<https://codeburst.io/quick-java-lambdas-and-method-references-3279b7690c07>

<https://www.javatpoint.com/java-8-features>

**Java 8 provides following features for Java Programming:**

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods,
* Base64 Encode Decode,
* Static methods in interface,
* Optional class,
* Collectors class,
* ForEach() method,
* Parallel array sorting,
* Nashorn JavaScript Engine,
* Parallel Array Sorting,
* Type and Repeating Annotations,
* IO Enhancements,
* Concurrency Enhancements,
* JDBC Enhancements etc

**Lambda expression**

Lambda expression helps us to write our code in functional style. It provides a clear and concise way to implement SAM interface (Single Abstract Method) by using an expression. If we do not use lambda expression then we have to create anonymous class object and overrides its method by writing full method definition But when you use lambda expression then you write () -> {lambda body}. If your lambda body has single line of code then you do not need enclose its body with brace bracket even in the case of return value similarly if your SAM method has one argument then argument parenthesis is optional.

A lambda expression consists of two parts: the parameter part and the expressions part separated by a forward arrow as below:

params -> expressions

1. Optional type declaration – when declaring the parameters on the left-hand side of the lambda, we don't need to declare their types as the compiler can infer them from their values. So int param -> … and param ->… are all valid
2. Optional parentheses – when only a single parameter is declared, we don't need to place it in parentheses. This means param -> … and (param) -> … are all valid. But when more than one parameter is declared, parentheses are required.
3. Optional curly braces – when the expressions part only has a single statement, there is no need for curly braces. This means that param – > statement and param – > {statement;} are all valid. But curly braces are required when there is more than one statement
4. Optional return statement – when the expression returns a value and it is wrapped inside curly braces, then we don't need a return statement. That means (a, b) – > {return a+b;} and (a, b) – > {a+b;} are both valid

**Method references**

Java 8 Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. When you are using lambda expression just to refer a method and do not want to override its functionality then, you can replace your lambda expression with method reference.

**JDK 8 Interface**

You can declare abstract methods, define default methods as well as static methods. Previous jdk versions do not allow definition of any method in interface but in case of jdk 8 onwards you can define default methods as well as static methods. You can also override default method of interface in implementing class and in the same you can hide the static method in implementing class by redefining it.

**Functional interfaces**

An Interface annotated with @FunctionalInterface annotation contains only one abstract method. It can have any number of default and static methods. Functional interfaces are also known as Single Abstract Method Interfaces (SAM Interfaces). Although @FunctionalInterface annotation usage is optional for functional interface. This annotation is used to ensure that the functional interface can’t have more than one abstract method. In that case compiler flags error thus we will be safe from accidently adding extra abstraction method in functional interface. SAM interface can also declare methods of object class or its super interface but it should be annotated with @Override annotation and in this case too you can use @FunctionalInterface without any error.

There are a lot of functional interfaces in the java.util.function package, the more common ones include but not limited to:

1. Function – it takes one argument and returns a result.
2. Consumer – it takes one argument and returns no result (represents a side effect)  
   Supplier – it does not takes argument and returns a result
3. Predicate – it takes one argument and returns a Boolean
4. BiFunction – it takes two arguments and returns a result
5. BinaryOperator – it is similar to a BiFunction, taking two arguments and returning a result. The two arguments and the result are all of the same types
6. typesUnaryOperator – it is similar to a Function, taking a single argument and returning a result of the same type

**Default methods**

Default method gives us the ability to add full implementations in interfaces besides abstract methods. You can create one or more default methods inside of the interface as well as functional interface (SAM interface). Methods which are defined inside of the interface and tagged with default keyword are known as default methods. These methods are non-abstract methods and can have method body. You can also override default methods in implementing class or its sub interface and this is how we can use a default method to add a new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface. Usually, when a new abstract method is added to an interface, all implementing classes will break until they implement the new abstract method. In Java 8, this problem has been solved by the use of default method. For example, Collection interface does not have forEach method declaration. Thus, adding such method would simply break the whole collections API. Java 8 introduces default method so that Collection interface can have a default implementation of forEach method without requiring the classes implementing this interface to implement the same.

