang Liu  
Nitin Joshi  
Sanvi Pandey  
Anuj Chettiar  
————————————–  
Employees In Security And Transport :  
————————————–  
Iqbal Hussain  
Jaden Dough  
————————————–  
Employees In Sales And Marketing :  
————————————–  
Paul Niksui  
Amelia Zoe  
Nicolus Den  
————————————–  
Employees In Infrastructure :  
————————————–  
Martin Theron  
Jasna Kaur  
Ali Baig  
————————————–  
Employees In HR :  
————————————–  
Jiya Brein  
Nima Roy  
————————————–  
Employees In Account And Finance :  
————————————–  
Manu Sharma  
Jyothi Reddy

**Query 3.13 : What is the average salary and total salary of the whole organization?**

For this query, we use *Collectors.summarizingDouble()* on *Employee::getSalary* which will return statistics of the employee salary like max, min, average and total.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | DoubleSummaryStatistics employeeSalaryStatistics=  employeeList.stream().collect(Collectors.summarizingDouble(Employee::getSalary));    System.out.println("Average Salary = "+employeeSalaryStatistics.getAverage());    System.out.println("Total Salary = "+employeeSalaryStatistics.getSum()); |

**Output :**

Average Salary = 21141.235294117647  
Total Salary = 359401.0

**Query 3.14 : Separate the employees who are younger or equal to 25 years from those employees who are older than 25 years.**

For this query, we will be using *Collectors.partitioningBy()* method which separates input elements based on supplied *Predicate*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | Map<Boolean, List<Employee>> partitionEmployeesByAge=  employeeList.stream().collect(Collectors.partitioningBy(e -> e.getAge() > 25));    Set<Entry<Boolean, List<Employee>>> entrySet = partitionEmployeesByAge.entrySet();    **for** (Entry<Boolean, List<Employee>> entry : entrySet)  {      System.out.println("----------------------------");    **if** (entry.getKey())      {          System.out.println("Employees older than 25 years :");      }  **else**      {          System.out.println("Employees younger than or equal to 25 years :");      }        System.out.println("----------------------------");        List<Employee> list = entry.getValue();    **for** (Employee e : list)      {          System.out.println(e.getName());      }  } |

**Output :**

—————————-  
Employees younger than or equal to 25 years :  
—————————-  
Paul Niksui  
Amelia Zoe  
Nitin Joshi  
Nicolus Den  
Ali Baig  
—————————-  
Employees older than 25 years :  
—————————-  
Jiya Brein  
Martin Theron  
Murali Gowda  
Nima Roy  
Iqbal Hussain  
Manu Sharma  
Wang Liu  
Jaden Dough  
Jasna Kaur  
Jyothi Reddy  
Sanvi Pandey  
Anuj Chettiar

**Query 3.15 : Who is the oldest employee in the organization? What is his age and which department he belongs to?**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Optional<Employee> oldestEmployeeWrapper = employeeList.stream().max(Comparator.comparingInt(Employee::getAge));    Employee oldestEmployee = oldestEmployeeWrapper.get();    System.out.println("Name : "+oldestEmployee.getName());    System.out.println("Age : "+oldestEmployee.getAge());    System.out.println("Department : "+oldestEmployee.getDepartment()); |

**Output :**

Name : Iqbal Hussain  
Age : 43  
Department : Security And Transport