**Java 8 Features**

* [Lambda Expressions](https://www.geeksforgeeks.org/java-8-features/#lambda)
* [Functional Interfaces](https://www.geeksforgeeks.org/java-8-features/#functional)
* [Method Reference](https://www.geeksforgeeks.org/java-8-features/#reference)
* [Streams](https://www.geeksforgeeks.org/java-8-features/#streams)
* [Comparable and Comparator](https://www.geeksforgeeks.org/java-8-features/#comp)
* [Optional Class](https://www.geeksforgeeks.org/java-8-features/#optional)
* [Date/Time API](https://www.geeksforgeeks.org/java-8-features/#datetime)

## **Lambda Expressions**

Lambda Expression basically expresses an instance of the functional interface (An interface with a single abstract method is called a functional interface), in other words, you can say it provides a clear and concise way to represent a method of the functional interface using an expression

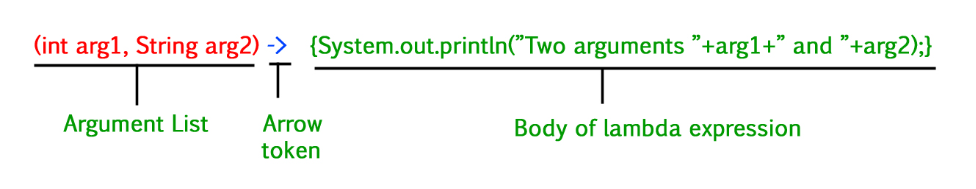
* [Lambda Expressions in Java 8](https://www.geeksforgeeks.org/lambda-expressions-java-8/)
* [Lambda Expressions Parameters](https://www.geeksforgeeks.org/java-lambda-expressions-parameters/)
* [Java Lambda Expression with Collections](https://www.geeksforgeeks.org/java-lambda-expression-with-collections/)
* [Lambda Expression Variable Capturing with Examples](https://www.geeksforgeeks.org/java-lambda-expression-variable-capturing-with-examples/)
* [How to Create Thread using Lambda Expressions in Java?](https://www.geeksforgeeks.org/how-to-create-thread-using-lambda-expressions-in-java/)
* [Serialization of Lambda Expression in Java](https://www.geeksforgeeks.org/serialization-of-lambda-expression-in-java/)
* [Block Lambda Expressions in Java](https://www.geeksforgeeks.org/block-lambda-expressions-in-java/)
* [Match Lambdas to Interfaces in Java](https://www.geeksforgeeks.org/match-lambdas-to-interfaces-in-java/)
* [Converting ArrayList to HashMap in Java 8 using a Lambda Expression](https://www.geeksforgeeks.org/converting-arraylist-to-hashmap-in-java-8-using-a-lambda-expression/)
* [Check if a String Contains Only Alphabets in Java Using Lambda Expression](https://www.geeksforgeeks.org/check-if-a-string-contains-only-alphabets-in-java-using-lambda-expression/)
* [Remove Elements From a List that Satisfy a Given Predicate in Java](https://www.geeksforgeeks.org/remove-elements-from-a-list-that-satisfy-given-predicate-in-java/)

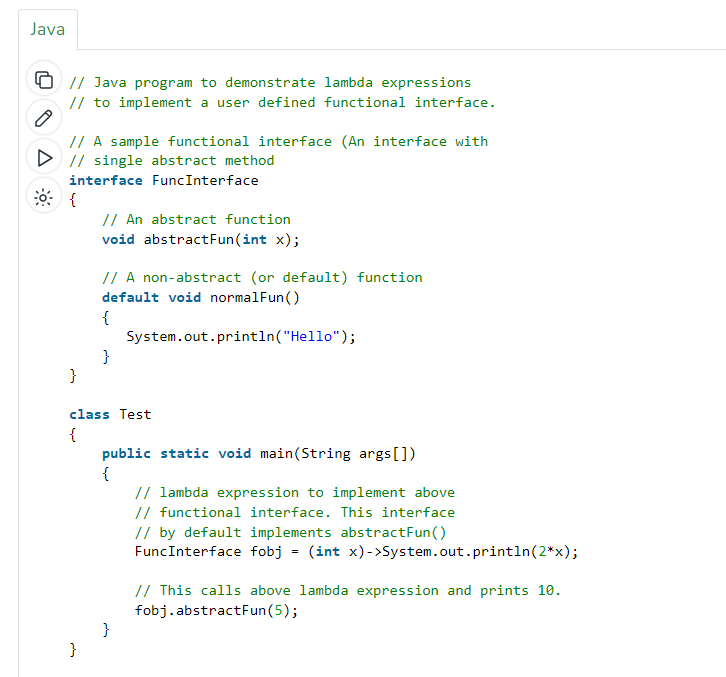
## **Functionalities of Lambda Expression in Java**

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





### **Lambda Expression Parameters**

1. **Zero Parameter**

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

1. **Single Parameter**

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

1. **Multiple Parameters**

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

# Java Lambda Expression with Collections

**Sorting Collections with Comparator (or without Lambda):** We can use [Comparator interface](https://www.geeksforgeeks.org/comparator-interface-java/) to sort, It only contains one abstract method: – compare(). An interface that only contains only a single abstract method then it is called a Functional Interface.