**Stream API**

A Java Stream enables us to use lambda express and provides functional style programming as well as capable of internal iteration of its elements, meaning it can iterate its elements itself. In contrast, when you are using the Java Collections iteration features (e.g a Java Iterator or the Java for-each loop used with a Java Iterable) you have to implement the iteration of the elements yourself.

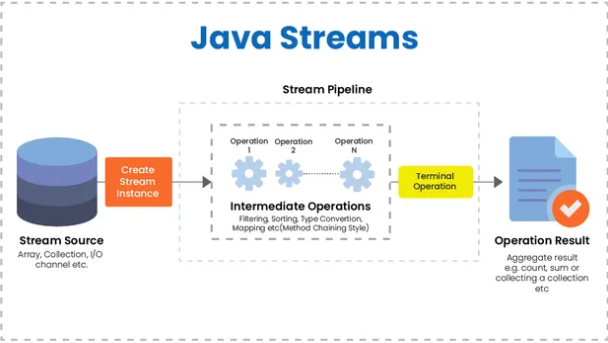
**Stream Processing**

You can attach lazy intermediate operations to a Stream. These lazy intermediate operations are called when the Stream iterates the elements internally. The lazy intermediate operations are called once for each element in the stream. That way each lazy intermediate operation gets to process each element in the stream. This is referred to as stream processing.

The lazy intermediate operations of a stream form a chain. The first lazy intermediate operation in the chain can process the element in the stream, and then return a new element for the next lazy intermediate operation in the chain to process. A lazy intermediate operation can either return the same element or a new, depending on what the purpose of that lazy intermediate operation is. These Lazy operation will execute when you want them to execute and you do by calling terminal operation at the of stream pipeline.

Stream represents a sequence of elements from a source, which supports aggregate operations. Streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast but **stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.** Before going to detail just have look of basic terminology and part of stream. Stream in consist of following part

**Source + One/more Intermediate Operations + Exactly One Terminal Operation**

****

Following are the characteristics of a Stream −

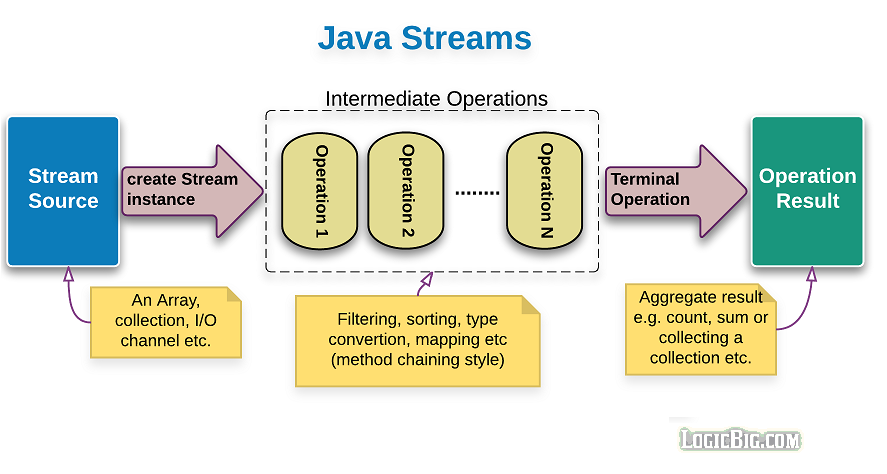
* **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
* **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
* **Aggregate operations** – aggregate operation can either be intermediate operation or terminal operation. Some examples of aggregate operations are filter(), map(), flatMap(), limit(), reduce(), find(), match() and collect and so on. Remember you can only add one aggregate operation on stream.
* **Intermediate operation**:- Intermediate operation takes input in the form of lambda expression or method reference. Intermediate operations are lazy in the sense that they execute when you call terminal operation on stream pipeline. When you add intermediate operation on stream then this operation return new stream and you add another intermediate operation on this newly returned stream and this is how you implement method chaining to add multiple intermediate operations in the middle part of stream pipeline. The newly returned stream from previous intermediate operation can be different from its previous stream with respect to data type or data format. This means stream data type and data format can be changed throughout the chaining of intermediate operations in stream pipeline. Remember you can add only one intermediate operation on the stream. If you want to add multiple intermediate operations then you have to add it on newly returned stream by previously added intermediate operation and this is how we add multiple intermediate operations using method chaining where each operation will be added to newly returned stream from previous operation. This method chaining is middle part of stream pipeline. Always remember that stream pipeline of intermediate operations does not executes and remain idle or lazy until you call terminal operation on it at the end. Call to terminal operation on stream pipeline triggers sequential of executions of all intermediate operations added to stream pipeline and then terminal operation consumes final stream that is returned by last intermediate operation in the stream pipeline and perform its business logic and close stream. Closed and consumed stream cannot be reused. If you do this then you will get IllegalStateException. Remember the order of intermediate operations in the stream pipeline matters because previously executed intermediate operation effects following intermediate operation hence always call filter() intermediate operation method before other intermediate method such as sort(), map and etc. It makes your stream processing faster. There are two types of intermediate operations, first is state-full and other is stateless. State-full operation effects other element in the stream hence state-full intermediate operation includes other elements in the stream while processing current element that means they operate inclusively not individually on elements. The other intermediate operations are stateless such as map(), flatMap() etc. These stateless operations execute on individual element and do not include other elements of stream while executing current element in stream. Beware of using parallel Stream for statefull operation because concurrency impact inclusive statefull operation negatively.
* **Terminal Operation**:- There will always be only one terminal operation call on stream pipeline and it always present at the end of stream. Terminal operation takes input in the form of lambda expression or method reference. Terminal operation call executes all pipelined intermediate operation sequentially. Each intermediate operation executes and return new processed stream that can contain different data from the stream where it is called and this process continue until last intermediate operation in pipeline line executes. Now Terminal operation consumes the final stream returned by last intermediate operation in the stream pipeline and performs its logic on consumed data after that terminal operation may collect data in some data structure such as collection api, convert it into map data structure or reduced it to single value and etc. It is all depend on the type of terminal operation you have used. After execution of terminal operation stream is closed and you cannot use that stream. If you do it then you will get IllegalStateException. Last thing to remember is that terminal operation never returns stream either existing or new.
* **Pipelining** – Intermediate stream operations return stream itself so that their result can be pipelined. Intermediate operations take input, process them lazily, and return new lazy stream that can be again pipelined. Terminal operation such as collect() method which is present at the end of the pipeline operation, executes all intermediate lazy operations and it consumes final and last stream returned by the last executing intermediate operation because it is added last in the pipeline. It should be noted the order of intermediate operation in stream pipeline is very important. This brings us up to the rule: intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element. So, keep such methods as skip(), filter(), distinct() at the top of your stream pipeline
* **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

**Note:- Remember Stream is lazy. Its intermediate operations will execute only when you call Terminal operation on stream pipeline. If you do not call terminal operation on the stream and it remains unused and you do not want to use it further then make sure to close stream by calling close() method on it else it will consume system resource unnecessarily. If you call terminal operation then no need to call close() method on stream because call to terminal operation consume whole stream aggregates or reduces stream data and perform final logic on it and then close it implicitly.**

**Sequential Stream Vs Parallel Stream**

Streams are of two types one is sequential stream and another is parallel stream. You can also switch mode of stream from stream to parallel stream visa versa. **Parallel Stream** has a much higher overhead compared to a sequential one. Parallel stream should be used only when there is a huge amount of elements to process and there is a performance problem with processing these elements. Also the parallel stream operations should be independent. **Parallel streams** divide the provided task into many and run them in different threads, utilizing multiple cores of the computer. On the other hand sequential **streams** work just like for-loop using a single core. **Parallel** execution of **streams** run multiple iterations simultaneously in different available cores hence **Parallel stream** enables **parallel** computing that involves processing elements concurrently in **parallel** with each element in a separate thread. But this does not guarantee high performance and faster execution every time. It again depends on the number of CPU cores available. Good thing is that developer is not required to code multiple thread management stuff for Parallel stream because it does this chorus under the hood. Parallel Steam uses the ForkJoinPool.commonPool(), **a Thread Pool shared by the entire application.** We can actually pass a custom *ThreadPool*when processing the *stream*. Do **remember**, Parallel Streams must be used only with *stateless*, *non-interfering*, and *associative operations* i.e.

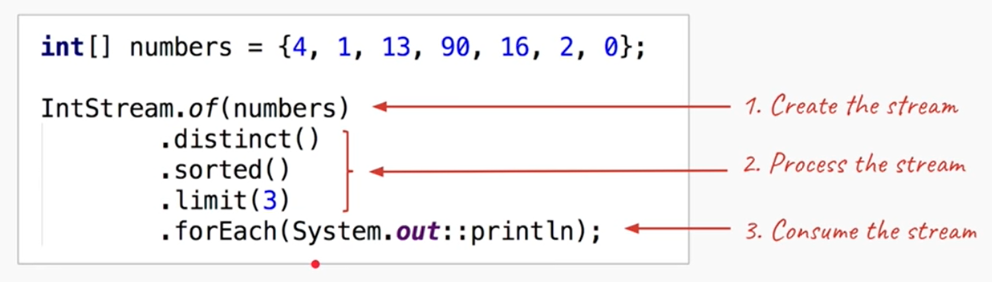
* A stateless operation is an operation in which the state of one element does not affect another element
* A non-interfering operation is an operation in which data source is not affected
* An associative operation is an operation in which the result is not affected by the order of operands

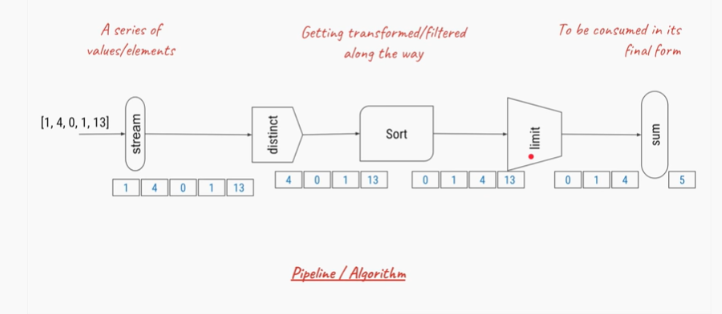


The Stream API is a powerful but simple to understand set of tools for processing sequence of elements. It allows us to reduce a huge amount of boilerplate code, create more readable programs and improve app’s productivity when used properly.

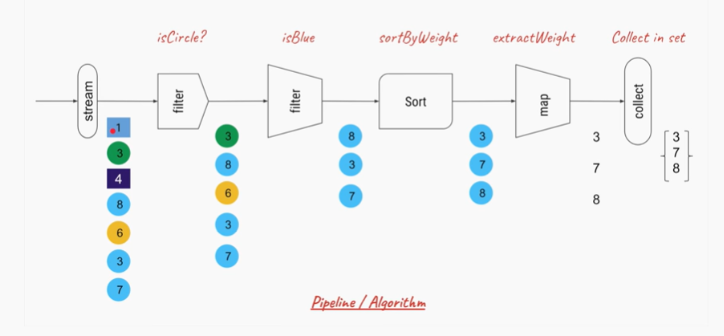
In most of the code samples shown in this article streams were left unconsumed (we didn't apply the close() method or a terminal operation). In a real app, **don't leave an instantiated streams unconsumed as that will lead to memory leaks.**

**Code Snippets and Diagrams**

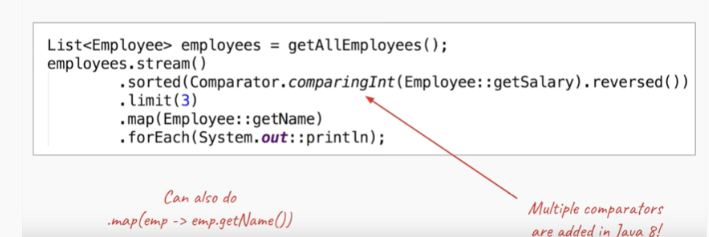


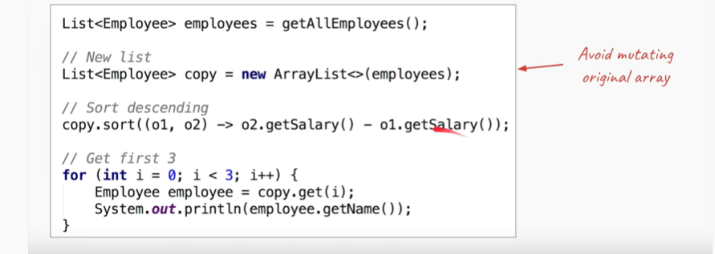




****

****

****

****

****

****

