LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Streams

COURSE AGENDA

- What is an Object
- The Basics
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Interfaces
- Polymorphism

- Collections
- Functional Programming
- Streams
- Exceptions
- Enums

INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
  - DTCC
  - Penske
  - HSBC
  - Government Agencies

in https://www.linkedin.com/in/hugoscavino/

REQUIRED SOFTWARE

- JDK™
  - JDK 8 or Newer (Open Source, Amazon, or Oracle)
  - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.
- IDE
  - IntelliJ Community Edition (Free) or Enterprise (Paid)
  - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind

RECOMMENDED TEXTS

- Java 5 Book
  - Thinking in Java – 4th Edition - Free – PDF - GitHub
  - Thinking in Java – 4th Edition – Hard Cover - Amazon
- Recommended Java 8 Book
  - Bruce Eckel on Java 8
    - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
  - Head First Java: A Brain-Friendly Guide 3rd Edition

TOPIC DESCRIPTION

Streams: (Java 8 above) A powerful abstraction found in the `java.util.stream` package. Streams provide a functional programming approach for processing collections of data in a concise and declarative way.

Key Characteristics:

- Declarative: Focuses on what to do rather than how to do it.
- Lazy Evaluation: Operations are not executed until a terminal operation is invoked.
- Pipeline Model: Stream operations can be chained together to form a processing pipeline. (*Choo-Choo*)
- Parallelism: Streams support parallel processing for improved performance.

STREAMS COMPONENTS

- Source: The data source for the stream (e.g., collections, arrays, files, or I/O channels).
- Intermediate Operations: These lazy operations transform a stream into another stream (e.g., map, filter). They do not execute until a terminal operation is called (the lazy part).
- Terminal Operations: At the end, terminal operations produce a result or a side effect (e.g., `forEach`, `collect`, `reduce`).

STREAM TYPES

- Sequential Streams: Process data in a single thread.
- Parallel Streams: Process data in multiple threads for better performance in large datasets

BENEFITS

- Readable Code: Reduce boilerplate code.
- Improved Productivity: Functional operations simplify common tasks like filtering and transformation.
- Parallel Processing: Easy to leverage multi-core architectures.

CHALLENGES

- Not Suitable for All Cases: Streams may not be the best choice for tasks involving stateful computations.
- Learning Curve: Initially challenging for developers unfamiliar with functional programming.
- Debugging Difficulty: Stream operations are more difficult to debug than traditional loops.

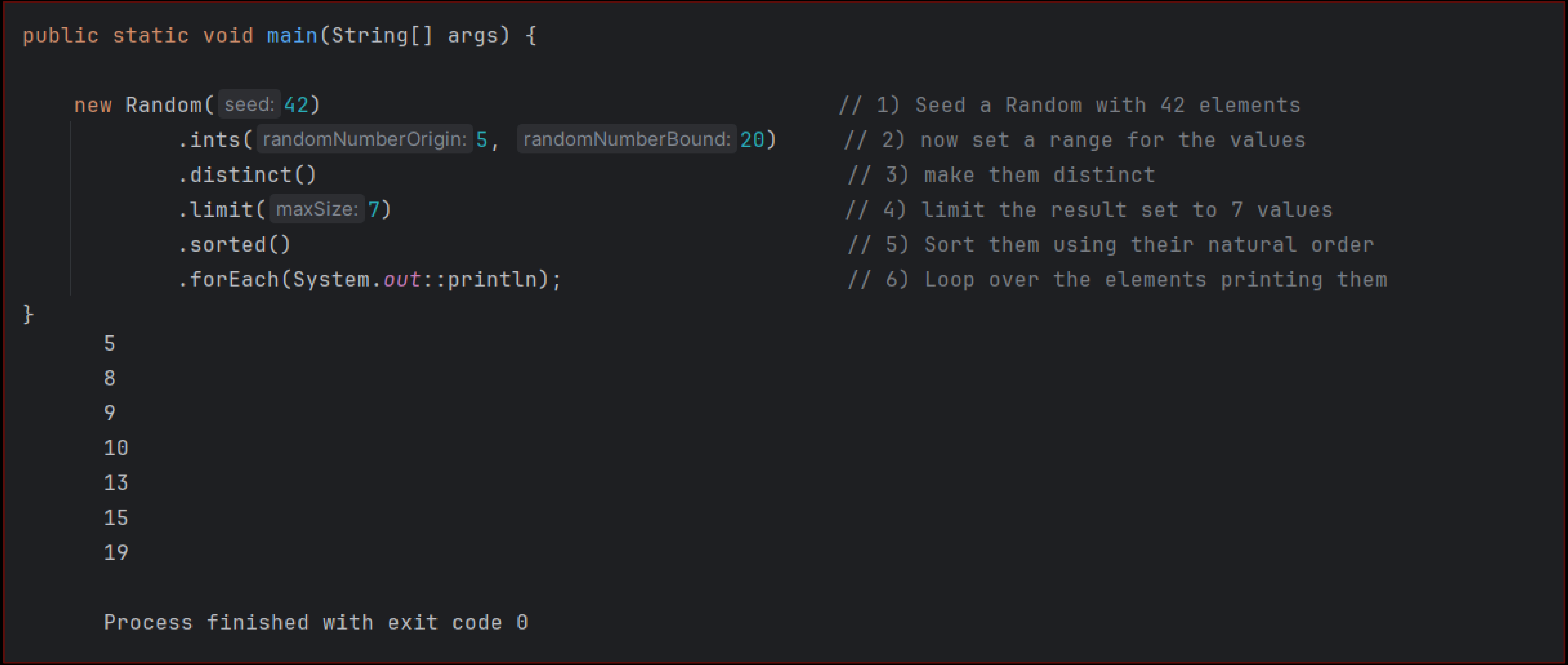
COMMON INTERMEDIATE OPERATIONS

- `filter(Predicate)`: Filters elements based on a condition.
- `map(Function)`: Transforms elements.
- `flatMap(Function)`: Flattens nested structures into a single stream.
- `distinct()`: Removes duplicate elements.
- `sorted()`: Sorts elements.
- `limit(long n)`: Limits the stream to n elements.
- `skip(long n)`: Skips the first n elements

