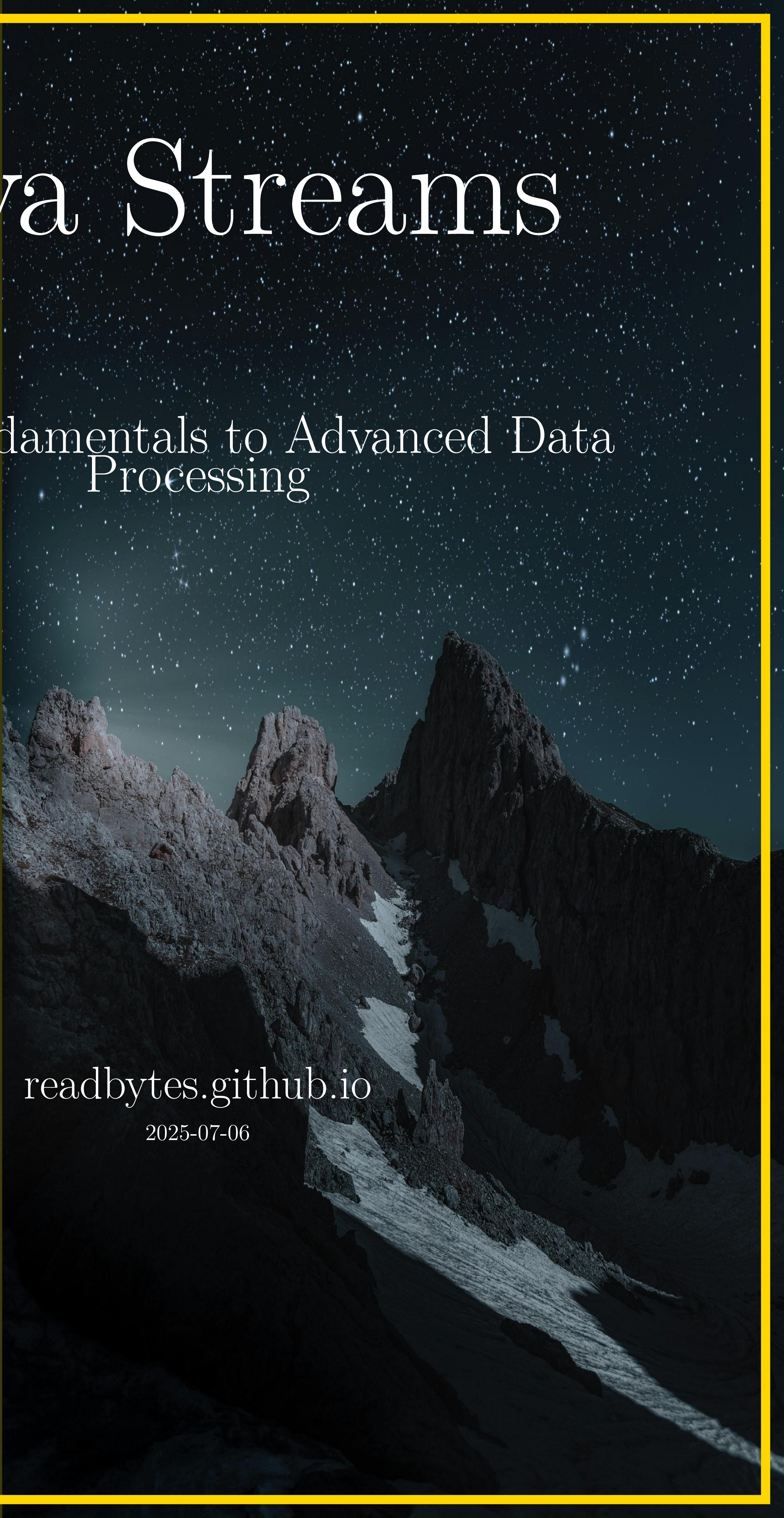
Java Streams

From Fundamentals to Advanced Data Processing

readbytes.github.io

2025-07-06



This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Java Streams

1. What is a Stream?
2. Benefits of Using Streams
3. Stream vs Collection
4. Writing Your First Stream Example

1 Introduction to Java Streams

1.1 What is a Stream?

In Java, a *Stream* is an abstraction that represents a sequence of elements supporting a wide range of *aggregate operations*, such as filtering, mapping, and reducing. Unlike traditional collections (like `List`, `Set`, or `Map`), a stream does not store data. Instead, it conveys elements from a data source—such as a collection, an array, or an I/O channel—through a pipeline of computational steps.

Streams are a central feature introduced in Java 8 as part of the *java.util.stream* package, enabling developers to write more declarative and functional-style code. A stream pipeline consists of three main components: a **source** (the origin of elements), **intermediate operations** (which transform or filter elements), and a **terminal operation** (which produces a result or side effect).

One of the key characteristics of streams is that they are *functional.* This means they operate on data through functions (like `map()`, `filter()`, `reduce()`), promoting immutability and avoiding side effects. Additionally, streams are *lazy*: intermediate operations are not performed until a terminal operation is invoked, which allows for performance optimizations such as short-circuiting and reduced traversals.

Streams can also be *possibly infinite.* For example, using `Stream.generate()` or `Stream.iterate()`, one can construct streams that produce unbounded sequences—useful in scenarios like simulation or data generation.

Importantly, a stream is *not a data structure.* It doesn't hold elements but processes them on-demand. This makes it fundamentally different from collections, which are primarily designed for storage and retrieval. While collections expose methods for manipulating stored data, streams focus on describing *what* computation should be performed, not *how* it should be done.

In essence, Java Streams provide a high-level, expressive, and efficient way to process data by chaining together a series of transformation steps over a sequence of elements.

1.2 Benefits of Using Streams

Java Streams offer several compelling benefits that make them a powerful tool for data processing. By shifting from imperative to declarative programming, Streams allow developers to write more concise, readable, and maintainable code.

1. Declarative Style and Readability Streams promote a high-level, declarative approach to expressing computation. Rather than writing detailed instructions on how to manipulate data (as with loops), you describe *what* should be done.

```java
// Imperative
List<String> result = new ArrayList<>();
for (String name : names) {
    if (name.startsWith("A")) {
        result.add(name.toUpperCase());
    }
}

// Declarative
List<String> result = names.stream()
    .filter(n -> n.startsWith("A"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

2. Lazy Evaluation Streams are lazy—operations like `filter()` and `map()` are only performed when a terminal operation (e.g., `collect()`, `forEach()`) is invoked. This allows for optimization, such as skipping unnecessary computations.

3. Ease of Parallelization Streams support easy parallel processing. By switching from `.stream()` to `.parallelStream()`, the pipeline can take advantage of multi-core CPUs for improved performance with large datasets.

```java
long count = names.parallelStream()
    .filter(n -> n.startsWith("A"))
    .count();
```

4. Composability and Conciseness Stream operations can be composed fluently into pipelines, reducing boilerplate code and enabling reuse of transformation logic.

Overall, Streams make it easier to build complex data processing pipelines that are clean, efficient, and easy to reason about—especially compared to traditional loop-based code.

1.3 Stream vs Collection

While both Streams and Collections deal with sequences of data, they serve fundamentally different purposes in Java.

A **Collection** (e.g., `List`, `Set`) is a data structure that stores elements in memory. It is designed for managing and organizing data—adding, removing, and accessing elements. Collections are *mutable* (unless explicitly made immutable) and support direct access to elements through iteration or indexing.

In contrast, a **Stream** is not a data structure. It represents a *pipeline of operations* on data. A stream is *not meant to store or modify data*, but to *process* it, often using a chain of functional-style operations. Streams are typically *immutable* and operate in a *lazy* and *declarative* manner.

Here's how you can convert a Collection to a Stream:

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Stream<String> nameStream = names.stream();
```

And here's how to collect the results of a Stream back into a Collection:

```java
List<String> filteredNames = names.stream()
    .filter(n -> n.startsWith("A"))
    .collect(Collectors.toList());
```

In summary, use **Collections** for storing and organizing data, and use **Streams** for transforming and processing that data in a fluent and expressive way.

1.4 Writing Your First Stream Example

Let's walk through a simple yet complete example that demonstrates how to create and use a Stream in Java. We'll start with a list of names, filter those that start with the letter "A", convert them to uppercase, and collect the results into a new list.

Here's the full runnable code:

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
    public static void main(String[] args) {
        // Step 1: Create a list of names
        List<String> names = Arrays.asList("Alice", "Bob", "Amanda", "Brian", "Alex");

        // Step 2: Use a stream to filter and transform the data
        List<String> filteredNames = names.stream()                // Create a stream from the list
            .filter(name -> name.startsWith("A"))                 // Keep names starting with "A"
            .map(String::toUpperCase)                             // Convert each to uppercase
            .collect(Collectors.toList());                        // Collect the result into a new list

        // Step 3: Print the result
        System.out.println(filteredNames); // Output: [ALICE, AMANDA, ALEX]
    }
}
```

1.4.1 Step-by-Step Explanation:

- **Step 1:** We start with a list of strings (`names`) using `Arrays.asList()` for simplicity.
- **Step 2:** We call `.stream()` on the list, which returns a Stream of elements.
  - `.filter(name -> name.startsWith("A"))` keeps only names beginning with the

letter “A”.
    - `.map(String::toUpperCase)` converts each remaining name to uppercase.
    - `.collect(Collectors.toList())` gathers the transformed results into a new `List<String>`.

- **Step 3:** We print the final list to the console.

This basic pipeline demonstrates the core power of Streams: **clean, readable, and declarative data processing**. As you become more familiar with Streams, you’ll be able to build much more complex transformations with similar simplicity.

Chapter 2.
Creating Streams

1. Creating Streams from Collections
2. Creating Streams from Arrays
3. Creating Streams from Files and I/O
4. Generating Streams with `Stream.of()`, `Stream.generate()`, `Stream.iterate()`

2 Creating Streams

2.1 Creating Streams from Collections

In Java, all core Collection types—such as `List`, `Set`, and `Queue`—provide convenient methods to create Streams: `.stream()` for sequential streams and `.parallelStream()` for parallel processing. These methods allow you to transform your existing collection data into a stream pipeline, unlocking powerful, functional-style operations.

For example, given a `List` of strings, you can easily create a stream and perform filtering:

```java
List<String> names = List.of("Alice", "Bob", "Charlie");
names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(System.out::println);  // Prints: Alice
```

Similarly, a `Set` can be streamed to perform mapping operations:

```java
Set<Integer> numbers = Set.of(1, 2, 3, 4);
numbers.stream()
       .map(n -> n * n)
       .forEach(System.out::println);  // Prints squares: 1, 4, 9, 16 (order not guaranteed)
```

Streams also work seamlessly with queues:

```java
Queue<Double> queue = new LinkedList<>(List.of(1.5, 2.5, 3.5));
queue.stream()
     .filter(d -> d > 2)
     .forEach(System.out::println);  // Prints: 2.5, 3.5
```

Tips and Pitfalls:

- The stream pipeline does *not* modify the underlying collection unless explicitly collected back.
- Use `parallelStream()` cautiously; it's beneficial for large datasets and CPU-intensive tasks but may add overhead for small or I/O-bound workloads.
- Be aware that some collections (like `HashSet`) do not guarantee element order in streams, which can affect downstream operations.

By leveraging `stream()` and `parallelStream()`, you can write expressive, concise code that efficiently processes collection data.

2.2 Creating Streams from Arrays

Java provides convenient ways to create streams from arrays, enabling powerful, functional-style data processing.

The most common method is **`Arrays.stream()`**, which can convert both object arrays and primitive arrays into streams. For example, given an array of `String`s:

```java
String[] fruits = {"apple", "banana", "cherry"};
Arrays.stream(fruits)
      .filter(f -> f.startsWith("b"))
      .forEach(System.out::println);
// Output: banana
```

For primitive arrays like `int[]`, `long[]`, or `double[]`, `Arrays.stream()` returns specialized streams: `IntStream`, `LongStream`, and `DoubleStream`, respectively. These allow efficient operations without boxing overhead:

```java
int[] numbers = {1, 2, 3, 4, 5};
int sumOfSquares = Arrays.stream(numbers)
                         .map(n -> n * n)
                         .sum();
System.out.println(sumOfSquares); // Output: 55
```

Another useful method is **`Stream.of()`**, which creates a stream from its arguments or an array:

```java
Stream.of("cat", "dog", "elephant")
      .filter(s -> s.length() > 3)
      .forEach(System.out::println);
// Output: elephant
```

Note that for primitive arrays, `Stream.of()` will create a stream of the entire array as a single element, not individual elements. Therefore, prefer `Arrays.stream()` for primitive arrays.

These methods provide flexible entry points to stream processing from arrays, enabling concise and readable code.

2.3 Creating Streams from Files and I/O

Java's `java.nio.file.Files` class provides powerful methods to create streams directly from files, making it easy to process file content using the Stream API. The most commonly used method is **`Files.lines(Path path)`**, which returns a `Stream<String>` where each element represents a line in the file.

Creating a Stream from a File

Here's how you can read a file line-by-line as a stream:

```java
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.stream.Stream;

public class FileStreamExample {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        // Using try-with-resources to ensure the stream and file are properly closed
        try (Stream<String> lines = Files.lines(path)) {
            lines.filter(line -> line.contains("Java"))
                 .map(String::toUpperCase)
                 .forEach(System.out::println);
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

In this example, `Files.lines(path)` lazily reads lines from the file `"example.txt"`. The stream is filtered to keep only lines containing `"Java"`, converted to uppercase, and then printed.

Resource Management and Exception Handling

When working with streams from files or other I/O sources, **resource management is crucial**. The stream must be closed to release the underlying file handle and system resources. This is why the stream is created inside a **try-with-resources** statement, which ensures it closes automatically—even if exceptions occur.

Since `Files.lines()` can throw an `IOException`, proper exception handling is necessary to handle errors like missing files or permission issues gracefully.

Other I/O Stream Sources

Besides files, streams can be created from other I/O sources like:

- `BufferedReader.lines()` for reading from any `Reader`
- `InputStream` can be adapted to streams, often requiring conversion from bytes to characters

Using streams with I/O opens up elegant and efficient ways to process data pipelines without manual iteration or buffering.

This combination of `Files.lines()`, try-with-resources, and Stream operations empowers concise and safe file processing in Java.

2.4 Generating Streams with `Stream.of()`, `Stream.generate()`, `Stream.iterate()`

Java provides several factory methods for generating streams directly, without relying on collections, arrays, or files. These methods—`Stream.of()`, `Stream.generate()`, and `Stream.iterate()`—offer flexible ways to produce both finite and infinite streams.

`Stream.of()`: Static Values

`Stream.of()` is used to create a stream from a known set of values. It is ideal for small, fixed-size sequences:

```java
import java.util.stream.Stream;

public class StreamOfExample {
    public static void main(String[] args) {
        Stream.of("apple", "banana", "cherry")
              .map(String::toUpperCase)
              .forEach(System.out::println);
        // Output: APPLE, BANANA, CHERRY
    }
}
```

Use `Stream.of()` when you have a predefined list of elements, not for dynamic or generated data.

`Stream.generate()`: Infinite Supplier-Based Streams

`Stream.generate(Supplier<T>)` produces an infinite stream by repeatedly calling a `Supplier`. To avoid infinite execution, always combine it with `limit()`:

```java
import java.util.stream.Stream;
import java.util.Random;

public class StreamGenerateExample {
    public static void main(String[] args) {
        Stream.generate(() -> new Random().nextInt(100))
              .limit(5)
              .forEach(System.out::println);
        // Output: 5 random integers
    }
}
```

This is useful for generating dynamic data like random values or timestamps.

`Stream.iterate()`: Sequences Based on Seed and UnaryOperator

`Stream.iterate(seed, unaryOperator)` creates a stream where each element is derived from the previous. It's excellent for number sequences or repeated patterns:

```java
import java.util.stream.Stream;

public class StreamIterateExample {
    public static void main(String[] args) {
        Stream.iterate(0, n -> n + 2)
              .limit(5)
              .forEach(System.out::println);
        // Output: 0, 2, 4, 6, 8
    }
}
```

As with `generate()`, use `limit()` to prevent infinite loops.

In summary:

- **`Stream.of()`**: finite and explicit values
- **`Stream.generate()`**: infinite, random/dynamic values
- **`Stream.iterate()`**: infinite sequences with predictable progression

These methods give developers powerful control over stream creation without relying on existing data sources.

Chapter 3.

Stream Operations Overview

3 Stream Operations Overview

3.1 Intermediate vs Terminal Operations

Understanding the distinction between **intermediate** and **terminal** operations is key to mastering Java Streams. These two types of operations define the lifecycle of a stream pipeline.

Intermediate Operations

- **Lazy evaluation**: These operations **do not process data immediately**.
- **Return a new Stream**: Each call creates a new pipeline stage.
- **Chainable**: Can be combined fluently into a stream pipeline.
- **Examples**:
  - `filter(Predicate<T>)`: Filters elements based on a condition.
  - `map(Function<T,R>)`: Transforms elements.
  - `sorted()`, `distinct()`, `limit(n)`

  Note: No work is done until a terminal operation is invoked.

Example: Intermediate operations only (no output):

```
Stream<String> names = Stream.of("Alice", "Bob", "Charlie")
                              .filter(name -> name.startsWith("A"))
                              .map(String::toUpperCase);
// Nothing happens yet!
```

Terminal Operations

- **Eager evaluation**: Triggers actual processing of the stream pipeline.
- **Consumes the stream**: Stream cannot be reused after a terminal operation.
- **Produces a result or side-effect**:
  - Result: `collect()`, `reduce()`, `count()`
  - Side-effect: `forEach()`, `forEachOrdered()`

Example: Complete pipeline with terminal operation:

```
Stream.of("Alice", "Bob", "Charlie")
      .filter(name -> name.startsWith("A"))
      .map(String::toUpperCase)
      .forEach(System.out::println);
// Output: ALICE
```

Summary Table

Feature	Intermediate Ops	Terminal Ops
Evaluation	Lazy	Eager
Return Type	Stream	Non-stream (or void)
Trigger Execution	NO No	YES Yes
Can Chain	YES Yes	NO No (ends stream)
Examples	`filter`, `map`, `limit`	`forEach`, `collect`, `count`

Understanding how these operations work together enables the construction of powerful, efficient, and readable data-processing pipelines.

3.2 Stream Laziness and Execution

One of the most powerful features of the Java Stream API is **lazy evaluation**. This means that intermediate operations—such as `filter()`, `map()`, and `sorted()`—are not executed immediately when called. Instead, they are **deferred** until a **terminal operation** (like `forEach()`, `collect()`, or `count()`) is invoked. This lazy behavior allows the stream pipeline to optimize execution, short-circuit operations, and avoid unnecessary computation.

Key Concept: Nothing Happens Without a Terminal Operation

Consider this example:

```
Stream<String> names = Stream.of("Alice", "Bob", "Charlie")
                             .filter(name -> {
                                 System.out.println("Filtering: " + name);
                                 return name.startsWith("A");
                             });
// No output yet!
```

Even though `filter()` contains a `println()`, **no output occurs** because no terminal operation has been called.

Tracing Execution with `peek()`

The `peek()` method is useful for debugging and observing how streams process elements. It behaves like `map()`, but without modifying the data—it simply lets you "peek" at each element.

```
Stream.of("apple", "banana", "cherry")
      .filter(s -> {
          System.out.println("Filtering: " + s);
          return s.contains("a");
      })
      .peek(s -> System.out.println("Peeking: " + s))
      .map(String::toUpperCase)
      .forEach(System.out::println);
```

Expected output:

```
Filtering: apple
Peeking: apple
APPLE
Filtering: banana
Peeking: banana
BANANA
Filtering: cherry
CHERRY
```

Notice:

- Operations are performed **one element at a time**, not stage-by-stage.
- The stream evaluates **just enough** to pass data to the terminal operation.
- `cherry` is filtered out early, so `peek()` and `map()` are never invoked for it.

This lazy and per-element evaluation makes streams both efficient and predictable when understood correctly.

3.3 Pipeline Construction and Processing

A **Stream pipeline** is a sequence of operations composed of three parts:

1. **Source** – Where the data comes from (e.g., collections, arrays, files).
2. **Intermediate operations** – Transformations (e.g., `filter()`, `map()`) that are lazy and return a new Stream.
3. **Terminal operation** – The trigger that executes the pipeline and produces a result or side effect (e.g., `collect()`, `forEach()`).

These operations are **chained** together fluently, forming a **pipeline** that is both expressive and efficient. Importantly, the entire pipeline is executed in a **single pass** over the data, meaning each element flows through the full chain of operations before the next one is processed. This design enables short-circuiting and optimization, reducing overhead and memory usage.

Example: Full Pipeline from Source to Terminal

```java
import java.util.List;
import java.util.stream.Collectors;

public class StreamPipelineExample {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Andrew", "Charlie", "Ann");
        List<String> result = names.stream()                      // Source
                                   .filter(name -> name.startsWith("A")) // Intermediate
                                   .map(String::toUpperCase)             // Intermediate
                                   .sorted()                             // Intermediate
                                   .collect(Collectors.toList());        // Terminal

        System.out.println(result); // Output: [ALICE, ANDREW, ANN]
    }
}
```

Benefits of Stream Pipelines

- **Readability**: Each transformation is clearly defined and chained.
- **Efficiency**: The pipeline processes data one element at a time, avoiding unnecessary operations.
- **Declarative style**: Focus on *what* to do, not *how* to do it.
- **Optimization**: Java can optimize execution using short-circuiting (`limit()`, `anyMatch()`, etc.) and lazy evaluation.

Stream pipelines encourage clean, modular, and performant data processing code. By chaining operations, you build expressive workflows that are easy to maintain and understand—an essential practice in modern Java programming.

Chapter 4.

Basic Stream Operations

1. Filtering Elements (`filter()`)
2. Mapping Elements (`map()`)
3. Sorting Elements (`sorted()`)
4. Distinct Elements (`distinct()`)
5. Limiting and Skipping Elements (`limit()`, `skip()`)

4 Basic Stream Operations

4.1 Filtering Elements (`filter()`)

The `filter()` method in Java Streams is used to **select elements** that match a given **predicate**—a condition expressed as a `boolean` test. It is an **intermediate, lazy operation**, meaning it doesn't execute until a terminal operation is called.

Filtering is one of the most common stream operations and is useful for **removing unwanted values**, **ignoring nulls**, or **selecting data that meets specific criteria**.

Syntax

```
Stream<T> filter(Predicate<? super T> predicate)
```

The method returns a new stream consisting of the elements that match the provided predicate.

Example 1: Filter Strings by Condition

Filter names starting with "A":

```
import java.util.List;

public class FilterExample1 {
    public static void main(String[] args) {
        List.of("Alice", "Bob", "Andrew", "Charlie")
            .stream()
            .filter(name -> name.startsWith("A"))
            .forEach(System.out::println);
        // Output: Alice, Andrew
    }
}
```

Example 2: Remove Null or Empty Strings

```
import java.util.List;

public class FilterExample2 {
    public static void main(String[] args) {
        List.of("Java", "", null, "Streams", " ")
            .stream()
            .filter(s -> s != null && !s.trim().isEmpty())
            .forEach(System.out::println);
        // Output: Java, Streams
    }
}
```

Example 3: Filter Numbers Based on Condition

Select even numbers from a list:

```java
import java.util.List;

public class FilterExample3 {
    public static void main(String[] args) {
        List.of(1, 2, 3, 4, 5, 6)
            .stream()
            .filter(n -> n % 2 == 0)
            .forEach(System.out::println);
        // Output: 2, 4, 6
    }
}
```

Laziness of `filter()`

Because `filter()` is lazy, no elements are actually tested until a **terminal operation** like `forEach()` or `collect()` is invoked. This allows efficient and optimized processing, especially when combined with short-circuiting methods like `limit()`.

Filtering is foundational to stream processing and helps write clear, expressive, and functional-style code.

4.2 Mapping Elements (`map()`)

The `map()` operation in Java Streams is used to **transform** each element in a stream into another form. It performs a **one-to-one mapping**, meaning that for every input element, exactly one output element is produced.

This method is essential for data transformation in a pipeline, whether you're converting types, extracting object fields, or applying computations.