* Use of Comparator(I): –
* Prototype of compare() method: –

While defining our own sorting, JVM is always going to call Comparator to compare() method.

* returns negative value(-1), if and only if obj1 has to come before obj2.
* returns positive value(+1), if and only if obj1 has to come after obj2.
* returns zero(0), if and only if obj1 and obj2 are equal.

        // using lambda expression in place of comparator object

        Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 :

                                       (o1 < o2) ? 1 : 0);

        TreeSet<Integer> h =

                       new TreeSet<Integer>((o1, o2) -> (o1 > o2) ?

                                          -1 : (o1 < o2) ? 1 : 0);

        TreeMap<Integer, String> m =

                   new TreeMap<Integer, String>((o1, o2) -> (o1 > o2) ?

                                               -1 : (o1 < o2) ? 1 : 0);

  // Pass a reverse comparator to TreeSet() via a lambda expression

  TreeSet<String> ts=new TreeSet<String>((aStr,bStr) -> bStr.compareTo(aStr));

# Java – Lambda Expression Variable Capturing with Examples

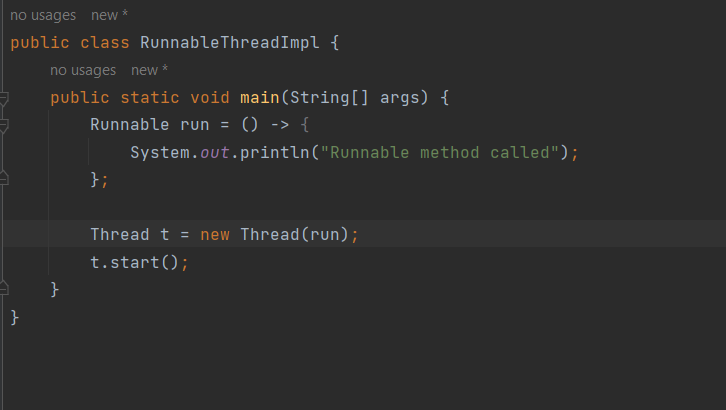
There are certain keypoints to be remembered, which are as follows:

1. Any local variable, formal parameter, or exception parameter used but not declared in a lambda expression must either be declared final or be effectively final , or a compile-time error occurs where the use is attempted.
2. Any local variable used but not declared in a lambda body must be definitely assigned before the lambda body, or a compile-time error occurs.
3. Similar rules on variable use apply in the body of an inner class . The restriction to effectively final variables prohibits access to dynamically-changing local variables, whose capture would likely introduce concurrency problems. Compared to the final restriction, it reduces the clerical burden on programmers.
4. The restriction to effectively final variables includes standard loop variables, but not enhanced-for loop variables, which are treated as distinct for each iteration of the loop.

# How to Create Thsread using Lambda Expressions in Java?

The following steps are performed to achieve the task:

* Create the [Runnable](https://www.geeksforgeeks.org/runnable-interface-in-java/) interface reference and write the Lambda expression for the run() method.
* Create a [Thread](https://www.geeksforgeeks.org/java-lang-thread-class-java/) class object passing the above-created reference of the Runnable interface since the start() method is defined in the Thread class its object needs to be created.
* Invoke the start() method to run the thread.



**Marker interface** (Interfaces does not contain any methods).

## **Functional Interfaces**

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

## **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**.

Functional interfaces are used and executed by representing the interface with an **annotation called *@FunctionalInterface***

## **Some Built-in Java Functional Interfaces**

* **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 ->** Takes one input returns no value

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

1. **Predicate ->** Takes one input returns Boolean value

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

1. **Function  ->** Takes one input returns and return value
2. **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.

Consumer -> Bi-Consumer Takes two input returns no value

Predicate -> Bi-Predicate Takes two input returns Boolean value

Function -> Bi-Function, Unary Operator, Binary Operator

### **Bi-Function**

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

***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 Reference**

Method reference is a shorthand notation of a lambda expression to call a method. There are four types of method references that are as follows:

* Static Method Reference
* Instance Method Reference of a particular object
* Instance Method Reference of an arbitrary object of a particular type
* Constructor Reference.

**Example:**

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

To refer to a method in an object

Object :: methodName

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

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

# Converting ArrayList to HashMap using Method Reference in Java 8

* **Function.identity()** refers to an element making itself as the key of the HashMap.
* **String::length** allows storing the length of the element as its respected value.

# 

## **Streams**

Stream API is introduced in Java 8 and is used to process collections of objects with the functional style of coding using the lambda expression.

* [Java 8 Stream](https://www.geeksforgeeks.org/java-8-stream-tutorial/)
* [Difference Between Streams and Collections in Java](https://www.geeksforgeeks.org/difference-between-streams-and-collections-in-java/)
* [Implement Filter Function using Reduce in Java 8 Streams](https://www.geeksforgeeks.org/implement-filter-function-using-reduce-in-java-8-streams/)
* [Java Stream API – Filters](https://www.geeksforgeeks.org/java-stream-api-filters/)
* [Parallel vs Sequential Stream in Java](https://www.geeksforgeeks.org/parallel-vs-sequential-stream-in-java/)
* [Functional Programming in Java 8+ using the Stream API with Example](https://www.geeksforgeeks.org/functional-programming-in-java-8-using-the-stream-api-with-example/)
* [Intermediate Methods of Stream in Java](https://www.geeksforgeeks.org/intermediate-methods-of-stream-in-java/)
* [Difference Between map() And flatMap() In Java Stream](https://www.geeksforgeeks.org/difference-between-map-and-flatmap-in-java-stream/)
* [Array to Stream in Java](https://www.geeksforgeeks.org/array-stream-java/)
* [10 Ways to Create a Stream in Java](https://www.geeksforgeeks.org/10-ways-to-create-a-stream-in-java/)
* [How to Print Elements of a Stream in Java 8](https://www.geeksforgeeks.org/how-to-print-elements-of-a-stream-in-java-8/)
* [Collecting a Stream to an Immutable Collection in Java](https://www.geeksforgeeks.org/collecting-a-stream-to-an-immutable-collection-in-java/)
* [Comparing Streams to Loops in Java](https://www.geeksforgeeks.org/comparing-streams-to-loops-in-java/)
* [Why You Need to Close the Java Streams in Finally Block?](https://www.geeksforgeeks.org/why-you-need-to-close-the-java-streams-in-finally-block/)
* [Convert an Iterable to Stream in Java](https://www.geeksforgeeks.org/convert-an-iterable-to-stream-in-java/)
* [Convert an Iterator to Stream in Java](https://www.geeksforgeeks.org/convert-an-iterator-to-stream-in-java/)
* [Difference Between Stream.of() and Arrays.stream() method in Java](https://www.geeksforgeeks.org/difference-between-stream-of-and-arrays-stream-method-in-java/)
* [Convert Stream to Set in Java](https://www.geeksforgeeks.org/convert-stream-set-java/)
* [Convert a Set to Stream in Java](https://www.geeksforgeeks.org/convert-set-stream-java/)
* [Streams on Arrays in Java 8](https://www.geeksforgeeks.org/streams-arrays-java-8/)

A ***stream is a sequence of objects that supports various methods*** which can be pipelined to produce the desired result.

***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.*

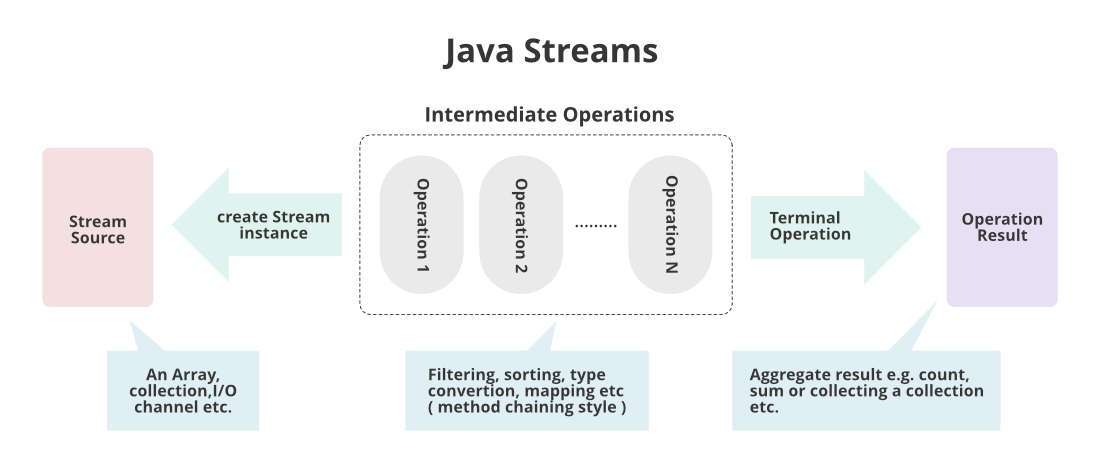
*Streams are present in java’s utility package named****java.util.stream***

Let us now start with the basic components involved in streams. They as listed and 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, Arrays**,** or I/O channels.
* 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.

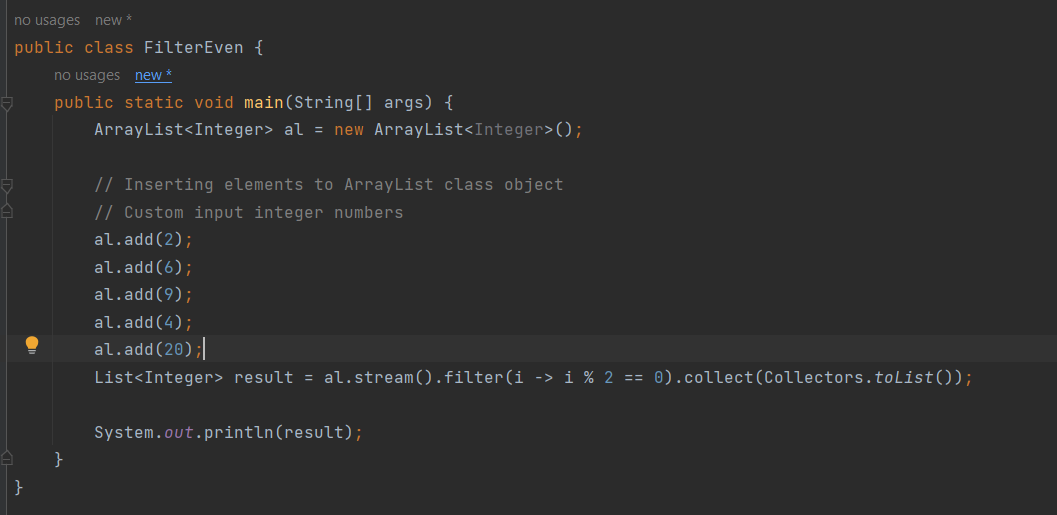


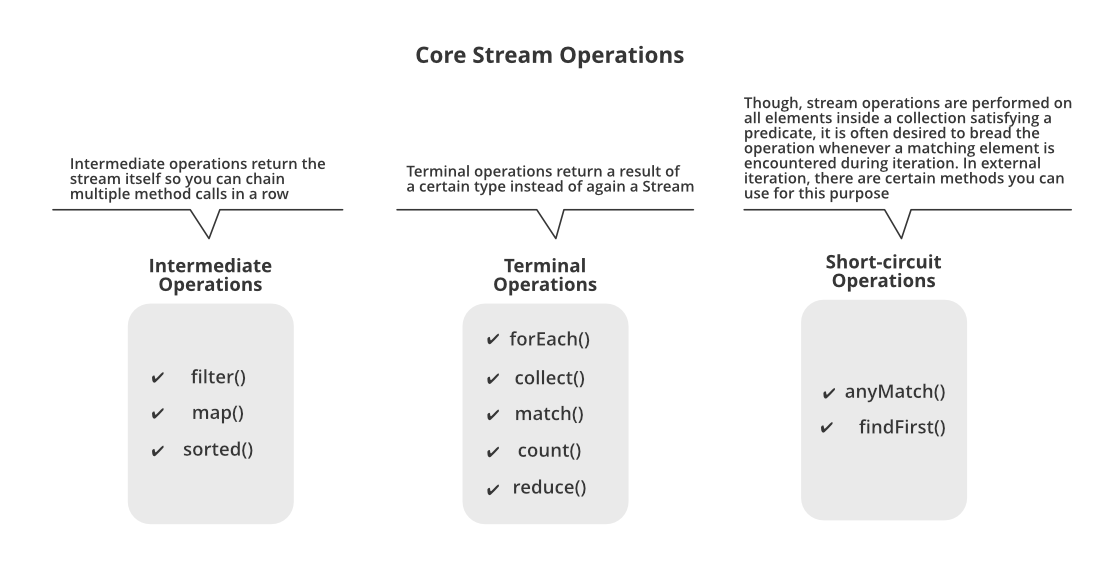
Before moving ahead in the concept consider an example in which we are having ArrayList of integers, and we suppose we apply a filter to get only even numbers from the object inserted.

**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.



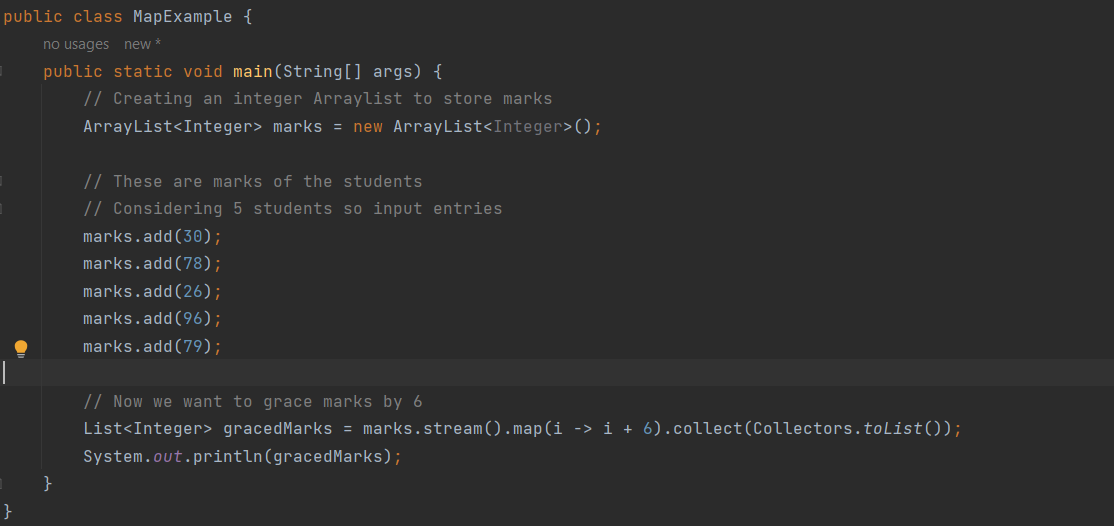
**Various core operations over Streams?**

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

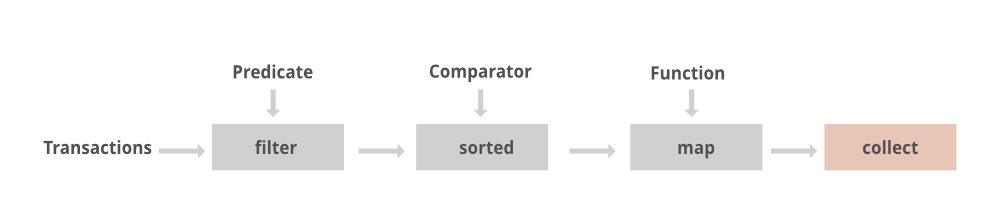
So, there are 3 types of **Intermediate operations** which are as follows:

* **Operation 1:** filter() method
* **Operation 2:**map() method
* **Operation 3:** sorted() method



***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.*

**Example 1:**In general, daily world, whenever the data is fetching from the database, it is more likely we will be using collection so there itself streams concept is must apply to deal with processed data.



**Example 2:**Streams in a Grocery store

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

List<Integer> transactionsIds =

transactions.stream().filter(t -> t.getType() == Transaction.GROCERY)

.sorted(comparing(Transaction::getValue).reversed())

.map(Transaction::getId).collect(toList());



# Difference Between Streams and Collections in Java

# 

# Implement Filter Function using Reduce in Java 8 Streams

# //TODO

# Java Stream API – Filters

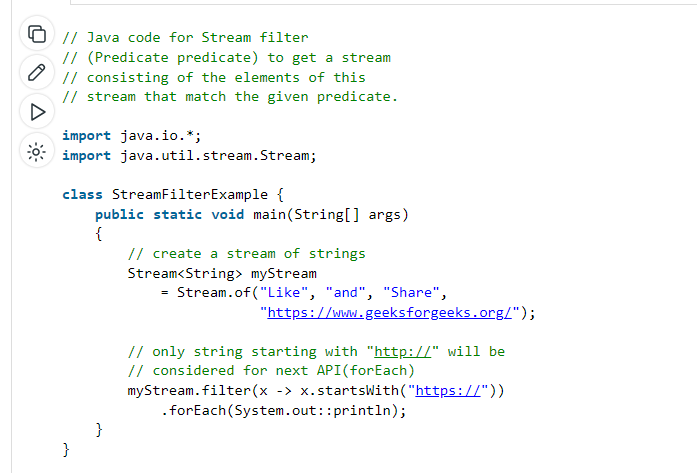
1. How [stream filter](https://www.geeksforgeeks.org/stream-filter-java-examples/) API works.

2. Filter by Object Properties.

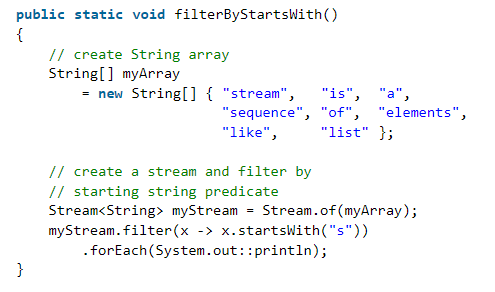
3. Filter by Index.

# 4. Filter by custom Object properties.

# Filter API takes a [Predicate](https://www.google.com/url?client=internal-element-cse&cx=009682134359037907028:tj6eafkv_be&q=https://www.geeksforgeeks.org/java-8-predicate-with-examples/&sa=U&ved=2ahUKEwiq8K-r2tvsAhV1zDgGHTy7DjcQFjACegQIBxAC&usg=AOvVaw0YJ2JCn2wZz2kTEUPUTlGe). The predicate is a Functional Interface. It takes an argument of any type and returns Boolean. The element will be considered for the next API in the pipeline if the function returns true. Otherwise, the element is filtered out.



**Filter by Object Properties**

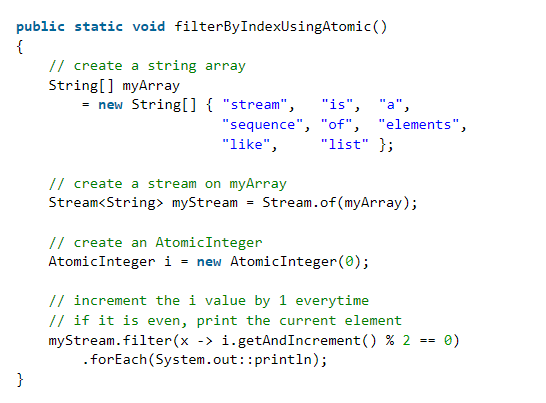


**Filter by Object Indices**

Filtering by indexes can be done in two ways.

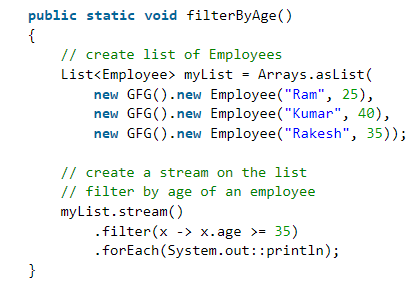
* 1. **Atomic Integers**
  2. **2. Intstream Approach**

**Atomic Integers**

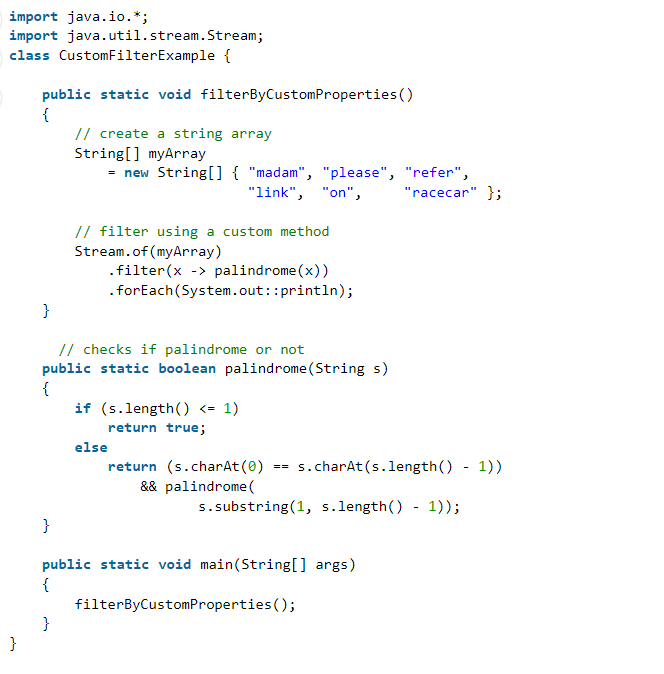
We need to use **AtomicInteger** because predicates expect [final variables](https://www.geeksforgeeks.org/final-keyword-java/) as parameters. As long as filter function(Predicate) returns boolean we can use any expression. Here, [getAndIncrement()](https://www.google.com/url?client=internal-element-cse&cx=009682134359037907028:tj6eafkv_be&q=https://www.geeksforgeeks.org/atomicinteger-getandincrement-method-in-java-with-examples/&sa=U&ved=2ahUKEwjd3s7B39vsAhV3yTgGHaO7C6MQFjAAegQIAhAB&usg=AOvVaw0WDKayNrCy7kRIioF1fLz6) method of AtomicInteger increments the current value by **1**and returns final int value.  
  


**Filter by Custom Properties**

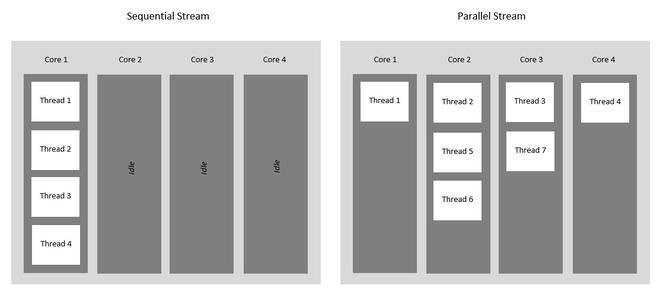
We can use any Java Object property for filtering. Here we are filtering by age.



**We can also create a custom function for filtering. The function must take a parameter and return a boolean value.**



# Parallel vs Sequential Stream in Java



### **Sequential Stream**

Sequential Streams are non-parallel streams that use a single thread to process the pipelining. Any stream operation without explicitly specified as parallel is treated as a sequential stream. Sequential stream’s objects are pipelined in a single stream on the same processing system hence it never takes the advantage of the multi-core system even though the underlying system supports parallel execution. Sequential stream performs operation one by one.



### **Parallel Stream**

It is a very useful feature of Java to use parallel processing, even if the whole program may not be parallelized. Parallel stream leverage multi-core processors, which increases its performance. Using parallel streams, our code gets divide into multiple streams which can be executed parallelly on separate cores of the system and the final result is shown as the combination of all the individual core’s outcomes. It is always not necessary that the whole program be parallelized, but at least some parts should be parallelized which handles the stream. The order of execution is not under our control and can give us unpredictably unordered results and like any other parallel programming, they are complex and error-prone.

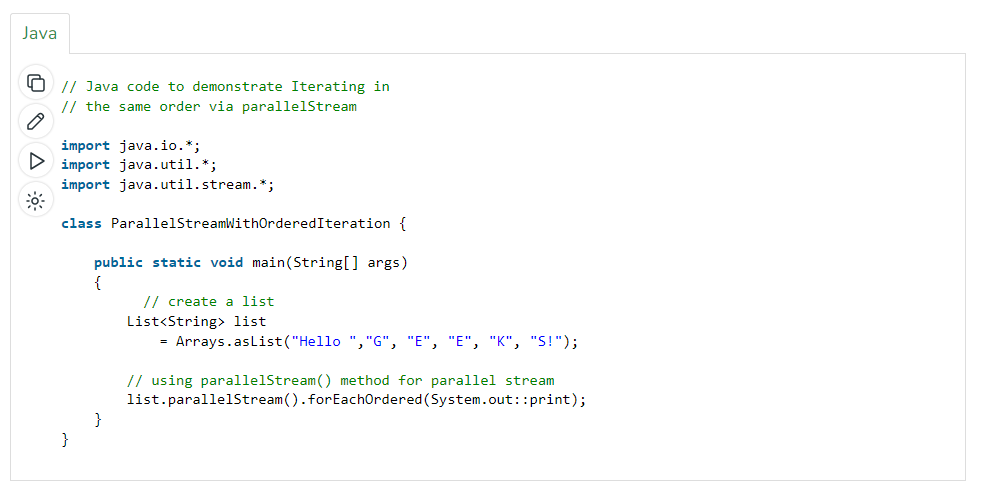
The Java stream library provides a couple of ways to do it. easily, and in a reliable manner.

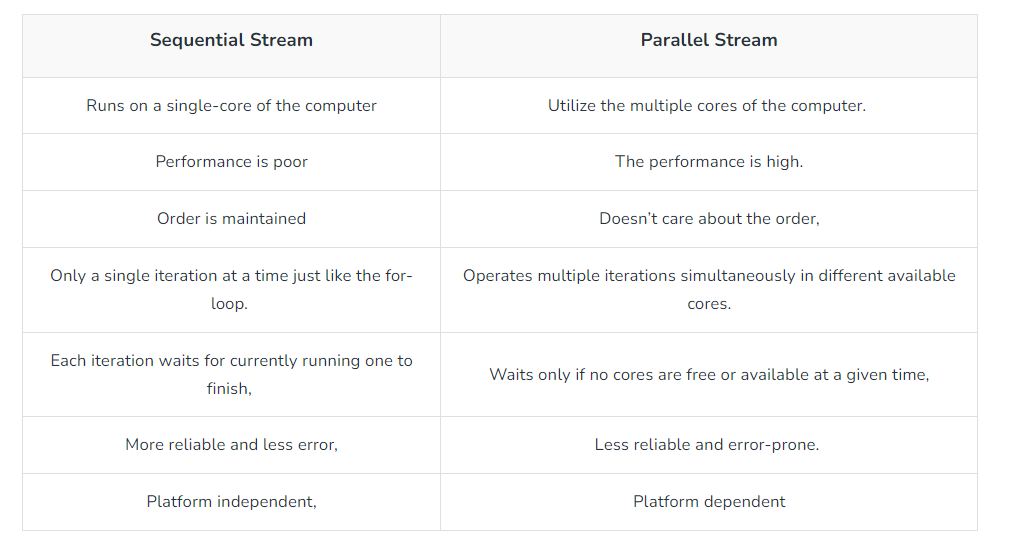
* One of the simple ways to obtain a parallel stream is by invoking the [parallelStream()](https://www.geeksforgeeks.org/what-is-java-parallel-streams/) method of **Collection**interface.
* Another way is to invoke the[parallel()](https://www.geeksforgeeks.org/what-is-java-parallel-streams/)method of **BaseStream** interface on a sequential stream.



Here we can see the order is not maintained as the list.parallelStream() works parallelly on multiple threads. If we run this code multiple times then we can also see that each time we are getting a different order as output but this parallel stream boosts the performance so the situation where the order is not important is the best technique to use.

**Note:** If we want to make each element in the parallel stream to be ordered, we can use the [forEachOrdered()](https://www.geeksforgeeks.org/stream-foreachordered-method-java-examples/) method, instead of the [forEach()](https://www.geeksforgeeks.org/stream-foreach-method-java-examples/) method.





# Intermediate Methods of Stream in Java

Intermediate operators do not execute until a terminal operation is invoked, i.e. they are not executed until a result of processing is actually needed. We will be discussing a few of the important and most frequently used:

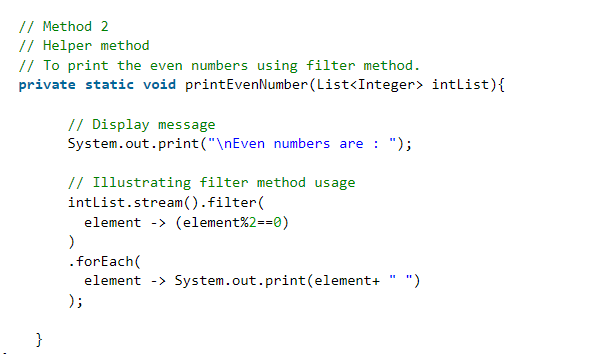
1. [*filter(predicate)*Method](https://www.geeksforgeeks.org/stream-filter-java-examples/)
2. [*sorted()*Method](https://www.geeksforgeeks.org/stream-sorted-in-java/)
3. [*distinct()*Method](https://www.geeksforgeeks.org/stream-distinct-java/)
4. [*map()*Method](https://www.geeksforgeeks.org/stream-map-java-examples/)

***Note:***

* *Intermediate functions return a stream back.*
* *On any stream you can execute any number of intermediate operations, but the terminal operation should be single and written at last. So following are the intermediate methods provided by the Stream*

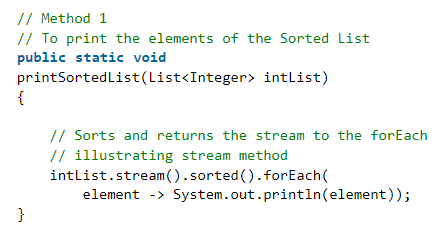
**Method 1:** [*filter(predicate)*](https://www.geeksforgeeks.org/stream-filter-java-examples/)

It returns a new stream consisting of the elements of the stream from which it is called which are according to the predicate (condition).



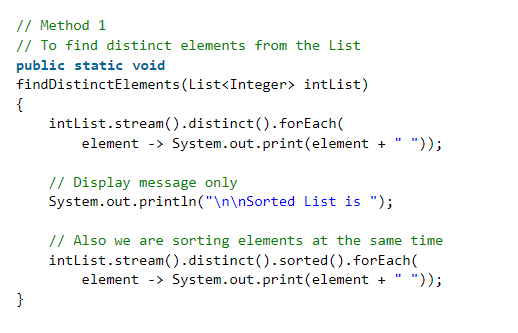
**Method 2:**[*sorted()*](https://www.geeksforgeeks.org/stream-sorted-in-java/)

Returns a stream consisting of the elements of the stream passed, sorted according to the natural order.  If the elements of this stream are not comparable, a  java.lang.ClassCastException may be thrown when the terminal operation is executed.



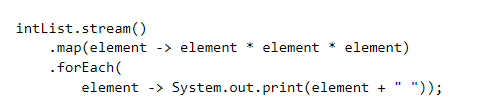
**Method 3:** [*distinct()*](https://www.geeksforgeeks.org/stream-distinct-java/)

It returns a stream consisting of the distinct(different) elements of the passed stream. For ordered stream, the selection of the distinct elements is stable (For duplicated elements, the element appearing first in the encounter order is preserved). While for non-ordered streams it does not make any guarantee for stability.



**Method 4:** [*map()*](https://www.geeksforgeeks.org/stream-map-java-examples/)

Mapper is a non-interfering, stateless function to apply to each element of the stream. It returns a stream consisting of the results of applying the given function to the element of the passed stream.



# flatMap() In Java Stream

# ****Stream flatMap(Function mapper)**** returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

**Note :**Each mapped stream is closed after its contents have been placed into this stream. If a mapped stream is null, an empty stream is used, instead.

flatMap() is the combination of a map and a flat operation i.e, it applies a function to elements as well as flatten them.

As already discussed in the post that flatMap() is the combination of a map and a flat operation i.e, it first applies map function and than flattens the result. Let us consider some examples to understand what exactly flattening a stream is.

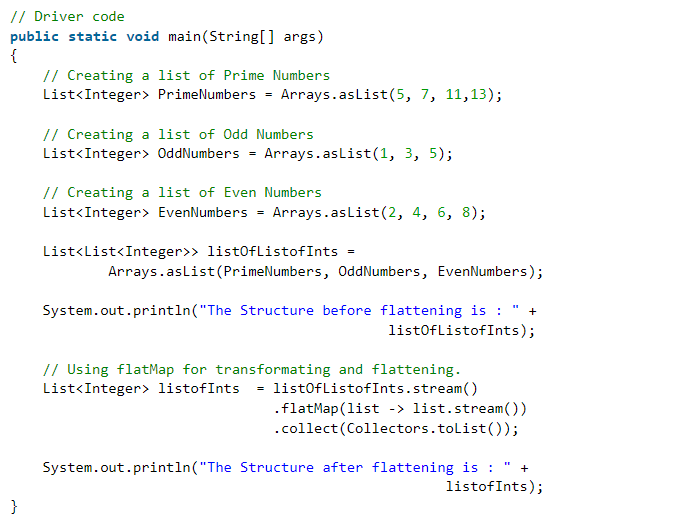
**Example 1 :**  
The list before flattening :

[ [2, 3, 5], [7, 11, 13], [17, 19, 23] ]

The list has 2 levels and consists of 3 small lists. After Flattening, it gets transformed into “one level” structure as shown :

[ 2, 3, 5, 7, 11, 13, 17, 19, 23 ]

In short, we can say that if there is a **Stream of List of <<Data Type>>** before flattening, then on applying flatMap(), **Stream of <<Data Type>>** is returned after flattening.



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