TERMINAL OPERATIONS

- `forEach(Consumer):` Acts on each element.
- `collect(Collector):` Gathers the stream's elements into a collection or another form.
- `reduce(BinaryOperator):` Reduces the stream to a single value.
- `count():` Counts the elements in the stream.
- `findFirst()` / `findAny():` Retrieves an element from the stream.
- `allMatch(Predicate)` / `anyMatch(Predicate)` / `noneMatch(Predicate):` Check conditions on elements.

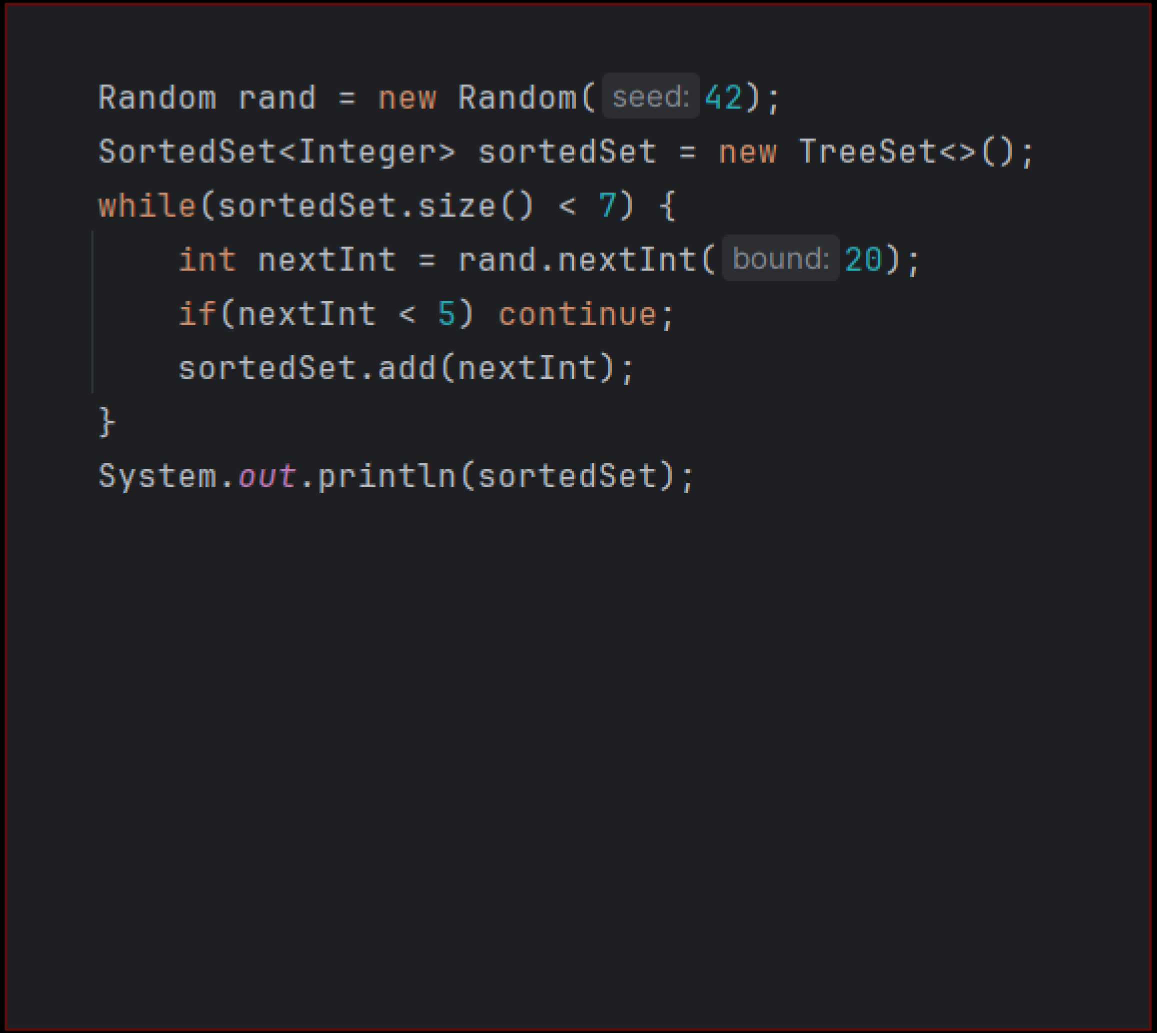
QUICK EXAMPLE



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

    new Random( seed: 42)                                                // 1) Seed a Random with 42 elements
            .ints( randomNumberOrigin: 5,  randomNumberBound: 20)        // 2) now set a range for the values
            .distinct()                                                  // 3) make them distinct
            .limit( maxSize: 7)                                          // 4) limit the result set to 7 values
            .sorted()                                                    // 5) Sort them using their natural order
            .forEach(System.out::println);                               // 6) Loop over the elements printing them
}
```

