**Knowledge of Java 8 Stream Features**

Java 8’s Stream API is one of the most significant and transformative features in the Java programming language. The Stream API allows you to process sequences of elements (such as collections) in a functional style, supporting operations like map, filter, and reduce. This API facilitates working with collections in a more declarative way, allowing developers to express complex transformations and computations in a clean, readable manner.

**1. Core Concepts of Java 8 Streams**

**What is a Stream?**

A **Stream** in Java is an abstraction that allows for functional-style operations on sequences of elements (such as collections, arrays, or I/O channels). Streams can be processed in a declarative way, using methods like map(), filter(), reduce(), collect(), and more. Unlike collections, streams do not store data; instead, they represent a sequence of computations that can be performed on the data.

**Characteristics of Streams:**

* **Laziness**: Streams are evaluated lazily, meaning intermediate operations (e.g., filter, map) are not executed until a terminal operation (e.g., forEach, collect) is invoked.
* **Immutability**: Streams do not modify the underlying data structures. They provide a new stream as the result of an operation.
* **Parallelism**: Streams allow for easy parallel execution by calling parallelStream() on a collection, which automatically divides the work across multiple threads.

**2. How Streams Work Internally**

The internal mechanics of Java Streams are built around the concept of **lazy evaluation** and **pipelining**. Here’s a breakdown of how they work internally:

**Pipeline Structure:**

Streams operate on pipelines that consist of:

* **Source**: A stream is created from a data source, which can be a collection, an array, a generator, or an I/O channel.
* **Intermediate Operations**: Operations like map(), filter(), and distinct() form the intermediate part of the pipeline. These operations are **lazy**, meaning they are not performed until a terminal operation is executed. Intermediate operations return a new stream, enabling method chaining.
* **Terminal Operations**: Operations like collect(), forEach(), and reduce() trigger the processing of the pipeline. Once a terminal operation is invoked, the stream is consumed, and no further operations can be performed.

**Lazy Evaluation:**

* Intermediate operations on streams are **lazy**. This means they don’t process data until a terminal operation is invoked. The data is only consumed when necessary.
* For example, if you perform a filter() and a map() operation on a stream, Java will **not** process the elements until you call a terminal operation like collect().

**Execution Flow:**

1. **Stream Creation**: A stream is created from a collection or other data sources.
2. **Pipeline Construction**: Intermediate operations are chained together to form a pipeline, but no computation occurs at this stage.
3. **Terminal Operation**: Once a terminal operation is invoked (e.g., collect()), the stream starts processing, applying intermediate operations on the data in a sequence.

**3. Pros of Java 8 Streams**

**3.1 Concise and Readable Code**

The Stream API enables developers to express complex data manipulations and transformations in a declarative and functional style. This results in **cleaner, more readable code** that abstracts away complex iteration logic and encourages concise expression of logic.

**Example:**

java

Copy

List<String> names = Arrays.asList("Steve", "John", "Anna");

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("S"))

.map(String::toUpperCase)

.collect(Collectors.toList());

This code is much more declarative compared to traditional loops.

**3.2 Functional Style**

Streams enable **functional programming** in Java. By leveraging higher-order functions like map(), filter(), reduce(), etc., developers can express complex data transformations in a functional manner, which can be easier to reason about, test, and maintain.

**3.3 Parallel Processing**

With **minimal effort**, Java Streams can execute in parallel. The parallelStream() method splits the task across multiple threads, enabling automatic multi-core utilization for data-intensive tasks. This makes it easy to scale up operations without needing to manually manage threads.

**Example:**

java

Copy

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int sum = numbers.parallelStream()

.mapToInt(Integer::intValue)

.sum();

**3.4 Improved Performance (Lazy Evaluation)**

Due to lazy evaluation, Streams can avoid unnecessary computations. Intermediate operations are only applied when necessary, and short-circuiting operations (such as findFirst(), anyMatch(), etc.) can terminate the pipeline early.

**3.5 Better Code Maintenance**

By using streams, developers can avoid writing boilerplate code for looping and filtering collections. This leads to fewer bugs and easier code maintenance.

**3.6 Immutable Collections and Thread Safety**

Streams encourage the use of immutable collections and make it easier to avoid issues associated with concurrency. The stream processing model allows parallelism, and the immutability of the underlying collections ensures that data is not inadvertently modified during stream processing.

**4. Cons of Java 8 Streams**

**4.1 Performance Overhead (For Simple Operations)**

While Streams can provide performance benefits for large data sets or complex operations, for **simple operations**, the overhead of creating streams, setting up pipelines, and invoking multiple method calls can **actually degrade performance** compared to traditional loops.

For example, filtering and transforming a small collection might be slower when using Streams compared to a simple for-loop due to the overhead involved.

**4.2 Debugging Difficulty**

Since Streams rely heavily on chaining and functional programming concepts, **debugging can be more challenging** compared to traditional loops. Stack traces in parallel streams, in particular, can sometimes be difficult to interpret.

**4.3 Memory Consumption**

Each intermediate operation (e.g., filter(), map()) results in a new stream, leading to the creation of additional objects in memory. This can lead to higher memory consumption, especially when using large data sets.

**4.4 Parallel Streams Overhead**

Parallel streams introduce thread management overhead. While they are beneficial for large datasets, the overhead of managing multiple threads can outweigh the performance benefits in smaller datasets or simple computations. If not used properly, parallel streams can **increase latency** rather than improve performance.

**4.5 Not Ideal for All Use Cases**

Streams are ideal for declarative transformations on data. However, they are **not well-suited for all types of tasks**, such as those that require intricate stateful operations, or tasks that need to be executed in a specific order (e.g., processing dependent data). In such cases, traditional imperative approaches (e.g., loops) might still be better.

**5. Why Were Streams Added in Java 8?**

Java 8 introduced Streams to address several long-standing issues and limitations in Java's approach to working with collections:

**5.1 Lack of Declarative Programming Style**

Before Java 8, Java was largely **imperative**, and developers had to manually manage iteration logic (e.g., using loops) to manipulate collections. The Stream API introduced a **functional, declarative programming style** to handle data processing more easily and concisely.

**5.2 Limited Support for Parallelism**

Parallelism was available in Java but was difficult to implement. Developers had to manually manage threads or use external libraries. Stream’s parallelStream() method provided **native parallelism** with minimal effort, enabling developers to leverage multiple cores without worrying about the underlying thread management.

**5.3 Boilerplate Code in Collection Handling**

Before Streams, developers often wrote a lot of **boilerplate code** to filter, transform, and collect data. This not only made the code more complex but also more error-prone. Streams allow developers to express such transformations in a **clear, concise way** with fewer lines of code.

**5.4 Avoiding Mutability Issues**

Streams enforce **immutable collections** and encourage the use of non-mutating operations. This helps avoid **concurrency problems** and the potential side effects of mutating data while iterating over collections.

**6. Internal Working of Streams (JVM Perspective)**

**6.1 Stream Creation**

When you create a stream from a collection, it’s simply a wrapper around the data source. Internally, the stream does not copy the collection's data but operates directly on the original collection through a series of computations. This means the stream is only a **view** of the data, not the data itself.

**6.2 Pipeline Execution**

Stream operations are divided into **intermediate operations** (such as filter() and map()) and **terminal operations** (such as collect() and forEach()). Intermediate operations are **lazy**, meaning they are not executed until a terminal operation is invoked.

When the terminal operation is invoked, the stream processes data in a **pipelined manner**. Each intermediate operation is executed only once per element in the stream, and it does not store the results of intermediate operations in memory unless explicitly requested (e.g., via collect()).

**6.3 Parallel Streams**

Parallel streams in Java utilize the **ForkJoinPool**, which splits the stream into smaller chunks and processes them in parallel across multiple threads. When parallel processing is applied, the elements of the stream are divided among multiple threads for concurrent execution. However, this requires careful handling of thread safety to ensure consistency.

**Stream Features for Experienced Java Professionals**

Java 8 Streams were added to streamline and modernize the way developers process collections and large datasets, aligning Java with functional programming paradigms. They provide the power of **declarative data manipulation**, **parallelism**, and **better readability**, with an internal model focused on **lazy evaluation** and **method chaining**.

**Advantages** include cleaner code, improved parallel processing capabilities, and better memory management with immutable collections. However, developers must be cautious of potential **performance issues** for smaller datasets, increased **memory overhead**, and the complexity of debugging parallel streams.

For **experienced developers**, Streams represent a significant shift in how we write Java code, promoting a shift from an imperative to a **functional** programming approach. Proper understanding of Streams and their inner workings—especially around lazy evaluation, parallelism, and memory usage—can make you far more effective in designing performant, modern Java applications.

**Lazy Evaluation in Java 8 Streams**

Lazy evaluation is one of the key features of Java 8 Streams, and it fundamentally changes how data is processed in Java. Understanding lazy evaluation in Streams is crucial for optimizing performance and properly leveraging Streams in your code.

**What is Lazy Evaluation?**

Lazy evaluation refers to the idea that **computations are deferred until they are absolutely necessary**. In the context of Java Streams, it means that intermediate operations (such as map(), filter(), distinct()) are not immediately executed when invoked. Instead, they are **set up in a pipeline** and are executed only when a terminal operation (such as collect(), forEach(), reduce()) is triggered.

I**ntermediate operations define the computation**, but they do not execute until a terminal operation is invoked. This allows for more efficient processing, as only the data necessary for the terminal operation will be processed.

**How Does Lazy Evaluation Work in Java 8 Streams?**

To better understand lazy evaluation in Streams, let’s break down the flow of operations:

1. **Stream Creation**: A stream is created from a data source (like a collection, array, or generator).
   * Example: List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
   * When we call numbers.stream(), it creates a stream but does not start processing the elements immediately.
2. **Intermediate Operations**: These operations, like map(), filter(), distinct(), and flatMap(), are **lazy**. They build a chain of transformations but do not perform any computation.
   * Example: numbers.stream().filter(x -> x > 2).map(x -> x \* 2);
   * Here, we are defining a pipeline of operations: filter() and map(). But none of the actual operations are performed until a terminal operation is invoked.
3. **Terminal Operation**: A terminal operation, like collect(), forEach(), or reduce(), **triggers the execution** of the intermediate operations. This is where the actual data processing happens.
   * Example: numbers.stream().filter(x -> x > 2).map(x -> x \* 2).collect(Collectors.toList());
   * The terminal operation (collect()) triggers the execution of the operations defined in the pipeline, and only then are the elements processed according to the transformations specified in the intermediate operations.
4. **Processing Data**: When the terminal operation is invoked, the stream processes the data one element at a time, **applying each intermediate operation lazily**. It only processes the elements needed by the terminal operation, and it does so in a **pipelined manner**, applying each operation step-by-step.
   * This means that for each element in the stream, **filtering, mapping, or any other transformation is performed in sequence** without reprocessing the entire dataset.
   * The stream pipeline executes **only when needed**, and as a result, it avoids unnecessary computation.

**Example of Lazy Evaluation:**

Simple example to understand lazy evaluation in action:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> result = numbers.stream()

.filter(x -> x > 2) // Intermediate operation: filter

.map(x -> x \* 2) // Intermediate operation: map

.collect(Collectors.toList()); // Terminal operation: collect

System.out.println(result); // Output: [6, 8, 10]

* **Step-by-Step Breakdown**:
  + When we call numbers.stream(), no data is processed yet.
  + The .filter(x -> x > 2) and .map(x -> x \* 2) operations define how we want to process the data, but they are not executed yet.
  + Only when the terminal operation .collect(Collectors.toList()) is invoked, the actual data processing begins.
  + The filter() operation checks each element for the condition x > 2, and only then does the map() operation double the values that pass the filter.

**Key Characteristics of Lazy Evaluation in Streams:**

1. **Pipeline Definition**: Intermediate operations (like map(), filter()) **define** the operations to be performed but are **not executed** until a terminal operation is triggered.
2. **On-Demand Execution**: Computation happens **on-demand**, meaning that the stream processes elements only when required by the terminal operation.
3. **Short-Circuiting**: Lazy evaluation allows for short-circuiting, which means that operations can stop early if a certain condition is met.
   * Example: anyMatch() or findFirst() operations can **stop the processing as soon as the first matching element is found**, thus saving time and resources.
4. **Memory Efficiency**: Lazy evaluation allows streams to be **memory efficient** because it doesn't require holding all data in memory at once. Intermediate operations like filtering and mapping do not create additional data structures; they just transform elements on the fly.

**Short-Circuiting Operations (An Example of Lazy Evaluation):**

Some intermediate operations in streams can short-circuit, meaning they will stop the pipeline as soon as a result is found. This is a key advantage of lazy evaluation in Java Streams.

**Example**:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

boolean result = numbers.stream()

.filter(x -> x > 2)

.anyMatch(x -> x == 4); // Short-circuiting operation

System.out.println(result); // Output: true

* The anyMatch() operation immediately stops processing when the first match is found (x == 4), avoiding unnecessary evaluation of further elements in the stream.
* This short-circuiting behavior significantly improves performance for large datasets or complex conditions.

**Advantages of Lazy Evaluation in Java 8 Streams**

**1. Performance Optimization:**

Lazy evaluation helps in **optimizing performance** by deferring computation until absolutely necessary. For example, if a stream pipeline is set up with multiple operations, the data is processed only when required by the terminal operation. This avoids unnecessary operations, and only the relevant data is computed.

**2. Short-Circuiting:**

Lazy evaluation enables **early termination** in certain scenarios through short-circuiting operations like findFirst(), anyMatch(), or noneMatch(). These operations stop processing as soon as the condition is satisfied, reducing the time spent on unnecessary computations.

**3. Memory Efficiency:**

Since the elements are processed one by one, and intermediate operations don’t produce a new collection, memory consumption remains low. The stream **doesn’t store the data**; it processes each element as it flows through the pipeline, which can be particularly useful when dealing with large datasets.

**4. Chaining Operations:**

The lazy evaluation model allows for **method chaining** in a natural and expressive way. Intermediate operations are composed in a pipeline, making the code concise and readable, without the need for complex loops or temporary collections.

**Disadvantages and Considerations**

**1. Increased Complexity for Debugging:**

While lazy evaluation improves performance, it can make debugging **more difficult**. Since the operations aren’t executed immediately, it can be harder to understand where the logic fails or where performance bottlenecks occur. Tools like peek() or logging inside the pipeline can help in debugging, but it can still be tricky.

**2. Overhead for Small Datasets:**

For small datasets or very simple operations, the overhead of setting up the stream pipeline and invoking the terminal operation may outweigh the benefits of lazy evaluation. In such cases, **traditional loops** may outperform streams in terms of both clarity and speed.

**3. Parallel Streams:**

While lazy evaluation benefits parallel streams, improper usage can lead to problems like **race conditions**, thread contention, or increased latency due to the overhead of managing multiple threads. Parallelism should be used only when necessary and in performance-critical applications.

**4. Garbage Collection:**

In cases where the stream is processing large amounts of data or using a high number of intermediate operations, the **garbage collector** might be impacted by the creation of additional objects (such as the internal representations of stream elements or the result of intermediate operations).

**Summary**

**Lazy evaluation** is a powerful concept in Java 8 Streams that helps with **performance optimization**, **memory efficiency**, and **concise code**. By deferring execution of intermediate operations until a terminal operation is triggered, Java Streams allow developers to write functional, declarative code without incurring the performance penalties of immediate evaluation.

Streams in Java 8 provide **short-circuiting** operations that enable early termination of operations, further optimizing performance, especially with large data sets. However, while this feature provides powerful optimizations, it does add complexity for debugging and may introduce overhead in small operations or simple use cases.

Understanding **lazy evaluation** is essential for Java developers, especially when working on large-scale applications or performance-critical code, and can be a key factor in passing interviews for senior-level Java roles.

**How Parallel Streams Work Internally with ForkJoinPool**

Java 8 introduced the concept of **parallel streams**, which is a feature designed to help developers harness the power of multi-core processors with minimal effort. By simply calling parallelStream() on a collection, Java will internally manage parallel execution. This parallelism is powered by the **ForkJoinPool**, a framework provided by Java to efficiently manage parallel tasks.

Let’s dive deep into how **parallel streams** work internally and how **ForkJoinPool** is utilized to manage and execute the tasks in parallel.

**1. ForkJoinPool: The Backbone of Parallel Streams**

The **ForkJoinPool** is the key component that drives parallelism in Java streams. It's a special type of thread pool introduced in Java 7, designed for tasks that can be broken down into smaller tasks (recursively) and run in parallel, making it ideal for parallel streams.

**Key Characteristics of ForkJoinPool:**

* **Work-Stealing Algorithm**: ForkJoinPool uses a **work-stealing** algorithm where idle threads try to "steal" work from busy threads, balancing the load.
* **Lightweight Threads (ForkJoinWorkerThread)**: The pool uses lightweight threads to manage tasks efficiently. These threads are more lightweight compared to normal threads, which helps with scalability and better resource utilization.
* **Recursive Task Execution**: ForkJoinPool is well-suited for tasks that can be broken down into smaller, independent subtasks (divide and conquer), which is the core idea behind parallel streams.

**2. Parallel Streams Internally Using ForkJoinPool**

Parallel streams leverage **ForkJoinPool** to divide the workload into smaller tasks that can be executed concurrently. Here’s a step-by-step breakdown of how parallel streams work internally:

**Step 1: Initiating the Parallel Stream**

When you call parallelStream() on a collection (e.g., List, Set), Java **wraps the stream** in a special **parallel stream** mode. Internally, this tells the system that operations on this stream should be executed in parallel rather than sequentially.

java

Copy

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.parallelStream()

.map(x -> x \* 2) // Operation to be performed in parallel

.forEach(System.out::println);

The stream pipeline is set up with **intermediate operations** (like map) and **terminal operations** (like forEach), but no actual computation occurs yet. It’s still a **logical pipeline**.

**Step 2: Splitting the Workload (Spliterator)**

Once the parallel stream is set up and a **terminal operation** (e.g., forEach, collect, reduce) is invoked, the stream needs to split the work into smaller pieces for parallel execution.

* **Spliterator**: A Spliterator is responsible for dividing the source data into smaller chunks. It is similar to an iterator but is designed for efficient splitting of data into sub-collections for parallel processing.
  + The Spliterator splits the data into **multiple segments** (chunks) based on the available parallelism.
  + The size of each chunk is typically determined by the **number of available cores** in the machine, or it’s dynamically adjusted depending on the data and task size.

**Step 3: Submitting Tasks to ForkJoinPool**

After splitting the data, Java submits each chunk to the **ForkJoinPool** for processing. The **ForkJoinPool** is the thread pool that manages the execution of these tasks. The pool contains multiple threads (often corresponding to the number of available CPU cores) that handle the tasks in parallel.

* **Recursive Task Execution**: The stream's internal implementation **recursively divides** the task until it reaches a threshold of granularity (typically one element per task).
* **Task Submission**: Each chunk of data (or task) is submitted to the ForkJoinPool using the fork() method, and the results of each chunk are **joined** back together after processing using the join() method.

**Step 4: Forking and Joining Tasks**

Each thread in the **ForkJoinPool** works on processing the chunks of data. These threads apply the operations on the stream (e.g., map, filter) to each chunk. Once the threads finish processing, they **join** their results back to a final output.

* **Forking**: Each task is "forked" (or broken down) into smaller subtasks that can be executed independently.
* **Joining**: Once the subtasks are processed, the results are joined back together in the correct order, combining the partial results from all the threads.

This is how a divide-and-conquer approach is used to parallelize the computation efficiently.

**Step 5: Work-Stealing**

The **work-stealing algorithm** in the **ForkJoinPool** allows threads to steal work from other threads if they become idle. This helps balance the workload dynamically, ensuring that no thread is left doing nothing while others are still busy.

* If a thread finishes its task, it can **steal** a task from another thread that is still working. This minimizes idle time and maximizes throughput, especially when the tasks are unbalanced or the data is not evenly distributed.

**3. Example of Parallel Stream Working Internally**

Let's take a simple example and break down how it would work internally using **ForkJoinPool**.

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.parallelStream()

.map(x -> x \* 2)

.forEach(System.out::println);

1. **Parallel Stream Creation**:
   * When numbers.parallelStream() is called, Java creates a **parallel stream** and internally sets up a **Spliterator** for the data (numbers).
   * The **Spliterator** will split the list into smaller chunks (say, one chunk per number if the dataset is small or more if the CPU cores are higher).
2. **Task Splitting (Forking)**:
   * The **Spliterator** divides the numbers list into **smaller chunks**.
   * If there are 4 CPU cores available, the data will be split into 4 parts (e.g., {1}, {2}, {3}, {4,5}) and assigned to separate threads in the **ForkJoinPool**.
3. **Parallel Task Execution**:
   * Each chunk is processed by a different thread in the **ForkJoinPool**.
   * For example:
     + Thread 1 processes {1}, mapping it to {2}.
     + Thread 2 processes {2}, mapping it to {4}.
     + Thread 3 processes {3}, mapping it to {6}.
     + Thread 4 processes {4,5}, mapping them to {8, 10}.
4. **Joining Results**:
   * After processing, the threads **join** their results back together. Since there is no specific ordering required in this case (for forEach), they are combined and printed.
5. **Work Stealing**:
   * If one of the threads finishes early (say, Thread 1 finishes processing {1} quickly), it might **steal** work from a thread that is still working (e.g., Thread 4, which might still be processing {4,5}).
   * This minimizes idle time for threads and ensures that all cores are fully utilized.

**4. Advantages and Challenges of Parallel Streams with ForkJoinPool**

**Advantages:**

1. **Automatic Parallelism**: Java makes it very easy to parallelize your code by simply calling parallelStream(). It abstracts away the complexity of thread management.
2. **Work Stealing**: The ForkJoinPool’s work-stealing algorithm ensures that threads are efficiently utilized, reducing the chances of thread idleness and ensuring high throughput.
3. **Simplified Code**: Parallelism is introduced declaratively, without requiring manual thread management, which makes your code cleaner and easier to maintain.

**Challenges:**

1. **Overhead for Small Tasks**: For small datasets or simple operations, the overhead of managing multiple threads can outweigh the benefits of parallelism, making sequential streams faster.
2. **Non-Deterministic Results**: Parallel streams may not guarantee the order of execution. For operations that depend on the order of elements (e.g., forEachOrdered()), you need to ensure the correct ordering in the final result.
3. **Thread Safety**: Since parallel streams use multiple threads, any mutable shared state should be handled carefully to avoid **concurrent modification issues**. You may need to ensure thread safety for shared variables (e.g., using AtomicInteger or synchronized blocks).

**5. Performance Considerations and Best Practices**

* **Small Datasets**: Parallel streams should not be used for small datasets where the overhead of splitting and managing tasks can be larger than the performance gains from parallelism.
* **Stateful Operations**: Avoid stateful operations (like collect()) in parallel streams, as they can lead to unpredictable behavior. Instead, use **stateless operations** that do not modify shared data.
* **Use forEachOrdered for Order**: If the operation depends on the order of elements, use forEachOrdered() instead of forEach() to preserve the order when processing elements in parallel.
* **ForkJoinPool Configuration**: By default, parallel streams use the common ForkJoinPool, but you can configure it to use a custom thread pool if necessary (e.g., with ForkJoinPool.commonPool()).

**Conclusion**

Java's **parallel streams** make it easier to parallelize your code with minimal effort by leveraging the **ForkJoinPool**. ForkJoinPool handles splitting the tasks, managing the threads, and utilizing the work-stealing algorithm to ensure efficient execution. However, parallel streams come with trade-offs and should be used judiciously, especially in performance-critical applications. Understanding how ForkJoinPool works and the overhead involved with parallelism will help you decide when to use parallel streams effectively and when to avoid them.

**Java 8 Stream Intermediate Operations**

Intermediate operations in Java Streams are used to transform, filter, or otherwise modify the data within the stream. These operations are **lazy**, meaning they are not executed until a terminal operation (such as collect(), forEach(), reduce()) is invoked. Each intermediate operation returns a new stream, allowing method chaining.

Here’s a list of **all the major intermediate operations** in Java 8 Streams, along with their usage, a brief description, and examples:

**1. filter()**

* **Usage**: Filters elements of a stream based on a given condition (predicate). It returns a new stream containing only the elements that satisfy the condition.
* **Description**: It takes a Predicate<T> and returns a stream of elements that match the given condition.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> filtered = numbers.stream()

.filter(n -> n > 2)

.collect(Collectors.toList()); // Output: [3, 4, 5]

**2. map()**

* **Usage**: Transforms each element in the stream using a provided function.
* **Description**: Takes a Function<T, R> and returns a stream of elements that are the result of applying the function to each element of the original stream.

**Example:**

List<String> names = Arrays.asList("John", "Jane", "Jack");

List<String> uppercased = names.stream()

.map(String::toUpperCase)

.collect(Collectors.toList()); // Output: [JOHN, JANE, JACK]

**3. flatMap()**

* **Usage**: Flattens a stream of collections (or other complex structures) into a single stream.
* **Description**: It takes a Function<T, Stream<R>> and flattens the result into a single stream. Useful when dealing with nested collections.

**Example:**

List<List<Integer>> numbers = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5, 6)

);

List<Integer> flattened = numbers.stream()

.flatMap(List::stream)

.collect(Collectors.toList()); // Output: [1, 2, 3, 4, 5, 6]

**4. distinct()**

* **Usage**: Removes duplicate elements from the stream.
* **Description**: Returns a new stream that contains only distinct elements, according to their equals() method.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 2, 4, 1);

List<Integer> distinct = numbers.stream()

.distinct()

.collect(Collectors.toList()); // Output: [1, 2, 3, 4]

**5. sorted()**

* **Usage**: Sorts the elements of the stream in natural order or using a provided comparator.
* **Description**: Returns a new stream with the elements sorted either in their natural order or according to a comparator.

**Example (natural order):**

List<Integer> numbers = Arrays.asList(5, 3, 8, 1);

List<Integer> sorted = numbers.stream()

.sorted()

.collect(Collectors.toList()); // Output: [1, 3, 5, 8]

**Example (custom comparator):**

List<String> words = Arrays.asList("apple", "banana", "cherry");

List<String> sorted = words.stream()

.sorted((s1, s2) -> s2.compareTo(s1))

.collect(Collectors.toList()); // Output: [cherry, banana, apple]

**6. peek()**

* **Usage**: Performs an action on each element of the stream without modifying the elements. It's primarily used for debugging purposes.
* **Description**: Returns a new stream consisting of the elements of the original stream, but with the action applied to each element.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream()

.peek(n -> System.out.println("Processing: " + n))

.collect(Collectors.toList());

* Note: peek() is typically used for debugging or logging and should not be used for modifying state.

**7. limit()**

* **Usage**: Limits the number of elements in the stream to a specified number.
* **Description**: Takes a number n and returns a stream with the first n elements of the original stream.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> limited = numbers.stream()

.limit(3)

.collect(Collectors.toList()); // Output: [1, 2, 3]

**8. skip()**

* **Usage**: Skips the first n elements of the stream and returns a new stream.
* **Description**: Takes a number n and returns a stream that skips the first n elements, effectively creating a sub-stream.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> skipped = numbers.stream()

.skip(2)

.collect(Collectors.toList()); // Output: [3, 4, 5]

**9. takeWhile() (Java 9)**

* **Usage**: Returns a new stream with elements taken as long as they match the provided predicate.
* **Description**: This method is available from **Java 9** and allows taking elements from the stream until the predicate no longer matches.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> taken = numbers.stream()

.takeWhile(n -> n < 4)

.collect(Collectors.toList()); // Output: [1, 2, 3]

**10. dropWhile() (Java 9)**

* **Usage**: Skips elements as long as they match the predicate and returns the remaining elements after the predicate no longer matches.
* **Description**: This method is available from **Java 9** and allows dropping elements until a predicate no longer holds true.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> dropped = numbers.stream()

.dropWhile(n -> n < 4)

.collect(Collectors.toList()); // Output: [4, 5]

**11. flatMapToInt() / flatMapToDouble() / flatMapToLong()**

* **Usage**: These methods are used to convert elements of the stream into a primitive type stream (such as IntStream, DoubleStream, or LongStream), which is necessary for better performance when working with primitive data types.
* **Description**: They flatten a stream of objects into a stream of primitive types.

**Example:**

List<List<Integer>> numbers = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5, 6)

);

numbers.stream()

.flatMapToInt(list -> list.stream().mapToInt(Integer::intValue))

.forEach(System.out::println); // Output: 1 2 3 4 5 6

**12. mapToInt() / mapToDouble() / mapToLong()**

* **Usage**: Converts the stream to a primitive stream (like IntStream, DoubleStream, or LongStream) based on the type of operation.
* **Description**: These are special forms of map() that return primitive streams, which are more memory efficient and faster than using object-based streams.

**Example:**

List<String> numbers = Arrays.asList("1", "2", "3", "4");

int sum = numbers.stream()

.mapToInt(Integer::parseInt)

.sum(); // Output: 10

**13. reduce()**

* **Usage**: Combines elements of the stream into a single result.
* **Description**: It takes a binary operator and performs a reduction on the stream elements, applying the operator cumulatively to the elements. It’s often used for sum, product, or concatenation.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

int sum = numbers.stream()

.reduce(0, Integer::sum); // Output: 10

**14. collect()**

* **Usage**: Collects the elements of the stream into a collection (e.g., List, Set) or another form (like a Map).
* **Description**: It’s a terminal operation that transforms the stream into a different form using a **Collector**.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

List<Integer> collected = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList()); // Output: [2, 4]

**Conclusion**

The **intermediate operations** in Java 8 Streams provide a powerful way to transform, filter, and manipulate data in a declarative manner. They offer flexibility, and when combined with terminal operations, they provide a streamlined and efficient way to process large datasets.

**Common Intermediate Operations**:

* filter()
* map()
* flatMap()
* distinct()
* sorted()
* peek()
* limit()
* skip()

**Java 8 Stream Terminal Operations**

Terminal operations in Java 8 Streams trigger the processing of the stream pipeline. Unlike intermediate operations (which are lazy), terminal operations cause the stream to be consumed and result in a final computation or collection of data. Once a terminal operation is invoked, the stream is considered **consumed**, and no further operations can be performed on it.

Here’s a list of **all major terminal operations** in Java 8 Streams, along with their usage, a brief description, and examples:

**1. forEach()**

* **Usage**: Performs an action for each element of the stream.
* **Description**: It’s an **action-oriented terminal operation** that applies a given action to each element of the stream. The action is performed in the order of the stream unless using parallelStream(), where order is not guaranteed.

**Example:**

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.stream()

.forEach(name -> System.out.println("Hello, " + name));

// Output: Hello, John

// Hello, Jane

// Hello, Jack

**2. collect()**

* **Usage**: Collects the elements of the stream into a collection or other form (e.g., a List, Set, Map, or a custom result).
* **Description**: The most common terminal operation for gathering results from the stream into a **mutable container**. It takes a Collector<T, A, R> argument that defines how the elements are collected.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> result = numbers.stream()

.filter(n -> n > 2)

.collect(Collectors.toList()); // Output: [3, 4, 5]

**3. reduce()**

* **Usage**: Combines elements of the stream into a single result using a **binary operator**.
* **Description**: Takes an initial identity value and a **binary operator** that reduces the elements of the stream. It’s often used for summing, multiplying, or concatenating values.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()

.reduce(0, Integer::sum); // Output: 15

**4. count()**

* **Usage**: Returns the number of elements in the stream.
* **Description**: It’s a **non-intermediate operation** that counts the number of elements in the stream and returns the count as a long. It’s useful when you want to know how many elements exist in the stream.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

long count = numbers.stream()

.filter(n -> n % 2 == 0)

.count(); // Output: 2 (because 2 and 4 are even)

**5. anyMatch()**

* **Usage**: Returns true if any element of the stream matches the provided predicate.
* **Description**: A **short-circuiting operation** that checks whether any element in the stream matches the given condition. It stops as soon as a match is found, making it more efficient than checking all elements.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

boolean hasEven = numbers.stream()

.anyMatch(n -> n % 2 == 0); // Output: true (because 2, 4 are even)

**6. allMatch()**

* **Usage**: Returns true if all elements of the stream match the provided predicate.
* **Description**: A **short-circuiting operation** that checks whether **all** elements match the given condition. It stops as soon as it finds an element that does not match the predicate.

**Example:**

List<Integer> numbers = Arrays.asList(2, 4, 6, 8);

boolean allEven = numbers.stream()

.allMatch(n -> n % 2 == 0); // Output: true

**7. noneMatch()**

* **Usage**: Returns true if no elements of the stream match the provided predicate.
* **Description**: Similar to allMatch() and anyMatch(), this operation checks that **no** elements in the stream match the given predicate. It is a **short-circuiting operation**.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

boolean noneEven = numbers.stream()

.noneMatch(n -> n % 2 != 0); // Output: false (because 1, 3, 5 are odd)

**8. findFirst()**

* **Usage**: Returns the **first element** of the stream, or an empty Optional if no elements are present.
* **Description**: Returns an Optional<T> that contains the first element of the stream. It is a **short-circuiting operation**.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> first = numbers.stream()

.filter(n -> n > 2)

.findFirst(); // Output: Optional[3]

**9. findAny()**

* **Usage**: Returns **any element** from the stream (in parallel streams, it may return any element from the subset of elements processed).
* **Description**: Similar to findFirst(), but in the case of parallel streams, it may return any element in the stream, not necessarily the first one.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> any = numbers.stream()

.filter(n -> n > 2)

.findAny(); // Output: Optional[3] (but can be 4 or 5 in parallel stream)

**10. max()**

* **Usage**: Returns the maximum element of the stream according to the provided comparator.
* **Description**: The max() operation uses a **Comparator** to determine which element is the largest in the stream. It returns an Optional<T>, as the stream could be empty.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> max = numbers.stream()

.max(Integer::compareTo); // Output: Optional[5]

**11. min()**

* **Usage**: Returns the minimum element of the stream according to the provided comparator.
* **Description**: Similar to max(), but returns the smallest element based on the comparator. It also returns an Optional<T> in case the stream is empty.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> min = numbers.stream()

.min(Integer::compareTo); // Output: Optional[1]

**12. toArray()**

* **Usage**: Collects the elements of the stream into an array.
* **Description**: Converts the stream into an array of the same type, using an array generator function that matches the stream’s element type.

**Example:**

List<String> names = Arrays.asList("John", "Jane", "Jack");

String[] nameArray = names.stream()

.toArray(String[]::new); // Output: ["John", "Jane", "Jack"]

**13. forEachOrdered()**

* **Usage**: Similar to forEach(), but guarantees that elements will be processed in the encounter order (important for parallel streams).
* **Description**: When used with parallel streams, forEachOrdered() ensures that elements are processed in the order they appear in the stream. Without this, parallel processing might change the order of processing.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream()

.forEachOrdered(System.out::println); // Output: 1 2 3 4 5 (preserving order)

**14. toList() (in Collectors)**

* **Usage**: Collects the stream elements into a List.
* **Description**: This is a commonly used collector in the Collectors class, which accumulates the elements of the stream into a list.

**Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> result = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList()); // Output: [2, 4]

**15. joining() (in Collectors)**

* **Usage**: Concatenates the elements of the stream into a single String.
* **Description**: Collects the stream into a single String, optionally using a separator, a prefix, and a suffix.

**Example:**

List<String> words = Arrays.asList("Java", "Streams", "are", "awesome");

String result = words.stream()

.collect(Collectors.joining(" ")); // Output: "Java Streams are awesome"

**Conclusion**

Java 8 Streams provide a wide range of **terminal operations** to perform calculations, transformations, and aggregations on the data. Here’s a recap of the most common terminal operations:

* **forEach()**
* **collect()**
* **reduce()**
* **count()**
* **anyMatch()**
* **allMatch()**
* **noneMatch()**
* **findFirst()**
* **findAny()**
* **max()**
* **min()**
* **toArray()**
* **forEachOrdered()**
* **joining()**
* **toList()**

These terminal operations are essential for processing streams effectively, and they allow you to perform a variety of tasks, from simple aggregation to complex transformations. Understanding and choosing the right terminal operation for your use case is key to optimizing the performance and readability of your Java Stream code.

**1. Explain the difference between intermediate and terminal operations in Java Streams.**

**Answer Expected:**

* **Intermediate operations**: These operations are **lazy** and return a new stream. They define the transformation of data but do not trigger processing. Examples include map(), filter(), distinct().
* **Terminal operations**: These operations **trigger the execution** of the stream pipeline and return a final result. Examples include collect(), reduce(), forEach().
* **Key Difference**: Intermediate operations are not executed until a terminal operation is invoked. Terminal operations consume the stream.

**2. How does lazy evaluation work in Java 8 streams? Why is it important?**

**Answer Expected:**

* **Lazy evaluation** means that the operations in the stream pipeline (e.g., map(), filter()) are not executed until a terminal operation is called. It prevents unnecessary processing and allows for **efficient chaining** of operations.
* **Example**: If the terminal operation is collect(), intermediate operations such as filter() and map() will only process data when needed.
* **Importance**: This improves **performance** by avoiding unnecessary intermediate operations. For example, map() will not execute if the stream is already short-circuited by findFirst().

**3. What is the difference between forEach() and forEachOrdered() in streams?**

**Answer Expected:**

* **forEach()**: Iterates over the elements in the stream, but in parallel streams, the order of processing is not guaranteed.
* **forEachOrdered()**: Guarantees that the order of elements is maintained even in parallel streams. This operation is more expensive in parallel streams because it ensures the order is preserved.

**Example**:

numbers.stream().forEach(System.out::println); // Order not guaranteed in parallel stream

numbers.stream().forEachOrdered(System.out::println); // Order guaranteed even in parallel stream

**4. What are the advantages and disadvantages of parallel streams?**

**Answer Expected:**

* **Advantages**:
  + **Improved performance** for large data sets on multi-core processors due to parallel execution.
  + Simplifies parallelization; the developer doesn't need to manage threads manually.
  + Automatically leverages the ForkJoinPool, providing **work-stealing** for efficient load balancing.
* **Disadvantages**:
  + **Overhead for small datasets**: Parallelization introduces overhead due to the splitting of data, which can actually **decrease performance** for small or simple datasets.
  + **Non-deterministic results**: Parallel streams may change the order of processing, so for operations that depend on the order, parallelism should be avoided.
  + **Thread safety issues**: Any shared mutable state between threads can lead to issues unless handled properly (e.g., using Atomic classes or synchronization).

**5. What is Spliterator in Java 8 streams? How is it used internally in parallel streams?**

**Answer Expected:**

* **Spliterator** is an interface in the Java 8 Streams API that is used for **splitting** and traversing the elements of a stream. It is designed for efficient processing of large collections and parallel processing.
* In parallel streams, the Spliterator is used to **split** the data into multiple substreams, which are then processed concurrently. Each substream is processed by a different thread in the ForkJoinPool.
* **Internal workings**:
  + The **trySplit()** method is used to break the stream into smaller chunks for parallel processing.
  + Spliterator supports both **sequential** and **parallel** traversal, depending on the stream's execution mode.

**Example**:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Spliterator<Integer> spliterator = numbers.spliterator();

Spliterator<Integer> secondPart = spliterator.trySplit(); // Splitting for parallel processing

**6. How does the ForkJoinPool work with parallel streams?**

**Answer Expected:**

* The **ForkJoinPool** is the **default thread pool** used by parallel streams in Java. It uses the **work-stealing algorithm** to efficiently divide tasks across multiple threads, ensuring that threads which finish early can "steal" work from other threads.
* The ForkJoinPool has a **pool of worker threads**, typically one per core of the CPU, and when a parallel stream is executed, tasks are divided into smaller chunks and distributed to available threads.
* **Key points**:
  + Parallel streams use the common ForkJoinPool by default.
  + Work-stealing ensures that idle threads pick up work from busy threads, improving throughput.
  + Developers can also configure a custom ForkJoinPool if needed.

**7. How would you handle ordering in parallel streams?**

**Answer Expected:**

* In **parallel streams**, the order of execution is **not guaranteed** unless explicitly handled. For operations where order is important, you can use **forEachOrdered()** or ensure the stream is processed **sequentially**.
* **Best practice**:
  + If order is not important, use parallel streams for better performance.
  + If order matters, use forEachOrdered() or use sequentialStream() instead of parallelStream().

**Example**:

numbers.parallelStream()

.forEachOrdered(System.out::println); // Maintains order even in parallel

**8. What is the impact of using mutable state with parallel streams?**

**Answer Expected:**

* When using parallel streams, **mutable state** (e.g., modifying shared variables) can cause **data race conditions** and lead to unpredictable behavior.
* **Avoid** modifying shared mutable variables inside stream operations, especially in parallel contexts. Instead, use **concurrent collections** or **atomic variables** for thread-safe operations.

**Example of problematic code**:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

AtomicInteger sum = new AtomicInteger();

numbers.parallelStream()

.forEach(n -> sum.addAndGet(n)); // Correct usage with AtomicInteger

**9. Can you explain how the map() and flatMap() methods differ?**

**Answer Expected:**

* **map()**: Transforms each element of the stream into another object of potentially different type.
  + The result is a **stream of transformed values**.
* **flatMap()**: Similar to map(), but it **flattens** the result. It can convert each element into a **stream** (or a collection) and then flatten the result into a single stream.
* **Key Difference**:
  + map() produces a **one-to-one** transformation, while flatMap() produces a **one-to-many** transformation, flattening nested streams into a single stream.

**Example:**

// map()

List<String> words = Arrays.asList("apple", "banana");

List<String> uppercased = words.stream()

.map(String::toUpperCase)

.collect(Collectors.toList()); // Output: ["APPLE", "BANANA"]

// flatMap()

List<List<String>> listOfLists = Arrays.asList(Arrays.asList("apple", "banana"), Arrays.asList("cherry"));

List<String> flatList = listOfLists.stream()

.flatMap(List::stream)

.collect(Collectors.toList()); // Output: ["apple", "banana", "cherry"]

**10. What are some performance concerns when using streams? How can you optimize stream operations?**

**Answer Expected:**

* **Overhead of Parallel Streams**: Parallel streams introduce overhead, especially for small datasets or simple operations. The time spent managing threads may outweigh the performance benefits.
* **Unnecessary Stream Creation**: Creating streams from collections with stream() or parallelStream() should be done judiciously, as it involves some overhead compared to using traditional loops for small collections.
* **Optimizations**:
  + **Avoid using parallel streams for small datasets**: Parallelism brings more overhead than benefit for small or simple datasets.
  + **Use reduce() and collect() efficiently**: When using collect(), use a **combiner function** for better aggregation.
  + **Minimize stateful operations**: Stateful operations like distinct() or sorted() may require additional memory, especially in parallel streams.

**11. How would you handle exceptions in streams?**

**Answer Expected:**

* Streams do not provide built-in support for exception handling because stream operations are intended to be functional and side-effect-free.
* To handle exceptions, you should **wrap the code inside a try-catch block** within stream operations or use **custom exception handling**.

**Example**:

List<String> numbers = Arrays.asList("1", "2", "a", "4");

numbers.stream()

.map(s -> {

try {

return Integer.parseInt(s);

} catch (NumberFormatException e) {

return 0; // handle exception by returning a default value

}

})

.collect(Collectors.toList()); // Output: [1, 2, 0, 4]

**12. How do you handle null values in streams?**

**Answer Expected:**

* Streams **do not handle null values by default**. Using null values in streams can result in NullPointerException. You should filter out null values before applying operations on them.

**Example**:

List<String> names = Arrays.asList("John", null, "Jane", "Jack");

names.stream()

.filter(Objects::nonNull)

.forEach(System.out::println); // Output: John, Jane, Jack

**1. Count the Occurrences of Each Word in a List of Strings**

**Question:**

Given a list of strings, count the occurrences of each word.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class WordCount {

public static void main(String[] args) {

List<String> words = Arrays.asList("apple", "banana", "apple", "orange", "banana", "apple");

Map<String, Long> wordCount = words.stream()

.collect(Collectors.groupingBy(word -> word, Collectors.counting()));

wordCount.forEach((word, count) -> System.out.println(word + ": " + count));

}

}

**Explanation**:

* **groupingBy**: Groups the stream by the word itself.
* **counting**: Counts the number of occurrences of each word.

**2. Find the Maximum and Minimum Value in a List of Numbers**

**Question:**

Given a list of integers, find the maximum and minimum values using streams.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class MaxMinValue {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Optional<Integer> max = numbers.stream().max(Integer::compareTo);

Optional<Integer> min = numbers.stream().min(Integer::compareTo);

System.out.println("Max Value: " + max.orElse(null));

System.out.println("Min Value: " + min.orElse(null));

}

}

**Explanation**:

* **max()** and **min()**: Terminal operations that return an Optional containing the maximum or minimum element.
* The Comparator used (Integer::compareTo) compares the elements.

**3. Remove Duplicates from a List Using Streams**

**Question:**

You are given a list of integers. Write a code to remove duplicates using streams and return a new list.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class RemoveDuplicates {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);

List<Integer> distinctNumbers = numbers.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(distinctNumbers); // Output: [1, 2, 3, 4, 5]

}

}

**Explanation**:

* **distinct()**: This intermediate operation removes duplicates from the stream.

**4. Convert a List of Strings to Uppercase and Filter Words with Length Greater than 3**

**Question:**

Given a list of strings, convert each string to uppercase and filter out the strings that have more than 3 characters.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class UppercaseFilter {

public static void main(String[] args) {

List<String> words = Arrays.asList("apple", "banana", "cherry", "kiwi", "mango");

List<String> result = words.stream()

.map(String::toUpperCase)

.filter(word -> word.length() > 3)

.collect(Collectors.toList());

System.out.println(result); // Output: [APPLE, BANANA, CHERRY]

}

}

**Explanation**:

* **map()**: Transforms each element to uppercase.
* **filter()**: Filters out words with length less than or equal to 3.

**5. Sum of All Even Numbers in a List**

**Question:**

Write a program that calculates the sum of all even numbers from a list using streams.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class SumEvenNumbers {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int sum = numbers.stream()

.filter(n -> n % 2 == 0)

.mapToInt(Integer::intValue)

.sum();

System.out.println("Sum of even numbers: " + sum); // Output: 30

}

}

**Explanation**:

* **filter()**: Filters out only the even numbers.
* **mapToInt()**: Converts the stream of Integer objects to an IntStream.
* **sum()**: Calculates the sum of the even numbers.

**6. Find the First Element that Matches a Condition**

**Question:**

Given a list of integers, find the first element that is greater than 5 using a stream.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class FirstMatchingElement {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

Optional<Integer> result = numbers.stream()

.filter(n -> n > 5)

.findFirst();

result.ifPresent(System.out::println); // