**JAVA STREAMS**

**Java 8 was introduced for features like Concise and minimal code, functional programming.**

**Also to enable parallel programming and more compatible code for the multicore processors, java 8 was introduced.**

**Features of Java 8:**

**A white and black text

AI-generated content may be incorrect.**

**A white and blue sign with black text

AI-generated content may be incorrect.**

**A close-up of a website

AI-generated content may be incorrect.**

**A close up of a box

AI-generated content may be incorrect.**

**A purple rectangular object with white text

AI-generated content may be incorrect.**

**A white background with black text

AI-generated content may be incorrect.**

**A white and purple rectangle with text

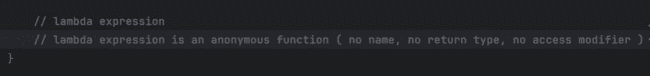
AI-generated content may be incorrect.**

**A white background with black text

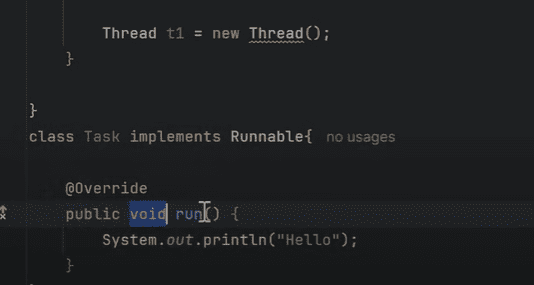
AI-generated content may be incorrect.**

**A close up of a paper

AI-generated content may be incorrect.**

****

**Instead of doing all the code that has been written:**

****

**We can use the following one:**

**A computer screen with text

AI-generated content may be incorrect.**

**Functional interfaces are those which have public abstract method and can be implemented using lambda expressions.**

**Now if we need to provide the implementation of following interface for add, subtract, multiply and divide.**

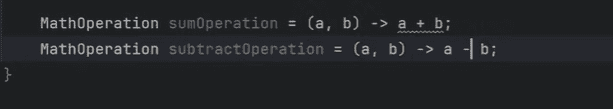
**A screen shot of a computer program

AI-generated content may be incorrect.**

**But if we use the lambda expressions, we can provide the implementation for the all the methods in the one line.**

**MathOperation sumOperation = (int a, int b)-> a+b;**

**Which can also be written as:**

****

**And we can do whatever operation we want here:**

**A screen shot of a computer

AI-generated content may be incorrect.**

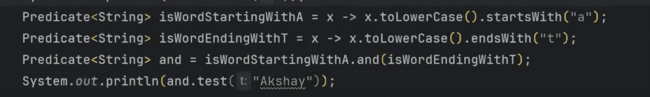
**Predicate-> is a Boolean value function which has the Boolean test(T t) function which we can call to check the true or false.**

**Example:**

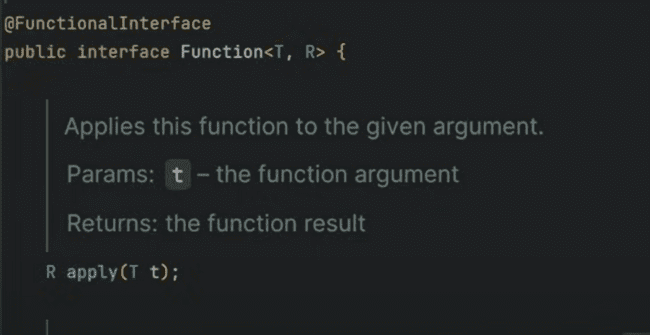
**A black background with white text and numbers

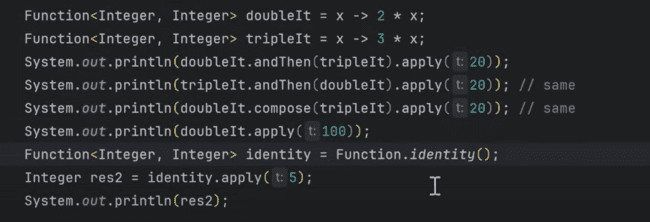
AI-generated content may be incorrect.**