Syntax

```java
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

It takes a `Function` that transforms elements from type `T` to type `R`.

Example 1: Mapping Strings to Their Lengths

```java
import java.util.List;

public class MapExample1 {
    public static void main(String[] args) {
        List.of("Java", "Stream", "API")
            .stream()
            .map(String::length)
            .forEach(System.out::println);
        // Output: 4, 6, 3
    }
}
```

Each string is mapped to its integer length.

Example 2: Extracting Object Fields

Suppose you have a list of `Person` objects and want to get their names:

```
import java.util.List;

class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name; this.age = age;
    }
}

public class MapExample2 {
    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Alice", 30),
            new Person("Bob", 25)
        );

        people.stream()
              .map(p -> p.name)
              .forEach(System.out::println);
        // Output: Alice, Bob
    }
}
```

Example 3: Type Conversion and Handling Nulls

Mapping string numbers to integers:

```
import java.util.List;

public class MapExample3 {
    public static void main(String[] args) {
        List.of("1", "2", "three", "4")
            .stream()
            .map(s -> {
                try {
                    return Integer.parseInt(s);
                } catch (NumberFormatException e) {
                    return null;
                }
            })
            .filter(n -> n != null)
            .forEach(System.out::println);
        // Output: 1, 2, 4
    }
}
```

This example highlights a **common pitfall**: returning `null` in a `map()` function. While possible, it must be handled carefully—typically with a `filter()` step to remove nulls.

Key Takeaways

- `map()` is **transformational**, not filtering.
- Always ensure type safety and handle potential `null` values explicitly.
- It's widely used in field extraction, type conversion, and data enrichment.

Mapping is central to making stream pipelines powerful and expressive.

4.3 Sorting Elements (`sorted()`)

The `sorted()` method in Java Streams is used to **order elements** in a stream. It can sort using **natural ordering** (like alphabetical or numerical) or a **custom `Comparator`**. Sorting is a **stateful intermediate operation**, which means it needs to examine the entire stream before it can produce results—this makes it inherently **less lazy** than operations like `map()` or `filter()`.

Syntax

```
Stream<T> sorted();                              // Natural order (Comparable)
Stream<T> sorted(Comparator<? super T> c);    // Custom Comparator
```

4.3.1 Stable Sorting

Java's `sorted()` operation is **stable**, meaning if two elements are considered equal under the sorting criteria, their **original order is preserved**. This is important when chaining multiple sorts or preserving input consistency.

4.3.2 Example 1: Natural Sorting (Strings)

```
import java.util.List;

public class SortedExample1 {
    public static void main(String[] args) {
        List.of("Banana", "Apple", "Cherry")
            .stream()
            .sorted()
            .forEach(System.out::println);
        // Output: Apple, Banana, Cherry
    }
}
```

Here, strings are sorted alphabetically using their natural order (defined by `Comparable`).

4.3.3 Example 2: Sorting Integers with a Comparator

```java
import java.util.List;
import java.util.Comparator;

public class SortedExample2 {
    public static void main(String[] args) {
        List.of(5, 2, 9, 1)
            .stream()
            .sorted(Comparator.reverseOrder())
            .forEach(System.out::println);
        // Output: 9, 5, 2, 1
    }
}
```

This example demonstrates sorting numbers in **descending order** using a custom comparator.

4.3.4 Example 3: Sorting Complex Objects by Multiple Fields

```java
import java.util.List;
import java.util.Comparator;

class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name; this.age = age;
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class SortedExample3 {
    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Alice", 30),
            new Person("Bob", 25),
            new Person("Alice", 22)
        );
        people.stream()
              .sorted(Comparator.comparing((Person p) -> p.name)
                                .thenComparing(p -> p.age))
              .forEach(System.out::println);
        // Output:
        // Alice (22)
        // Alice (30)
        // Bob (25)
    }
}
```

This demonstrates a **stable, multi-level sort**: first by name, then by age.

4.3.5 Summary

- `sorted()` without arguments uses natural order (must implement `Comparable`).
- `sorted(Comparator)` provides full control over ordering.
- Sorting is **stable** and **stateful**, so consider performance on large streams.
- It's a powerful tool when combined with custom classes and complex criteria.

4.4 Distinct Elements (`distinct()`)

The `distinct()` method in Java Streams is used to **remove duplicate elements** from a stream. It retains **only the first occurrence** of each element, as determined by the `equals()` method. This makes `distinct()` an effective way to enforce uniqueness in a stream pipeline.

How It Works

- Uses **`equals()` and `hashCode()`** for comparison.
- Only applicable to **object streams**, not primitive streams like `IntStream`—though similar results can be achieved using `boxed()` and then `distinct()`.

4.4.1 Example 1: Distinct Strings

```
import java.util.List;

public class DistinctExample1 {
    public static void main(String[] args) {
        List.of("apple", "banana", "apple", "cherry", "banana")
            .stream()
```

```java
            .distinct()
            .forEach(System.out::println);
        // Output: apple, banana, cherry
    }
}
```

4.4.2 Example 2: Distinct on Custom Objects (Override `equals()` and `hashCode()`)

```java
import java.util.List;
import java.util.Objects;

class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name; this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person p)) return false;
        return age == p.age && Objects.equals(name, p.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class DistinctExample2 {
    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Alice", 30),
            new Person("Bob", 25),
            new Person("Alice", 30)
        );

        people.stream()
              .distinct()
              .forEach(System.out::println);
        // Output: Alice (30), Bob (25)
    }
}
```

Without properly overriding `equals()` and `hashCode()`, the above example would treat all objects as distinct—even if their contents are identical.

4.4.3 Example 3: Distinct Primitive Values

Use `boxed()` to convert primitive streams to object streams:

```java
import java.util.stream.IntStream;

public class DistinctExample3 {
    public static void main(String[] args) {
        IntStream.of(1, 2, 2, 3, 3, 3)
                 .boxed()
                 .distinct()
                 .forEach(System.out::println);
        // Output: 1, 2, 3
    }
}
```

4.4.4 Performance Considerations

- Internally, `distinct()` uses a `Set` to track duplicates.
- On large streams, this can increase **memory usage** and **processing time**.
- For custom types, ensure `equals()` and `hashCode()` are efficient and correct to avoid subtle bugs or poor performance.

Using `distinct()` effectively allows for clean, deduplicated data streams, especially when handling input from user lists, files, or APIs.

4.5 Limiting and Skipping Elements (`limit()`, `skip()`)

The `limit()` and `skip()` methods in Java Streams provide control over **how many elements are processed** or **where processing starts** in a stream. These operations are particularly useful in scenarios like **pagination**, **data sampling**, or working with **infinite streams**.

`limit(n)`

- Truncates the stream to contain **at most `n` elements**.
- If the stream has fewer than `n` elements, it returns all of them.

`skip(n)`

- Discards the **first `n` elements** from the stream.
- The remaining elements are passed downstream.

Both are **intermediate**, lazy operations that are **evaluated only when a terminal operation is invoked**.

4.5.1 Example 1: Limiting Results After Filtering

```java
import java.util.List;

public class LimitExample {
    public static void main(String[] args) {
        List.of("apple", "apricot", "banana", "avocado", "almond", "blueberry")
            .stream()
            .filter(s -> s.startsWith("a"))
            .limit(3)
            .forEach(System.out::println);
        // Output: apple, apricot, avocado
    }
}
```

Here, we filter for strings starting with `"a"` and take only the first **3** matching results.

4.5.2 Example 2: Pagination with `skip()` and `limit()`

```java
import java.util.List;

public class PaginationExample {
    public static void main(String[] args) {
        List<String> items = List.of("Item1", "Item2", "Item3", "Item4", "Item5", "Item6");

        int page = 2;
        int pageSize = 2;

        items.stream()
             .skip((page - 1) * pageSize)
             .limit(pageSize)
             .forEach(System.out::println);
        // Output (Page 2): Item3, Item4
    }
}
```

This simulates **pagination**, retrieving page 2 of a 2-item-per-page listing.

4.5.3 Example 3: Handling Infinite Streams

`limit()` is essential when working with **infinite streams** to avoid non-terminating execution.

```java
import java.util.stream.Stream;

public class InfiniteStreamExample {
    public static void main(String[] args) {
        Stream.iterate(1, n -> n + 1)
              .filter(n -> n % 2 == 0)
              .skip(3)       // Skip first 3 even numbers: 2, 4, 6
              .limit(5)      // Take next 5: 8, 10, 12, 14, 16
              .forEach(System.out::println);
    }
}
```

4.5.4 Summary

- Use `limit(n)` to **constrain output size**.
- Use `skip(n)` to **ignore early elements**, enabling **offsets**.
- Both are powerful tools for **efficient stream slicing**, especially when combined with filters or infinite sequences.

Chapter 5.

Working with Primitive Streams

1. `IntStream`, `LongStream`, `DoubleStream`
2. Conversion Between Object and Primitive Streams
3. Primitive Stream Operations and Examples

5 Working with Primitive Streams

5.1 IntStream, LongStream, DoubleStream

Java provides specialized stream interfaces—`IntStream`, `LongStream`, and `DoubleStream`—to efficiently handle **primitive data types** without the overhead of boxing and unboxing that occurs when using generic `Stream<T>` with wrapper classes like `Integer` or `Double`.

Why Use Primitive Streams?

- **Avoids boxing overhead:** Operations on primitives are faster and use less memory.
- **Provides specialized methods:** Such as `sum()`, `average()`, `range()`, which are optimized for primitives.
- **Improves performance** when processing large amounts of numeric data.

Key Differences from `StreamT`

Feature	`Stream<T>`	Primitive Streams (`IntStream`, etc.)
Element Type	Object (e.g., `Integer`)	Primitive (`int`, `long`, `double`)
Boxing/Unboxing Overhead	Yes	No
Specialized Methods	General methods	Numeric-specific (e.g., `sum()`, `average()`)
Conversion Methods	`mapToInt()`, `mapToLong()`, etc.	Can convert to/from `Stream<T>`

Creating and Using Primitive Streams

`IntStream` example: sum and average

```java
import java.util.stream.IntStream;

public class IntStreamExample {
    public static void main(String[] args) {
        IntStream numbers = IntStream.of(1, 2, 3, 4, 5);

        int sum = numbers.sum();
        System.out.println("Sum: " + sum); // Output: Sum: 15

        // To reuse the stream, recreate it:
        double average = IntStream.of(1, 2, 3, 4, 5).average().orElse(0);
        System.out.println("Average: " + average); // Output: Average: 3.0
    }
}
```

`LongStream` example: generating ranges

```java
import java.util.stream.LongStream;

public class LongStreamExample {
    public static void main(String[] args) {
        long sum = LongStream.rangeClosed(1, 5).sum();
        System.out.println("Sum of 1 to 5: " + sum); // Output: 15
    }
}
```

`DoubleStream` example: statistics

```java
import java.util.stream.DoubleStream;

public class DoubleStreamExample {
    public static void main(String[] args) {
        DoubleStream doubles = DoubleStream.of(1.5, 2.5, 3.5);

        double max = doubles.max().orElse(Double.NaN);
        System.out.println("Max: " + max); // Output: Max: 3.5
    }
}
```

Performance Benefits

Primitive streams **avoid the overhead of boxing/unboxing** that occurs in generic streams, which can significantly improve performance in numerical computations, especially on large datasets or in tight loops.

Using these specialized streams is a best practice when dealing with primitive data, combining clean code with efficient execution.

5.2 Conversion Between Object and Primitive Streams

Java Streams provide convenient methods to convert between **object streams** (e.g., `Stream<Integer>`) and **primitive streams** (`IntStream`, `LongStream`, `DoubleStream`). This conversion is essential to combine the expressive power of object streams with the performance benefits of primitive streams.

Converting Object Streams to Primitive Streams

- `mapToInt(ToIntFunction<? super T> mapper)`
- `mapToLong(ToLongFunction<? super T> mapper)`
- `mapToDouble(ToDoubleFunction<? super T> mapper)`

These methods apply a mapping function that extracts the primitive value from each object and returns the corresponding primitive stream.

Boxing Primitive Streams Back to Object Streams

- `boxed()` converts a primitive stream back into a `Stream` of wrapper objects, e.g., from `IntStream` to `Stream<Integer>`.
- Useful when you want to apply object-based operations or collect into collections of wrapper types.

5.2.1 Example 1: Object Stream to `IntStream` and Back

```java
import java.util.List;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class ConversionExample1 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4);

        // Convert Stream<Integer> to IntStream
        IntStream intStream = numbers.stream()
                                     .mapToInt(Integer::intValue);

        // Sum of primitive ints
        int sum = intStream.sum();
        System.out.println("Sum: " + sum); // Output: Sum: 10

        // Convert IntStream back to Stream<Integer>
        Stream<Integer> boxedStream = IntStream.range(1, 5).boxed();
        boxedStream.forEach(System.out::println);
    }
}
```

5.2.2 Example 2: Mapping Objects to `DoubleStream`

```java
import java.util.List;

public class ConversionExample2 {
    public static void main(String[] args) {
        List<String> prices = List.of("9.99", "19.95", "5.50");

        // Convert Stream<String> to DoubleStream
        double total = prices.stream()
                             .mapToDouble(Double::parseDouble)
                             .sum();

        System.out.println("Total price: " + total); // Output: Total price: 35.44
    }
}
```

5.2.3 Example 3: Handling Nulls When Mapping to Primitive Streams

```java
import java.util.List;
import java.util.Objects;

public class ConversionExample3 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, null, 3, 4);

        // Map to IntStream carefully, filtering out nulls first
        int sum = numbers.stream()
                         .filter(Objects::nonNull)
                         .mapToInt(Integer::intValue)
                         .sum();

        System.out.println("Sum ignoring nulls: " + sum); // Output: 8
    }
}
```

5.2.4 Summary

- Use `mapToInt()`, `mapToLong()`, and `mapToDouble()` to convert **object streams to primitive streams**.
- Use `boxed()` to convert **primitive streams back to object streams**.
- Handle null values carefully before converting to primitive streams to avoid `NullPointerException`.
- These conversions combine the **efficiency** of primitives with the **flexibility** of object streams for complex operations.

5.3 Primitive Stream Operations and Examples

Primitive streams such as `IntStream`, `LongStream`, and `DoubleStream` provide specialized operations tailored to numeric data processing. These operations simplify common statistical calculations, making the code more concise and efficient by avoiding boxing overhead.

Key Operations

- **`sum()`**: Computes the total of all elements.
- **`average()`**: Calculates the mean as an `OptionalDouble`.
- **`min()` / `max()`**: Finds the smallest or largest element, returned as `OptionalInt`, `OptionalLong`, or `OptionalDouble`.
- **`range(startInclusive, endExclusive)`**: Generates a stream of numbers from a start (inclusive) to an end (exclusive).

5.3.1 Example 1: Sum and Average of IntStream

```java
import java.util.stream.IntStream;

public class PrimitiveOperations1 {
    public static void main(String[] args) {
        IntStream numbers = IntStream.of(10, 20, 30, 40, 50);

        int sum = numbers.sum();
        System.out.println("Sum: " + sum); // Output: Sum: 150

        // Need to recreate stream for average, as streams are single-use
        double average = IntStream.of(10, 20, 30, 40, 50).average().orElse(0);
        System.out.println("Average: " + average); // Output: Average: 30.0
    }
}
```

5.3.2 Example 2: Finding Min and Max in a LongStream

```java
import java.util.stream.LongStream;

public class PrimitiveOperations2 {
    public static void main(String[] args) {
        LongStream values = LongStream.of(100L, 200L, 50L, 400L, 300L);

        long min = values.min().orElseThrow();
        System.out.println("Min value: " + min); // Output: Min value: 50

        // Recreate stream for max
        long max = LongStream.of(100L, 200L, 50L, 400L, 300L).max().orElseThrow();
        System.out.println("Max value: " + max); // Output: Max value: 400
    }
}
```

5.3.3 Example 3: Using `range()` to Generate and Sum a Range of Numbers

```java
import java.util.stream.IntStream;

public class PrimitiveOperations3 {
    public static void main(String[] args) {
        int sumRange = IntStream.range(1, 6)  // 1 to 5 inclusive of 1, exclusive of 6
                                .sum();
        System.out.println("Sum of range 1 to 5: " + sumRange); // Output: 15
    }
}
```

5.3.4 Summary

Primitive stream operations provide a **clean and efficient** way to perform numeric calculations without manual looping or boxing. Methods like `sum()`, `average()`, `min()`, and `max()` are concise and return meaningful results wrapped in optionals when necessary, reducing boilerplate and improving readability. Additionally, `range()` and `rangeClosed()` help quickly generate numeric sequences for processing, enhancing productivity when working with numeric datasets.

Chapter 6.
Collecting Stream Results

6 Collecting Stream Results

6.1 Introduction to Collectors

The `Collector` interface in Java Streams defines a **strategy for reducing** a stream of elements into a **final container or result**. It abstracts how elements are accumulated, combined, and finished to produce a desired output, such as a collection, a summary value, or a concatenated string.

The **`Collectors` utility class** provides a rich set of **predefined collector implementations** that cover common reduction scenarios like collecting to lists, sets, maps, joining strings, or computing summary statistics. This allows developers to write concise, readable code without manually implementing reduction logic.

Mutable vs Immutable Collections

- Most collectors provided by `Collectors` produce **mutable collections**, such as `ArrayList` or `HashSet`, which can be modified after creation.
- This mutability often improves performance during collection since elements can be added incrementally.
- However, when immutability is required, you can create unmodifiable wrappers around the collected result or use third-party libraries.

Simple Example: Collecting to a List

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorIntroExample {
    public static void main(String[] args) {
        List<String> fruits = Stream.of("apple", "banana", "cherry")
                                    .collect(Collectors.toList());

        System.out.println(fruits);
        // Output: [apple, banana, cherry]
    }
}
```

In this example, the stream of fruit names is **collected into a mutable `List`** using the `Collectors.toList()` collector. This pattern is one of the most common ways to transform streams back into collections for further use.

By understanding the `Collector` interface and the power of built-in collectors, you unlock the full potential of stream processing and elegant data transformation in Java.

6.2 Collecting to Lists, Sets, Maps

Java Streams provide powerful built-in collectors to accumulate stream elements into different collection types: **lists**, **sets**, and **maps**. These are available through the `Collectors` utility class as `toList()`, `toSet()`, and `toMap()`.

Collecting to a List: `Collectors.toList()`

- Collects elements into a **`List`**.
- The returned list is usually mutable.
- **Preserves encounter order** when the stream has a defined order (e.g., from a `List`).

Example:

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToListExample {
    public static void main(String[] args) {
        List<String> fruits = Stream.of("apple", "banana", "cherry", "apple")
                                    .collect(Collectors.toList());

        System.out.println(fruits);
        // Output: [apple, banana, cherry, apple]
    }
}
```

Collecting to a Set: `Collectors.toSet()`

- Collects elements into a **`Set`**.
- Removes duplicates based on `equals()`.
- **No guaranteed order** — typically a `HashSet` is returned, so iteration order is unpredictable.

Example:

```java
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToSetExample {
    public static void main(String[] args) {
        Set<String> fruits = Stream.of("apple", "banana", "cherry", "apple")
                                   .collect(Collectors.toSet());

        System.out.println(fruits);
        // Output: [banana, cherry, apple] (order may vary)
    }
}
```

Collecting to a Map: `Collectors.toMap()`

- Creates a **`Map<K, V>`** from stream elements.

- Requires two functions:
  - A **key mapper** to produce keys.
  - A **value mapper** to produce values.
- By default, duplicate keys cause an `IllegalStateException`.
- To handle duplicate keys, provide a **merge function** to resolve conflicts.

Example 1: Simple Map without duplicates

```java
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToMapExample1 {
    public static void main(String[] args) {
        Map<String, Integer> nameLengths = Stream.of("apple", "banana", "cherry")
            .collect(Collectors.toMap(
                s -> s,         // key mapper: string itself
                String::length // value mapper: string length
            ));

        System.out.println(nameLengths);
        // Output: {apple=5, banana=6, cherry=6}
    }
}
```

Example 2: Handling duplicate keys with merge function

```java
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToMapExample2 {
    public static void main(String[] args) {
        Map<Character, String> initialsMap = Stream.of("apple", "banana", "apricot")
            .collect(Collectors.toMap(
                s -> s.charAt(0),  // key: first character
                s -> s,            // value: string itself
                (existing, replacement) -> existing + ", " + replacement // merge duplicates
            ));

        System.out.println(initialsMap);
        // Output: {a=apple, apricot, b=banana}
    }
}
```

6.2.1 Summary

- `toList()` preserves order and allows duplicates.
- `toSet()` removes duplicates but does not guarantee order.

- `toMap()` requires key/value mappers and needs careful handling of duplicate keys via merge functions.
- Understanding these collectors helps transform streams flexibly into standard Java collections for further processing.

6.3 Joining Strings

The `Collectors.joining()` method is a convenient way to **concatenate strings** from a stream into a single result. It allows you to specify optional **delimiters**, **prefixes**, and **suffixes**, making it ideal for producing readable and well-formatted string outputs.

Basic Usage

- **`Collectors.joining()`** — concatenates elements directly.
- **`Collectors.joining(delimiter)`** — inserts a delimiter between elements.
- **`Collectors.joining(delimiter, prefix, suffix)`** — adds delimiter, prefix, and suffix around the concatenated string.

6.3.1 Example 1: Simple Joining with Comma Delimiter

```java
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class JoiningExample1 {
    public static void main(String[] args) {
        String result = Stream.of("apple", "banana", "cherry")
                              .collect(Collectors.joining(", "));

        System.out.println(result);
        // Output: apple, banana, cherry
    }
}
```

This example joins stream elements into a **comma-separated list**, commonly used for display or CSV formatting.

6.3.2 Example 2: Joining with Prefix, Suffix, and Delimiter

```java
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

```java
public class JoiningExample2 {
    public static void main(String[] args) {
        String result = Stream.of("Java", "Python", "C++")
                              .collect(Collectors.joining(" | ", "[Languages: ", "]"));

        System.out.println(result);
        // Output: [Languages: Java | Python | C++]
    }
}
```

Here, the joining operation produces a **formatted string** with a prefix and suffix, and uses `" | "` as the delimiter, making the output clearer and more descriptive.

6.3.3 Summary

`Collectors.joining()` is a simple yet powerful tool to convert streams of strings into a **single formatted string**. Its flexibility with delimiters, prefixes, and suffixes lets you customize output for user interfaces, logs, reports, or file exports with minimal code.

6.4 Summary Statistics Collectors

Java Streams provide convenient built-in collectors such as `Collectors.summarizingInt()`, `summarizingDouble()`, and `summarizingLong()` that produce **summary statistics objects**. These objects encapsulate useful aggregate information about numeric data streams, enabling concise and efficient statistical calculations.

What Do These Collectors Provide?

The summary statistics objects (`IntSummaryStatistics`, `DoubleSummaryStatistics`, and `LongSummaryStatistics`) contain the following fields:

- **count** — number of elements processed
- **sum** — total sum of the elements
- **min** — smallest element value
- **max** — largest element value
- **average** — mean value (as a `double`)

These stats allow quick insights into the dataset without manually writing loops or multiple operations.

6.4.1 Example 1: Using `summarizingInt()` to Analyze Ages

```java
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;

public class SummaryStatsExample1 {
    public static void main(String[] args) {
        List<Integer> ages = List.of(25, 30, 22, 40, 28);

        IntSummaryStatistics stats = ages.stream()
                                         .collect(Collectors.summarizingInt(Integer::intValue));

        System.out.println("Count: " + stats.getCount());
        System.out.println("Sum: " + stats.getSum());
        System.out.println("Min: " + stats.getMin());
        System.out.println("Max: " + stats.getMax());
        System.out.println("Average: " + stats.getAverage());
    }
}
```

Output:

```
Count: 5
Sum: 145
Min: 22
Max: 40
Average: 29.0
```

6.4.2 Example 2: Using `summarizingDouble()` to Summarize Product Prices

```java
import java.util.DoubleSummaryStatistics;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class SummaryStatsExample2 {
    public static void main(String[] args) {
        DoubleSummaryStatistics priceStats = Stream.of(19.99, 9.99, 4.95, 29.99)
                                                    .collect(Collectors.summarizingDouble(Double::double

        System.out.println("Count: " + priceStats.getCount());
        System.out.println("Sum: " + priceStats.getSum());
        System.out.println("Min: " + priceStats.getMin());
        System.out.println("Max: " + priceStats.getMax());
        System.out.println("Average: " + priceStats.getAverage());
    }
}
```

Output:

```
Count: 4
Sum: 64.92
Min: 4.95
Max: 29.99
Average: 16.23
```

6.4.3 Summary

Using summary statistics collectors dramatically simplifies the process of computing key numeric metrics. They provide a single, immutable object encapsulating multiple statistics, improving code clarity and reducing boilerplate when analyzing datasets in Java Streams.

Chapter 7.

Reducing and Aggregating Data

1. Using `reduce()` for Aggregation
2. Common Reduction Patterns (sum, max, min)
3. Custom Reduction Operations

7 Reducing and Aggregating Data

7.1 Using `reduce()` for Aggregation

The `reduce()` method in Java Streams is a **powerful terminal operation** that combines stream elements into a single result by repeatedly applying an operation. It's often used for aggregation tasks like summing numbers, concatenating strings, or computing product.

Three Variations of `reduce()`

1. **`Optional<T> reduce(BinaryOperator<T> accumulator)`**
   - Reduces elements using the accumulator function.
   - Returns an `Optional` because the stream might be empty.
   - Example: Summing integers without an initial value.
2. **`T reduce(T identity, BinaryOperator<T> accumulator)`**
   - Uses an **identity value** as the initial result.
   - Returns the reduced value directly.
   - The identity is a neutral element for the accumulator (e.g., 0 for addition).
3. **`<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`**
   - Supports **parallel execution** by separating accumulation and combining.
   - `accumulator` processes each element into the result type.
   - `combiner` merges partial results.
   - Useful for complex reductions or when result type differs from stream element type.

Key Concepts

- **Identity**: A value that does not change the result of the operation (e.g., 0 for sum).
- **Accumulator**: A function that takes the partial result and the next element to produce a new partial result.
- **Combiner**: In parallel streams, merges two partial results into one.

7.1.1 Examples

Example 1: String Concatenation with reduce()

```java
import java.util.Optional;
import java.util.stream.Stream;

public class ReduceExample {
    public static void main(String[] args) {
        Optional<String> result = Stream.of("Java", "Streams", "Reduce")
                                        .reduce((a, b) -> a + " " + b);

        result.ifPresent(System.out::println);  // Output: Java Streams Reduce
    }
}
```

Example 2: Numerical Sum with Identity and Method Reference

```
import java.util.stream.Stream;

public class ReduceExample2 {
    public static void main(String[] args) {
        int sum = Stream.of(1, 2, 3, 4, 5)
                        .reduce(0, Integer::sum);

        System.out.println("Sum: " + sum); // Output: Sum: 15
    }
}
```

Example 3: Using Three-Parameter reduce() for Parallel Processing

```
import java.util.Arrays;
import java.util.List;

public class ReduceExample3 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "cherry");

        String result = words.parallelStream()
                             .reduce(
                                 "",                            // identity
                                 (partial, word) -> partial + word.toUpperCase(), // accumulator
                                 (s1, s2) -> s1 + s2          // combiner
                             );

        System.out.println(result); // Output: APPLEBANANACHERRY
    }
}
```

7.1.2 When to Prefer `reduce()` Over `collect()`

- Use **`reduce()`** for **immutable reduction** when the result is a single value and you don't need mutable accumulation.
- Use **`collect()`** when you want to accumulate elements into a **mutable container** (like a list or map), or need more flexible reduction logic.

By mastering `reduce()`, you gain fine-grained control over aggregation, enabling concise yet expressive data summarization in your stream pipelines.

7.2 Common Reduction Patterns (sum, max, min)

Reduction operations like calculating the **sum**, **maximum**, **minimum**, or **product** are some of the most frequent tasks in stream processing. Java Streams allow you to perform these aggregations using the flexible `reduce()` method, but also provide specialized terminal operations for primitive streams that offer better readability and performance.

Using `reduce()` for Common Aggregations

Sum using `reduce()`

```java
import java.util.stream.Stream;

public class SumReduce {
    public static void main(String[] args) {
        int sum = Stream.of(1, 2, 3, 4, 5)
                        .reduce(0, Integer::sum);

        System.out.println("Sum: " + sum);  // Output: Sum: 15
    }
}
```

This example sums integers with an identity value of 0 and the built-in method reference `Integer::sum`.

Maximum using `reduce()`

```java
import java.util.Optional;
import java.util.stream.Stream;

public class MaxReduce {
    public static void main(String[] args) {
        Optional<Integer> max = Stream.of(3, 7, 2, 9, 5)
                                      .reduce(Integer::max);

        max.ifPresent(m -> System.out.println("Max: " + m));  // Output: Max: 9
    }
}
```

Here, no identity is provided, so the result is wrapped in an `Optional`.

Minimum using `reduce()`

```java
import java.util.Optional;
import java.util.stream.Stream;

public class MinReduce {
    public static void main(String[] args) {
        Optional<Integer> min = Stream.of(3, 7, 2, 9, 5)
                                      .reduce(Integer::min);

        min.ifPresent(m -> System.out.println("Min: " + m));  // Output: Min: 2
    }
}
```

Product using `reduce()`

```java
import java.util.stream.Stream;

public class ProductReduce {
    public static void main(String[] args) {
        int product = Stream.of(1, 2, 3, 4)
                            .reduce(1, (a, b) -> a * b);

        System.out.println("Product: " + product);  // Output: Product: 24
    }
}
```

Product does not have a built-in shortcut method, so `reduce()` is ideal here.

Comparing with Primitive Stream Terminal Operations

For primitive streams like `IntStream`, Java provides direct methods:

- `sum()`
- `max()` (returns `OptionalInt`)
- `min()` (returns `OptionalInt`)

Example:

```java
import java.util.stream.IntStream;

public class PrimitiveSumMaxMin {
    public static void main(String[] args) {
        int sum = IntStream.of(1, 2, 3, 4, 5).sum();
        int max = IntStream.of(3, 7, 2, 9, 5).max().orElseThrow();
        int min = IntStream.of(3, 7, 2, 9, 5).min().orElseThrow();

        System.out.printf("Sum: %d, Max: %d, Min: %d%n", sum, max, min);
        // Output: Sum: 15, Max: 9, Min: 2
    }
}
```

7.2.1 Readability, Performance, and Precision

- **Readability:** Using built-in methods like `sum()`, `max()`, and `min()` for primitives is more concise and expressive.
- **Performance:** Primitive stream methods avoid boxing/unboxing overhead compared to object streams using `reduce()`.
- **Precision:** For numeric types like `double`, be mindful of floating-point precision in custom reductions.

7.2.2 Summary

- Use `reduce()` for **custom operations** like product or when working with objects.
- Prefer built-in **primitive terminal operations** for standard numeric reductions due to better clarity and efficiency.
- Understanding both approaches helps you choose the right tool depending on your data and requirements.

7.3 Custom Reduction Operations

Beyond simple numeric reductions, the `reduce()` method enables **complex aggregations** involving custom objects and multi-step logic. This is particularly useful when you want to combine multiple fields or accumulate rich summaries from a stream of data.

Reduction in Parallel Streams: Key Requirements

- The **accumulator** and **combiner** functions must be **associative**, meaning the order of operations doesn't change the result.
- The operations should be **stateless** and free of side effects to work correctly with parallel streams.
- The **combiner** merges partial results produced by parallel subtasks, ensuring thread-safe, correct aggregation.

7.3.1 Example: Aggregating User Records into a Summary Report

Suppose we have a `User` class and want to produce a summary containing the total count of users, the combined length of all usernames, and a concatenated list of unique domains in their emails.

```java
import java.util.*;
import java.util.stream.*;

class User {
    String username;
    String email;

    User(String username, String email) {
        this.username = username;
        this.email = email;
    }
}

class UserSummary {
    int userCount;
    int totalNameLength;
    Set<String> emailDomains;

    UserSummary() {
        this.userCount = 0;
        this.totalNameLength = 0;
        this.emailDomains = new HashSet<>();
    }

    UserSummary accumulate(User user) {
        userCount++;
        totalNameLength += user.username.length();

        String domain = user.email.substring(user.email.indexOf('@') + 1);
        emailDomains.add(domain);

        return this;
    }

    UserSummary combine(UserSummary other) {
        userCount += other.userCount;
        totalNameLength += other.totalNameLength;
        emailDomains.addAll(other.emailDomains);
        return this;
    }

    @Override
    public String toString() {
        return String.format("UserCount: %d, TotalNameLength: %d, Domains: %s",
                             userCount, totalNameLength, emailDomains);
    }
}

public class CustomReduceExample {
    public static void main(String[] args) {
        List<User> users = List.of(
            new User("alice", "alice@example.com"),
            new User("bob", "bob@openai.com"),
            new User("charlie", "charlie@example.com")
        );

        UserSummary summary = users.parallelStream()
                                   .reduce(
                                       new UserSummary(),        // identity
                                       UserSummary::accumulate,  // accumulator
                                       UserSummary::combine      // combiner
                                   );

        System.out.println(summary);
    }
}
```

7.3.2 Step-by-Step Walkthrough

- **Identity:** Starts with a fresh, empty `UserSummary` instance.

- **Accumulator:** For each `User`, increments count, adds username length, and extracts email domain.
- **Combiner:** When running in parallel, merges two partial summaries by summing counts, lengths, and combining domains.

7.3.3 Why Use This Approach?

- **Flexibility:** Custom objects allow accumulating complex, multi-field results.
- **Parallel Safety:** Proper combiner function ensures correct results even with parallel streams.
- **Efficiency:** Avoids creating many intermediate objects by mutating the accumulator.

7.3.4 Summary

Custom reduction with `reduce()` empowers you to aggregate sophisticated data structures by clearly defining how to accumulate and combine partial results. When designing these operations, always ensure **associativity** and **statelessness** to maintain correctness and maximize parallel processing benefits.

Chapter 8.

Optional and Streams

8 Optional and Streams

8.1 Using `Optional` in Stream Pipelines

In Java Streams, many terminal operations such as `findFirst()`, `findAny()`, `min()`, and `max()` return an `Optional<T>`. This design reflects the fact that these operations may not find any matching element, so instead of returning `null`, they wrap the result in an `Optional`. This helps **avoid null-related errors** and encourages safer, more expressive code.

Why Use `Optional`?

- **Explicit absence**: `Optional` clearly signals that a value may or may not be present.
- **No null checks needed**: Methods like `ifPresent()`, `orElse()`, and `map()` enable elegant handling of possible absence.
- **Functional style**: Integrates smoothly with stream pipelines and lambdas.

8.1.1 Common Optional Methods in Stream Context

- **`ifPresent(Consumer<? super T> action)`** — executes the action only if a value is present.
- **`orElse(T other)`** — returns the value if present; otherwise returns the provided default.
- **`map(Function<? super T,? extends U> mapper)`** — transforms the contained value if present.

8.1.2 Example 1: Using `findFirst()` with `ifPresent()`

```java
import java.util.stream.Stream;

public class OptionalExample1 {
    public static void main(String[] args) {
        Stream<String> fruits = Stream.of("apple", "banana", "cherry");

        fruits.filter(f -> f.startsWith("b"))
              .findFirst()
              .ifPresent(fruit -> System.out.println("Found: " + fruit));
        // Output: Found: banana
    }
}
```

Here, `findFirst()` returns an `Optional<String>`. If a fruit starting with “b” exists, the lambda inside `ifPresent()` runs.

8.1.3 Example 2: Using `min()` with `orElse()`

```java
import java.util.stream.Stream;

public class OptionalExample2 {
    public static void main(String[] args) {
        Stream<Integer> numbers = Stream.of(5, 8, 2, 10);

        int min = numbers.min(Integer::compareTo)
                         .orElse(-1);  // default if stream empty

        System.out.println("Min: " + min);
        // Output: Min: 2
    }
}
```

If the stream is empty, `orElse(-1)` provides a safe fallback.

8.1.4 Example 3: Using `map()` to Transform an Optional Result

```java
import java.util.stream.Stream;

public class OptionalExample3 {
    public static void main(String[] args) {
        Stream<String> words = Stream.of("hello", "world");

        int length = words.filter(w -> w.contains("o"))
                          .findAny()
                          .map(String::length)
                          .orElse(0);

        System.out.println("Length: " + length);
        // Output: Length: 5
    }
}
```

Here, if a word containing “o” is found, its length is computed; otherwise, the default length 0 is returned.

8.1.5 Summary

Using `Optional` in stream pipelines makes it easier and safer to handle potentially missing results from terminal operations. Instead of dealing with `null` values, you work with a rich API that encourages expressive and null-safe code. This enhances readability and robustness in stream processing.

8.2 Handling Absent Values

When working with streams, it's common to encounter **missing or absent values**—whether from filtering, searching, or external data sources. Java's `Optional` provides a clean and safe way to handle these cases, avoiding null pointer exceptions and enabling clear fallback strategies.

Using `Optional.empty()`

`Optional.empty()` represents the explicit absence of a value. It's often returned when no matching element is found in a stream (e.g., after a filter or a search operation).

Filtering Out Nulls or Unwanted Values

Sometimes your source data may contain `null` elements or invalid values. You can safely remove these using a `filter()` before further processing:

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class NullFilteringExample {
    public static void main(String[] args) {
        List<String> data = List.of("apple", null, "banana", null, "cherry");

        List<String> filtered = data.stream()
                                    .filter(s -> s != null)  // filter out nulls
                                    .collect(Collectors.toList());

        System.out.println(filtered);
        // Output: [apple, banana, cherry]
    }
}
```

Default Values with `orElse()` and `orElseGet()`

When an `Optional` might be empty, you can specify a default value:

- **`orElse(value)`** — returns the default value if empty (eager evaluation).
- **`orElseGet(supplier)`** — returns the default value from a supplier function (lazy evaluation).

Example:

```java
import java.util.Optional;
import java.util.stream.Stream;

public class OptionalFallbackExample {
    public static void main(String[] args) {
        Optional<String> found = Stream.<String>empty()
                                       .findAny();

        String result = found.orElse("Default Value");
        System.out.println(result);  // Output: Default Value
        // Lazy fallback example
        String lazyResult = found.orElseGet(() -> expensiveFallback());
        System.out.println(lazyResult);  // Output: Expensive fallback value
    }

    private static String expensiveFallback() {
        System.out.println("Computing fallback...");
        return "Expensive fallback value";
    }
}
```

Note: `orElseGet()` is preferred when the fallback value is costly to compute.

When to Use Fallbacks vs Propagating Absence

- Use **fallbacks** (`orElse()`, `orElseGet()`) when a sensible default improves robustness or user experience.
- Propagate absence (e.g., returning an empty `Optional`) when the caller needs to explicitly handle missing data or when the absence signals an error or exceptional case.

8.2.1 Summary

Handling absent values safely in streams is essential for robust applications. By filtering out `null`s early, using `Optional.empty()` to represent missing data, and employing `orElse()` or `orElseGet()` for fallbacks, you can write clean, null-safe, and expressive stream pipelines that gracefully handle missing elements.

8.3 Combining Streams and Optionals

In some scenarios, stream operations produce a **stream of `Optional` values** (`Stream<Optional<T>>`), especially when intermediate steps might or might not yield results. Handling these nested optionals effectively requires flattening them to work with the actual contained values, and Java Streams provide a powerful way to do this using `flatMap()`.

Why Flatten a `StreamOptionalT`?

A `Stream<Optional<T>>` means each element is wrapped in an `Optional`. To process only the **present** values inside these optionals as a normal stream, you need to **unwrap** and flatten them into a `Stream<T>`. This avoids having to deal with nested optionals and makes downstream processing simpler.

8.3.1 Visual Explanation of Flattening

Imagine:

```
Stream<Optional<T>>:     [Optional[A], Optional.empty(), Optional[B]]
Flattened to Stream<T>: [A, B]
```

Using `flatMap()` with `Optional::stream` converts each `Optional<T>` to either a **single-element stream** (if present) or an **empty stream** (if absent), effectively removing empty optionals from the pipeline.

8.3.2 Example 1: Flattening Optional Stream of Strings

```java
import java.util.Optional;
import java.util.stream.Stream;
import java.util.List;

public class FlattenOptionalExample {
    public static void main(String[] args) {
        Stream<Optional<String>> optionalStream = Stream.of(
            Optional.of("apple"),
            Optional.empty(),
            Optional.of("banana"),
            Optional.empty(),
            Optional.of("cherry")
        );

        List<String> fruits = optionalStream
            .flatMap(Optional::stream)  // flatten optionals
            .toList();

        System.out.println(fruits);
        // Output: [apple, banana, cherry]
    }
}
```

8.3.3 Example 2: Mapping Elements to Optional and Filtering Present Results

Suppose you map strings to integers but some might fail (returning empty):

```java
import java.util.List;
import java.util.Optional;
import java.util.stream.Stream;

public class OptionalMappingExample {
    public static void main(String[] args) {
        List<String> inputs = List.of("1", "two", "3", "four", "5");

        List<Integer> numbers = inputs.stream()
            .map(OptionalMappingExample::parseIntOptional)  // returns Optional<Integer>
            .flatMap(Optional::stream)                       // flatten present integers
            .toList();

        System.out.println(numbers);
        // Output: [1, 3, 5]
    }

    private static Optional<Integer> parseIntOptional(String s) {
        try {
            return Optional.of(Integer.parseInt(s));
        } catch (NumberFormatException e) {
            return Optional.empty();
        }
    }
}
```

Here, only the successfully parsed integers are collected, while invalid inputs are filtered out safely.

8.3.4 Example 3: Combining Nested Optionals in a Stream Pipeline

You can chain multiple steps returning optionals and flatten at the end:

```java
import java.util.Optional;
import java.util.stream.Stream;

public class NestedOptionalExample {
    public static void main(String[] args) {
        Stream<String> data = Stream.of("  42  ", "  ", "100", "abc");

        var results = data
            .map(String::trim)
            .map(NestedOptionalExample::parseIntOptional)
            .flatMap(Optional::stream)
            .map(i -> i * 2)
            .toList();

        System.out.println(results);
        // Output: [84, 200]
    }

    private static Optional<Integer> parseIntOptional(String s) {
        if (s.isEmpty()) return Optional.empty();
        try {
            return Optional.of(Integer.parseInt(s));
        } catch (NumberFormatException e) {
            return Optional.empty();
        }
    }
}
```

8.3.5 Summary

When working with `Stream<Optional<T>>`, use `flatMap(Optional::stream)` to **flatten** the stream and work only with present values. This pattern helps you elegantly handle optional values scattered through your stream pipeline, simplifying aggregation and downstream processing without null checks or complicated conditionals.

Chapter 9.

Parallel Streams Basics

9 Parallel Streams Basics

9.1 Introduction to Parallel Streams

Parallel streams provide a **simple and powerful way to perform concurrent data processing** in Java. By splitting a stream's workload across multiple threads, parallel streams can leverage multi-core processors to improve performance for large data sets or computationally intensive tasks — all without explicit thread management.

How to Create Parallel Streams

You can create a parallel stream in two ways:

- By calling **`parallelStream()`** on a collection:

  ```
  List<String> data = List.of("a", "b", "c");
  data.parallelStream()...;
  ```

- By converting an existing sequential stream using **`.parallel()`**:

  ```
  Stream<String> stream = data.stream();
  stream.parallel()...;
  ```

Internally, the Java runtime uses the **common ForkJoinPool** to divide the stream elements into smaller chunks, processing them in parallel threads and combining results efficiently.

Benefits of Parallel Streams

- **Improved performance:** Tasks that can be broken into independent units often run faster by utilizing multiple CPU cores.
- **Simple concurrency:** Parallelism is achieved with minimal code changes—just switch from `stream()` to `parallelStream()` or `.parallel()`.
- **Better CPU utilization:** Especially effective for CPU-bound operations on large collections or data sources.

Example: Sequential vs Parallel Processing

The following example demonstrates summing the squares of numbers sequentially and in parallel, measuring the time taken for each.

```
import java.util.List;
import java.util.stream.IntStream;

public class ParallelStreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = IntStream.rangeClosed(1, 10_000_000).boxed().toList();

        long startSeq = System.currentTimeMillis();
        long sumSeq = numbers.stream()
                            .mapToLong(n -> n * n)
                            .sum();
        long durationSeq = System.currentTimeMillis() - startSeq;
        System.out.println("Sequential sum: " + sumSeq + ", Time: " + durationSeq + " ms");
```

```
        long startPar = System.currentTimeMillis();
        long sumPar = numbers.parallelStream()
                        .mapToLong(n -> n * n)
                        .sum();
        long durationPar = System.currentTimeMillis() - startPar;
        System.out.println("Parallel sum: " + sumPar + ", Time: " + durationPar + " ms");
    }
}
```

In this example, the parallel stream often completes faster by dividing the workload, but the actual speedup depends on hardware, data size, and task complexity.

9.1.1 Summary

Parallel streams allow you to harness multicore CPUs for faster data processing with minimal coding effort. By understanding how to switch between sequential and parallel streams, you can optimize your applications for performance while maintaining readable, declarative code.

9.2 When to Use Parallel Streams

Parallel streams can significantly speed up data processing by distributing work across multiple threads. However, **not all tasks benefit from parallelization**, so understanding when to use parallel streams is essential for writing efficient and correct code.

Guidelines for Using Parallel Streams

1. **Large Data Sets** Parallel streams shine when processing large collections or data sources. Small or trivial data sets often incur more overhead in thread management than the performance gains they bring.
2. **CPU-Bound Operations** If the operation on each element requires significant computation (e.g., complex calculations, transformations), parallel processing can reduce overall time by leveraging multiple cores.
3. **Independence of Elements** Operations should be **stateless and independent** per element, meaning the processing of one item does not depend on or affect others. This avoids race conditions and synchronization bottlenecks.
4. **Cost Per Element** The work done per element must be sufficiently expensive to justify the overhead of parallel execution. Simple, fast operations like incrementing integers or light filtering may not benefit.

When to Avoid Parallel Streams

- **I/O-bound tasks:** Tasks blocked by I/O (file, network) might not gain much from parallel streams.
- **Small or ordered data:** Parallelizing small streams or streams with strict ordering requirements might degrade performance.
- **Shared mutable state:** If the operation involves shared mutable objects, thread safety becomes complex.

Comparison to Other Concurrency Tools

While parallel streams abstract away thread management, traditional frameworks like **ForkJoinPool** and **ExecutorService** provide finer control over thread creation, task scheduling, and exception handling. Parallel streams are easier to use but less flexible.

- Use **parallel streams** for straightforward data-parallel computations.
- Use **ForkJoinPool** or **ExecutorService** when you need custom thread pools, more control, or complex concurrency patterns.

9.2.1 Example: When Parallel Streams Help

```java
import java.util.List;
import java.util.stream.IntStream;

public class ParallelStreamGuidelineExample {
    public static void main(String[] args) {
        List<Integer> largeNumbers = IntStream.rangeClosed(1, 1_000_000).boxed().toList();

        // CPU-intensive operation: compute sum of squares
        long sumParallel = largeNumbers.parallelStream()
                                       .mapToLong(n -> heavyComputation(n))
                                       .sum();

        System.out.println("Sum with parallel stream: " + sumParallel);
    }

    private static long heavyComputation(int n) {
        // Simulate expensive operation
        long result = 0;
        for (int i = 0; i < 1000; i++) {
            result += Math.sqrt(n * i);
        }
        return result;
    }
}
```

9.2.2 Example: When Not to Use Parallel Streams

```java
import java.util.List;

public class SmallDatasetExample {
    public static void main(String[] args) {
        List<String> smallList = List.of("a", "b", "c");

        // Parallel overhead may outweigh benefits here
        long count = smallList.parallelStream()
                              .filter(s -> s.startsWith("a"))
                              .count();

        System.out.println("Count: " + count);
    }
}
```

9.2.3 Summary

Parallel streams are best suited for **large, CPU-bound, independent data processing tasks** with non-trivial per-element costs. For smaller, I/O-bound, or stateful operations, sequential streams or other concurrency mechanisms may be more appropriate. Understanding these trade-offs helps you choose the right tool for efficient and safe parallelism.

9.3 Example: Parallel vs Sequential Performance

To understand the real benefits of parallel streams, it helps to compare their performance with sequential streams on a sufficiently large dataset. This example processes a large list of numbers, applying a moderately expensive transformation, and measures the execution time of both approaches.

Setup: Squaring Numbers with a Simulated Workload

We'll square numbers from 1 to 10 million, simulating some CPU work with a small delay, then compare sequential and parallel execution times.

```java
import java.util.List;
import java.util.stream.IntStream;

public class ParallelVsSequentialPerformance {
    public static void main(String[] args) {
        List<Integer> numbers = IntStream.rangeClosed(1, 10_000_000).boxed().toList();

        // Sequential processing
        long startSeq = System.currentTimeMillis();
        long sumSeq = numbers.stream()
                           .mapToLong(ParallelVsSequentialPerformance::simulateWork)
                           .sum();
        long durationSeq = System.currentTimeMillis() - startSeq;
        System.out.println("Sequential sum: " + sumSeq + ", Time: " + durationSeq + " ms");

        // Parallel processing
        long startPar = System.currentTimeMillis();
        long sumPar = numbers.parallelStream()
                           .mapToLong(ParallelVsSequentialPerformance::simulateWork)
                           .sum();
        long durationPar = System.currentTimeMillis() - startPar;
        System.out.println("Parallel sum: " + sumPar + ", Time: " + durationPar + " ms");
    }

    private static long simulateWork(int n) {
        // Simulate moderate CPU workload
        double result = 0;
        for (int i = 0; i < 50; i++) {
            result += Math.sqrt(n + i);
        }
        return (long) result;
    }
}
```

Sample Output

```
Sequential sum: 670962544, Time: 2300 ms
Parallel sum: 670962544, Time: 800 ms
```

9.3.1 Analysis

- The **parallel stream** typically runs faster, using multiple CPU cores to divide the workload.
- The **sequential stream** processes each element one by one on a single thread.
- The speedup depends on:
  - **Number of CPU cores:** More cores mean more parallelism.
  - **Task complexity:** Heavier computations per element justify parallel overhead.
  - **Data size:** Smaller datasets may see no benefit or even slower execution due to overhead.

9.3.2 When Parallelism May Hurt Performance

- Overhead of thread management and task splitting can outweigh benefits for:

  - Small datasets.
  - Simple, fast operations.
  - Streams requiring strict ordering.

- I/O-bound operations may not improve since bottleneck is external resource, not CPU.

9.3.3 Summary

This example highlights how parallel streams can significantly reduce processing time for CPU-intensive, large-scale tasks. Benchmarking with your own data and operations is essential to decide if parallel streams are appropriate for your use case.

9.4 Pitfalls and Thread Safety

While parallel streams simplify concurrent data processing, they also introduce **risks related to thread safety and correctness**. Understanding these pitfalls is crucial to avoid subtle bugs and unpredictable behavior.

Common Pitfalls with Parallel Streams

1. **Non-Associative Operations** Operations like `reduce()` require the accumulator to be **associative** and **stateless**. Non-associative or stateful reductions can produce incorrect results or exceptions in parallel execution.

2. **Shared Mutable State** Modifying shared objects (like collections or counters) inside stream operations—especially intermediate or terminal ones—can cause race conditions and data corruption when streams run in parallel.

3. **Side-Effects in Intermediate Operations** Intermediate operations like `map()`, `filter()`, or `peek()` should be **stateless** and without side effects, as they might be invoked multiple times or out of order in parallel.

9.4.1 Example 1: Unsafe Shared Mutable State

This example demonstrates a common mistake—updating a non-thread-safe collection inside a parallel stream, leading to unpredictable output:

```java
import java.util.ArrayList;
import java.util.List;

public class UnsafeParallelExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);
        List<Integer> results = new ArrayList<>();

        // Parallel stream modifying shared list - NOT thread-safe!
        numbers.parallelStream()
               .map(n -> n * 2)
               .forEach(results::add);

        System.out.println("Results: " + results);
    }
}
```

Problem: `ArrayList` is not thread-safe. Concurrent `add()` calls cause data races and may miss elements or throw exceptions.

9.4.2 Example 2: Safe Alternative Using Thread-Safe Collector

Use built-in thread-safe collectors like `Collectors.toList()` to safely gather results in parallel streams:

```java
import java.util.List;
import java.util.stream.Collectors;

public class SafeParallelExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        List<Integer> results = numbers.parallelStream()
                                       .map(n -> n * 2)
                                       .collect(Collectors.toList()); // thread-safe

        System.out.println("Results: " + results);
    }
}
```

Here, `collect()` handles thread-safe accumulation internally, avoiding shared mutable state issues.

9.4.3 Additional Tips for Thread Safety

- Prefer **stateless** operations without side effects.
- Avoid modifying external state inside streams.
- Use **immutable data structures** or **thread-safe collectors**.
- For custom reduction, ensure accumulator and combiner are associative and stateless.

9.4.4 Summary

Parallel streams can introduce concurrency bugs when shared mutable state or non-associative operations are involved. Avoid side-effects in stream pipelines and rely on thread-safe collectors and stateless functions to ensure correct, predictable parallel processing.

Chapter 10.

Advanced Mapping Techniques

1. FlatMapping with `flatMap()`
2. Mapping to Multiple Collections
3. Complex Transformations

10 Advanced Mapping Techniques

10.1 FlatMapping with `flatMap()`

The `flatMap()` operation is a powerful tool in the Java Streams API used to **flatten nested structures** and combine multiple streams into a single continuous stream. While `map()` transforms each element into exactly one output element (one-to-one mapping), `flatMap()` transforms each element into **zero or more elements** (one-to-many mapping) and then flattens the results into a single stream.

How `flatMap()` Differs from `map()`

- `map()` takes a function that returns a single element and produces a stream of those elements.
- `flatMap()` takes a function that returns a **stream or collection** for each element, then concatenates (flattens) all those inner streams into one flat stream.

This is useful when you deal with nested collections or want to transform each element into multiple elements.

10.1.1 Example 1: Flattening a List of Lists

Suppose you have a list of lists and want to process all inner elements as a single stream.

```java
import java.util.List;

public class FlatMapExample1 {
    public static void main(String[] args) {
        List<List<String>> listOfLists = List.of(
            List.of("apple", "banana"),
            List.of("cherry", "date"),
            List.of("elderberry")
        );

        System.out.println("Before flatMap (listOfLists): " + listOfLists);

        List<String> flattened = listOfLists.stream()
                                            .flatMap(List::stream)  // flatten inner lists
                                            .toList();

        System.out.println("After flatMap (flattened): " + flattened);
    }
}
```

Output:

```
Before flatMap (listOfLists): [[apple, banana], [cherry, date], [elderberry]]
After flatMap (flattened): [apple, banana, cherry, date, elderberry]
```

10.1.2 Example 2: Splitting Strings into Words

Transform a stream of sentences into a stream of words by splitting each sentence and flattening the resulting arrays.

```java
import java.util.List;
import java.util.Arrays;

public class FlatMapExample2 {
    public static void main(String[] args) {
        List<String> sentences = List.of(
            "Java streams are powerful",
            "flatMap is very useful",
            "functional programming rocks"
        );

        System.out.println("Before flatMap (sentences): " + sentences);

        List<String> words = sentences.stream()
                                      .flatMap(sentence -> Arrays.stream(sentence.split(" ")))
                                      .toList();

        System.out.println("After flatMap (words): " + words);
    }
}
```

Output:

```
Before flatMap (sentences): [Java streams are powerful, flatMap is very useful, function
After flatMap (words): [Java, streams, are, powerful, flatMap, is, very, useful, functic
```

10.1.3 Example 3: Processing Nested Objects

Assume you have a list of `User` objects, each with a list of email addresses, and you want to get all emails in a single stream.

```java
import java.util.List;

public class FlatMapExample3 {
    static class User {
        String name;
        List<String> emails;
        User(String name, List<String> emails) {
            this.name = name;
            this.emails = emails;
        }
    }

    public static void main(String[] args) {
        List<User> users = List.of(
            new User("Alice", List.of("alice@example.com", "alice.work@example.com")),
            new User("Bob", List.of("bob@example.com")),
            new User("Carol", List.of())
        );

        System.out.println("Before flatMap (users): " + users.size() + " users");

        List<String> allEmails = users.stream()
                                      .flatMap(user -> user.emails.stream())
                                      .toList();

        System.out.println("After flatMap (allEmails): " + allEmails);
    }
}
```

Output:

```
Before flatMap (users): 3 users
After flatMap (allEmails): [alice@example.com, alice.work@example.com, bob@example.com]
```

10.1.4 Summary

- Use `map()` for **one-to-one** transformations.
- Use `flatMap()` when each element maps to **multiple elements** or a nested stream, and you want to flatten them into a single stream.
- Common use cases include flattening nested collections, splitting strings into words, or extracting nested properties from objects.

By mastering `flatMap()`, you can write concise, expressive code that handles complex data structures with ease.

10.2 Mapping to Multiple Collections

In some situations, you want to **process a stream once but collect the results into multiple collections simultaneously**—for example, splitting data into categories or producing different aggregations from the same data source.

Java 12 introduced **`Collectors.teeing()`**, a powerful utility that allows you to run two collectors in parallel on the same stream and then combine their results. Before Java 12, this kind of logic required multiple stream passes or manual splitting.

Using `Collectors.teeing()`

`teeing()` takes three arguments:

- **Two downstream collectors** that process the stream elements.

- A **merger function** to combine their results into a single output object.

10.2.1 Example 1: Splitting Even and Odd Numbers into Separate Lists

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class TeeingExample1 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        var result = numbers.stream()
            .collect(Collectors.teeing(
                // Collector for even numbers
                Collectors.filtering(n -> n % 2 == 0, Collectors.toList()),
                // Collector for odd numbers
                Collectors.filtering(n -> n % 2 != 0, Collectors.toList()),
                // Merge into a Map
                (evens, odds) -> Map.of("evens", evens, "odds", odds)
            ));

        System.out.println("Evens: " + result.get("evens"));
        System.out.println("Odds: " + result.get("odds"));
    }
}
```

Output:

```
Evens: [2, 4, 6, 8, 10]
Odds: [1, 3, 5, 7, 9]
```

10.2.2 Example 2: Computing Multiple Summaries (Count and Sum) from the Same Stream

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class TeeingExample2 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(10, 20, 30, 40, 50);

        var summary = numbers.stream()
            .collect(Collectors.teeing(
                Collectors.counting(),
                Collectors.summingInt(Integer::intValue),
                (count, sum) -> Map.of("count", count, "sum", sum)
            ));

        System.out.println("Count: " + summary.get("count"));
        System.out.println("Sum: " + summary.get("sum"));
    }
}
```

Output:

```
Count: 5
Sum: 150
```

10.2.3 Alternative: Splitting Streams Without `teeing()` (Pre-Java 12)

If `Collectors.teeing()` is unavailable, you can process the stream twice or use `partitioningBy()` for boolean splits:

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class PartitioningExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

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

        System.out.println("Evens: " + partitioned.get(true));
        System.out.println("Odds: " + partitioned.get(false));
    }
}
```

10.2.4 Summary

- Use **`Collectors.teeing()`** to collect stream data into multiple collections or summaries in one pass.
- It reduces the need for multiple stream traversals and keeps code concise.
- For boolean splits, **`partitioningBy()`** is a useful shortcut.
- Mapping to multiple collections improves efficiency and clarity when analyzing or categorizing data.

These techniques empower you to write expressive, high-performance data processing pipelines with Java Streams.

10.3 Complex Transformations

Complex transformations in streams go beyond simple one-to-one mappings. They often involve **conditional logic**, **extracting multiple fields**, and **nesting maps** to reshape data into new forms like DTOs (Data Transfer Objects) or JSON-ready structures. By composing smaller transformations, you can build modular and readable pipelines for sophisticated data processing.

Key Patterns in Complex Transformations

- **Conditional mapping:** Apply different transformations based on element state or attributes.
- **Multi-field extraction:** Extract multiple properties from objects to construct new structures.
- **Nested mapping:** Build hierarchical or grouped data, such as reports or JSON-like objects.
- **DTO mapping:** Convert domain objects into simplified or formatted versions for output or transport.

10.3.1 Example 1: Summarizing Transactions by Customer and Category

Given a list of transactions, group them by customer, then by category, and calculate total amounts per group.

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class ComplexTransformationExample1 {

    static class Transaction {
        String customer;
        String category;
        double amount;

        Transaction(String customer, String category, double amount) {
            this.customer = customer;
            this.category = category;
            this.amount = amount;
        }
    }

    public static void main(String[] args) {
        List<Transaction> transactions = List.of(
            new Transaction("Alice", "Books", 120.0),
            new Transaction("Alice", "Electronics", 200.0),
            new Transaction("Bob", "Books", 80.0),
            new Transaction("Bob", "Books", 50.0),
            new Transaction("Alice", "Books", 30.0)
        );

        Map<String, Map<String, Double>> report = transactions.stream()
            .collect(Collectors.groupingBy(
                t -> t.customer,
                Collectors.groupingBy(
                    t -> t.category,
                    Collectors.summingDouble(t -> t.amount)
                )
            ));

        System.out.println("Customer spending report:");
        System.out.println(report);
    }
}
```

Output:

```
Customer spending report:
{Alice={Books=150.0, Electronics=200.0}, Bob={Books=130.0}}
```

This nested grouping and summing provides a clear multi-level summary of spending per customer and category.

10.3.2 Example 2: Mapping Entities to JSON-Ready Nested DTOs with Conditional Fields

Suppose you have `Person` objects with optional addresses and phone numbers. You want to convert them into nested DTOs suitable for JSON serialization, skipping empty data.

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class ComplexTransformationExample2 {

    static class Person {
        String name;
        String email;
        List<String> phones;
        Address address;

        Person(String name, String email, List<String> phones, Address address) {
            this.name = name;
            this.email = email;
            this.phones = phones;
            this.address = address;
        }
    }

    static class Address {
        String street;
        String city;

        Address(String street, String city) {
            this.street = street;
            this.city = city;
        }
    }

    static class PersonDTO {
        String name;
        String email;
        List<String> phones;
        Map<String, String> address;

        PersonDTO(String name, String email, List<String> phones, Map<String, String> address) {
            this.name = name;
            this.email = email;
            this.phones = phones;
            this.address = address;
        }

        @Override
        public String toString() {
            return "PersonDTO{" +
                   "name='" + name + '\'' +
                   ", email='" + email + '\'' +
                   ", phones=" + phones +
                   ", address=" + address +
                   '}';
        }
    }

    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("John Doe", "john@example.com", List.of("123-456-7890"), new Address("123 Elm St
            new Person("Jane Smith", "jane@example.com", List.of(), null)
        );

        List<PersonDTO> dtos = people.stream()
            .map(p -> new PersonDTO(
                p.name,
                p.email,
                p.phones.isEmpty() ? null : p.phones,
                p.address == null ? null : Map.of(
                    "street", p.address.street,
                    "city", p.address.city
                )
            ))
            .collect(Collectors.toList());

        dtos.forEach(System.out::println);
    }
}
```

Output:

```
PersonDTO{name='John Doe', email='john@example.com', phones=[123-456-
7890], address={street=123 Elm St, city=Springfield}}
PersonDTO{name='Jane Smith', email='jane@example.com', phones=null, address=null}
```

This example shows conditional mapping (skipping empty phones and addresses) and nested object construction using maps, suitable for JSON serialization frameworks.

10.3.3 Summary

Complex transformations often combine:

- **Conditional logic** to handle optional or varying data.
- **Nested mappings** to create hierarchical or grouped structures.
- **DTO conversion** for clean data transfer or output.

By modularizing these transformations and composing smaller functions, you create maintainable and expressive stream pipelines that fit real-world data processing needs.

Chapter 11.

Grouping and Partitioning Data

1. Grouping with `Collectors.groupingBy()`
2. Multi-level Grouping
3. Partitioning with `Collectors.partitioningBy()`

11 Grouping and Partitioning Data

11.1 Grouping with `Collectors.groupingBy()`

The `Collectors.groupingBy()` method is one of the most powerful collectors in the Java Streams API, used to **group elements of a stream based on a classification function**. It partitions the input elements into a `Map` whose keys are the classification results, and whose values are collections (or other results) of the grouped elements.

Basic Syntax

```
Map<K, List<T>> grouped = stream.collect(Collectors.groupingBy(classifier));
```

- **classifier**: a function that maps each element to a key (the grouping criterion).
- By default, elements with the same key are collected into a `List`.

You can also specify downstream collectors to change the result collection type (e.g., `Set`, `Map`) or perform further reduction.

11.1.1 Example 1: Grouping Users by Age Bracket

Suppose we want to group a list of users by their age bracket (e.g., “Youth”, “Adult”, “Senior”):

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class GroupingExample1 {
    static class User {
        String name;
        int age;

        User(String name, int age) {
            this.name = name;
            this.age = age;
        }

        @Override
        public String toString() {
            return name + "(" + age + ")";
        }
    }

    public static void main(String[] args) {
        List<User> users = List.of(
            new User("Alice", 23),
            new User("Bob", 45),
```

```java
            new User("Charlie", 17),
            new User("Diana", 65),
            new User("Eve", 34)
        );

        Map<String, List<User>> groupedByAgeBracket = users.stream()
            .collect(Collectors.groupingBy(user -> {
                if (user.age < 18) return "Youth";
                else if (user.age < 60) return "Adult";
                else return "Senior";
            }));

        System.out.println("Users grouped by age bracket:");
        groupedByAgeBracket.forEach((ageGroup, groupUsers) -> {
            System.out.println(ageGroup + ": " + groupUsers);
        });
    }
}
```

Output:

```
Users grouped by age bracket:
Youth: [Charlie(17)]
Adult: [Alice(23), Bob(45), Eve(34)]
Senior: [Diana(65)]
```

11.1.2 Example 2: Grouping Orders by Status with Different Collection Types

This example groups orders by their status and collects results into a `Set` to avoid duplicates.

```java
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

public class GroupingExample2 {
    static class Order {
        String id;
        String status;

        Order(String id, String status) {
            this.id = id;
            this.status = status;
        }

        @Override
        public String toString() {
            return id;
        }
    }
    public static void main(String[] args) {
        List<Order> orders = List.of(
            new Order("1001", "SHIPPED"),
            new Order("1002", "PENDING"),
            new Order("1003", "SHIPPED"),
            new Order("1004", "CANCELLED"),
            new Order("1005", "PENDING")
        );

        Map<String, Set<Order>> groupedByStatus = orders.stream()
            .collect(Collectors.groupingBy(
                order -> order.status,
                Collectors.toSet()  // Collect into a Set instead of default List
            ));

        System.out.println("Orders grouped by status:");
        groupedByStatus.forEach((status, orderSet) -> {
            System.out.println(status + ": " + orderSet);
        });
    }
}
```

Output:

```
Orders grouped by status:
SHIPPED: [1003, 1001]
PENDING: [1005, 1002]
CANCELLED: [1004]
```

11.1.3 Summary

- `Collectors.groupingBy()` groups stream elements according to a **classification function**, producing a `Map<K, List<T>>` by default.
- You can customize the downstream collector to change the type of the values, e.g., `Set`, `Map`, or aggregate results.
- Grouping is essential for data categorization, reporting, and aggregation tasks, enabling concise and expressive processing pipelines.

By mastering `groupingBy()`, you can transform flat streams into structured maps that reflect the logical organization of your data.

11.2 Multi-level Grouping

Multi-level grouping allows you to **group data hierarchically** by applying `Collectors.groupingBy()` multiple times in a nested fashion. This technique produces nested maps, where the value of one grouping is itself a map resulting from a further grouping operation.

Why Multi-level Grouping?

Sometimes, a single classification is not enough. For example:

- Grouping employees **first by department**, then by role within each department.
- Grouping customers **by country**, then by city.
- Grouping products **by category**, then by brand.

Syntax Overview

```
Map<K1, Map<K2, List<T>>> nestedGrouping = stream.collect(
    Collectors.groupingBy(
        classifier1,
        Collectors.groupingBy(classifier2)
    )
);
```

Here, `classifier1` is the outer grouping function, and `classifier2` is the inner grouping function applied within each outer group.

11.2.1 Example 1: Grouping Employees by Department, then Role

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class MultiLevelGroupingExample1 {
    static class Employee {
        String name;
        String department;
        String role;

        Employee(String name, String department, String role) {
            this.name = name;
            this.department = department;
            this.role = role;
        }

        @Override
        public String toString() {
            return name;
        }
    }

    public static void main(String[] args) {
        List<Employee> employees = List.of(
            new Employee("Alice", "HR", "Manager"),
            new Employee("Bob", "HR", "Recruiter"),
            new Employee("Charlie", "IT", "Developer"),
            new Employee("Diana", "IT", "Developer"),
            new Employee("Eve", "IT", "Manager")
        );

        Map<String, Map<String, List<Employee>>> grouped = employees.stream()
            .collect(Collectors.groupingBy(
                emp -> emp.department,
                Collectors.groupingBy(emp -> emp.role)
            ));

        System.out.println("Employees grouped by department and role:");
        grouped.forEach((dept, roleMap) -> {
            System.out.println(dept + ":");
            roleMap.forEach((role, emps) -> {
                System.out.println("  " + role + " -> " + emps);
            });
        });
    }
}
```

Output:

```
Employees grouped by department and role:
HR:
  Manager -> [Alice]
  Recruiter -> [Bob]
IT:
  Developer -> [Charlie, Diana]
  Manager -> [Eve]
```

11.2.2 Example 2: Grouping Customers by Country and City

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class MultiLevelGroupingExample2 {
    static class Customer {
        String name;
        String country;
        String city;

        Customer(String name, String country, String city) {
            this.name = name;
            this.country = country;
            this.city = city;
        }
```

```java
        @Override
        public String toString() {
            return name;
        }
    }

    public static void main(String[] args) {
        List<Customer> customers = List.of(
            new Customer("John", "USA", "New York"),
            new Customer("Jane", "USA", "Boston"),
            new Customer("Pierre", "France", "Paris"),
            new Customer("Marie", "France", "Lyon"),
            new Customer("Steve", "USA", "New York")
        );

        Map<String, Map<String, List<Customer>>> groupedByCountryCity = customers.stream()
            .collect(Collectors.groupingBy(
                c -> c.country,
                Collectors.groupingBy(c -> c.city)
            ));

        System.out.println("Customers grouped by country and city:");
        groupedByCountryCity.forEach((country, cityMap) -> {
            System.out.println(country + ":");
            cityMap.forEach((city, custs) -> {
                System.out.println("  " + city + " -> " + custs);
            });
        });
    }
}
```

Output:

```
Customers grouped by country and city:
USA:
  New York -> [John, Steve]
  Boston -> [Jane]
France:
  Paris -> [Pierre]
  Lyon -> [Marie]
```

11.2.3 Traversing Nested Groupings

The resulting nested map is a structure like:

```
Map<OuterKey, Map<InnerKey, List<Element>>>
```

You can traverse it using nested loops or stream operations, as shown in the examples. This structure allows you to drill down from coarse to fine groupings easily.

11.2.4 Summary

- Multi-level grouping is a natural extension of `groupingBy()` for hierarchical data classification.
- The outer collector defines the first grouping level, and the inner collector defines subsequent levels.
- The result is a nested `Map` structure that supports flexible data navigation and aggregation.

Mastering multi-level grouping empowers you to organize complex datasets intuitively and efficiently within your Java Streams workflows.

11.3 Partitioning with `Collectors.partitioningBy()`

The `Collectors.partitioningBy()` method splits a stream into **two groups based on a boolean predicate**. Unlike `groupingBy()`, which can produce multiple groups keyed by any value, `partitioningBy()` always returns a `Map<Boolean, List<T>>`, dividing elements into those that satisfy the predicate (`true` key) and those that don't (`false` key).

Key Differences from `groupingBy()`

Aspect	`partitioningBy()`	`groupingBy()`
Number of groups	Always 2 (true and false)	Arbitrary number of groups
Return type	`Map<Boolean, List<T>>` (by default)	`Map<K, List<T>>`
Use case	Binary classification	Multi-class classification
Performance	Slightly more efficient for boolean grouping	General-purpose grouping

`partitioningBy()` is simpler and often more performant when you only need a **true/false split**.

11.3.1 Example 1: Partitioning Numbers into Even and Odd

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class PartitioningExample1 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        Map<Boolean, List<Integer>> partitioned = numbers.stream()
            .collect(Collectors.partitioningBy(n -> n % 2 == 0));

        System.out.println("Even numbers: " + partitioned.get(true));
        System.out.println("Odd numbers: " + partitioned.get(false));
    }
}
```

Output:

```
Even numbers: [2, 4, 6, 8, 10]
Odd numbers: [1, 3, 5, 7, 9]
```

11.3.2 Example 2: Partitioning Users into Active and Inactive

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class PartitioningExample2 {
    static class User {
        String name;
        boolean active;

        User(String name, boolean active) {
            this.name = name;
            this.active = active;
        }

        @Override
        public String toString() {
            return name;
        }
    }

    public static void main(String[] args) {
        List<User> users = List.of(
            new User("Alice", true),
            new User("Bob", false),
            new User("Charlie", true),
            new User("Diana", false)
        );

        Map<Boolean, List<User>> partitioned = users.stream()
            .collect(Collectors.partitioningBy(user -> user.active));

        System.out.println("Active users: " + partitioned.get(true));
        System.out.println("Inactive users: " + partitioned.get(false));
    }
}
```

Output:

```
Active users: [Alice, Charlie]
Inactive users: [Bob, Diana]
```

11.3.3 Using Downstream Collectors

You can further customize the collection of each partition with a downstream collector. For example, counting how many users are active vs inactive:

```java
Map<Boolean, Long> countByActivity = users.stream()
    .collect(Collectors.partitioningBy(
        user -> user.active,
        Collectors.counting()
    ));

System.out.println("Count by activity: " + countByActivity);
```

Output:

```
Count by activity: {false=2, true=2}
```

11.3.4 Summary

- `partitioningBy()` splits stream elements into **two groups based on a boolean predicate**, returning a `Map<Boolean, List<T>>`.
- It is simpler and more efficient than `groupingBy()` when only binary partitioning is needed.
- Downstream collectors can be used to perform further reductions on each partition.
- Common use cases include separating even/odd numbers, active/inactive users, or any yes/no classification.

Using `partitioningBy()` makes binary data splits clear and concise, enhancing both performance and code readability.

Chapter 12.

Custom Collectors

12 Custom Collectors

12.1 Writing Your Own Collector

Java Streams provide many built-in collectors, but sometimes you need a **custom collector** to handle special data aggregation or formatting needs. Writing your own collector involves implementing the `Collector` interface or using the convenient `Collector.of()` factory method by specifying five core components:

The Five Required Components of a Collector

Component	Description
Supplier	Creates a new mutable container to hold partial results (e.g., a `StringBuilder` or list).
Accumulator	Adds an element from the stream to the mutable container.
Combiner	Merges two partial containers (needed for parallel processing).
Finisher	Converts the container into the final desired result (may be identity if no conversion needed).
Characteristics	Provides hints about the collector's behavior, like `UNORDERED` or `IDENTITY_FINISH`.

12.1.1 Complete Example: Custom Collector to Join Strings as a CSV Line

Suppose you want to collect a stream of strings into a **single CSV line**, enclosed in quotes and separated by commas, such as:

```
"apple","banana","cherry"
```

Here's how to build this collector step-by-step:

```java
import java.util.Set;
import java.util.stream.Collector;
import java.util.stream.Collectors;

public class CustomCollectorExample {

    public static void main(String[] args) {
        var fruits = java.util.List.of("apple", "banana", "cherry");

        String csv = fruits.stream()
            .collect(csvCollector());

        System.out.println(csv);
    }
    public static Collector<String, StringBuilder, String> csvCollector() {
        return Collector.of(
            // Supplier: create a new StringBuilder to accumulate results
            StringBuilder::new,

            // Accumulator: add each string wrapped in quotes with trailing comma
            (sb, s) -> {
                if (sb.length() > 0) sb.append(",");
                sb.append("\"").append(s).append("\"");
            },

            // Combiner: merge two StringBuilders (needed for parallel streams)
            (sb1, sb2) -> {
                if (sb1.length() == 0) return sb2;
                if (sb2.length() == 0) return sb1;
                sb1.append(",").append(sb2);
                return sb1;
            },

            // Finisher: convert StringBuilder to String (final output)
            StringBuilder::toString,

            // Characteristics: no special behavior, so empty set
            Collector.Characteristics.IDENTITY_FINISH
        );
    }
}
```

12.1.2 Step-by-Step Breakdown

1. **Supplier:** Creates an empty `StringBuilder` to accumulate elements.
2. **Accumulator:** For each element `s`, appends a comma if needed, then adds the string enclosed in quotes.
3. **Combiner:** Joins two partial `StringBuilder` results by appending a comma between them, crucial for parallel execution.
4. **Finisher:** Converts the `StringBuilder` into the final `String`.
5. **Characteristics:** Specifies `IDENTITY_FINISH` since the finisher is a simple conversion and the container isn't reused beyond this.

12.1.3 Why Create a Custom Collector?

- Built-in collectors like `Collectors.joining()` work well for simple joins but may not handle complex formatting or aggregation logic.
- Custom collectors enable full control over intermediate state, combination logic, and

final result.
- They are reusable and integrate seamlessly with the Stream API.

By understanding these components and how to implement them, you can create efficient, thread-safe, and flexible collectors tailored to your data processing needs.

12.2 Collector Interface Explained

The `Collector` interface in Java Streams defines how to **reduce** a stream of elements into a single summary result. It abstracts the process of mutable reduction — accumulating elements, combining partial results (for parallel processing), and finishing with a final transformation.

Generic Parameters: `T, A, R`

- `T`: The **type of input elements** in the stream.
- `A`: The **mutable accumulation type**, i.e., the container or intermediate data structure that holds partial results during processing.
- `R`: The **result type** returned after the final transformation.

Core Collector Components

A `Collector<T, A, R>` consists of these essential components:

Component	Role
supplier()	Provides a fresh **mutable container** (`A`) to hold partial results.
accumulator()	Takes the container (`A`) and an element (`T`) and incorporates the element into the container.
combiner()	Merges two containers (`A` and `A`) into one — critical for parallel execution where partial results are combined.
finisher()	Transforms the container (`A`) into the final result type (`R`). Often an identity function if no transformation is needed.
characteristics()	Describes behavioral hints about the collector, like concurrency or ordering.

Characteristics

The `characteristics()` method returns a `Set<Collector.Characteristics>` with zero or more of these flags:

- **CONCURRENT:** Indicates that the accumulator function can be called concurrently from multiple threads on the same container. Allows the Stream framework to perform parallel reductions without additional combining steps.
- **UNORDERED:** Declares that the collector does not care about the encounter order of the stream elements. This can enable optimizations.

- **IDENTITY_FINISH:** Means the finisher is the identity function, so the accumulator type `A` and the result type `R` are the same, avoiding an additional transformation step.

Simplified Pseudocode Example

Here's an abstract view of how the collector's components fit together during reduction:

```
// 1. Create a new container to hold partial results
A container = supplier.get();

// 2. Process each element in the stream:
for (T element : stream) {
    accumulator.accept(container, element);
}

// 3. For parallel streams, combine partial containers:
A combined = combiner.apply(container1, container2);

// 4. Final transformation to result:
R result = finisher.apply(combined);
```

Minimal Collector Interface (Simplified)

```
public interface Collector<T, A, R> {

    Supplier<A> supplier();

    BiConsumer<A, T> accumulator();

    BinaryOperator<A> combiner();

    Function<A, R> finisher();

    Set<Characteristics> characteristics();

    enum Characteristics {
        CONCURRENT, UNORDERED, IDENTITY_FINISH
    }
}
```

12.2.1 Summary

- The `Collector` interface defines how streams are **reduced** by accumulating elements into a mutable container and then producing a final result.
- Its generic parameters `<T, A, R>` represent the input element type, the mutable accumulator type, and the final result type.
- Characteristics provide important hints for **optimization and parallelization**.
- Understanding these parts helps you grasp the power behind built-in collectors and design your own custom ones effectively.

12.3 Examples of Custom Collectors

Custom collectors empower you to tailor stream reductions to your unique needs. Here are three practical examples demonstrating how to implement and use such collectors for common real-world tasks.

12.3.1 Example 1: Building a Histogram from a Stream of Words

A histogram counts occurrences of each distinct element. This collector accumulates frequencies in a `Map<String, Integer>`.

```java
import java.util.*;
import java.util.stream.Collector;

public class HistogramCollector {

    public static Collector<String, Map<String, Integer>, Map<String, Integer>> toHistogram() {
        return Collector.of(
            HashMap::new,                                // Supplier: create a new map
            (map, word) -> map.merge(word, 1, Integer::sum), // Accumulator: increment count
            (map1, map2) -> {                           // Combiner: merge two maps
                map2.forEach((k, v) -> map1.merge(k, v, Integer::sum));
                return map1;
            }
        );
    }

    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "apple", "orange", "banana", "banana");

        Map<String, Integer> histogram = words.stream()
            .collect(toHistogram());

        System.out.println(histogram);
    }
}
```

Expected Output:

```
{orange=1, banana=3, apple=2}
```

12.3.2 Example 2: Creating a String Prefix/Suffix Wrapper Collector

This collector joins strings with a delimiter and adds a custom prefix and suffix, useful for generating formatted output.

```java
import java.util.List;
import java.util.Set;
import java.util.stream.Collector;

public class WrappedStringCollector {

    public static Collector<String, StringBuilder, String> wrappedCollector(
        String prefix, String delimiter, String suffix) {

        return Collector.of(
            StringBuilder::new,
            (sb, s) -> {
                if (sb.length() > 0) sb.append(delimiter);
                sb.append(s);
            },
            (sb1, sb2) -> {
                if (sb1.length() == 0) return sb2;
                if (sb2.length() == 0) return sb1;
                return sb1.append(delimiter).append(sb2.toString());
            },
            sb -> prefix + sb.toString() + suffix
            // Removed Characteristics.UNORDERED - order matters
        );
    }

    public static void main(String[] args) {
        List<String> items = List.of("red", "green", "blue");

        String result = items.stream()
            .collect(wrappedCollector("[", "; ", "]"));

        System.out.println(result); // Output: [red; green; blue]
    }
}
```

Expected Output:

```
[red; green; blue]
```

12.3.3 Example 3: Aggregating a Stream of Objects into a Formatted Table

Suppose you have a list of `Person` objects and want to collect their details into a neatly formatted table string.

```java
import java.util.*;
import java.util.stream.Collector;

public class TableCollector {

    static class Person {
        String name;
        int age;
        Person(String name, int age) {
            this.name = name;
            this.age = age;
        }
    }

    public static Collector<Person, StringBuilder, String> toTable() {
        return Collector.of(
            StringBuilder::new,
            (sb, p) -> sb.append(String.format("%-10s | %3d%n", p.name, p.age)),
            (sb1, sb2) -> {
                sb1.append(sb2);
                return sb1;
            },
            StringBuilder::toString // <-- finisher added here
        );
    }

    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Alice", 30),
            new Person("Bob", 25),
            new Person("Carol", 28)
        );

        String table = people.stream()
            .collect(toTable());

        System.out.println("Name       | Age\n------------------");
        System.out.print(table);
    }
}
```

Expected Output:

```
Name       | Age
------------------
Alice      |  30
Bob        |  25
Carol      |  28
```

12.3.4 Summary

- **Histogram Collector:** Efficient frequency counting using a mutable `Map`.
- **Wrapped String Collector:** Flexible string joining with prefix and suffix.
- **Table Collector:** Custom formatting of objects into a structured table.

Each example demonstrates clean, reusable designs and highlights how custom collectors can be tailored to solve varied data processing tasks seamlessly within the Streams API.

Chapter 13.
Stream Composition and Pipelines

13 Stream Composition and Pipelines

13.1 Chaining Multiple Streams

One of the most powerful features of Java Streams is the ability to **chain multiple operations** together, forming a clear and concise data processing pipeline. Each intermediate operation returns a new stream, enabling fluent composition of transformations without modifying the original data source.

Immutability and Laziness in Stream Chaining

- **Immutability:** Streams are immutable; operations don't change the source data but produce new streams with the applied transformations. This avoids side effects and makes code easier to reason about.
- **Laziness:** Intermediate operations like `filter()`, `map()`, and `sorted()` are lazy — they don't execute immediately. Instead, they build up a pipeline of transformations that only run when a **terminal operation** (e.g., `collect()`, `forEach()`) is invoked. This allows optimization and efficient processing.

Practical Examples

Here are two real-world examples illustrating chaining of multiple stream operations:

Example 1: Filter, Map, and Sort a List of Products

Imagine a product list where you want to find all available items priced above $20, convert their names to uppercase, and then sort alphabetically.

```java
import java.util.*;
import java.util.stream.*;

public class ProductPipeline {
    static class Product {
        String name;
        double price;
        boolean available;

        Product(String name, double price, boolean available) {
            this.name = name; this.price = price; this.available = available;
        }

        @Override public String toString() {
            return name + " ($" + price + ")";
        }
    }

    public static void main(String[] args) {
        List<Product> products = List.of(
            new Product("Laptop", 999.99, true),
            new Product("Mouse", 19.99, true),
            new Product("Keyboard", 29.99, false),
            new Product("Monitor", 199.99, true),
            new Product("USB Cable", 10.99, true)
        );

        List<String> result = products.stream()
            .filter(p -> p.available)                    // Keep only available
            .filter(p -> p.price > 20)                   // Price > 20
            .map(p -> p.name.toUpperCase())              // Map to uppercase names
            .sorted()                                    // Sort alphabetically
            .collect(Collectors.toList());

        System.out.println(result);
    }
}
```

Output:

```
[LAPTOP, MONITOR]
```

Example 2: FlatMapping Nested Lists and Sorting

Suppose you have a list of orders, each containing multiple items. You want to extract all unique item names, filter those starting with "B", and sort them.

```
import java.util.*;
import java.util.stream.*;

public class OrderPipeline {
    static class Order {
        List<String> items;

        Order(List<String> items) {
            this.items = items;
        }
    }

    public static void main(String[] args) {
        List<Order> orders = List.of(
            new Order(List.of("Banana", "Apple", "Bread")),
            new Order(List.of("Butter", "Orange", "Banana")),
            new Order(List.of("Bread", "Blueberry", "Apple"))
        );

        List<String> filteredItems = orders.stream()
            .flatMap(order -> order.items.stream())   // Flatten all items into one stream
            .filter(item -> item.startsWith("B"))     // Items starting with 'B'
            .distinct()                               // Remove duplicates
            .sorted()                                 // Sort alphabetically
            .collect(Collectors.toList());

        System.out.println(filteredItems);
    }
}
```

Output:

```
[Banana, Blueberry, Bread, Butter]
```

13.1.1 Best Practices for Stream Chaining

- **Keep pipelines readable:** Avoid overly long chains; break complex logic into well-named methods if necessary.
- **Minimize side effects:** Stream operations should be pure functions without side effects.
- **Use lazy intermediate operations:** Leverage laziness for efficiency; only trigger processing with terminal operations.
- **Use method references when clear:** For cleaner code, prefer `ClassName::method` syntax over lambdas when applicable.

13.1.2 Summary

Chaining multiple stream operations lets you build clear, modular, and efficient data processing pipelines. Immutability and laziness ensure safe, optimized execution. By combining operations like `filter()`, `map()`, `flatMap()`, and `sorted()`, you can express complex logic in a fluent, readable style that's easy to maintain and understand.

13.2 Combining Streams with `concat()`

The `Stream.concat()` method allows you to **combine two separate streams into a single stream**. This is useful when you have multiple data sources or intermediate streams you want to merge seamlessly and then continue processing as one.

How `Stream.concat()` Works

- It takes two streams of the **same element type** and returns a new stream that will sequentially emit all elements from the first stream, followed by all elements from the second stream.
- Both input streams remain unchanged — the operation produces a new combined stream without mutating existing streams.
- It supports both **sequential** and **parallel** streams.
- **Important:** Neither of the streams passed to `concat()` can be `null`. If there is a possibility of `null`, you should handle it explicitly before concatenation.

Limitations

- `Stream.concat()` can only combine **two streams** at a time. To concatenate more streams, you need to chain multiple calls or use other approaches (e.g., `Stream.of()` with `flatMap()`).
- It does **not automatically flatten nested collections**; the streams must be of the same element type.
- Performance-wise, concatenation introduces minimal overhead but be mindful when working with very large or infinite streams.

Examples

Example 1: Concatenating Numeric Streams

Combine two `IntStream`s of numbers and sum all elements.

```java
import java.util.stream.*;

public class NumericConcatExample {
    public static void main(String[] args) {
        IntStream first = IntStream.range(1, 4);   // 1, 2, 3
        IntStream second = IntStream.range(4, 7);  // 4, 5, 6

        IntStream combined = IntStream.concat(first, second);

        int sum = combined.sum();  // 1+2+3+4+5+6 = 21
        System.out.println("Sum of combined streams: " + sum);
    }
}
```

Expected Output:

```
Sum of combined streams: 21
```

Example 2: Concatenating Streams from Lists

Merge two lists of strings into one stream, convert all to uppercase, and collect.

```java
import java.util.*;
import java.util.stream.*;

public class ListConcatExample {
    public static void main(String[] args) {
        List<String> list1 = List.of("apple", "banana");
        List<String> list2 = List.of("cherry", "date");

        Stream<String> combined = Stream.concat(list1.stream(), list2.stream());

        List<String> result = combined
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        System.out.println(result);
    }
}
```

Expected Output:

```
[APPLE, BANANA, CHERRY, DATE]
```

Example 3: Concatenating Streams of Custom Objects

Suppose you have two employee streams and want to concatenate them, then filter by department.

```java
import java.util.*;
import java.util.stream.*;

public class EmployeeConcatExample {
    static class Employee {
        String name;
        String department;

        Employee(String name, String department) {
            this.name = name; this.department = department;
        }

        @Override
        public String toString() {
            return name + " (" + department + ")";
        }
    }

    public static void main(String[] args) {
        Stream<Employee> teamA = Stream.of(
            new Employee("Alice", "Sales"),
            new Employee("Bob", "HR")
        );

        Stream<Employee> teamB = Stream.of(
            new Employee("Carol", "Sales"),
            new Employee("Dave", "IT")
        );

        List<Employee> salesTeam = Stream.concat(teamA, teamB)
            .filter(e -> "Sales".equals(e.department))
            .collect(Collectors.toList());

        System.out.println("Sales team: " + salesTeam);
    }
}
```

Expected Output:

```
Sales team: [Alice (Sales), Carol (Sales)]
```

13.2.1 Summary

- `Stream.concat()` is a simple and effective way to merge two streams of the same type.
- It works with sequential or parallel streams but requires non-null inputs.
- The combined stream can be further processed with filters, maps, or collectors as usual.
- For concatenating multiple streams, chain `concat()` calls or consider other composition methods.

By mastering `Stream.concat()`, you can flexibly merge data sources and build more expressive stream pipelines.

13.3 Stream Builders and Custom Stream Sources

Java Streams offer flexible ways to create streams programmatically beyond collections and arrays. This includes `Stream.builder()`, `Stream.generate()`, `Stream.iterate()`, and custom sources for dynamic or infinite data. Each method serves different needs, such as deferred computation, on-demand data, or constructing values programmatically.

13.3.1 Using `Stream.builder()`

`Stream.builder()` allows you to build a stream manually by adding elements one at a time. It's useful when the data isn't already in a collection or needs to be constructed conditionally.

```java
import java.util.stream.*;

public class BuilderExample {
    public static void main(String[] args) {
        Stream<String> names = Stream.<String>builder()
            .add("Alice")
            .add("Bob")
            .add("Carol")
            .build();

        names.forEach(System.out::println);
    }
}
```

13.3.2 Using `Stream.generate()`

`Stream.generate()` creates an **infinite stream** where each element is supplied by a `Supplier<T>`. It's ideal for producing repeated or computed values, such as random numbers.

```java
import java.util.stream.*;
import java.util.Random;

public class GenerateExample {
    public static void main(String[] args) {
        Random rand = new Random();

        Stream.generate(() -> rand.nextInt(100))  // infinite stream of random numbers
              .limit(5)
              .forEach(System.out::println);
    }
}
```

Use `limit()` to safely cap infinite streams.

13.3.3 Using `Stream.iterate()`

`Stream.iterate()` is used to produce elements by applying a function repeatedly, often for sequences.

```java
import java.util.stream.*;

public class IterateExample {
    public static void main(String[] args) {
        Stream.iterate(1, n -> n + 2)   // odd numbers
              .limit(5)
              .forEach(System.out::println);
    }
}
```

From Java 9 onward, you can also pass a predicate to create bounded `iterate()` streams.

13.3.4 Creating Custom Stream Sources

Custom streams are useful when reading from non-standard sources like sensors, logs, or APIs. This often involves wrapping a custom `Iterator` or using `Spliterator` and `StreamSupport`.

Example: Stream from a Custom Iterator (simulated sensor data)

```java
import java.util.*;
import java.util.stream.*;

public class CustomIteratorExample {
    public static void main(String[] args) {
        Iterator<Double> sensorSimulator = new Iterator<>() {
            private int count = 0;

            @Override
            public boolean hasNext() {
                return count++ < 5; // Simulate 5 sensor readings
            }

            @Override
            public Double next() {
                return 20 + Math.random() * 10; // Random temperature between 20-30°C
            }
        };

        Iterable<Double> iterable = () -> sensorSimulator;

        Stream<Double> sensorStream = StreamSupport.stream(iterable.spliterator(), false);

        sensorStream.forEach(temp -> System.out.println("Sensor: " + temp + "°C"));
    }
}
```

13.3.5 When to Use Each Approach

Method	Best Use Case
`Stream.builder()`	Dynamically building small or condition-based streams
`Stream.generate()`	Infinite or lazy values like UUIDs, timestamps, sensors
`Stream.iterate()`	Sequences with clear progression rules
Custom Iterator	External/dynamic data like sensors, APIs, logs

13.3.6 Summary

Java provides several flexible ways to build streams beyond collections:

- `Stream.builder()` for programmatic element addition,
- `Stream.generate()` for infinite streams from suppliers,
- `Stream.iterate()` for functional sequences,
- Custom stream sources via `Iterator` or `Spliterator` for dynamic or external input.

These tools unlock the full power of streams for modeling complex and evolving data sources.

Chapter 14.

Exception Handling in Streams

14 Exception Handling in Streams

14.1 Handling Checked Exceptions in Lambdas

Java Streams do not directly support lambdas that throw **checked exceptions** (e.g., `IOException`, `SQLException`). Since standard functional interfaces like `Function`, `Predicate`, and `Consumer` don't declare any checked exceptions, trying to throw one inside a lambda causes a **compilation error**.

This restriction often becomes problematic when integrating I/O or other checked-exception-producing logic into stream pipelines.

14.1.1 Why Lambdas and Checked Exceptions Clash

```java
// Compilation error!
List<String> paths = List.of("file1.txt", "file2.txt");

paths.stream()
     .map(path -> Files.readString(Path.of(path))) // IOException not allowed
     .forEach(System.out::println);
```

14.1.2 Strategy 1: Wrap Checked Exception in a RuntimeException

A simple workaround is to wrap the checked exception in an unchecked one (`RuntimeException`). This shifts the handling responsibility downstream.

```java
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class WrapRuntimeExample {
    public static void main(String[] args) {
        List<String> files = List.of("file1.txt", "file2.txt");

        files.stream()
             .map(path -> {
                 try {
                     return Files.readString(Path.of(path));
                 } catch (Exception e) {
                     throw new RuntimeException(e); // wrap checked exception
                 }
             })
             .forEach(System.out::println);
    }
}
```

Pros: Simple to implement. WARNING *Cons*: Loses specific error typing and forces downstream handling or crashes at runtime.

14.1.3 Strategy 2: Use a Utility Method for Exception Handling

To promote reuse and cleaner code, you can create a helper function that converts a lambda that throws checked exceptions into a standard functional interface.

```java
import java.io.*;
import java.nio.file.*;
import java.util.function.*;
import java.util.stream.*;

public class WithHandlerExample {

    // Functional interface that allows checked exceptions
    @FunctionalInterface
    interface CheckedFunction<T, R> {
        R apply(T t) throws Exception;
    }

    // Helper to wrap checked exception
    public static <T, R> Function<T, R> handleChecked(CheckedFunction<T, R> func) {
        return t -> {
            try {
                return func.apply(t);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        };
    }

    public static void main(String[] args) {
        Stream<String> fileNames = Stream.of("file1.txt", "file2.txt");

        fileNames
            .map(handleChecked(path -> Files.readString(Path.of(path))))
            .forEach(System.out::println);
    }
}
```

Pros: Reusable and clean. WARNING *Cons*: Still propagates as runtime exceptions, but improves composability.

14.1.4 Other Strategies

- **Custom Functional Interfaces**: Define interfaces like `ThrowingFunction<T, R, E extends Exception>` and add specific `try-catch` logic.

- **Libraries**: Use external libraries like Vavr that provide enhanced functional interfaces supporting checked exceptions.
- **FlatMap with Try/Optional**: Encapsulate results in `Optional`, `Try`, or custom wrapper types to capture success/failure.

14.1.5 Summary

Handling checked exceptions in streams requires extra effort because lambdas can't throw them directly. You can:

- Wrap them in unchecked exceptions,
- Use reusable helper utilities,
- Or leverage external libraries for richer handling.

These techniques allow you to maintain robust, readable pipelines even in the presence of checked exceptions.

14.2 Strategies for Robust Stream Pipelines

When working with real-world data, stream pipelines must be **resilient** to malformed, missing, or inconsistent input. Streams offer elegant ways to **validate, filter, and recover** without interrupting the entire pipeline. This section outlines key strategies for writing fault-tolerant and robust stream-based code.

14.2.1 Filter Invalid or Null Data Early

The `filter()` operation can remove invalid or `null` elements before they cause errors downstream.

```java
import java.util.*;
import java.util.stream.*;

public class FilterNullExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", null, "Bob", "", "Carol");

        names.stream()
             .filter(Objects::nonNull)         // filter out nulls
             .filter(s -> !s.isBlank())        // filter out empty/blank
             .map(String::toUpperCase)
             .forEach(System.out::println);
    }
}
```

Tip: Defensive filtering early in the pipeline prevents runtime exceptions during transformations.

14.2.2 Use `Optional` to Represent Absence

`Optional` allows you to express missing or invalid values without nulls. Use it with `flatMap()` to seamlessly handle optional returns.

```java
import java.util.List;
import java.util.Optional;

public class OptionalParseExample {

    public static Optional<Integer> parseIntSafe(String s) {
        try {
            return Optional.of(Integer.parseInt(s));
        } catch (NumberFormatException e) {
            return Optional.empty();
        }
    }

    public static void main(String[] args) {
        List<String> numbers = List.of("42", "invalid", "17", "NaN");

        numbers.stream()
               .map(OptionalParseExample::parseIntSafe)
               .flatMap(Optional::stream) // skips Optional.empty()
               .forEach(System.out::println);
    }
}
```

Benefit: Avoids crashes by skipping invalid elements gracefully.

14.2.3 Validate Before Transforming

Use conditional logic to validate data before applying expensive or error-prone transformations.

```java
import java.util.List;

public class ValidateBeforeTransform {
    public static void main(String[] args) {
        List<String> values = List.of("12", "-5", "abc", "8");

        values.stream()
              .filter(s -> s.matches("\\d+"))   // only positive integers
              .map(Integer::parseInt)
              .filter(n -> n > 0)              // business rule: must be > 0
              .forEach(System.out::println);
    }
}
```

14.2.4 Wrap Operations with Try/Catch or Fallbacks

Use helper methods to wrap risky operations and provide default values or error logging.

```java
import java.util.List;

public class FallbackExample {
    public static int safeParse(String s) {
        try {
            return Integer.parseInt(s);
        } catch (NumberFormatException e) {
            return -1; // sentinel or fallback value
        }
    }

    public static void main(String[] args) {
        List<String> inputs = List.of("100", "oops", "250");

        inputs.stream()
              .map(FallbackExample::safeParse)
              .filter(n -> n >= 0)
              .forEach(System.out::println);
    }
}
```

14.2.5 Summary of Best Practices

Strategy	Benefit
`filter(Objects::nonNull)`	Avoids `NullPointerException`
Early validation	Prevents downstream errors
Use of `Optional` and `flatMap()`	Handles absence clearly
Exception-wrapping helper methods	Keeps pipelines clean and safe
Provide fallback/default values	Maintains continuity of processing

By applying these strategies, your stream pipelines become more **robust, readable, and reliable**, even in the face of imperfect or inconsistent data.

14.3 Examples with File I/O and Streams

Working with files in stream pipelines often involves handling **checked exceptions**, particularly `IOException`. Java provides the `Files.lines(Path)` method, which returns a lazily populated `Stream<String>` for reading lines of text from a file. However, because it can throw an exception, it's important to use proper **exception handling** techniques such as **try-with-resources** to ensure safety and resource management.

14.3.1 Recommended Pattern: Try-with-Resources

Using try-with-resources ensures that the file stream is **automatically closed**, even if an exception occurs during stream processing.

14.3.2 Example 1: Reading and Filtering Log Entries

Suppose you have a log file where each line is a log entry, and you want to extract only the `ERROR` lines.

```java
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.*;

public class ErrorLogFilter {
    public static void main(String[] args) {
        Path logFile = Path.of("application.log");

        try (Stream<String> lines = Files.lines(logFile)) {
            lines.filter(line -> line.contains("ERROR"))
                 .map(String::trim)
                 .forEach(System.out::println);
        } catch (IOException e) {
            System.err.println("Failed to read log file: " + e.getMessage());
        }
    }
}
```

Explanation:

- `Files.lines()` returns a `Stream<String>` from the file.
- `try-with-resources` ensures the file is closed after processing.
- If an `IOException` occurs, it's caught and logged.

14.3.3 Example 2: Reading CSV Records with Fallback

This example processes a CSV file where each line contains a user record (e.g., `name,email`). Some lines may be malformed.

```java
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class UserCsvProcessor {
    public static void main(String[] args) {
        Path file = Path.of("users.csv");

        try (Stream<String> lines = Files.lines(file)) {
            List<User> users = lines
                .skip(1) // skip header
                .map(UserCsvProcessor::parseUserSafely)
                .flatMap(Optional::stream) // filter out failed parses
                .collect(Collectors.toList());

            users.forEach(System.out::println);
        } catch (IOException e) {
            System.err.println("Unable to read users.csv: " + e.getMessage());
        }
    }

    static Optional<User> parseUserSafely(String line) {
        try {
            String[] parts = line.split(",");
            if (parts.length < 2) return Optional.empty();
            return Optional.of(new User(parts[0].trim(), parts[1].trim()));
        } catch (Exception e) {
            return Optional.empty(); // skip bad record
        }
    }

    static class User {
        String name;
        String email;

        User(String name, String email) {
            this.name = name;
            this.email = email;
        }

        public String toString() {
            return "User{name='" + name + "', email='" + email + "'}";
        }
    }
}
```

Key Features:

- Skips invalid lines using `Optional` and `flatMap`.
- Catches and logs `IOException` for file access.

- Uses clear fallback logic for malformed input.

14.3.4 Summary

When working with file-based streams:

- Use `try-with-resources` with `Files.lines()` to manage file handles safely.
- Handle checked exceptions like `IOException` gracefully.
- Filter or recover from malformed input using `Optional`, custom parsers, or default values.

These practices ensure robust stream pipelines when reading from disk or other I/O sources.

Chapter 15.

Processing Text Files and Data Sources

15 Processing Text Files and Data Sources

15.1 Reading and Processing CSV Files with Streams

Java Streams offer a powerful and efficient way to read and process CSV files, especially when combined with `Files.lines()`. By leveraging the stream API, you can read large datasets line by line, transform them into structured objects, filter or group them, and collect results with minimal memory usage and high readability.

15.1.1 Key Steps for CSV Processing

1. **Open the file** using `Files.lines(Path)` in a try-with-resources block.
2. **Skip headers** if present.
3. **Split each line** using `String.split()` or a CSV parser.
4. **Map lines to structured objects**.
5. **Filter or transform** data as needed.
6. **Collect results** into a list or other structure.

15.1.2 Example: Parse CSV into a `Person` Class

CSV file: `people.csv`

```
name,age,email
Alice,30,alice@example.com
Bob,25,bob@example.com
Charlie,invalid,charlie@example.com
```

```java
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class CsvPersonExample {
    public static void main(String[] args) {
        Path file = Path.of("people.csv");

        try (Stream<String> lines = Files.lines(file)) {
            List<Person> people = lines
                .skip(1) // Skip header
                .map(CsvPersonExample::parsePerson)
                .flatMap(Optional::stream) // Filter out failed parses
                .filter(p -> p.age >= 18) // Filter adults
                .collect(Collectors.toList());
            people.forEach(System.out::println);
        } catch (IOException e) {
            System.err.println("Failed to read CSV file: " + e.getMessage());
        }
    }

    static Optional<Person> parsePerson(String line) {
        try {
            String[] parts = line.split(",", -1);
            if (parts.length < 3) return Optional.empty();
            String name = parts[0].trim();
            int age = Integer.parseInt(parts[1].trim());
            String email = parts[2].trim();
            return Optional.of(new Person(name, age, email));
        } catch (Exception e) {
            return Optional.empty(); // Skip malformed line
        }
    }

    static class Person {
        String name;
        int age;
        String email;

        Person(String name, int age, String email) {
            this.name = name;
            this.age = age;
            this.email = email;
        }

        public String toString() {
            return name + " (" + age + ") - " + email;
        }
    }
}
```

15.1.3 Error Handling & Best Practices

- Use `Optional` to gracefully skip malformed records.
- Always wrap `Files.lines()` in a try-with-resources block to close the stream properly.
- For large files, avoid loading everything into memory unless necessary.
- Consider using a CSV library like OpenCSV for complex formats with quoted values and delimiters.

15.1.4 Summary

Processing CSV files with streams enables concise, readable, and efficient data transformation pipelines. With careful error handling and attention to performance, you can parse large datasets into structured objects with minimal overhead.

15.2 Processing Large Text Files Efficiently

Java Streams are especially powerful for processing **large text files** thanks to their **lazy evaluation** and **line-by-line streaming** capabilities. Unlike traditional approaches that read the entire file into memory, `Files.lines(Path)` returns a lazily populated `Stream<String>`, allowing efficient processing of massive files without exhausting system resources.

15.2.1 Why Streams Are Efficient for Large Files

- **Lazy Evaluation**: Data is read and processed only when needed.
- **Line Streaming**: Only one line is held in memory at a time.
- **Parallel Processing**: Can be parallelized for performance (with care).

This is ideal for logs, large CSVs, or text analytics.

15.2.2 Example 1: Count Lines Matching a Pattern

```java
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.*;

public class ErrorCounter {
    public static void main(String[] args) {
        Path logPath = Path.of("server.log");

        try (Stream<String> lines = Files.lines(logPath)) {
            long errorCount = lines
                .filter(line -> line.contains("ERROR"))
                .count();

            System.out.println("Total ERROR lines: " + errorCount);
        } catch (IOException e) {
            System.err.println("Failed to read file: " + e.getMessage());
        }
    }
}
```

Explanation: Only lines containing `"ERROR"` are processed. No need to load the full log file into memory.

15.2.3 Example 2: Extract Statistics from Lines

Suppose each line contains a numeric value. You want to compute summary statistics efficiently.

```
values.txt:
23
42
17
invalid
58
```

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class FileStats {
    public static void main(String[] args) {
        Path path = Path.of("values.txt");

        try (Stream<String> lines = Files.lines(path)) {
            IntSummaryStatistics stats = lines
                .map(String::trim)
                .filter(s -> s.matches("\\d+")) // Filter valid numbers
                .mapToInt(Integer::parseInt)
                .summaryStatistics();

            System.out.println("Count: " + stats.getCount());
            System.out.println("Min: " + stats.getMin());
            System.out.println("Max: " + stats.getMax());
            System.out.println("Average: " + stats.getAverage());
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Efficient Design:

- Filters invalid data early.
- Uses `mapToInt()` for primitive stream processing.
- Avoids materializing all lines at once.

15.2.4 Best Practices

- Use `try-with-resources` to ensure file closure.
- Avoid stateful operations or collecting entire files unnecessarily.
- Use `parallel()` cautiously—file I/O is typically I/O-bound, not CPU-bound.

15.2.5 Summary

Java Streams combined with `Files.lines()` provide a scalable, elegant solution for processing large text files. By processing data lazily and efficiently, you can analyze logs, parse files, and compute summaries without memory overhead, even on gigabyte-scale datasets.

15.3 Example: Word Count with Streams

Word counting is a classic problem that demonstrates the power of Java Streams for text processing. This example walks through reading a file, tokenizing text into words, normalizing and cleaning input, and then computing word frequencies using collectors.

We'll use `Files.lines()` to read a file lazily, process each line to extract words, and count occurrences with `Collectors.groupingBy()`.

15.3.1 Key Steps

1. Read lines from a file using `Files.lines()`.
2. Convert all text to lowercase (normalization).
3. Split lines into words using a regex.
4. Filter out blanks or invalid entries.
5. Use `Collectors.groupingBy()` and `Collectors.counting()` to tally words.

15.3.2 Example File: `sample.txt`

```
Hello, world!
This is a test. This is only a test.
hello HELLO? test!
```

15.3.3 Runnable Code: Word Frequency Counter

```java
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
import java.util.function.Function;
import java.util.stream.*;
```

```java
public class WordCountExample {
    public static void main(String[] args) {
        Path file = Path.of("sample.txt");

        try (Stream<String> lines = Files.lines(file)) {
            Map<String, Long> wordCounts = lines
                .flatMap(line -> Arrays.stream(line
                    .toLowerCase()                  // Normalize case
                    .replaceAll("[^a-z\\s]", "") // Remove punctuation
                    .split("\\s+")))             // Split by whitespace
                .filter(word -> !word.isBlank()) // Skip empty strings
                .collect(Collectors.groupingBy(
                    Function.identity(),
                    Collectors.counting()));

            // Print sorted result
            wordCounts.entrySet().stream()
                .sorted(Map.Entry.<String, Long>comparingByValue(Comparator.reverseOrder()))
                .forEach(entry ->
                    System.out.printf("%-10s -> %d%n", entry.getKey(), entry.getValue()));
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

15.3.4 Breakdown of the Pipeline

- **Normalization**: `toLowerCase()` ensures that "Hello" and "hello" are treated the same.
- **Cleaning**: `replaceAll("[^a-z\\s]", "")` strips punctuation.
- **Splitting**: `split("\\s+")` tokenizes each line into words.
- **Filtering**: Removes blank strings from extra spaces or lines.
- **Collecting**: Groups by word and counts each occurrence.
- **Sorting**: Final output is sorted by frequency in descending order.

15.3.5 Output for `sample.txt`

```
hello      -> 3
test       -> 3
this       -> 2
is         -> 2
a          -> 2
only       -> 1
```

```
world       -> 1
```

15.3.6 Summary

This example highlights how to build a complete and efficient word count pipeline using Java Streams. With just a few transformations and collectors, you can process complex text input, handle edge cases like punctuation and blank lines, and produce clean, sorted output. This pattern is easily extendable to more advanced natural language processing tasks.

Chapter 16. Working with Dates and Times in Streams

1. Stream Operations on `LocalDate`, `LocalDateTime`
2. Filtering and Grouping by Date Fields
3. Practical Examples: Event Processing

16 Working with Dates and Times in Streams

16.1 Stream Operations on `LocalDate`, `LocalDateTime`

Java Streams work seamlessly with the `java.time` package, allowing powerful manipulation and analysis of date and time data. Two of the most commonly used types are `LocalDate` (date without time) and `LocalDateTime` (date with time).

Streams can be used to **sort**, **filter**, **group**, and **transform** collections of date/time objects. Common use cases include parsing string-based dates, computing time differences, or filtering based on a date range.

16.1.1 Example 1: Mapping Strings to `LocalDate` and Sorting

```java
import java.time.LocalDate;
import java.util.*;
import java.util.stream.*;

public class SortDates {
    public static void main(String[] args) {
        List<String> dateStrings = List.of("2024-03-15", "2025-01-01", "2023-12-31");

        List<LocalDate> sortedDates = dateStrings.stream()
            .map(LocalDate::parse)              // Convert to LocalDate
            .sorted()                           // Natural order (chronological)
            .collect(Collectors.toList());

        sortedDates.forEach(System.out::println);
    }
}
```

Output:

```
2023-12-31
2024-03-15
2025-01-01
```

16.1.2 Example 2: Filter Dates in the Past

```java
import java.time.LocalDate;
import java.util.*;
import java.util.stream.*;
```

```java
public class FilterPastDates {
    public static void main(String[] args) {
        List<LocalDate> dates = List.of(
            LocalDate.of(2023, 5, 1),
            LocalDate.now().plusDays(1),
            LocalDate.now().minusDays(10)
        );

        LocalDate today = LocalDate.now();

        List<LocalDate> pastDates = dates.stream()
            .filter(d -> d.isBefore(today))      // Keep only past dates
            .collect(Collectors.toList());

        pastDates.forEach(System.out::println);
    }
}
```

16.1.3 Example 3: Add Days to Each Date

```java
import java.time.LocalDate;
import java.util.*;
import java.util.stream.*;

public class AddDays {
    public static void main(String[] args) {
        List<LocalDate> dates = List.of(
            LocalDate.of(2024, 10, 10),
            LocalDate.of(2024, 12, 25)
        );

        List<LocalDate> futureDates = dates.stream()
            .map(date -> date.plusDays(7))       // Add 7 days to each date
            .collect(Collectors.toList());

        futureDates.forEach(System.out::println);
    }
}
```

16.1.4 Summary

Java's `LocalDate` and `LocalDateTime` integrate cleanly with streams. Use cases include:

- Mapping from strings to typed date/time.
- Filtering by comparison using `isBefore()`, `isAfter()`, and `equals()`.
- Transforming dates using `plusDays()`, `minusMonths()`, etc.
- Sorting chronologically with `.sorted()`.

These operations enable expressive, readable pipelines for time-based logic in domains like scheduling, logging, and event processing.

16.2 Filtering and Grouping by Date Fields

Java Streams, in combination with `LocalDate` and `LocalDateTime`, make it simple to filter and group data based on time components like **year**, **month**, or **day of week**. This is useful in scenarios such as grouping events by month, filtering records from a specific year, or analyzing weekly patterns.

We'll use methods like `getYear()`, `getMonth()`, and `getDayOfWeek()` and combine them with `Collectors.groupingBy()` to build structured date-based groupings.

16.2.1 Example 1: Filter Events by Year

```java
import java.time.LocalDate;
import java.util.List;
import java.util.stream.Collectors;

public class FilterByYear {
    public static void main(String[] args) {
        List<LocalDate> events = List.of(
            LocalDate.of(2023, 6, 10),
            LocalDate.of(2024, 1, 5),
            LocalDate.of(2024, 7, 20),
            LocalDate.of(2025, 2, 14)
        );

        List<LocalDate> events2024 = events.stream()
            .filter(date -> date.getYear() == 2024)
            .collect(Collectors.toList());

        events2024.forEach(System.out::println);
    }
}
```

Output:

```
2024-01-05
2024-07-20
```

16.2.2 Example 2: Group Tasks by Month

```
import java.time.LocalDate;
import java.time.Month;
import java.util.*;
import java.util.stream.Collectors;

public class GroupByMonth {
    public static void main(String[] args) {
        Map<String, LocalDate> tasks = Map.of(
            "Submit Report", LocalDate.of(2024, 3, 15),
            "Doctor Visit", LocalDate.of(2024, 3, 28),
            "Vacation", LocalDate.of(2024, 7, 10),
            "Conference", LocalDate.of(2024, 7, 25),
            "New Year Prep", LocalDate.of(2024, 12, 29)
        );

        Map<Month, List<String>> tasksByMonth = tasks.entrySet().stream()
            .collect(Collectors.groupingBy(
                entry -> entry.getValue().getMonth(),
                Collectors.mapping(Map.Entry::getKey, Collectors.toList())
            ));

        tasksByMonth.forEach((month, taskList) -> {
            System.out.println(month + ": " + taskList);
        });
    }
}
```

Output:

```
MARCH: [Submit Report, Doctor Visit]
JULY: [Vacation, Conference]
DECEMBER: [New Year Prep]
```

16.2.3 Example 3: Group Events by Day of Week

```
import java.time.LocalDate;
import java.time.DayOfWeek;
import java.util.*;
import java.util.stream.Collectors;

public class GroupByDayOfWeek {
    public static void main(String[] args) {
        List<LocalDate> dates = List.of(
            LocalDate.of(2024, 6, 24),  // MONDAY
            LocalDate.of(2024, 6, 25),  // TUESDAY
            LocalDate.of(2024, 6, 26),  // WEDNESDAY
            LocalDate.of(2024, 7, 1),   // MONDAY
            LocalDate.of(2024, 7, 2)    // TUESDAY
        );

        Map<DayOfWeek, List<LocalDate>> grouped = dates.stream()
            .collect(Collectors.groupingBy(LocalDate::getDayOfWeek));

        grouped.forEach((day, dateList) -> {
            System.out.println(day + ": " + dateList);
        });
    }
}
```

Sample Output:

```
MONDAY: [2024-06-24, 2024-07-01]
TUESDAY: [2024-06-25, 2024-07-02]
WEDNESDAY: [2024-06-26]
```

16.2.4 Summary

- Use `getYear()`, `getMonth()`, or `getDayOfWeek()` on `LocalDate` or `LocalDateTime` to filter or classify dates.
- Use `Collectors.groupingBy()` to create maps organized by these time components.
- Combining date manipulation with stream operations makes time-based data analysis elegant and efficient.

These patterns are commonly used in reporting, scheduling systems, and analytics dashboards.

16.3 Practical Examples: Event Processing

Event processing is a common use case for Java Streams, especially when working with timestamps. With `LocalDateTime`, we can filter, group, and summarize events efficiently and expressively.

In this section, we'll walk through two real-world examples using streams:

1. **Filtering and grouping scheduled events by date**
2. **Counting log events per day**

16.3.1 Example 1: Scheduling Events Filter Future Events and Group by Day

```java
import java.time.LocalDateTime;
import java.time.LocalDate;
import java.util.*;
import java.util.stream.Collectors;

class Event {
    String title;
    LocalDateTime timestamp;

    Event(String title, LocalDateTime timestamp) {
        this.title = title;
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return title + " @ " + timestamp;
    }
}

public class ScheduleProcessor {
    public static void main(String[] args) {
        List<Event> events = List.of(
            new Event("Team Meeting", LocalDateTime.now().plusDays(1)),
            new Event("Project Deadline", LocalDateTime.now().plusDays(3)),
            new Event("Code Review", LocalDateTime.now().minusDays(1)),
            new Event("Client Call", LocalDateTime.now().plusDays(1)),
            new Event("System Maintenance", LocalDateTime.now().plusDays(5))
        );

        LocalDateTime now = LocalDateTime.now();

        // Filter future events and group by event date
        Map<LocalDate, List<Event>> futureEventsByDate = events.stream()
            .filter(e -> e.timestamp.isAfter(now))
            .collect(Collectors.groupingBy(e -> e.timestamp.toLocalDate()));

        futureEventsByDate.forEach((date, evts) -> {
            System.out.println("Date: " + date);
            evts.forEach(e -> System.out.println("  " + e));
        });
    }
}
```

Output (example):

```
Date: 2025-06-24
  Team Meeting @ 2025-06-24T08:30
  Client Call @ 2025-06-24T14:00
Date: 2025-06-26
  Project Deadline @ 2025-06-26T09:00
Date: 2025-06-28
```

```
System Maintenance @ 2025-06-28T23:00
```

16.3.2 Example 2: Log Event Count by Day

```java
import java.time.LocalDateTime;
import java.time.LocalDate;
import java.util.*;
import java.util.stream.Collectors;

class LogEntry {
    LocalDateTime timestamp;
    String level;

    LogEntry(LocalDateTime timestamp, String level) {
        this.timestamp = timestamp;
        this.level = level;
    }
}

public class LogAnalyzer {
    public static void main(String[] args) {
        List<LogEntry> logs = List.of(
            new LogEntry(LocalDateTime.of(2024, 6, 23, 10, 15), "INFO"),
            new LogEntry(LocalDateTime.of(2024, 6, 23, 12, 40), "ERROR"),
            new LogEntry(LocalDateTime.of(2024, 6, 24, 9, 0), "INFO"),
            new LogEntry(LocalDateTime.of(2024, 6, 24, 16, 30), "WARN"),
            new LogEntry(LocalDateTime.of(2024, 6, 25, 11, 20), "INFO")
        );

        Map<LocalDate, Long> logCountPerDay = logs.stream()
            .collect(Collectors.groupingBy(
                log -> log.timestamp.toLocalDate(),
                Collectors.counting()
            ));

        logCountPerDay.forEach((date, count) ->
            System.out.println(date + ": " + count + " log entries")
        );
    }
}
```

Output:

```
2024-06-23: 2 log entries
2024-06-24: 2 log entries
2024-06-25: 1 log entries
```

16.3.3 Summary

- Use `LocalDateTime` for timestamps and convert to `LocalDate` using `.toLocalDate()` for grouping.
- Stream pipelines can efficiently **filter** past/future events and **group** by any date field.
- Combining collectors and time APIs enables **powerful reporting**, **monitoring**, and **scheduling** use cases.

These techniques scale well even for large event datasets and are widely used in production applications such as schedulers, log analyzers, and activity trackers.

Chapter 17.

Real-world Data Processing Examples

1. Stream-Based Data Filtering and Reporting
2. Transforming and Exporting Data
3. Stream Processing in Web Applications

17 Real-world Data Processing Examples

17.1 Stream-Based Data Filtering and Reporting

Streams provide a powerful, declarative way to filter large datasets and generate insightful reports. By chaining operations like `filter()`, `map()`, `collect()`, and aggregation methods, you can transform raw data into meaningful summaries efficiently and readably.

Below are two practical examples from common domains: employee performance filtering and product expiration reporting.

17.1.1 Example 1: Filtering Top-Performing Employees

Suppose you have a list of employees with sales numbers and want to generate a report of employees who exceeded a sales threshold.

```java
import java.util.*;
import java.util.stream.Collectors;

class Employee {
    String name;
    double sales;

    Employee(String name, double sales) {
        this.name = name;
        this.sales = sales;
    }

    public double getSales() {
        return sales;
    }

    @Override
    public String toString() {
        return name + " (Sales: " + sales + ")";
    }
}

public class EmployeeReport {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
            new Employee("Alice", 15000),
            new Employee("Bob", 9000),
            new Employee("Carol", 20000),
            new Employee("David", 12000),
            new Employee("Eve", 8000)
        );

        double threshold = 10000;

        List<Employee> topPerformers = employees.stream()
```

```
            .filter(e -> e.getSales() > threshold)
            .sorted(Comparator.comparingDouble(Employee::getSales).reversed())
            .collect(Collectors.toList());

        System.out.println("Top Performing Employees:");
        topPerformers.forEach(System.out::println);
    }
}
```

Output:

```
Top Performing Employees:
Carol (Sales: 20000.0)
Alice (Sales: 15000.0)
David (Sales: 12000.0)
```

Explanation: We start by filtering employees whose sales exceed the threshold, then sort them in descending order by sales, and finally collect them into a list for reporting.

17.1.2 Example 2: Extracting All Expired Products from a Catalog

Assume you manage a product catalog with expiration dates. The goal is to extract and report all products that have expired as of today.

```
import java.time.LocalDate;
import java.util.*;
import java.util.stream.Collectors;

class Product {
    String name;
    LocalDate expirationDate;

    Product(String name, LocalDate expirationDate) {
        this.name = name;
        this.expirationDate = expirationDate;
    }

    @Override
    public String toString() {
        return name + " (Expires: " + expirationDate + ")";
    }
}

public class ProductReport {
    public static void main(String[] args) {
        List<Product> products = List.of(
            new Product("Milk", LocalDate.now().minusDays(1)),
            new Product("Bread", LocalDate.now().plusDays(2)),
            new Product("Cheese", LocalDate.now().minusDays(5)),
            new Product("Butter", LocalDate.now().plusDays(10))
        );

        LocalDate today = LocalDate.now();

        List<Product> expiredProducts = products.stream()
            .filter(p -> p.expirationDate.isBefore(today) || p.expirationDate.isEqual(today))
            .collect(Collectors.toList());

        System.out.println("Expired Products:");
        expiredProducts.forEach(System.out::println);
    }
}
```

Output:

```
Expired Products:
Milk (Expires: 2025-06-22)
Cheese (Expires: 2025-06-18)
```

Explanation: The stream filters products whose expiration date is before or equal to the current date, gathering all expired items for the report.

17.1.3 Summary

- Use **filter** to narrow down datasets by conditions.
- Use **sorted** to order results meaningfully.
- Collect results into lists, sets, or maps for further processing.
- Streams enable building readable, efficient pipelines for data filtering and reporting.

These patterns apply broadly—from HR analytics to inventory management—helping transform large data into actionable insights.

17.2 Transforming and Exporting Data

Streams are ideal for reshaping and preparing data for export into formats such as JSON, CSV, or custom string representations. By combining mapping operations and collectors, you can easily transform domain objects into Data Transfer Objects (DTOs) or formatted strings, and then write them to files or other output streams.