```
      5
      8
      9
      10
      13
      15
      19

      Process finished with exit code 0
```

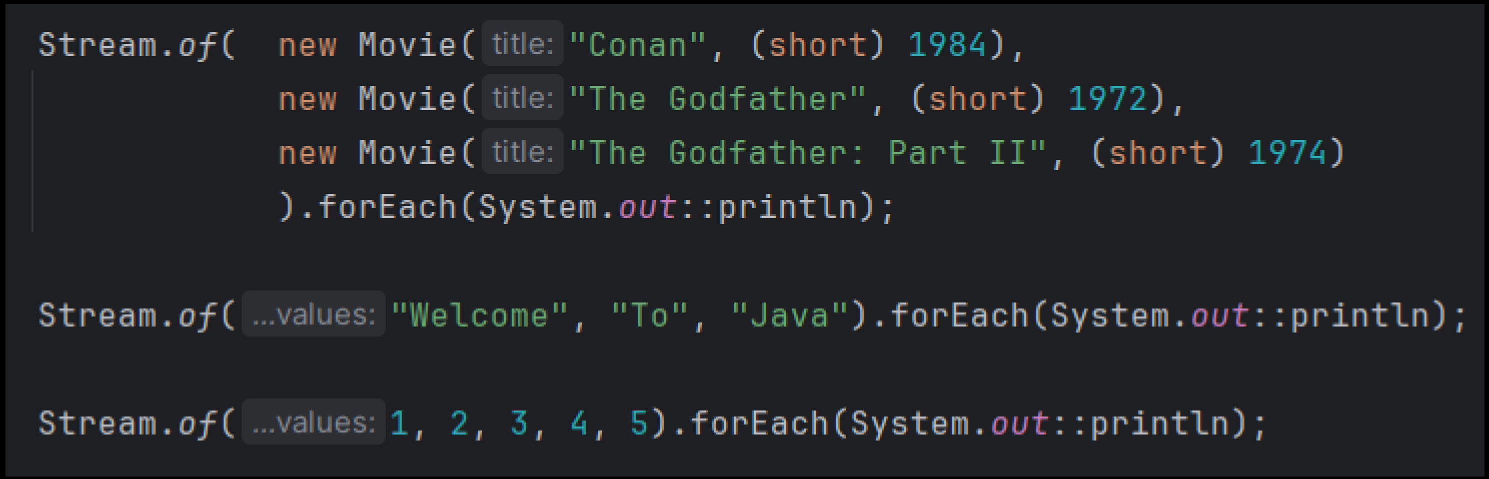
COMPARED TO



```
Random rand = new Random( seed: 42);
SortedSet<Integer> sortedSet = new TreeSet<>();
while(sortedSet.size() < 7) {
    int nextInt = rand.nextInt( bound: 20);
    if(nextInt < 5) continue;
    sortedSet.add(nextInt);
}
System.out.println(sortedSet);
```

CREATING YOUR OWN STREAM

Using the Stream.of() operator

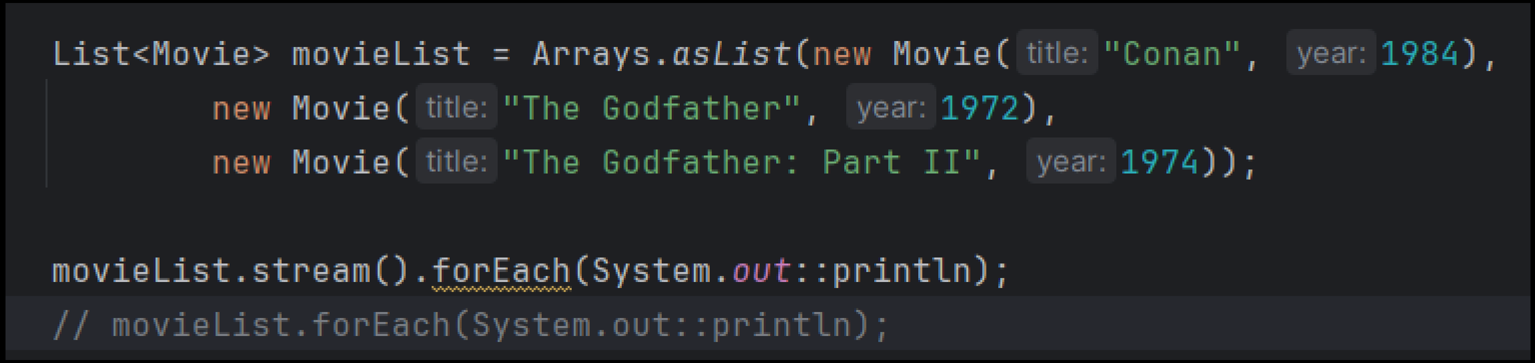


```
Stream.of(  new Movie( title: "Conan", (short) 1984),
            new Movie( title: "The Godfather", (short) 1972),
            new Movie( title: "The Godfather: Part II", (short) 1974)
            ).forEach(System.out::println);

Stream.of( ...values: "Welcome", "To", "Java").forEach(System.out::println);