**We can also combine the predicates as well as shown below:**

****

**Function is a function which takes the value and returns the value, and its method is:**

****

****

**andThen combines both functions from left to right but compose calculates from right first and then left.**

**Function.identity() returns the same output that has been input.**

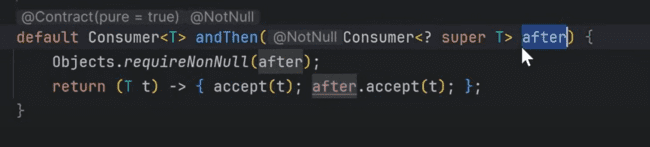
**CONSUMER:**

**It takes an argument but does not provide any output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**In the consumer, for the andThen method, it simply does the accept on one argument and then on the other argument as shown below:**

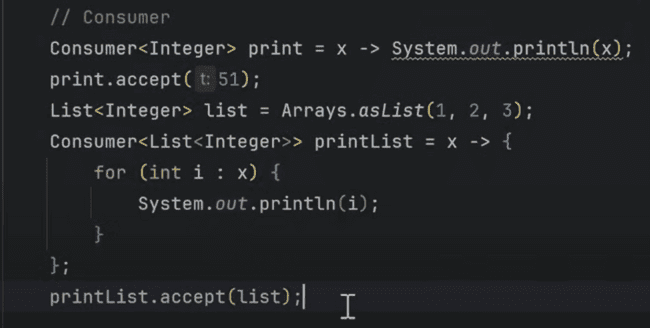
****

**Whereas in function, it combines return type of one argument and then apply on the other argument as shown below:**

**A screenshot of a computer

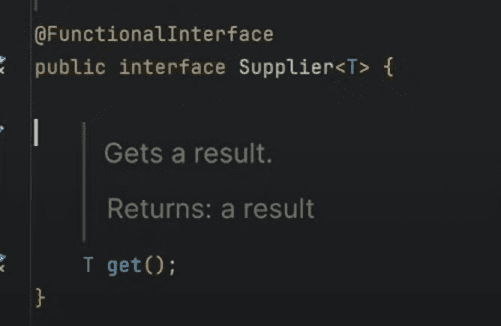
AI-generated content may be incorrect.**

**So, in the consumer, we can simply print the result of anything or print the result of the Array List as shown below:**

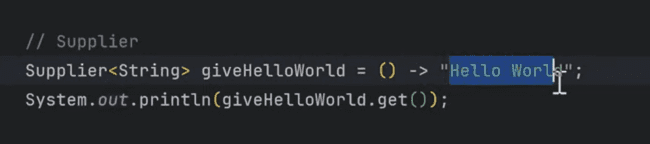
****

**SUPPLIER:**

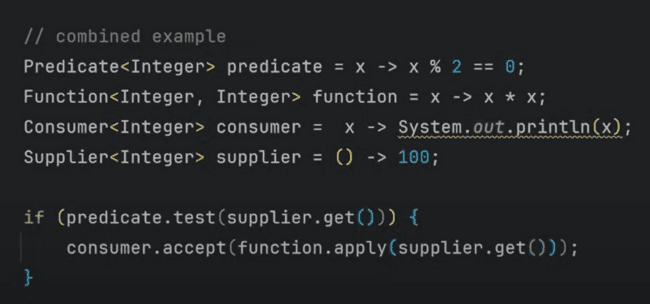
**Has only one get(), method in the interface.**

****

**It does not take any argument but returns one.**

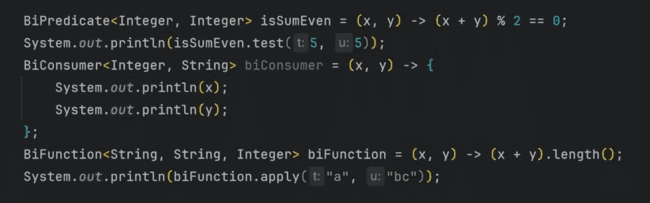
****

**Combined example of above 4:**

****

**There are others as:**

**BiConsumer, BiPredicate, BiFunction:**

****

**BiFunction takes 3 arguments, one and two are inputs and third is output.**

**And we can use UnaryOperator in place of Function and it extends Function as well.**

**And BinaryOperator in place of BiFunction as shown below, but there is one condition that, all should has same types.**

****

**This function can be replaced using:**

**A black background with white text

AI-generated content may be incorrect.**

**And BinaryOperator in place of BiFunction as shown below:**

**A black background with white text

AI-generated content may be incorrect.**

**MEHTOD REFERENCE:**

**This is using the method without invoking it and in place of lambda expressions.**

**Instead of calling or invoking the method, we can simply pass the reference of that method to the function or for Each loop shown below:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**System.out.println(x) is calling the method, but System.out::println is passing this as an argument without invoking the method and passing method as parameter.**

**Constructor reference:**

**We can call the constructor reference using the following:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Like the above example, we can pass the same using the constructor reference as shown below:**

****

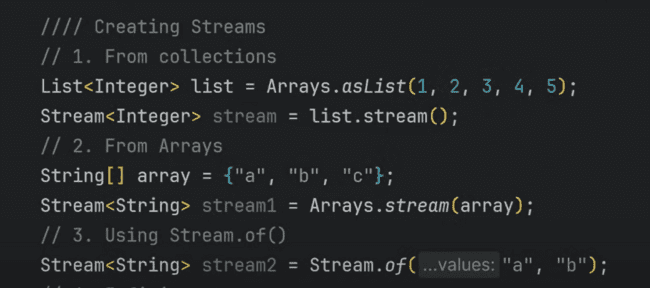
**STREAMS**

**Is a feature introduced in java8 and process the collection of data in a functional and declarative manner.**

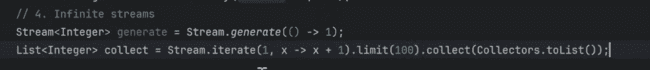
* **Simply Data processing, like the usage of if else.**
* **Without dealing the multithreading complexity, we can achieve the parallelism.**

**Consists of Source, then intermediate operations and then terminal operations.**

**Ways of creating the Streams:**

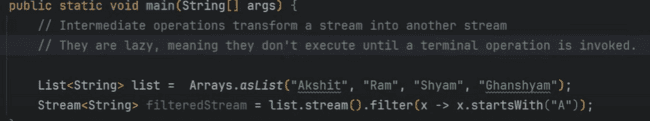
****

**Also, we can create the infinite stream, which would grow:**

****

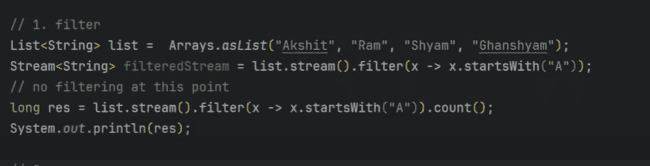
**We need to use generate and iterate methods for creating the infinite streams.**