17.2.1 Example 1: Mapping Domain Objects to CSV Format

Suppose you have a list of `Employee` objects and want to export their details as CSV lines.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.stream.Collectors;

class Employee {
    String name;
    String department;
    double salary;

    Employee(String name, String department, double salary) {
        this.name = name;
        this.department = department;
        this.salary = salary;
    }
}

public class CsvExportExample {
    public static void main(String[] args) throws IOException {
        List<Employee> employees = List.of(
            new Employee("Alice", "Engineering", 85000),
            new Employee("Bob", "Sales", 72000),
            new Employee("Carol", "Engineering", 91000)
        );

        // Map each employee to a CSV line
        List<String> csvLines = employees.stream()
            .map(e -> String.join(",", e.name, e.department, String.valueOf(e.salary)))
            .collect(Collectors.toList());

        // Add header at the beginning
        csvLines.add(0, "Name,Department,Salary");

        // Write to CSV file
        Path outputPath = Path.of("employees.csv");
        Files.write(outputPath, csvLines);

        System.out.println("CSV export completed: " + outputPath.toAbsolutePath());
    }
}
```

Explanation: We transform each `Employee` into a CSV string using `map()`, collect the lines into a list, prepend a header line, and write the list to a file with `Files.write()`. This approach preserves the data structure and is easy to maintain or extend.

17.2.2 Example 2: Exporting Data as JSON-like Strings

For lightweight JSON export without external libraries, streams can build JSON representations manually.

```java
import java.util.List;
import java.util.stream.Collectors;

class Product {
    String name;
    double price;

    Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
}

public class JsonExportExample {
    public static void main(String[] args) {
        List<Product> products = List.of(
            new Product("Laptop", 1200.50),
            new Product("Phone", 650.00),
            new Product("Tablet", 300.99)
        );

        String json = products.stream()
            .map(p -> String.format("{\"name\":\"%s\",\"price\":%.2f}", p.name, p.price))
            .collect(Collectors.joining(", ", "[", "]"));

        System.out.println("JSON output:");
        System.out.println(json);
    }
}
```

Explanation: This example transforms each `Product` into a JSON-like string, then joins them with commas, wrapping the entire collection in square brackets to form a JSON array. This pattern is useful for simple JSON export without external dependencies.

17.2.3 Key Points

- **Mapping:** Use `map()` to convert domain objects into strings or DTOs for the desired output format.
- **Collecting:** Use collectors like `Collectors.toList()` or `Collectors.joining()` to assemble the output.
- **Exporting:** Use `Files.write()` or similar APIs to persist results into files.
- **Formatting:** Properly format fields (e.g., escaping CSV, JSON quotes) for correctness and readability.

Streams provide a clean, functional approach to transform and export data efficiently, supporting formats ranging from CSV and JSON to any custom output your application requires.

17.3 Stream Processing in Web Applications

In modern web applications, Java Streams offer a powerful, concise way to handle common data processing tasks such as filtering API request payloads, transforming query results, and preparing data for JSON responses. Their fluent, declarative style fits naturally within service layers or controllers, enabling clean, readable, and maintainable code.

17.3.1 Example 1: Filtering User-submitted Form Data

Imagine a REST endpoint receives a list of user roles submitted from a form. The service needs to validate and filter out invalid or duplicate roles before further processing.

```java
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

class UserRoleService {
    private static final Set<String> VALID_ROLES = Set.of("ADMIN", "USER", "MODERATOR");

    public List<String> filterValidRoles(List<String> submittedRoles) {
        return submittedRoles.stream()
            .map(String::toUpperCase)                 // Normalize case
            .filter(VALID_ROLES::contains)            // Keep only valid roles
            .distinct()                              // Remove duplicates
            .collect(Collectors.toList());
    }
}

// Example usage in a controller or service
public class UserController {
    public static void main(String[] args) {
        UserRoleService service = new UserRoleService();

        List<String> submittedRoles = List.of("admin", "user", "guest", "user");
        List<String> filteredRoles = service.filterValidRoles(submittedRoles);

        System.out.println("Filtered Roles: " + filteredRoles);
    }
}
```

Explanation: This example uses a stream pipeline to clean and validate input, demonstrating how Streams can be integrated into business logic for form processing or API input validation.

17.3.2 Example 2: Transforming Database Query Results for JSON Response

Suppose your backend retrieves a list of `Product` entities from the database, but the API response requires a simplified DTO with only name and price, formatted as strings.

```java
import java.util.List;
import java.util.stream.Collectors;

class Product {
    String name;
    double price;
    String description; // extra field not needed in response

    Product(String name, double price, String description) {
        this.name = name;
        this.price = price;
        this.description = description;
    }
}

class ProductDTO {
    String name;
    String price;

    ProductDTO(String name, String price) {
        this.name = name;
        this.price = price;
    }
}

class ProductService {
    public List<ProductDTO> getProductDTOs(List<Product> products) {
        return products.stream()
            .map(p -> new ProductDTO(p.name, String.format("$%.2f", p.price)))
            .collect(Collectors.toList());
    }
}

// Simulated Controller method
public class ProductController {
    public static void main(String[] args) {
        List<Product> products = List.of(
            new Product("Laptop", 1200.99, "High-end laptop"),
            new Product("Mouse", 25.50, "Wireless mouse"),
            new Product("Keyboard", 45.00, "Mechanical keyboard")
        );

        ProductService productService = new ProductService();
        List<ProductDTO> response = productService.getProductDTOs(products);

        // In a real app, response would be serialized as JSON by the web framework
        response.forEach(dto -> System.out.println(dto.name + ": " + dto.price));
    }
}
```

Explanation: Here, streams cleanly handle the conversion of entity objects to DTOs tailored for API responses, demonstrating separation of concerns and data shaping before JSON serialization.

17.3.3 Why Use Streams in Web Apps?

- **Conciseness:** Reduces boilerplate loops and conditional logic in controllers/services.
- **Readability:** Declarative style improves clarity on data flow and transformations.
- **Composability:** Easy to chain multiple operations such as filtering, mapping, and sorting.
- **Integration:** Works seamlessly with collections returned by repositories or incoming JSON payloads.

17.3.4 Best Practices

- Keep stream logic in service layers rather than directly in controllers for testability.
- Handle null and empty inputs gracefully within streams (e.g., using `Optional` or `filter`).
- Use DTOs or view models to decouple internal domain objects from API contracts.

Streams empower Java web applications to efficiently process, validate, and transform data, enhancing code maintainability and performance in APIs and user interfaces alike.

Chapter 18.

Performance Considerations and Best Practices

18 Performance Considerations and Best Practices

18.1 Stream Performance Tips

Using Java Streams effectively requires attention to performance nuances, especially when working with large datasets or performance-critical applications. Below are key tips to optimize stream processing for both sequential and parallel streams.

Minimize Intermediate Operations

Each intermediate operation (like `map()`, `filter()`, `sorted()`) adds processing overhead. Chaining many intermediate steps can degrade performance, especially if some operations are expensive or redundant.

- **Why it matters:** More operations mean more per-element processing before the terminal operation triggers execution.
- **Tip:** Combine operations logically and avoid unnecessary transformations. For example, filter early to reduce elements processed by subsequent operations.

```java
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
int sum = numbers.stream()
    .filter(n -> n % 2 == 0)  // Filter early to reduce downstream processing
    .mapToInt(n -> n * 2)
    .sum();
```

Use Primitive Streams (`IntStream`, `LongStream`, `DoubleStream`) Whenever Possible

Primitive streams avoid the overhead of boxing/unboxing wrapper objects (`Integer`, `Long`, `Double`), leading to faster execution and less memory pressure.

- **Why it matters:** Autoboxing incurs object allocation and additional CPU cycles, harming throughput.
- **Tip:** Use `mapToInt()`, `mapToLong()`, or `mapToDouble()` to convert object streams to primitive streams when dealing with numbers.

```java
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
int sum = numbers.stream()
    .mapToInt(Integer::intValue)  // Avoid boxing by using primitive stream
    .sum();
```

Avoid Unnecessary Boxing and Unboxing

Avoid converting primitives to their wrapper classes unless necessary, especially in large pipelines or inside loops.

- **Why it matters:** Boxing/unboxing causes additional allocations and garbage collection overhead.
- **Example Pitfall:** Using `Stream<Integer>` with heavy numerical computations instead

of `IntStream`.

Prefer Short and Simple Pipelines

Long, complicated pipelines can be harder to optimize and may cause performance degradation due to overhead in the stream framework and increased function calls.

- **Why it matters:** Simpler pipelines allow JVM optimizations like inlining and better CPU cache usage.
- **Tip:** Break complex logic into reusable methods or intermediate collections if needed for clarity and performance.

Use Parallel Streams Judiciously

Parallel streams can boost performance for CPU-intensive and large data workloads, but introduce overhead for small datasets or IO-bound operations.

- **Why it matters:** Thread management and splitting costs may outweigh benefits for small or simple streams.
- **Tip:** Benchmark both sequential and parallel versions on realistic datasets. Use parallel streams for:
  - Large collections (thousands+ elements)
  - CPU-bound tasks (e.g., heavy computation)
  - Independent operations with no shared state

Be Careful with Stateful Operations

Operations like `sorted()`, `distinct()`, and `limit()` are stateful and may reduce parallel performance due to synchronization and ordering requirements.

- **Tip:** Use stateful operations sparingly and as late as possible in the pipeline.

Avoid Side-Effects in Intermediate Operations

Side-effects can cause unpredictable performance and thread-safety issues, especially in parallel streams.

- **Tip:** Keep intermediate operations pure and side-effect-free to maximize optimization potential.

18.1.1 Summary Table

Tip	Why It Matters	Example/Note
Minimize intermediate ops	Reduce processing overhead	Filter early in pipeline

Tip	Why It Matters	Example/Note
Use primitive streams	Avoid boxing/unboxing costs	Use `mapToInt()` instead of `map()`
Avoid unnecessary boxing	Decrease memory and CPU overhead	Avoid `Stream<Integer>` for numbers
Prefer short pipelines	Easier JVM optimization	Break complex steps into methods
Use parallel streams wisely	Overhead can negate benefits	Benchmark before adopting
Minimize stateful ops	Statefulness hinders parallelism	Use `sorted()` near pipeline end
Avoid side-effects	Prevent thread-safety and performance bugs	Keep operations pure

18.1.2 Illustrative Microbenchmark (Conceptual)

```java
// Sum numbers using boxed vs primitive streams
List<Integer> numbers = IntStream.rangeClosed(1, 1_000_000).boxed().collect(Collectors.toList());

// Boxed sum - slower due to boxing/unboxing
long startBoxed = System.currentTimeMillis();
int sumBoxed = numbers.stream().mapToInt(Integer::intValue).sum();
long durationBoxed = System.currentTimeMillis() - startBoxed;

// Primitive sum - faster
long startPrimitive = System.currentTimeMillis();
int sumPrimitive = IntStream.rangeClosed(1, 1_000_000).sum();
long durationPrimitive = System.currentTimeMillis() - startPrimitive;

System.out.println("Boxed sum time: " + durationBoxed + " ms");
System.out.println("Primitive sum time: " + durationPrimitive + " ms");
```

By following these performance tips, developers can harness the power and expressiveness of Java Streams while maintaining efficient and scalable applications.

18.2 Avoiding Common Pitfalls

While Java Streams offer powerful and expressive APIs for data processing, several common pitfalls can lead to bugs, poor performance, or unexpected behavior. Recognizing and avoiding these mistakes is crucial for writing robust and maintainable stream code.

Pitfall 1: Modifying External State Inside Streams

Problem: Modifying external mutable state within stream operations (especially intermediate ones) violates the functional programming principles streams promote. This causes unpredictable behavior, especially with parallel streams, leading to race conditions and incorrect results.

Faulty example:

```java
List<String> names = List.of("Alice", "Bob", "Charlie");
List<String> upperNames = new ArrayList<>();

// Modifying external state inside forEach (terminal operation)
names.stream()
     .map(String::toUpperCase)
     .forEach(upperNames::add);

System.out.println(upperNames);
```

While this may appear to work in sequential streams, it is unsafe in parallel streams and harder to debug.

Corrected version:

```java
List<String> names = List.of("Alice", "Bob", "Charlie");

List<String> upperNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(upperNames);
```

Explanation: Use built-in collectors instead of mutating external collections. This approach is thread-safe and expressive.

Pitfall 2: Using Streams on Null Collections or Null Elements

Problem: Streams do not handle `null` collections or elements gracefully. Calling `.stream()` on a `null` collection throws `NullPointerException`. Also, stream operations may fail unexpectedly when elements are `null`.

Faulty example:

```java
List<String> names = null;

// Throws NullPointerException immediately
names.stream()
     .filter(n -> n.startsWith("A"))
     .forEach(System.out::println);
```

Corrected version:

```java
List<String> names = null;

List<String> safeNames = Optional.ofNullable(names)
    .orElseGet(Collections::emptyList);

safeNames.stream()
    .filter(Objects::nonNull)  // Filter out null elements if present
    .filter(n -> n.startsWith("A"))
    .forEach(System.out::println);
```

Explanation: Use `Optional.ofNullable()` or null checks to avoid NPE. Also, consider filtering null elements before processing.

Pitfall 3: Overusing `peek()` for Side Effects

Problem: The `peek()` method is primarily intended for debugging or logging intermediate elements. Using it for business logic or mutating state can cause confusing behavior, especially since intermediate operations are lazy and may not execute as expected.

Faulty example:

```java
List<String> names = List.of("Alice", "Bob", "Charlie");

names.stream()
    .filter(n -> n.length() > 3)
    .peek(n -> System.out.println("Filtered name: " + n))  // Debugging okay
    .peek(n -> someSideEffect(n))  // Side effect - discouraged
    .collect(Collectors.toList());
```

If the stream is never consumed, `peek()` won't run, causing silent bugs.

Corrected version:

```java
List<String> names = List.of("Alice", "Bob", "Charlie");

names.stream()
    .filter(n -> n.length() > 3)
    .peek(n -> System.out.println("Filtered name: " + n))  // Only for debugging/logging
    .collect(Collectors.toList());

// For side effects, use terminal operation explicitly:
names.stream()
    .filter(n -> n.length() > 3)
    .forEach(n -> someSideEffect(n));
```

Explanation: Reserve `peek()` for debugging or logging only. Perform side effects in terminal operations like `forEach()`.

Pitfall 4: Forgetting Stream Laziness Leading to Surprising Behavior

Problem: Stream intermediate operations are lazy and won't execute until a terminal operation triggers them. This can confuse developers expecting immediate results.

Faulty example:

```java
Stream<String> stream = Stream.of("a", "b", "c")
    .filter(s -> {
        System.out.println("Filtering: " + s);
        return true;
    });

// No output yet, filter not executed because no terminal operation
```

Corrected version:

```java
Stream<String> stream = Stream.of("a", "b", "c")
    .filter(s -> {
        System.out.println("Filtering: " + s);
        return true;
    });

stream.forEach(System.out::println);  // Triggers the filtering and prints output
```

Explanation: Always remember streams are lazy. Terminal operations like `forEach()`, `collect()`, or `count()` trigger processing.

18.2.1 Summary

Pitfall	Why It Matters	How to Fix
Modifying external state	Causes race conditions & bugs	Use collectors, avoid shared mutable state
Streaming null collections/elements	Throws NullPointerException	Null checks or use `Optional` and filter nulls
Overusing `peek()` for side-effects	Unreliable side effects, lazy eval	Use `peek()` only for debugging; side effects in terminal ops
Ignoring stream laziness	Confusing silent behavior	Always include terminal operations to trigger pipeline

By keeping these pitfalls in mind and adopting correct patterns, you can write safer, clearer, and more maintainable stream-based code that behaves predictably in real-world applications.

18.3 When Not to Use Streams

While Java Streams provide a powerful, expressive API for data processing, they are not a silver bullet for every programming scenario. Knowing when *not* to use streams is just as important as mastering their use. Here are common situations where traditional approaches might be more suitable than streams:

When You Need Imperative Control or Early Exit

Streams are designed for declarative, pipeline-style data processing without explicit control flow. However, scenarios requiring early exits—like searching for the first element matching a condition with complex break logic—or intricate stateful iteration are often clearer and more efficient with imperative loops.

Example: Early exit on complex conditions

Using Streams (limited):

```java
Optional<String> result = list.stream()
    .filter(s -> {
        // complex condition with side-effects
        return s.startsWith("A");
    })
    .findFirst();
```

This works if only finding first matching element. But if the exit condition depends on external state or multi-step criteria, streams get cumbersome.

Using traditional loop:

```java
String result = null;
for (String s : list) {
    if (complexCondition(s)) {
        result = s;
        break;  // early exit immediately
    }
}
```

The imperative loop offers explicit control and may improve readability and performance in such cases.

Indexed or Position-Sensitive Operations

Streams do not natively support accessing elements by index during processing. If your algorithm requires knowing element positions or working with adjacent elements, using indexed for-loops or specialized data structures is often simpler.

Example: Summing pairs of adjacent numbers

Stream-based approach (awkward):

```java
IntStream.range(0, list.size() - 1)
    .map(i -> list.get(i) + list.get(i + 1))
    .forEach(System.out::println);
```

This requires creating an IntStream over indices and accessing list elements by index, which can be less readable.

Traditional loop:

```java
for (int i = 0; i < list.size() - 1; i++) {
    int sum = list.get(i) + list.get(i + 1);
    System.out.println(sum);
}
```

Here, the classic loop expresses the logic directly and cleanly.

Performance-Critical Tight Loops or Simple Transformations

Streams can introduce overhead through object creation (especially with boxed types), lambda invocation, and pipeline setup. In micro-benchmarks or performance-critical sections with simple logic and tight loops, classic loops can outperform streams due to lower overhead.

Example: Summing an `int[]`

Stream approach:

```java
int sum = Arrays.stream(array).sum();
```

While concise, for very small arrays or high-frequency calls, a simple for-loop might be more performant:

```java
int sum = 0;
for (int i : array) {
    sum += i;
}
```

Complex Stateful or Side-Effecting Algorithms

Streams encourage stateless operations. Algorithms that require maintaining and updating complex external state or performing side effects (e.g., interacting with UI or hardware in each iteration) are better expressed imperatively.

18.3.1 Summary Table

Scenario	Why Streams Might Not Fit	Alternative Recommendation
Early exit with complex logic	Streams limited to short-circuit methods	Use imperative loops with breaks
Index or position-dependent ops	No built-in indexing in streams	Use indexed for-loops
Tight performance-critical loops	Overhead in lambdas and object creation	Use classic loops or specialized libs
Complex stateful or side-effect	Streams prefer stateless, side-effect free	Use imperative code for clarity

18.3.2 Developing Judgment

Streams shine when processing collections declaratively with clear, stateless transformations and aggregations. However, blindly applying streams in all scenarios can lead to complex, less efficient, or harder-to-maintain code.

Before using streams:

- Ask if the task is mostly stateless, transformation-based, or aggregation-centric.
- Consider if early exit or position awareness is critical.
- Measure performance impact if in a hot path.
- Prioritize code readability and maintainability.

In summary, streams are a powerful tool but not a one-size-fits-all solution. Recognizing their limitations and complementing them with traditional techniques when appropriate leads to better, more efficient, and more understandable codebases.

Chapter 19.

Reactive Streams Introduction

1. Overview of Reactive Programming
2. Differences Between Java Streams and Reactive Streams
3. Simple Examples Using `Flow` API (Java 9+)

19 Reactive Streams Introduction

19.1 Overview of Reactive Programming

Reactive programming is a paradigm focused on **asynchronous data flow**, **non-blocking execution**, and **responsive systems** that can handle data streams dynamically and efficiently. Unlike traditional programming models that are often synchronous and blocking, reactive programming embraces *events* and *streams* of data that can be processed as they arrive, enabling highly scalable and resilient applications.

Core Principles of Reactive Programming

- **Asynchronous Data Flow:** Data is emitted over time, and consumers react to new data as it becomes available. The flow is event-driven rather than sequential.
- **Non-blocking Execution:** Reactive systems avoid waiting (blocking) on operations. Instead, they register callbacks or handlers that are triggered when data or events occur, freeing up threads for other tasks.
- **Backpressure:** A mechanism that allows consumers to control the rate at which they receive data from producers, preventing overload and ensuring system stability.

The Reactive Streams Specification

To standardize reactive programming in Java, the **Reactive Streams** specification defines a minimal API to handle asynchronous stream processing with backpressure. This specification is the foundation for many reactive libraries and is included as the `java.util.concurrent.Flow` API in Java 9+.

Key Components and Terminology

- **Publisher:** Produces data asynchronously and publishes it to one or more Subscribers.
- **Subscriber:** Consumes data items emitted by a Publisher. It receives notifications for new data, completion, or errors.
- **Subscription:** Represents a link between Publisher and Subscriber. It allows the Subscriber to request data and cancel the subscription.
- **Processor<T, R>:** A hybrid component that acts both as a Subscriber (consumes data) and Publisher (emits transformed data), enabling data processing pipelines.

Conceptual Flow: Reactive vs. Synchronous

Synchronous Model	Reactive Streams Model
Caller requests data and waits (blocking).	Data flows asynchronously without blocking.
Data is pulled on demand or pushed without control.	Backpressure allows subscriber to control demand.
Sequential, step-by-step execution.	Event-driven, asynchronous processing pipeline.

Visualizing Reactive Streams Flow

```
Publisher  --->  Subscription  --->  Subscriber
   |                ↑                  |
   |                |   request(n)     |
   |-------------> |-----------------|
   |                |                  |
   | emit data (onNext)                |
   |-------------> |-----------------|
```

- The Subscriber requests n items via the `Subscription`.
- The Publisher emits up to n items asynchronously.
- This cycle continues until the stream completes or errors out.
- The backpressure mechanism ensures the Subscriber is never overwhelmed.

In essence, reactive programming transforms how we deal with data streams by making them **asynchronous, non-blocking, and controllable** through backpressure, enabling systems that are highly responsive and resilient to varying workloads. This new paradigm addresses the limitations of traditional synchronous or event-driven push/pull models by explicitly coordinating the data flow between producers and consumers.

19.2 Differences Between Java Streams and Reactive Streams

Java Streams and Reactive Streams both deal with processing sequences of data, but they differ fundamentally in design, execution model, and use cases. Understanding these differences helps choose the right tool for your needs.

Conceptual Differences

Feature	Java Streams	Reactive Streams
Model	Pull-based (consumer requests data)	Push-based (producer pushes data)
Execution	Synchronous and blocking	Asynchronous and non-blocking
Data Size	Finite, typically eager processing	Potentially infinite or unbounded
Evaluation	Lazy intermediate ops, eager terminal op	Fully asynchronous, event-driven
Error Handling	Errors thrown immediately, stopping the pipeline	Errors propagated asynchronously through callbacks
Backpressure Support	None (consumer pulls at its own pace)	Built-in backpressure controls data flow
Lifecycle	Single-use pipeline, terminates after terminal op	Long-lived, supports subscription, cancellation, multiple events

Feature	Java Streams	Reactive Streams
Parallelism	Supports parallel streams via `parallel()`	Designed for concurrent, non-blocking streams

Key Technical Differences

- **Pull vs Push:** Java Streams pull elements on demand during terminal operations. Reactive Streams push data asynchronously as it becomes available, requiring Subscribers to signal readiness via backpressure.
- **Finite vs Potentially Infinite:** Java Streams are designed for finite data sources (collections, arrays). Reactive Streams can handle infinite data flows like sensor data, user events, or live feeds.
- **Error Handling:** In Java Streams, exceptions interrupt the flow immediately. Reactive Streams propagate errors through the `onError` callback, allowing subscribers to react gracefully.
- **Lifecycle and Cancellation:** Reactive Streams provide `Subscription` that allows subscribers to cancel or limit data flow. Java Streams terminate naturally after processing all elements.

Code Contrast: Filtering and Summing Integers

Java Streams (Synchronous, Pull-based):

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);

int sum = numbers.stream()
    .filter(n -> n % 2 == 0)
    .mapToInt(Integer::intValue)
    .sum();

System.out.println("Sum of even numbers: " + sum);
```

Reactive Streams (`Flow` API, Asynchronous, Push-based):

```
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.Flow.*;

public class ReactiveExample {
    public static void main(String[] args) throws InterruptedException {
        SubmissionPublisher<Integer> publisher = new SubmissionPublisher<>();

        Subscriber<Integer> subscriber = new Subscriber<>() {
            private Subscription subscription;
            private int sum = 0;

            @Override
            public void onSubscribe(Subscription subscription) {
                this.subscription = subscription;
                subscription.request(1); // Request one item
            }

            @Override
            public void onNext(Integer item) {
                if (item % 2 == 0) {
                    sum += item;
                }
                subscription.request(1); // Request next item
            }

            @Override
            public void onError(Throwable throwable) {
                throwable.printStackTrace();
            }

            @Override
            public void onComplete() {
                System.out.println("Sum of even numbers: " + sum);
            }
        };

        publisher.subscribe(subscriber);

        // Publish numbers asynchronously
        for (int i = 1; i <= 5; i++) {
            publisher.submit(i);
        }
        publisher.close();

        Thread.sleep(100); // Give time for async processing
    }
}
```

Summary

Aspect	Java Streams	Reactive Streams
Data Flow	Pull-based, synchronous	Push-based, asynchronous
Data Source	Finite collections/arrays	Potentially infinite streams
Execution Model	Lazy intermediates, eager terminal	Fully async with backpressure
Error Handling	Immediate exceptions	Asynchronous error callbacks
Lifecycle	Single use pipeline	Long-lived, subscribable stream
Use Case Examples	Batch data processing	Event-driven, live data streams

Java Streams excel in simple, synchronous batch processing, while Reactive Streams provide powerful tools for asynchronous, event-driven, and scalable stream processing—especially suited to modern distributed and interactive systems.