Stream.of( ...values: 1, 2, 3, 4, 5).forEach(System.out::println);
```

When manually creating objects, Strings, or primitives

STREAM FROM A COLLECTION

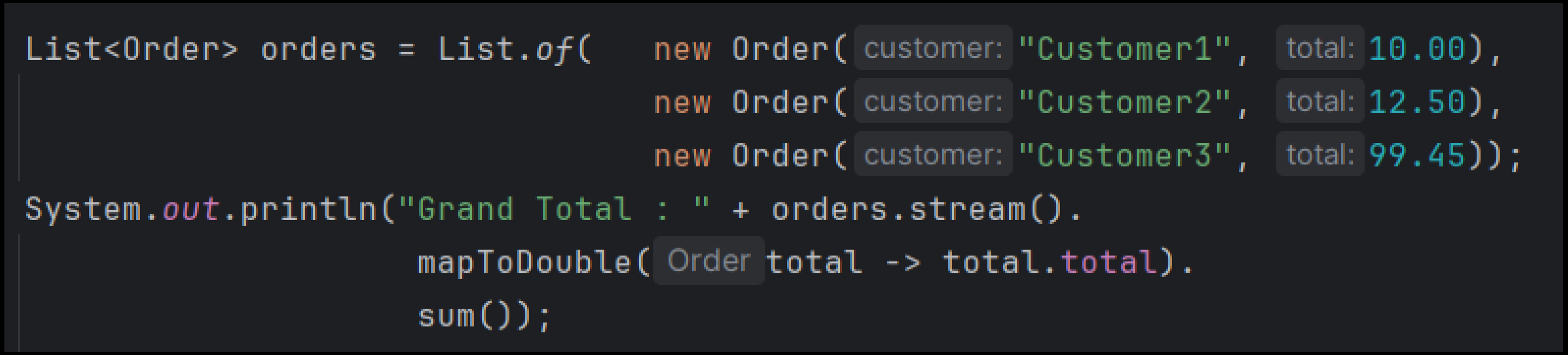
Using a `List<Movie>`



```
List<Movie> movieList = Arrays.asList(new Movie( title: "Conan",  year: 1984),
        new Movie( title: "The Godfather",  year: 1972),
        new Movie( title: "The Godfather: Part II",  year: 1974));

movieList.stream().forEach(System.out::println);
// movieList.forEach(System.out::println);
```

STREAM FROM A COLLECTION AND MAP

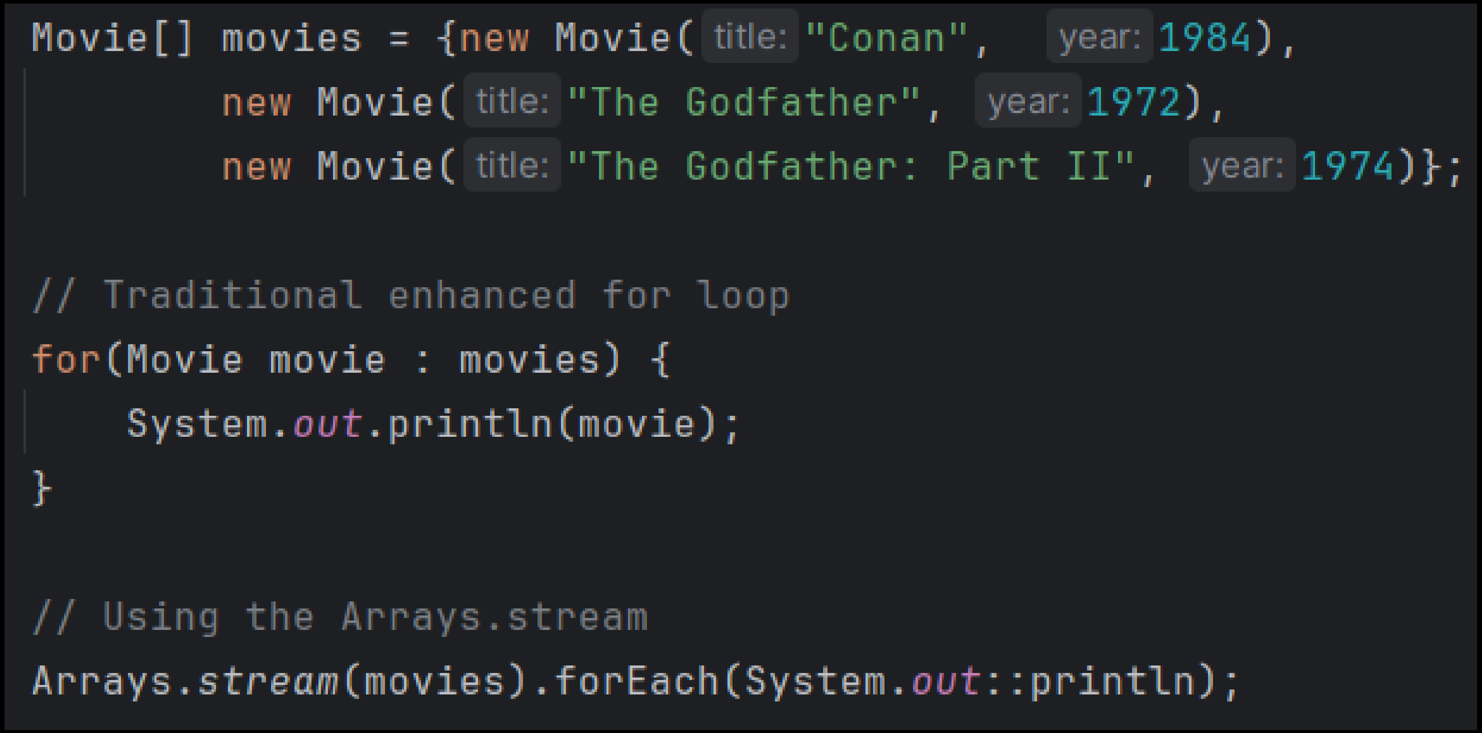
Using a `List<Order>` and `mapToDouble()`



```
List<Order> orders = List.of(   new Order( customer: "Customer1",  total: 10.00),
                                new Order( customer: "Customer2",  total: 12.50),
                                new Order( customer: "Customer3",  total: 99.45));
System.out.println("Grand Total : " + orders.stream().
                    mapToDouble( Order total -> total.total).
                    sum());
```

STREAM FROM AN ARRAY

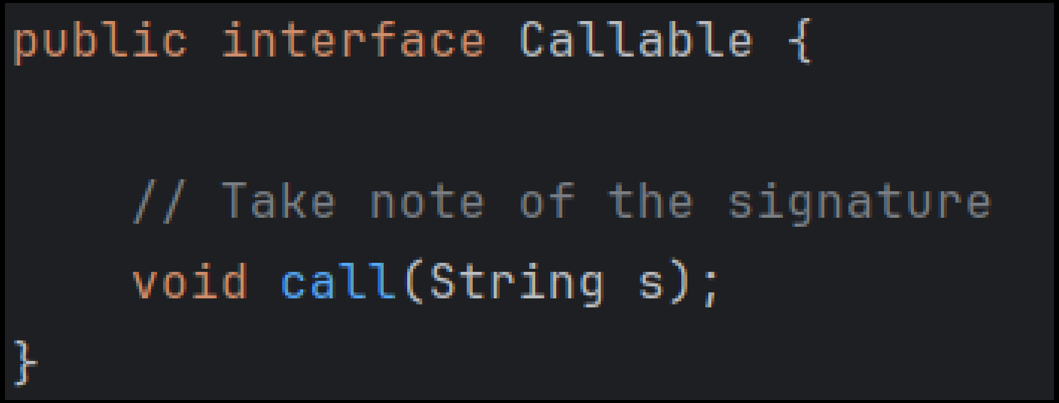
Using an `Array` of type `Movie` or primitives



```
Movie[] movies = {new Movie( title: "Conan",   year: 1984),
        new Movie( title: "The Godfather",  year: 1972),
        new Movie( title: "The Godfather: Part II",  year: 1974)};

// Traditional enhanced for loop
for(Movie movie : movies) {
    System.out.println(movie);
}

// Using the Arrays.stream
Arrays.stream(movies).forEach(System.out::println);
```

METHOD REFERENCE EXAMPLE

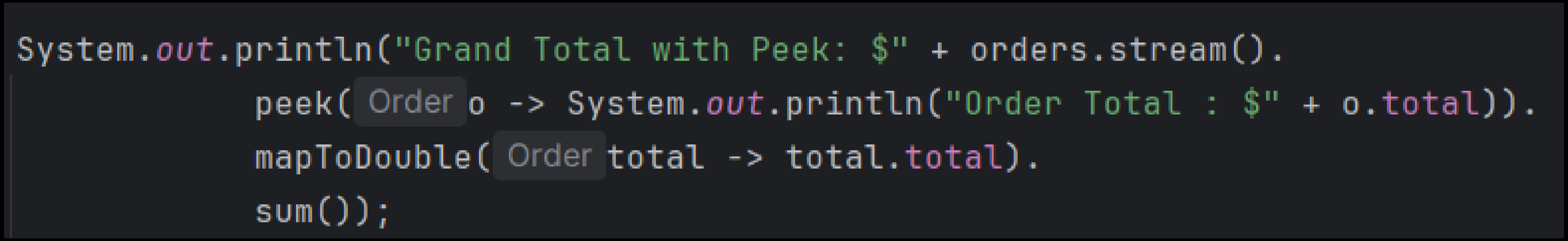
Start with the Callable interface with one void method that takes one parameter of type String



```
public interface Callable {

    // Take note of the signature
    void call(String s);
}
```

DEBUGGING WITH PEEK

Using a `List<Order>`, `peek()`, and then `mapToDouble()`

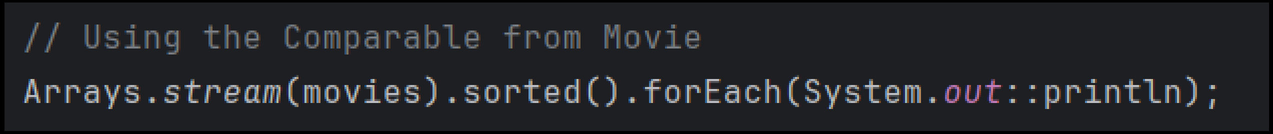


```
System.out.println("Grand Total with Peek: $" + orders.stream().
            peek( Order o -> System.out.println("Order Total : $" + o.total)).
            mapToDouble( Order total -> total.total).
            sum());
```

```
Order Total : $10.0
Order Total : $12.5
Order Total : $99.45
Grand Total with Peek: $121.95
```

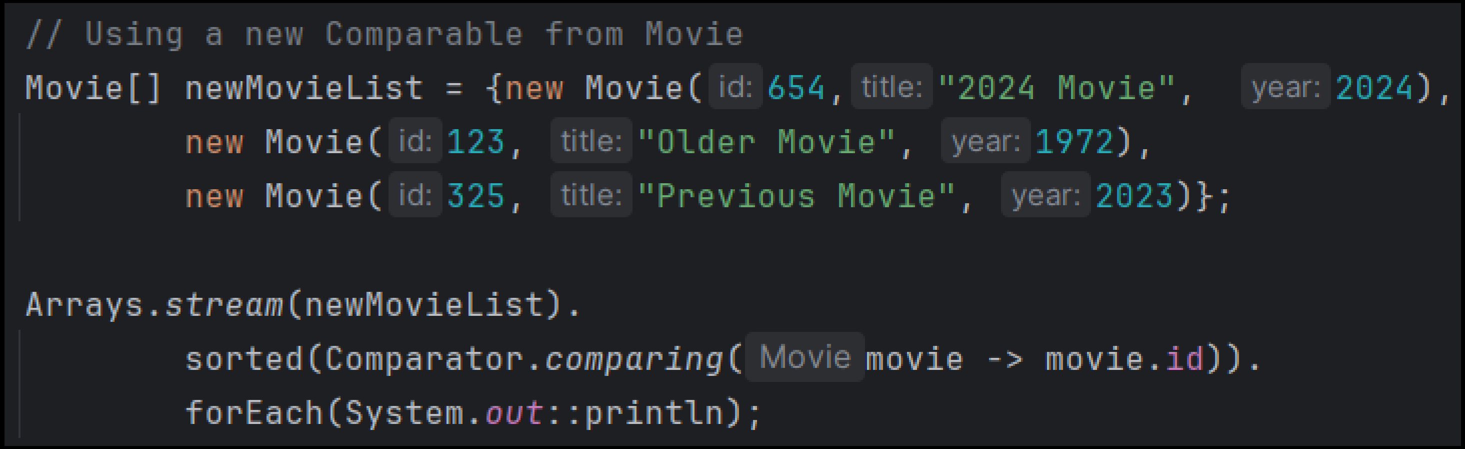
SORTING WITH COMPARATOR

Using `Comparable` from the `Movie` class



```
// Using the Comparable from Movie
Arrays.stream(movies).sorted().forEach(System.out::println);
```

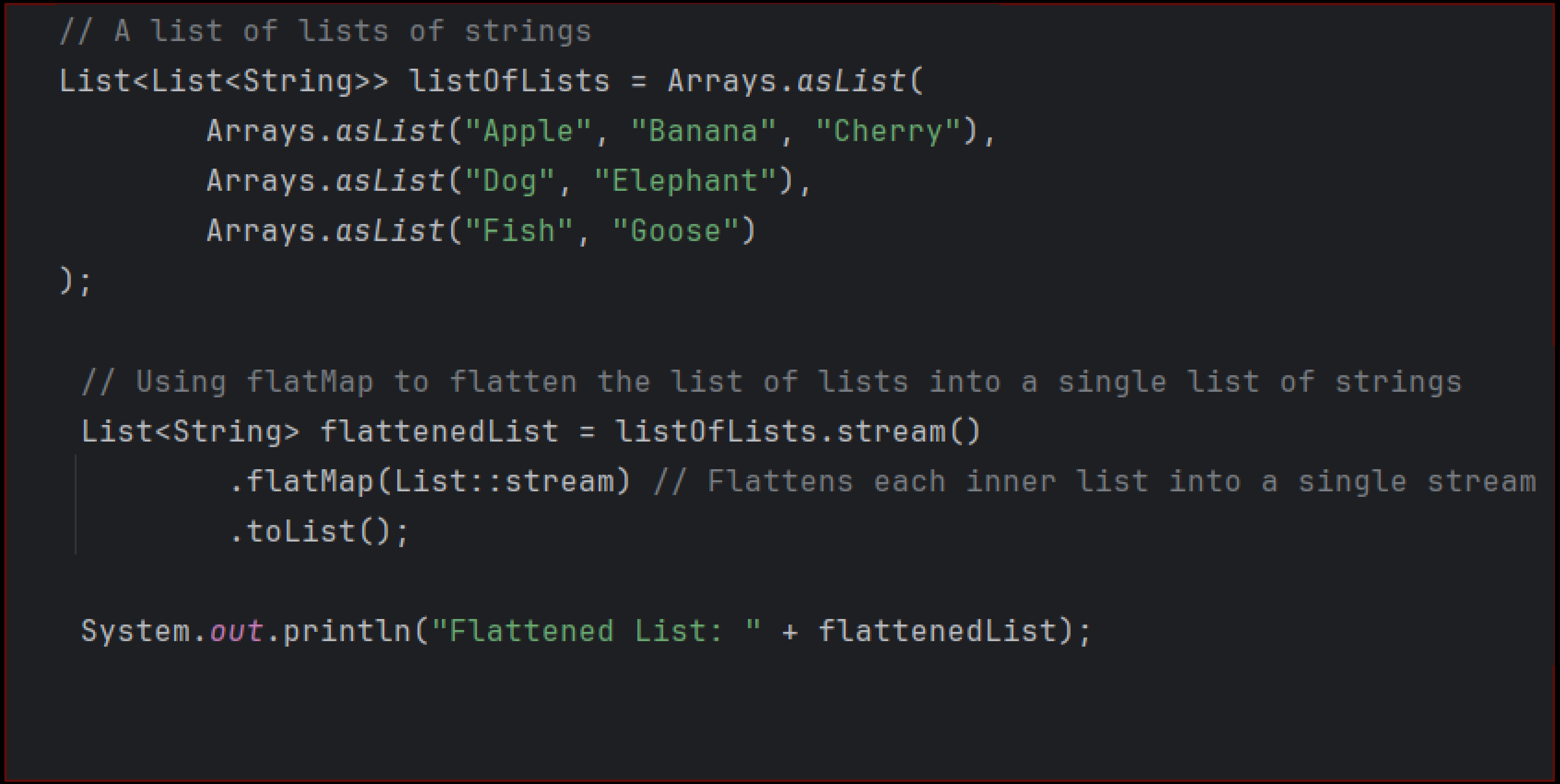
Using a new `Comparator` applied to the `Movie` stream



```
// Using a new Comparable from Movie
Movie[] newMovieList = {new Movie( id: 654, title: "2024 Movie",  year: 2024),
        new Movie( id: 123,  title: "Older Movie",  year: 1972),
        new Movie( id: 325,  title: "Previous Movie",  year: 2023)};

Arrays.stream(newMovieList).
        sorted(Comparator.comparing( Movie movie -> movie.id)).
        forEach(System.out::println);
```

USING FLATMAP

Transforms each stream element into another stream and then flattens the resulting streams into a single stream.



```java
// A list of lists of strings
List<List<String>> listOfLists = Arrays.asList(
        Arrays.asList("Apple", "Banana", "Cherry"),
        Arrays.asList("Dog", "Elephant"),
        Arrays.asList("Fish", "Goose")
);

 // Using flatMap to flatten the list of lists into a single list of strings
 List<String> flattenedList = listOfLists.stream()
         .flatMap(List::stream) // Flattens each inner list into a single stream
         .toList();

 System.out.println("Flattened List: " + flattenedList);
```

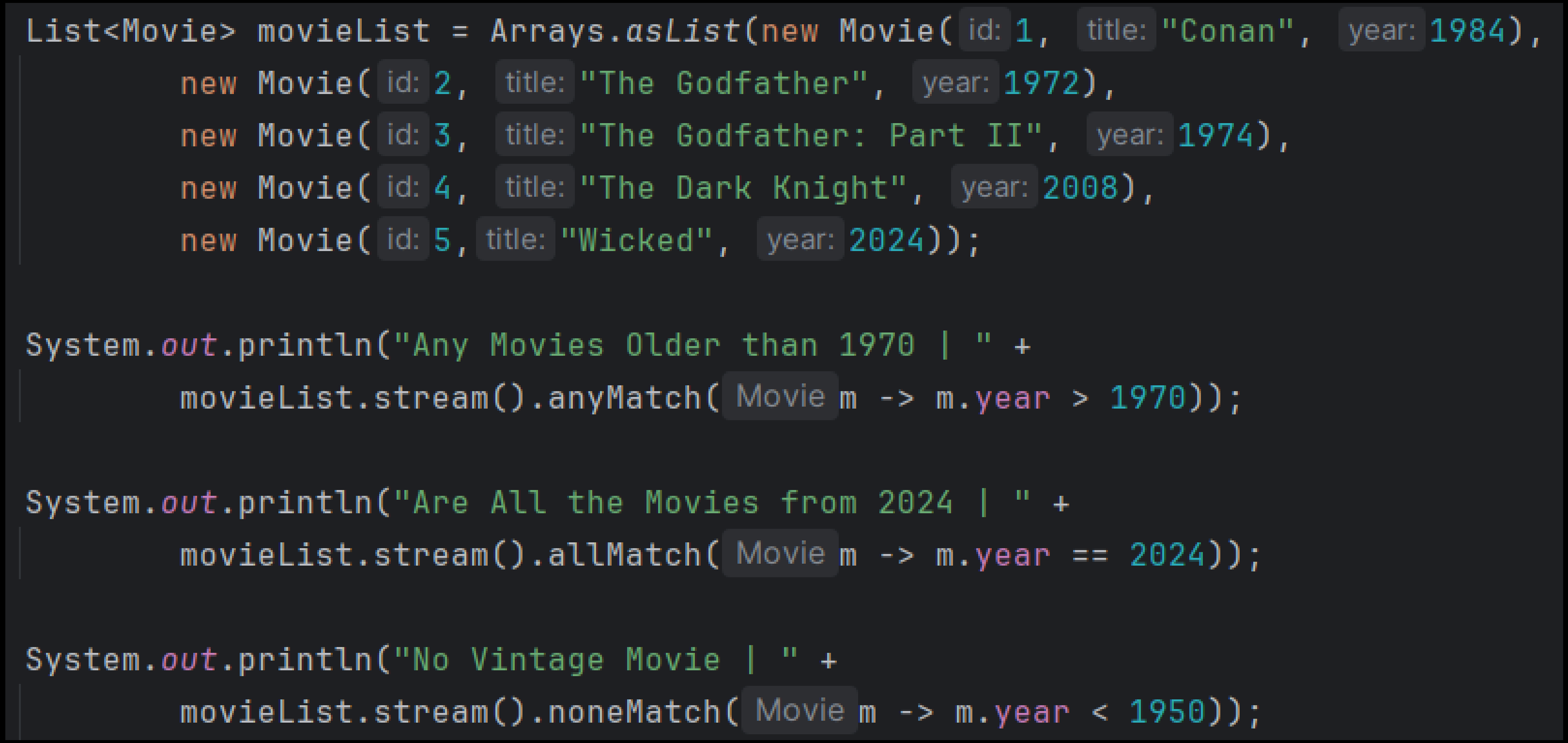
MATCHING

`allMatch(Predicate)`: Returns `true` if every stream element produces true when provided to the supplied Predicate. Short-circuits upon the first false.

`anyMatch(Predicate)`: Returns `true` if any stream element produces true when provided to the supplied Predicate. Short-circuits upon the first true.

`noneMatch(Predicate)`: Returns `true` if no stream element produces true when provided to the supplied Predicate. Short-circuits upon the first true.

MATCHING DEMO



```java
List<Movie> movieList = Arrays.asList(new Movie( id: 1, title: "Conan", year: 1984),
        new Movie( id: 2, title: "The Godfather", year: 1972),
        new Movie( id: 3, title: "The Godfather: Part II", year: 1974),
        new Movie( id: 4, title: "The Dark Knight", year: 2008),
        new Movie( id: 5, title: "Wicked", year: 2024));

System.out.println("Any Movies Older than 1970 | " +
        movieList.stream().anyMatch( Movie m -> m.year > 1970));

System.out.println("Are All the Movies from 2024 | " +
        movieList.stream().allMatch( Movie m -> m.year == 2024));

System.out.println("No Vintage Movie | " +
        movieList.stream().noneMatch( Movie m -> m.year < 1950));
```

```
Any Movies Older than 1970 | true
Are All the Movies from 2024 | false
No Vintage Movie | true
```

INFORMATIONAL

`count()`: The number of elements in this stream.

`max(Comparator)`: This stream's "maximum" element is determined by the Comparator.

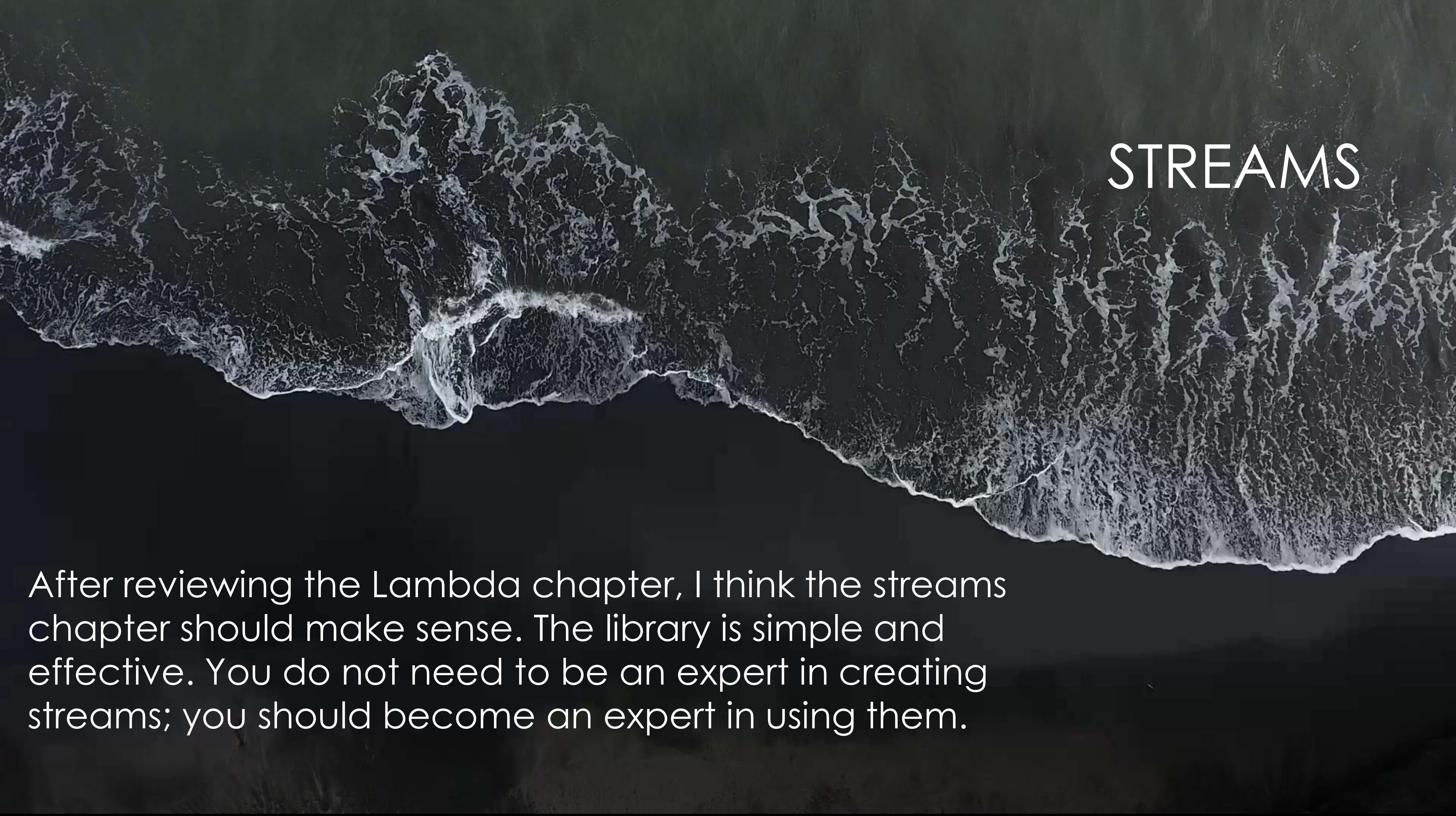
`min(Comparator)`: This stream's "minimum" element is determined by the Comparator.

SELECTING AN ELEMENT

`findFirst():` returns an `Optional` containing the first element of the stream, or `Optional.empty` if the stream has no elements.

`findAny():` returns an `Optional` containing some element of the `stream`, or

`Optional.empty` if the stream has no elements.



STREAMS

After reviewing the Lambda chapter, I think the streams chapter should make sense. The library is simple and effective. You do not need to be an expert in creating streams; you should become an expert in using them.

TOPIC SUMMARY - STREAMS

After reviewing the Lambda chapter, I think the streams chapter should make sense. The library is simple and effective. You do not need to be an expert in creating streams; you should become an expert in using them.