# Chapter 2. Data Streams

## What is a Stream?

In Java, a stream is defined as a typed interface that extends the **BaseStream** interface with a generic type **<T>**, along with its subtype **Stream<T>**. This technical definition points to the fact that a stream is a sequence of elements that can be processed in a functional style, with a strong emphasis on types.

## Why are Streams Efficient?

**Parallel Execution**: Streams can be processed in parallel, which means that the tasks can be split up and run simultaneously across multiple cores of a CPU. This is particularly advantageous when dealing with large data sets as it can significantly reduce the time required for processing.

**Pipelining**: Streams are designed to be pipelined. This means that a series of operations can be linked together, and they will be executed in a way that minimizes intermediate processing. This is achieved by lazy evaluation, where intermediate operations are merged and processed only when necessary, typically at the terminal operation.

A diagram of a cylinder

Description automatically generated

A stream pipeline consists of three parts:

* **Source**: This is the data source from which the stream is created. It could be a **collection**, an **array**, or any other data structure that provides a stream of elements.

List<String> names = Arrays.asList("John", "Jane", "Adam", "Tom");  
Stream<String> namesStream = names.stream();

* **Intermediate Operations**: These are the operations that transform the stream into another one, such as **map** or **filter**. They are usually lazy, meaning they don't actually perform any computation until a terminal operation is invoked.

Stream<Integer> lengths = namesStream.map(String::length);

* **Terminal Operation**: This operation, such as **reduce** or **collect**, triggers the processing of the stream and produces a result. It is eager, meaning it processes the data and produces an outcome immediately upon invocation.

long count = lengths.count();

### Why Can't a Collection be a Stream?

The key difference between a Stream and a Collection lies in the approach to processing: **collections** are typically **processed eagerly**, meaning each operation on a collection is computed **as soon as it's invoked**. Streams, by contrast, are **processed lazily** for most operations, meaning **computations are deferred** **until the result is needed**. This allows for more efficient processing, as unnecessary computations can be avoided, especially when pipelining multiple operations.

By embracing the characteristics of streams—such as their lazy nature, the ability to pipeline operations, and the potential for parallel execution—Java provides a powerful abstraction for data processing that can lead to more readable and potentially more efficient code.

## Lazy and Terminal Operations

A diagram of a program

Description automatically generated

**Lazy Operations**:

* + **stream()**: This operation initiates the stream process. It's the starting point where a collection or array is converted into a stream.
  + **map()**: This transformational operation applies a function to each element of the stream, converting each element into another form.
  + **filter()**: This operation applies a predicate to each element to determine if it should be included in the resultant stream.

These operations are lazy because they don't perform any computation until a terminal operation is invoked. They set up a pipeline of operations that Java can optimize, for instance, by merging operations or skipping unnecessary computations.

**Terminal Operations**:

Stream processing finishes with a single terminal operation.   
There are several options for these operations:

* + **aggregate**: These operations compute a single result from the stream elements, such as **max()**, **min()**, **sum()**, and **average()**. They provide a way to obtain a final result from the processed stream.
  + **reduce**: The **reduce** operation combines all stream elements using a provided associative function to produce a single summary result. For instance, **reduce(0, (a, b) -> a+b)** computes the sum of all elements in the stream, starting with an initial value of 0.
  + **collect**: Collection operations like **collect(toSet())** gather the final stream results into a collection, such as a set or a list.
  + **forEach**: This operation applies a provided function, such as **System.out::println**, to each element of the stream. It is often used for invoking side effects.

For instance, to calculate the average age of persons over 20 years old, the stream approach allows for a more declarative style:

double averageAge = persons.stream()  
 .mapToInt(Person::getAge)  
 .filter(age -> age > 20)  
 .average()  
 .orElseThrow(NoSuchElementException::new);

## Aggregation Functions in Streams

**Available Aggregations:**

* **max()**, **min()**, **count()**: These functions are used to compute the maximum, minimum, or total count of elements in a stream, respectively.

**Example: Finding Maximum Age**

OptionalInt maxAge = persons.stream()  
 .mapToInt(Person::getAge)  
 .max();

**Boolean Reductions:**

* **allMatch(predicate)**: Checks if all elements of the stream satisfy the given predicate.
* **noneMatch(predicate)**: Checks if no elements of the stream satisfy the given predicate.
* **anyMatch(predicate)**: Checks if any elements of the stream satisfy the given predicate.

**Example: Checking if All Persons are Above a Certain Age**

boolean areAllAdults = persons.stream()  
 .allMatch(person -> person.getAge() > 20);

**Reductions that return an Optional:**

* **findFirst()**: Returns an **Optional** describing the first element of the stream, or an empty **Optional** if the stream is empty.
* **findAny()**: Returns an **Optional** describing some element of the stream, or an empty **Optional** if the stream is empty. This is particularly useful in parallel streams as it allows for more flexibility in which element is returned.

**Example 3: Finding the First Adult Person**

Optional<Person> firstAdult = persons.stream()  
 .filter(person -> person.getAge() > 20)  
 .findFirst();

**Reductions are terminal operations**: They are used to conclude the processing of the data in a stream. When a terminal operation is invoked, it triggers the internal iteration of the stream and returns a result.

## Reducing Streams

The **reduce()** method is a general-purpose reduction operation. It takes an identity value and a BinaryOperator to combine two elements and produce a new value. It's a terminal operation that triggers the processing of the stream.

Here's an example that demonstrates the use of **reduce()** to sum a list of integers:

int sum = Stream.of(4, 5, 3, 9, 8)  
 .reduce(0, (a,b) -> a+b);

**Identity element**

In the previous example, we pass 0 as the first argument to the reduce() method.

This argument is named an identity element.  
But why do we need to do that? This is important for several reasons:

1. **Empty Streams**: If the stream is empty, the reduction has nothing to operate on. In such cases, the identity element serves as a default result. For example,

Stream.empty().reduce(0, (a,b) -> a+b )

will return 0 because the identity element is 0.

1. **Single Element Streams**: If the stream contains only one element, the reduction operation would have only that single value to work with. The identity element becomes a no-op in this case, and the single value in the stream is the result of the reduction. For example,

Stream.of(1).reduce(0, (a,b) -> a+b )

will return 1, because method (a,b) -> a+b will be applied to arguments (0, 1) which correspond to the identity element and the only one element of the stream, 1.

1. **Parallel Streams**: When streams are processed in parallel, the elements are split into sub-streams that are reduced separately. The identity element ensures that each sub-stream can be reduced independently without affecting the final result.

The requirements for an identity element in a reduction operation are that it must be an element that, when combined with another element using the provided reduction function, leaves the other element unchanged. For example, it is the number 0 in addition (since x + 0 = x) or the number 1 in multiplication (x \* 1 = x).

**Parallel Processing with reduce() method**

When using **reduce()** in parallel streams, the operation should be commutative to ensure consistent results. In parallel processing, elements are processed in chunks, with the reduction applied within each chunk before combining the results. However, non-commutative operations may yield unpredictable results when used in parallel.

Here's an example of using **reduce()** in a parallel stream:

int parallelSum = Stream.of(4, 5, 3, 9, 8)  
 .parallel()  
 .reduce(0, Integer::*sum*);

The requirements for the function passed to the **reduce** method are as follows:

1. **Commutativity**: The function used for the **reduce** method must be commutative. This means that when the function is applied to multiple pairs of elements, the order in which the operations are performed does not affect the final result, like for addition: a + b = b + a. This is crucial, especially for parallel streams, because it ensures that the result will be the same regardless of the order in which the elements are processed. For example, **addition and multiplication are commutative, whereas subtraction and division are not**. For example, a – b ≠ b – a, and subtraction will return different results depending on how it was used.

**(a, b) -> a + b**

**(a, b)** -> a + “,” + b;

1. **Non-interfering**: The function should not interfere with the source of the elements in the stream. This means that it **should not modify the underlying source** (such as a collection) while the stream operations are in progress.

Let's look at an example of what you should not do, which would violate the non-interfering requirement:

List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));  
  
int sum = numbers.stream()  
 .reduce(0, (a, b) -> {  
 if (b % 2 == 0) numbers.remove(b);

*// This is interfering and can cause undefined behavior* return a + b;  
 });

In the above code, the lambda expression passed to the **reduce** method modifies the source list by removing elements that are even. This is an interfering operation and can lead to unpredictable results or throw a **ConcurrentModificationException** if a parallel stream is used.

1. **Statelessness**: the execution of the function should not depend on any state outside of its input arguments and it should not modify any state outside of its scope. When using **reduce**, the provided reduction function must be stateless to ensure that the result is deterministic and consistent, particularly important when running operations in parallel.

Here's an example of a function used in a **reduce** operation that violates statelessness:

AtomicInteger multiplier = new AtomicInteger(1);  
  
int sum = Stream.of(1, 2, 3, 4, 5)  
 .reduce(0, (a, b) -> {  
 *// The result depends on the external state (multiplier)* int result = a + b \* multiplier.getAndIncrement();

*// This is NOT stateless* return result;  
 });  
  
System.out.println(sum); *// The result will be different on each run*

For the same stream data, you should always get the same results.

Never change the external data inside reductor function.

Here the result depends not only on data in the stream but also on some external state (multiplier). The result of the reduction will vary depending on the order in which the elements are processed, making the function's behavior non-deterministic, especially in a parallel stream where the order of element processing is not guaranteed.

Statelessness should be satisfied for several reasons:

1. **Determinism**: Stateless functions ensure that the result is the same for the same input, regardless of the order in which elements are processed. This is especially crucial in parallel streams where operations may be performed out of order or concurrently.
2. **Concurrency Safety**: Stateless functions do not share state between threads, preventing race conditions or the need for synchronization, which can degrade performance.
3. **Reusability and Predictability**: Stateless functions can be reused in different contexts and provide predictable results, which simplifies debugging and testing.

## Using Pure functions for Stream processing

A pure function is a concept borrowed from functional programming that refers to a function that has the following properties:

1. **Deterministic**: Given the same input, the function will always produce the same output.
2. **No Side Effects**: The function does not cause any observable side effects, such as modifying an external state or interacting with input/output streams.

These properties ensure that a function is predictable and can safely be executed in parallel without causing unexpected behavior due to shared state or side effects.

Having side effects in a function means that the function does more than just return a value; it interacts with some state outside of its scope or changes the system in some way. Both the properties of non-interference and statelessness (discussed before) are related to the concept of side effects in functions.

When you process streams, especially parallel streams, the operations might be split across multiple threads. If your functions are not pure, the result might differ due to race conditions or other interference between threads. Using pure functions eliminates these concerns, making parallel processing safe and predictable.

**Example of Pure function**

public int add(int a, int b) {  
 return a + b;  
}

This **add** function is pure because it always returns the same result for the same input, it depends only on the arguments, and it has no side effects.

**Example of Impure Function with Side Effects:**

private int sum = 0; *// external state*public void addToSum(int value) {  
 sum += value; *// modifies external state*}

**Example of Impure Function with Non-Determinism:**

public double addRandom(double number) {  
 return number + Math.*random*(); *// non-deterministic output*}

**Using Functions in Stream Processing**

When processing streams, especially in parallel, you should aim to use pure functions. For example, consider the following stream operation:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
  
*// Using a pure function with the map operation.*List<Integer> doubled = numbers.stream()  
 .map(n -> n \* 2) *// pure function: deterministic and no side effects* .collect(Collectors.toList());

In contrast, using an impure function in a stream operation can lead to unpredictable results and is generally not safe:

List<Integer> randomAdded = numbers.parallelStream()  
 .map(n -> addRandom(n)) *// impure function: non-deterministic* .collect(Collectors.toList());  
*// The output is unpredictable and different on each run.*

While streams do not enforce the use of pure functions, using them leads to safer, more maintainable, and parallelism-friendly code. Functions with side effects or non-deterministic behavior can lead to hard-to-debug issues, especially when used with parallel streams.

## Immutability in Stream Processing

An object is considered immutable if its state cannot be modified after it has been created. This means that once an instance of an immutable class is created, the object's data cannot be changed.

The most well-known example of an immutable object is class String.

String greeting = "Hello";

String greet = "Hello";

String greet2 = new String("Hello");

greet2.intern() == greet; // true

greet == greeting; // true  
greeting = greeting.toUpperCase();

*// Creates a new String object, does not modify the original*

Even though it might seem like **greeting** is being modified, what actually happens is that **toUpperCase()** creates a new **String** object with all characters in uppercase. The original **greeting** string remains unchanged.

The wrapper classes Integer, Long, or Double are also immutable.

Convert list to immutable list: Collections.unmodifiableList(…)

Here’s an example of how to create a custom immutable class:

public final class ImmutablePerson {  
 private final String name;  
 private final int age;  
  
 public ImmutablePerson(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 *// No setters to change state after construction* public ImmutablePerson setAge(int age) {  
 return new ImmutablePerson(name, age);  
 }

}

As you can see, here we cannot change the state of the class without recreating it.

Here's how you might use this immutable class:

public class Main {  
 public static void main(String[] args) {  
 ImmutablePerson person = new ImmutablePerson("Alice", 30);  
  
 *// Since ImmutablePerson is immutable, we cannot change Alice's age.  
 // Instead, if we need a different age, we have to create a new instance.* ImmutablePerson olderPerson =   
 new ImmutablePerson(person.getName(), person.getAge() + 1);

System.*out*.println(person.getName() +   
 " is " + person.getAge()); *// Alice is 30* System.*out*.println(olderPerson.getName() +   
 " is " + olderPerson.getAge()); *// Alice is 31* }  
}

To "modify" an **ImmutablePerson**, you create a new instance with the new desired state. This is a cornerstone of functional programming and aligns with the principles of stream processing in Java.

### Benefits of Immutability

1. **Predictability**: Immutability ensures that the data remains consistent throughout its lifecycle. When using streams, particularly parallel streams, you can be assured that the data being processed will not change unexpectedly due to side effects from other operations, which might otherwise lead to hard-to-track bugs.
2. **Thread Safety**: In a concurrent or multi-threaded environment, mutable data is prone to race conditions, where multiple threads attempt to modify the same data concurrently. Immutable data structures are inherently thread-safe because their state cannot change after creation, negating the need for synchronization.
3. **Ease of Reasoning**: When you know that an object cannot change state, it becomes easier to understand and reason about how your code operates. There's a clear data flow that can be followed, where data transformations produce new versions of the data rather than mutating the original. This clarity is particularly beneficial when chaining multiple operations in a stream.
4. **Functional Programming Semantics**: Streams are designed to work with pure functions—functions where the return value is only determined by its input values, without observable side effects. Immutability fits perfectly into this paradigm by preventing side effects, which could occur if the data were mutable.
5. **Reduction of Errors**: Mutable state is one of the primary sources of errors in programming because it introduces dependencies between different parts of an application. These dependencies can lead to situations where changing the state in one place can unintentionally affect other parts of the application. With immutability, once an object is created, it can be passed around freely without worrying about it being altered.
6. **Efficient Memory Use in Streams**: Stream operations often rely on creating new modified versions of objects rather than changing the original objects. Immutability can lead to more efficient memory usage patterns, especially in modern JVM implementations that optimize for immutable data structures.
7. **Optimization Opportunities**: Immutable objects are also candidates for various optimizations such as memoization (caching the result of a function based on its input parameters) and sharing. Since immutable objects cannot change, they can be shared freely between different parts of a program without the risk of one part changing the data for all the others.

### Practical example

When working with streams, especially in parallel processing, it's important to avoid modifying the state of the objects being processed because it could lead to unpredictable behavior. Instead, we create new instances with the updated state, leaving the original instances unchanged.

Assume we have a list of Person instances and need to make changes, for example, increase the age of each person by one. How can we do it in an immutable way?

Instead of modifying the original Person instances, we may create new instances of **Person** objects with the updated ages.

Assuming a **Person** class defined as follows:

public class Person {  
 private final String name;  
 private final int age;  
  
 *// Constructor* public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 *// Getter methods* public String getName() { return name; }  
 public int getAge() { return age; }  
  
 *// Method to "change" age - returns a new instance* public Person withIncreasedAge() {  
 return new Person(this.name, this.age + 1);  
 }  
}

To increase the age of each **Person** in an immutable way:

public class ImmutableExample {  
 public static void main(String[] args) {  
 List<Person> people = List.*of*(  
 new Person("Alice", 30),  
 new Person("Bob", 25),  
 new Person("Charlie", 35)  
 );  
  
 *// Using Stream to create a new list with updated ages* List<Person> updatedPeople = people.stream()  
 .map(Person::withIncreasedAge)  
 .collect(Collectors.toList());  
  
 *// Original list remains unchanged,*

*// updatedPeople contains Persons with ages incremented by 1* updatedPeople.forEach(p -> System.*out*.println(

p.getName() + " is now " + p.getAge()));  
 }  
}

In this example, **withIncreasedAge()** is crucial. It adheres to the principles of immutability by returning a new **Person** instance with the incremented age, rather than modifying the existing instance. This approach ensures that the original list and its objects remain unchanged, demonstrating how immutability works in practice.

## Method peek()

The **peek()** method in Java Streams is an intermediate operation that allows you to perform a specified action on each element of a stream as elements are consumed from the resulting stream. Typically, **peek()** is used for debugging purposes, as it allows you to observe the elements as they flow past a certain point in a stream pipeline without altering them.

Here is the signature of the **peek()** method:

Stream<T> peek(Consumer<? super T> action)

The method takes a **Consumer** functional interface, which means it accepts a lambda expression or method reference that takes one parameter and returns no result.

**Use Cases for peek():**

1. **Debugging**: The most common use case for **peek()** is to print out the elements of the stream to debug or understand what the stream contains at a given point in the pipeline.

List<String> myList = Arrays.asList("a", "b", "c", "d");  
  
List<String> result = myList.stream()  
 .peek(System.*out*::println) *// Prints each element of the stream* .collect(Collectors.toList());

1. **Monitoring**: You can use **peek()** to observe the state of elements at various stages in the stream without modifying them, such as before and after a **map()** or **filter()** operation.

myList.stream()  
 .peek(e -> System.out.println("Before filter: " + e))  
 .filter(s -> s.startsWith("a"))  
 .peek(e -> System.out.println("After filter: " + e))  
 .collect(Collectors.toList());

1. **Performing Actions**: Even though it's not recommended, **peek()** can be used to perform actions on the elements of the stream, as long as these actions do not interfere with the stream itself (like modifying the underlying source). This is not a common practice as it can lead to side effects that go against the functional programming principles that streams are based on.

Since **peek()** is an intermediate operation, it will not be executed until a terminal operation is invoked. Without a terminal operation, the **peek()** will not do anything because intermediate operations are lazy.

Using **peek()** to modify the underlying source of the stream can lead to unpredictable behavior and is strongly discouraged. This is because streams are designed to work with a non-interfering data source.

**peek()** is often used in a debugging scenario where you want to see the elements without changing the stream. For production code, side-effect operations should generally be avoided in favor of pure functions.

## Stream Collectors

Collectors are a final step in the processing of data using Java Streams, where we transition from the stream world back into a data structure or a desired result. They are most commonly used with the **collect()** terminal operation and provide various utility functions to put the results of the stream into a List, Set, Map, or even reduce the elements down to a single value.

The **java.util.stream.Collectors** class provides a number of pre-defined collectors to handle common mutable reduction tasks:

**Examples of Commonly Used Collectors:**

1. **toList()**: Collects the elements of a stream into a **List**.

List<String> list = stream.collect(Collectors.toList());

1. **toSet()**: Collects the elements of a stream into a **Set** which removes any duplicate elements.

Set<String> set = stream.collect(Collectors.toSet());

1. **toMap()**: Collects elements into a **Map**. This requires two functions: one for the key and one for the value.

Map<Integer, String> map = stream.collect(

Collectors.toMap(String::length, Function.identity()));

1. **joining()**: Joins the elements of a stream into a single **String**.

String result = stream.collect(Collectors.joining(", "));

1. **groupingBy()**: Groups the elements of the stream by a classifier function.

Map<Integer, List<String>> groupedByLength =

stream.collect(Collectors.groupingBy(String::length));

1. **partitioningBy()**: Partitions the elements into a **Map** of **List**s based on a predicate. The map will have two keys: **true** and **false**.

Map<Boolean, List<Integer>> isEven =   
 stream.collect(Collectors.partitioningBy(i -> i % 2 == 0));

1. **counting()**: Returns a collector that counts the number of elements.

long count = stream.collect(Collectors.counting());

1. **summarizingInt()**: Collects statistics, such as count, sum, min, average, and max, for the **int** data type.

IntSummaryStatistics stats =

stream.collect(Collectors.summarizingInt(Integer::intValue));

1. **reducing()**: Performs a reduction on the elements of the stream using an associative accumulation function and returns an **Optional**.

Optional<Integer> total = stream.collect(

Collectors.reducing((i1, i2) -> i1 + i2));

**Usage Examples:**

Here is how you might use some of these collectors in a real-world scenario.

*// Assume we have a stream of strings*Stream<String> namesStream = Stream.of("John", "Jane", "Doe", "Jane");  
  
*// Collect into a list*List<String> namesList = namesStream.collect(Collectors.toList());  
  
*// Collect into a set (removing duplicates)*Set<String> namesSet = namesStream.collect(Collectors.toSet());  
  
*// Joining collector with delimiter, prefix and suffix*String namesString = namesStream.collect(Collectors.joining(", ", "[", "]"));  
  
*// Group by the length of the string*Map<Integer, List<String>> namesByLength =

namesStream.collect(Collectors.groupingBy(String::length));  
  
*// Partition students by passing grade*Map<Boolean, List<Student>> passingStudents =

studentsStream.collect(Collectors.partitioningBy(s -> s.getGrade() >= 50));  
  
*// Get the summary statistics for ages of persons*IntSummaryStatistics ageSummary =

personsStream.collect(Collectors.summarizingInt(Person::getAge));

## Downstream Collectors

Downstream collectors in Java are collectors that act on the results of another collector, typically used in conjunction with grouping or partitioning operations. They enable you to perform additional transformations and operations on the results of a primary collector. The idea is to "pipe" the output of one collector into another, much like a "downstream" process in a pipeline.

Here's how downstream collectors commonly come into play:

1. **groupingBy**: When using **groupingBy**, you often want to do more than just group the elements into lists. Downstream collectors allow you to perform additional operations such as counting, summarizing, or further grouping on the elements of each group.
2. **partitioningBy**: Similarly, with **partitioningBy**, you can use a downstream collector to process the elements that fall into the **true** or **false** partitions of your partitioning function.

**Examples of Downstream Collectors:**

1. **groupingBy with counting**:

Map<Integer, Long> countByLength = strings.stream()  
 .collect(Collectors.groupingBy(

String::length, Collectors.counting()));

In the above example, **Collectors.counting()** is a downstream collector that counts the number of elements in each group created by **groupingBy**.

1. **groupingBy with mapping:**

Map<Integer, Set<String>> firstLetterByLength =

strings.stream()  
 .collect(Collectors.groupingBy(  
 String::length,  
 Collectors.mapping(

s -> s.substring(0, 1), Collectors.toSet())  
 ));

Here, Collectors.mapping() is the downstream collector that transforms each string to its first letter, and then Collectors.toSet() ensures that the result is a set of these first letters.

1. **groupingBy with summarizingInt**:

Map<Boolean, IntSummaryStatistics> ageStatsByAdult = persons.stream()  
 .collect(Collectors.partitioningBy(  
 p -> p.getAge() >= 18,  
 Collectors.summarizingInt(Person::getAge)  
 ));

In this case, **Collectors.summarizingInt()** is the downstream collector used to gather statistics about the ages in each partition (adults vs non-adults).

1. **groupingBy with reducing**:

Map<Integer, Optional<String>> longestStringByLength = strings.stream()  
 .collect(Collectors.groupingBy(  
 String::length,  
 Collectors.reducing(

(s1, s2) -> s1.length() > s2.length() ? s1 : s2)  
 ));

Here, **Collectors.reducing()** is a downstream collector that finds the longest string in each group of strings of the same length.

Downstream collectors are very powerful because they add a layer of processing that allows for more complex aggregations and transformations.