19.3 Simple Examples Using `Flow` API (Java 9+)

Java 9 introduced the `java.util.concurrent.Flow` API to provide a standard interface for reactive streams. This API defines four key interfaces:

- **Publisher**: Produces items asynchronously.
- **Subscriber**: Consumes items, reacting to events.
- **Subscription**: Represents the link between Publisher and Subscriber, enabling back-pressure.
- **Processor<T, R>**: Both a Subscriber and Publisher, acting as a data transformer.

Here, we'll focus on implementing a simple reactive stream pipeline by creating a custom `Publisher` and `Subscriber` and connecting them.

Basic Flow

1. **Publisher** sends data asynchronously.
2. **Subscriber** subscribes to Publisher and requests data.
3. **Subscription** controls the flow, allowing Subscriber to request or cancel.
4. Data and signals flow via `onNext()`, `onError()`, and `onComplete()` methods.

19.3.1 Example 1: `StringPublisher` and `LoggingSubscriber`

This minimal example demonstrates a publisher sending a fixed list of strings to a subscriber that logs received messages.

```java
import java.util.List;
import java.util.concurrent.Flow.*;
import java.util.concurrent.SubmissionPublisher;

public class SimpleFlowExample {
    public static void main(String[] args) throws InterruptedException {
        // Publisher that asynchronously submits strings
        SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

        // Subscriber that logs received items
        Subscriber<String> subscriber = new Subscriber<>() {
            private Subscription subscription;

            @Override
            public void onSubscribe(Subscription subscription) {
                this.subscription = subscription;
                subscription.request(1);  // Request the first item
            }

            @Override
            public void onNext(String item) {
                System.out.println("Received: " + item);
                subscription.request(1);  // Request next item after processing
            }
            @Override
            public void onError(Throwable throwable) {
                System.err.println("Error occurred: " + throwable.getMessage());
            }

            @Override
            public void onComplete() {
                System.out.println("All items processed.");
            }
        };

        // Connect subscriber to publisher
        publisher.subscribe(subscriber);

        // Publish some items asynchronously
        List.of("Hello", "Reactive", "Streams", "with", "Flow API")
            .forEach(publisher::submit);

        // Close the publisher to signal end of stream
        publisher.close();

        // Wait briefly to allow async processing to complete
        Thread.sleep(1000);
    }
}
```

Explanation:

- `SubmissionPublisher` is a built-in Publisher that handles asynchronous submission.
- The `Subscriber` implements the 4 required methods.
- On `onSubscribe()`, the Subscriber requests one item.
- After receiving each item in `onNext()`, it processes and requests the next.
- `onComplete()` signals the stream has ended.
- The `Thread.sleep()` ensures the main thread waits for async processing.

19.3.2 Example 2: Custom Publisher and Subscriber

To understand internals, here is a minimal custom Publisher and Subscriber illustrating manual data flow control.

```
import java.util.concurrent.Flow.*;

public class ManualFlowExample {
    public static void main(String[] args) {
        Publisher<String> publisher = new Publisher<>() {
            @Override
            public void subscribe(Subscriber<? super String> subscriber) {
                subscriber.onSubscribe(new Subscription() {
                    private boolean completed = false;
                    @Override
                    public void request(long n) {
                        if (!completed) {
                            subscriber.onNext("Custom");
                            subscriber.onNext("Flow");
                            subscriber.onNext("Example");
                            subscriber.onComplete();
                            completed = true;
                        }
                    }
                    @Override
                    public void cancel() {
                        System.out.println("Subscription cancelled");
                    }
                });
            }
        };

        Subscriber<String> subscriber = new Subscriber<>() {
            private Subscription subscription;
            @Override
            public void onSubscribe(Subscription subscription) {
                this.subscription = subscription;
                subscription.request(1);
            }
            @Override
            public void onNext(String item) {
                System.out.println("Received: " + item);
                subscription.request(1);
            }
            @Override
            public void onError(Throwable throwable) {
                System.err.println("Error: " + throwable.getMessage());
            }
            @Override
            public void onComplete() {
                System.out.println("Processing complete.");
            }
        };

        publisher.subscribe(subscriber);
    }
}
```

Explanation:

- The custom `Publisher` calls `subscriber.onSubscribe()` with a `Subscription` implementation.
- The `request(long n)` method pushes predefined data items on demand.
- The `Subscriber` requests one item at a time, processing and then requesting the next.
- This shows explicit backpressure: the subscriber controls when it receives more data.

19.3.3 Summary

- The `Flow` API provides a simple but powerful abstraction for asynchronous, non-blocking stream processing.
- Publishers emit data asynchronously, Subscribers consume it, and Subscriptions manage flow control.
- `SubmissionPublisher` helps create async publishers with minimal code.
- Custom implementations offer flexibility for tailored reactive pipelines.

With these basics, you can build scalable reactive data flows integrated into modern Java applications.

Chapter 20.

Integrating Streams with Functional Programming

1. Using Streams with Functional Interfaces
2. Composing Functions and Stream Pipelines
3. Currying and Partial Application Concepts

20 Integrating Streams with Functional Programming

20.1 Using Streams with Functional Interfaces

Java Streams are built on the foundation of **functional interfaces**—special interfaces with a single abstract method—that enable concise and declarative data processing pipelines. The most commonly used functional interfaces in streams are:

- **Predicate** — represents a boolean-valued function of one argument, commonly used for filtering.
- **Function<T, R>** — represents a function that accepts one argument and produces a result, used for mapping.
- **Consumer** — represents an operation that accepts a single input argument and returns no result, used for actions like printing.
- **Supplier** — represents a supplier of results, used to generate or provide values on demand.

These interfaces empower streams to express complex transformations, filters, and side effects with minimal boilerplate.

How Functional Interfaces Enable Declarative Logic in Streams

Streams leverage these functional interfaces to build pipelines where each step declares *what* should be done rather than *how*. For example:

- `filter(Predicate<T> predicate)` only lets through elements matching a condition.
- `map(Function<T, R> mapper)` transforms elements from one form to another.
- `forEach(Consumer<T> action)` performs side effects like printing.

Using lambdas or method references makes this code clean, readable, and highly expressive.

Example 1: Filtering with `Predicate`

```java
import java.util.List;

public class PredicateExample {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie", "David");

        // Filter names that start with 'A' using a lambda Predicate
        names.stream()
            .filter(name -> name.startsWith("A"))
            .forEach(System.out::println);
    }
}
```

Output:

```
Alice
```

Here, the lambda `name -> name.startsWith("A")` implements `Predicate<String>` to test each element.

Example 2: Mapping with `Function`

```java
import java.util.List;

public class FunctionExample {
    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "cherry");

        // Convert each word to uppercase using a method reference Function
        words.stream()
            .map(String::toUpperCase)
            .forEach(System.out::println);
    }
}
```

Output:

```
APPLE
BANANA
CHERRY
```

`String::toUpperCase` is a concise method reference that matches `Function<String, String>`.

Example 3: Consuming with `Consumer`

```java
import java.util.List;

public class ConsumerExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4);

        // Print each number with a prefix using a custom Consumer lambda
        numbers.stream()
            .forEach(n -> System.out.println("Number: " + n));
    }
}
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
```

The lambda `n -> System.out.println("Number: " + n)` defines a `Consumer<Integer>` for side effects.

20.1.1 Summary

By combining **functional interfaces** with streams:

- You write **declarative** code that focuses on *what* to do, not *how*.
- Lambdas and method references make stream pipelines **concise** and **expressive**.
- Core interfaces like `Predicate`, `Function`, and `Consumer` are the building blocks of common stream operations such as filtering, mapping, and consuming.

This synergy between Streams and functional interfaces is key to mastering modern Java data processing patterns.

20.2 Composing Functions and Stream Pipelines

Function composition is a powerful technique that enhances the clarity, reusability, and modularity of stream pipelines. By combining smaller functions into larger ones, you can build complex transformations or filters in a clean, declarative manner. Java's functional interfaces like `Function` and `Predicate` provide built-in methods to facilitate this composition:

- **`Function.andThen()`**: Chains two functions, applying the first, then the second.
- **`Function.compose()`**: Chains two functions, applying the second, then the first.
- **`Predicate.and()` / `Predicate.or()`**: Combines multiple boolean conditions logically.

Using these methods allows you to write reusable, composable logic blocks that can be passed directly to stream operations like `map()` or `filter()`.

How Function Composition Works

- **`andThen`**: Applies the current function, then passes the result to the next function.

- **`compose`**: Applies the argument function first, then the current function.

This distinction is important when chaining multiple transformations, enabling flexible ordering.

20.2.1 Example 1: Chained Transformations with `Function.andThen()`

Suppose you want to process a list of names by trimming whitespace and then converting to uppercase. Instead of writing a single lambda, compose two reusable functions:

```java
import java.util.List;
import java.util.function.Function;

public class FunctionCompositionExample {
    public static void main(String[] args) {
        List<String> names = List.of("  Alice ", " Bob", "Charlie  ");

        Function<String, String> trim = String::trim;
        Function<String, String> toUpperCase = String::toUpperCase;

        // Compose functions: first trim, then convert to uppercase
        Function<String, String> trimAndUpperCase = trim.andThen(toUpperCase);

        names.stream()
             .map(trimAndUpperCase)
             .forEach(System.out::println);
    }
}
```

Output:

```
ALICE
BOB
CHARLIE
```

By composing `trim` and `toUpperCase`, the code becomes modular and easy to maintain.

20.2.2 Example 2: Complex Filtering with `Predicate.and()` and `Predicate.or()`

Imagine filtering a list of integers to include only those that are positive and even, or greater than 50:

```
import java.util.List;
import java.util.function.Predicate;

public class PredicateCompositionExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(10, 25, 42, 55, 60, -4, 0);

        Predicate<Integer> isPositive = n -> n > 0;
        Predicate<Integer> isEven = n -> n % 2 == 0;
        Predicate<Integer> isGreaterThan50 = n -> n > 50;

        // Compose predicates: (positive AND even) OR greater than 50
        Predicate<Integer> complexCondition = isPositive.and(isEven).or(isGreaterThan50);

        numbers.stream()
               .filter(complexCondition)
               .forEach(System.out::println);
    }
}
```

Output:

```
10
```

```
42
55
60
```

This example cleanly expresses a complex conditional filter by composing simple predicates.

20.2.3 Benefits of Composing Functions in Stream Pipelines

- **Reusability:** Small functions can be reused across different parts of the code.
- **Readability:** Composed functions clearly express the intent without nested lambdas.
- **Maintainability:** Modifications to a single component function propagate naturally.
- **Modularity:** Separates concerns by isolating individual transformation or filtering steps.

20.2.4 Summary

Function composition using `Function.andThen()`, `compose()`, and `Predicate.and()/or()` empowers you to build elegant, reusable logic blocks for stream pipelines. This approach leads to cleaner, more declarative code when performing chained transformations or complex filtering — essential for writing maintainable and expressive Java stream-based data processing.

20.3 Currying and Partial Application Concepts

Currying and partial application are foundational concepts in functional programming that enable building flexible, reusable functions by breaking down functions with multiple arguments into chains of single-argument functions. These techniques can greatly simplify and modularize logic when working with Java Streams.

What Is Currying?

Currying transforms a function that takes multiple arguments into a sequence of functions each taking a single argument. For example, a function of two arguments `(A, B) -> R` becomes `A -> (B -> R)` — a function that returns another function.

In Java, currying is often implemented with lambdas that return other lambdas.

What Is Partial Application?

Partial application fixes a few arguments of a multi-argument function, producing a new function of fewer arguments. For example, partially applying the first argument of a twoargument function `f(A, B)` yields a single-argument function `g(B)`.

20.3.1 Why Does This Matter for Streams?

Currying and partial application help build reusable and parameterized stream operations, such as:

- Generating customized predicates (filters) with dynamic parameters.
- Creating comparators with configurable criteria.
- Building transformation functions on the fly.

This leads to concise, composable pipelines without repetitive boilerplate.

20.3.2 Example 1: Curried Predicate Generator for Filtering

Suppose you want to create a filter predicate to check if strings have length greater than a dynamic threshold. Currying helps create a reusable predicate factory:

```java
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

public class CurryingExample {
    // Curried function: int -> Predicate<String>
    static Function<Integer, Predicate<String>> lengthGreaterThan =
        length -> str -> str.length() > length;

    public static void main(String[] args) {
        List<String> words = List.of("apple", "pear", "banana", "kiwi", "pineapple");

        // Partially apply length = 4
        Predicate<String> longerThan4 = lengthGreaterThan.apply(4);

        words.stream()
             .filter(longerThan4)
             .forEach(System.out::println);
    }
}
```

Output:

```
apple
banana
pineapple
```

Here, `lengthGreaterThan` is a curried function that produces a predicate configured by a threshold. This keeps filtering logic reusable and clean.

20.3.3 Example 2: Partial Application for Custom Mapping

Imagine mapping integers to strings with a customizable prefix. You can partially apply the prefix to create reusable mappers:

```java
import java.util.List;
import java.util.function.Function;

public class PartialApplicationExample {
    // Function that takes prefix and returns function from Integer to String
    static Function<String, Function<Integer, String>> prefixer =
        prefix -> number -> prefix + number;

    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        // Partially apply prefix "Item-"
        Function<Integer, String> itemNamer = prefixer.apply("Item-");

        numbers.stream()
               .map(itemNamer)
               .forEach(System.out::println);
    }
}
```

Output:

```
Item-1
Item-2
Item-3
Item-4
Item-5
```

This pattern can generate various mapping functions dynamically based on context.

20.3.4 Summary

Currying and partial application bring powerful abstraction to stream processing by enabling you to build parameterized, reusable functions that cleanly integrate with `map()`, `filter()`, and other stream operations. By structuring lambdas as chains of single-argument functions, you achieve more expressive and maintainable pipelines that adapt flexibly to different use cases.

Chapter 21.

Debugging and Testing Stream Pipelines

1. Debugging Techniques for Streams
2. Using Peek for Debugging
3. Unit Testing Streams with JUnit

21 Debugging and Testing Stream Pipelines

21.1 Debugging Techniques for Streams

Debugging stream pipelines can be challenging due to their fluent, declarative nature and the lazy evaluation model of Java Streams. Unlike traditional loops, streams process data only when a terminal operation is invoked, making it harder to trace intermediate states and understand where logic might be failing or producing unexpected results. This section outlines practical strategies to help you effectively debug streams.

Understand Laziness and Chaining

One core challenge is that streams are **lazy**—intermediate operations like `filter()`, `map()`, and `sorted()` are not executed until a terminal operation like `collect()`, `forEach()`, or `reduce()` is called. This means:

- No data processing occurs until the pipeline is triggered.
- Errors or unexpected results may not be obvious at the operation where they happen.
- Debugging requires insight into what each stage does *when* it runs.

Recognizing this helps you avoid confusion when no output or side-effects appear during pipeline setup.

Strategy 1: Break Pipelines into Named Steps

Complex pipelines are easier to debug when broken into multiple, named intermediate streams or variables:

```java
Stream<String> source = Stream.of("apple", "banana", "cherry", "date");

Stream<String> filtered = source.filter(s -> s.length() > 5);
Stream<String> mapped = filtered.map(String::toUpperCase);
List<String> result = mapped.collect(Collectors.toList());
```

By splitting the pipeline, you can isolate issues by inspecting the contents at each step via debugging tools or additional print statements.

Strategy 2: Print Intermediate Results

Use logging or print statements to observe the elements flowing through the pipeline. The most straightforward way is to insert `.peek()` calls that execute side-effects without modifying the stream:

```java
List<String> result = Stream.of("apple", "banana", "cherry", "date")
    .filter(s -> s.length() > 5)
    .peek(s -> System.out.println("Filtered: " + s))
    .map(String::toUpperCase)
    .peek(s -> System.out.println("Mapped: " + s))
    .collect(Collectors.toList());
```

This technique lets you track what elements pass each stage without disrupting the flow.

Strategy 3: Simplify Logic Temporarily

When pipelines get too complex, temporarily simplify the logic by:

- Removing or commenting out transformations.
- Replacing lambdas with constants or identity functions.
- Testing individual operations separately.

This isolates faulty logic and ensures each component behaves as expected before combining.

Strategy 4: Use Debuggers and Breakpoints

Set breakpoints inside lambda expressions or method references in IDEs like IntelliJ IDEA or Eclipse. Modern IDEs let you step through stream processing and inspect values at runtime, even inside fluent chains.

Summary

Debugging streams requires adapting your approach compared to traditional imperative code. By breaking pipelines into manageable parts, printing intermediate data with `peek()`, simplifying complex logic, and using IDE debugging tools, you gain better visibility into how your stream processes data. This step-by-step reasoning helps uncover subtle bugs and understand transformations in your pipelines more effectively.

21.2 Using Peek for Debugging

The `peek()` method in Java Streams serves as a powerful diagnostic tool designed to help developers observe the elements flowing through a stream pipeline without altering the stream itself. It's essentially an **intermediate operation** that takes a `Consumer<T>` and allows you to perform side-effects—commonly used for logging or debugging.

What Does `peek()` Do?

- `peek()` lets you **inspect elements as they pass through the pipeline**.
- It does **not** modify the stream contents or affect the final result.
- Because it's an intermediate operation, the stream remains **lazy** until a terminal operation triggers processing.

This makes `peek()` perfect for tracing and understanding what happens at each step in your stream pipeline, especially when debugging complex transformations or filters.

When to Use `peek()`

- Insert `peek()` **between intermediate operations** to log or print elements.
- Use it to verify which elements reach a certain stage.

- Confirm filtering, mapping, or sorting behavior in complex pipelines.

Warning: Avoid Misuse in Production

While `peek()` is handy during development and debugging, avoid leaving it in production code for:

- Side effects that affect state or program flow.
- Performance overhead due to extra logging.
- Unintended consequences from relying on side effects.

If you need to perform meaningful side effects as part of your logic, use `forEach()` or proper downstream collectors instead.

Example 1: Tracing Filtered Elements

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class PeekExample1 {
    public static void main(String[] args) {
        List<String> fruits = Stream.of("apple", "banana", "cherry", "date")
            .filter(s -> s.length() > 5)
            .peek(s -> System.out.println("Filtered fruit: " + s))
            .map(String::toUpperCase)
            .peek(s -> System.out.println("Mapped fruit: " + s))
            .collect(Collectors.toList());

        System.out.println("Final list: " + fruits);
    }
}
```

Output:

```
Filtered fruit: banana
Filtered fruit: cherry
Mapped fruit: BANANA
Mapped fruit: CHERRY
Final list: [BANANA, CHERRY]
```

This example clearly shows which elements passed the filter and how they transform in the mapping stage.

Example 2: Debugging Sorted Stream

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class PeekExample2 {
    public static void main(String[] args) {
        List<Integer> numbers = Stream.of(5, 3, 9, 1, 7)
            .peek(n -> System.out.println("Original number: " + n))
            .sorted()
            .peek(n -> System.out.println("After sorting: " + n))
            .collect(Collectors.toList());

        System.out.println("Sorted list: " + numbers);
    }
}
```

Output:

```
Original number: 5
Original number: 3
Original number: 9
Original number: 1
Original number: 7
After sorting: 1
After sorting: 3
After sorting: 5
After sorting: 7
After sorting: 9
Sorted list: [1, 3, 5, 7, 9]
```

Here, `peek()` helps confirm that the stream first emits numbers in original order, then outputs them sorted, illustrating the flow clearly.

Example 3: Tracing Lazy Evaluation

```
import java.util.stream.Stream;

public class PeekExample3 {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("one", "two", "three", "four")
            .peek(s -> System.out.println("Before filter: " + s))
            .filter(s -> s.length() > 3)
            .peek(s -> System.out.println("After filter: " + s));

        System.out.println("No terminal operation yet, so no output");

        // Trigger the pipeline:
        stream.forEach(s -> System.out.println("ForEach: " + s));
    }
}
```

Output:

```
No terminal operation yet, so no output
Before filter: one
Before filter: two
Before filter: three
After filter: three
ForEach: three
Before filter: four
After filter: four
ForEach: four
```

This example shows how no processing happens before the terminal operation, highlighting **stream laziness** and how `peek()` fits into that.

21.2.1 Summary

`peek()` is a great ally for inspecting stream elements during development. It lets you trace data as it flows through filters, maps, sorts, and more, without modifying the pipeline results. Just remember to remove or limit its use in production code to avoid side-effects and performance hits. Using `peek()` wisely helps you gain insight into your stream's behavior and fix issues faster.

21.3 Unit Testing Streams with JUnit

Unit testing stream pipelines is essential to ensure your data transformations behave correctly, handle edge cases, and produce expected results. This section outlines best practices and demonstrates how to write effective JUnit tests for stream-based logic.

Key Guidelines for Testing Streams

- **Isolate stream logic:** Encapsulate stream pipelines inside well-named methods. This makes the pipelines independently testable.
- **Assert on final results:** Use assertions to check the output collections, aggregated values, or object properties.
- **Test edge cases:** Include tests for empty streams, null elements, boundary conditions, and error scenarios.
- **Avoid side effects:** Ensure that the pipeline is pure (no hidden state mutations), which simplifies testing.
- **Mock external dependencies:** If the pipeline relies on external resources (e.g., databases, file systems), mock those to keep tests focused and fast.
- **Handle exceptions gracefully:** Test how your streams behave when exceptions occur during mapping or filtering.

Example 1: Testing a Simple Filtering and Mapping Pipeline

Suppose you have a method that filters users over a certain age and returns their uppercase names:

```java
import java.util.List;
import java.util.stream.Collectors;

public class UserService {
    public List<String> getAdultUserNamesUppercase(List<User> users) {
        return users.stream()
                .filter(user -> user.getAge() >= 18)
                .map(user -> user.getName().toUpperCase())
                .collect(Collectors.toList());
    }
}

class User {
    private final String name;
    private final int age;
    public User(String name, int age) { this.name = name; this.age = age; }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```

Now, a JUnit test validating this logic:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.Arrays;
import java.util.List;

public class UserServiceTest {
    @Test
    public void testGetAdultUserNamesUppercase() {
        UserService service = new UserService();
        List<User> users = Arrays.asList(
            new User("Alice", 17),
            new User("Bob", 18),
            new User("Charlie", 20)
        );

        List<String> result = service.getAdultUserNamesUppercase(users);

        assertEquals(2, result.size());
        assertTrue(result.contains("BOB"));
        assertTrue(result.contains("CHARLIE"));
        assertFalse(result.contains("ALICE"));
    }
}
```

Example 2: Testing Aggregations and Edge Cases

Here's a method that calculates the total price of products after filtering those in stock:

```java
import java.util.List;

public class ProductService {
    public double totalInStockValue(List<Product> products) {
        return products.stream()
                .filter(Product::isInStock)
                .mapToDouble(Product::getPrice)
                .sum();
    }
}

class Product {
    private final String name;
    private final double price;
    private final boolean inStock;
    public Product(String name, double price, boolean inStock) {
        this.name = name; this.price = price; this.inStock = inStock;
    }
    public double getPrice() { return price; }
    public boolean isInStock() { return inStock; }
}
```

JUnit test including empty list and all out-of-stock cases:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class ProductServiceTest {
    @Test
    public void testTotalInStockValue() {
        ProductService service = new ProductService();

        List<Product> products = Arrays.asList(
            new Product("Pen", 1.5, true),
            new Product("Notebook", 3.0, false),
            new Product("Pencil", 0.5, true)
        );
        assertEquals(2.0, service.totalInStockValue(products), 0.0001);

        assertEquals(0.0, service.totalInStockValue(Collections.emptyList()), "Empty list should yield

        List<Product> outOfStock = Arrays.asList(
            new Product("Marker", 2.0, false)
        );
        assertEquals(0.0, service.totalInStockValue(outOfStock), "All out-of-stock should yield zero");
    }
}
```

Tips for Testing Pipelines with External Dependencies or Exceptions

- **Mock external calls**: If your pipeline fetches data from a DB or API, mock these sources using tools like Mockito to isolate logic.
- **Handle exceptions in streams**: If your pipeline might throw exceptions (e.g., parsing

failures), write tests that supply invalid input and assert exception handling or fallback behavior.
- **Use `assertThrows`** in JUnit 5 to verify expected exceptions.
- **Consider using `peek()` temporarily** to log data during test debugging, but remove before finalizing tests.

21.3.1 Summary

Writing unit tests for stream pipelines involves isolating the pipeline logic, asserting on final outputs, and considering edge cases and error handling. By structuring streams inside testable methods and using JUnit assertions effectively, you can ensure your stream-based code remains robust, maintainable, and easy to debug.