**For operations on streams, we perform intermediate operation on streams as shown below:**

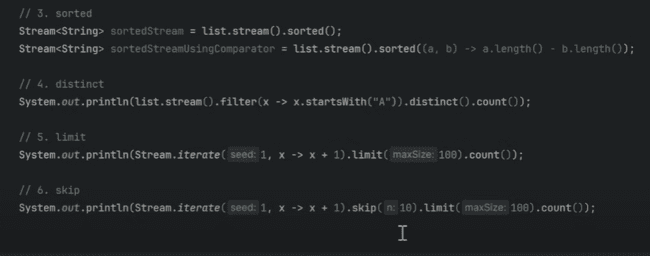
****

**Now if there are not terminal operations on the streams, then no filtering is done in the above picture.**

**We need to use either count(), collect() or any other to make filtering.**

****

**Intermediate operations:**

****

**We can use the toUpperCase function directly instead of using the**

**.map(x-> x.toUpperCase())**

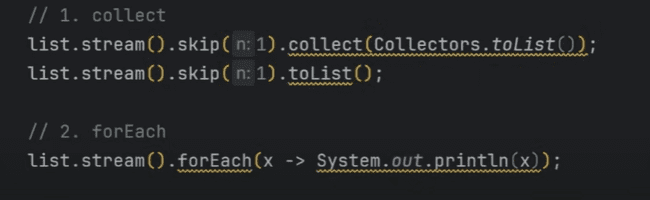
****

**Terminal Operations:**

**Collect and forEach:**

**Short circuit operations: As soon as they find the match, they stop processing rest of the elements. Example: anyMatch(), allMatch(), noneMatch(), findFirst(), findAny().**

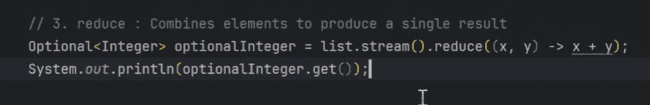
**The collection returned by toList() is unmodifiable list.**

****

**Skip() is basically to skip the elements and then perform the operation.**

**Reduce(): reduces the elements to the single element.**

**Accumulators:**

****

**As the above is lambda expressions, we can rewrite like the following:**

**A computer screen shot of a black screen

AI-generated content may be incorrect.**

**Return type of reduce is Optional, which we can process using get().**

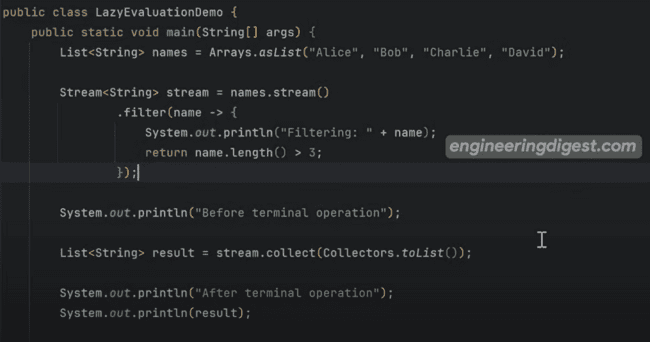
To use streams on the String, we need to use the .chars(), which would be returning the Integer stream as shown below:

A screen shot of a computer

AI-generated content may be incorrect.

Stateful and Stateless operations:

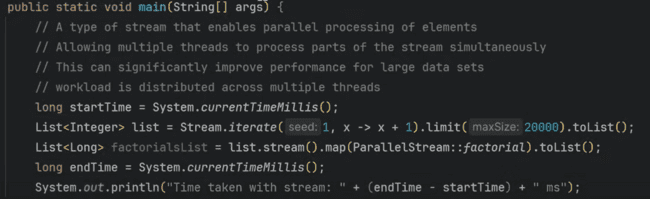
Like sorted() and distinct() operations wants to know regarding the states of the previous elements, but like map() and filter(), does not need to know about the previous and next elements.



Above is the example of Lazy operation in the streams, as it would not print “Filtering”, it would be printing from “Before terminal operation” to result.

**PARALLEL STREAM:**

Definition of parallel stream and time taken for the sequential stream is shown as below:



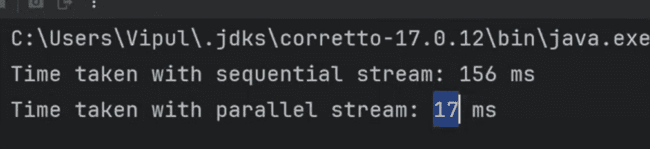
A screen shot of a computer

AI-generated content may be incorrect.

Same time taken with parallel stream are:

**A screen shot of a computer program

AI-generated content may be incorrect.**



Parallel streams are effective for the CPU-insensitive, large data sets where tasks are independent.

But they add overhead for the small tasks or datasets.

Example:

A computer screen shot of a program code

AI-generated content may be incorrect.

***Important: Variable use in lambda expression should be final or effectively final.***

So, we can use the Atomic Integer here in this case as shown below:

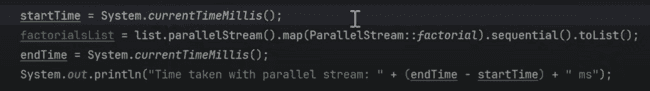
A screenshot of a computer program

AI-generated content may be incorrect.

And the result would be like as shown below as the parallel stream is not independent.

A screen shot of a computer

AI-generated content may be incorrect.

We can also use the .sequential() to convert the parallel stream to sequential one. Like you want the final result to convert to the sequential, you can also do that. 

**Peek():**

**forEach is a terminal operator, but peek is an intermediate operator.**

**Like in this case, peek prints the elements, but does not terminate the operation, so we have used the count, but does not push that into one element.**

**So, we can perform any operation using peek in between and does not do printing.**

****

**toArray(): is a terminal operator and used to covert stream to array and also need termination:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Min, max: return type of this is optional.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**You can also insert the lambda expression in between the sorting order:**

**This would provide you max element:**

****

**While this would provide you the min element:**

****

**Flatmap: Do the flatten of the nested list and transform them**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Like we have list of lists:**

**Now if we need to convert this to uppercase and flat the whole list, then we need to do:**

**A screenshot of a computer

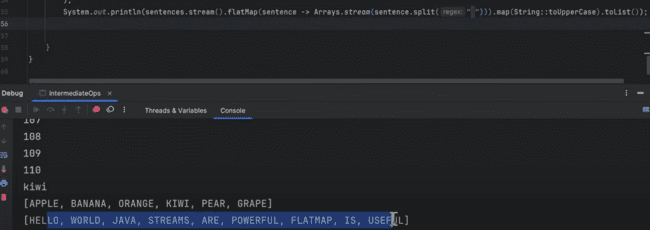
AI-generated content may be incorrect.**

**Now, if we can also apply operations on the map as well like following:**

**A screen shot of a computer

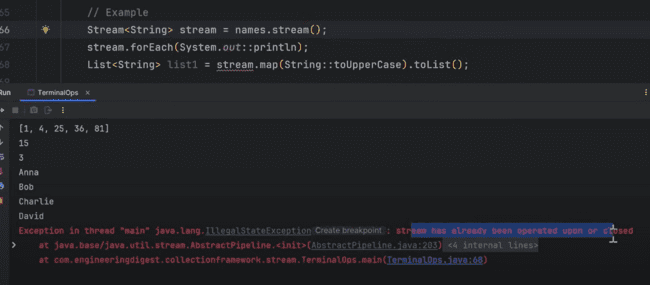
AI-generated content may be incorrect.**

**Now if we want to convert the above in the one list and also make this to uppercase, then we can do this using the following:**

****

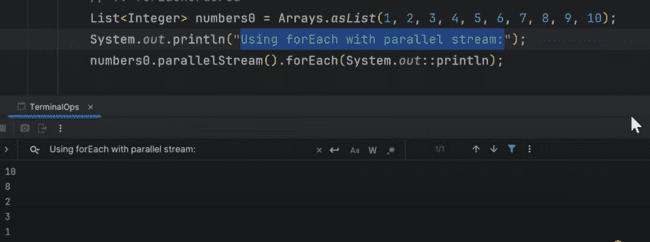
**One important point: Streams cannot be used again, when the terminal operation has been performed on them:**

**As forEach has been performed on the stream, so toList() cannot be performed:**

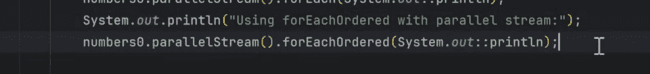
****

**forEachOrdered:**

**With parallel streams, there won’t be any ordering of printing the numbers which is not in sequence**

****

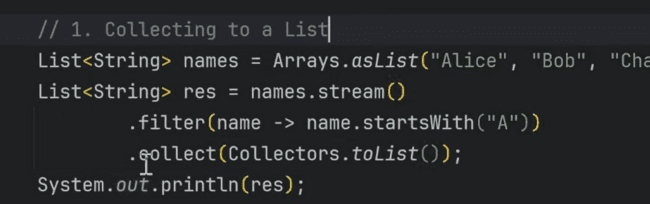
**So, we can use the forEachOrdered using parallel stream, the printing would be done in sequential:**

****

**COLLECTORS:**

**These are used to collect and transform them to a specific collection:**

**toList():**

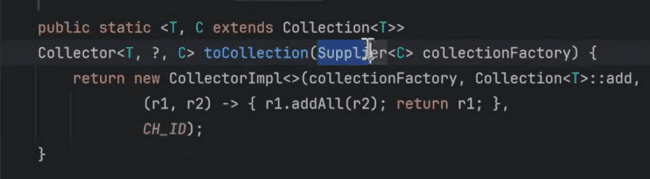
****

**ToSet():**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Collecting to specific collection: Collection.toColection() takes the supplier which is shown as below:**

****

**So, to any collection, we can convert that stream into. Example:**

**A computer screen with a black background

AI-generated content may be incorrect.**

**Joining the names array using joining(): means contatinating streams into a single string.**

****

**Which would print alice, Charlie and bob together as:**

**A close up of a clock

AI-generated content may be incorrect.**

**Using the space and separator it prints like this:**

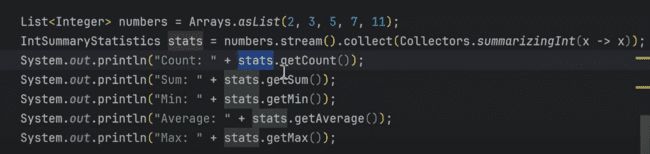
**A screen shot of a computer code

AI-generated content may be incorrect.**

**A black background with white text

AI-generated content may be incorrect.**

**Summarizing the data: which gets the input and takes the output like count, sum, min, max, average.**

****

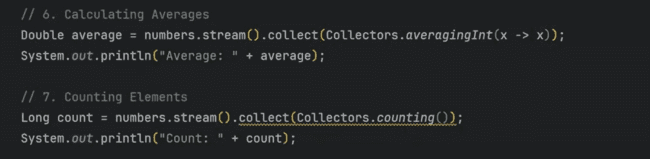
**It would print like:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Calculating the average and count directly without using summarizing, that can be done using following:**

**Question: why in averagingInt(x->x) has been used:**

****

**Grouping the elements:**

**Like printing the elements on the basis of their length, that can be done using groupingBy which takes the classifiers.**

****

**Which can also be written like this:**

****

**The output would be:**

****

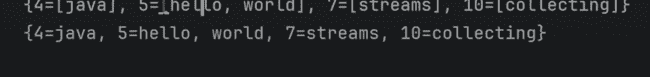
**The other part is classifier with downstream collector:**

**A screenshot of a computer

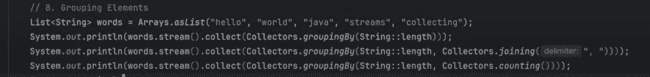
AI-generated content may be incorrect.**

**Which classifies the stream based on length and then perform any operation on them, like joining the lists using the joining method:**

****

****

**Or we can also count them like:**

**A black background with white text

AI-generated content may be incorrect.**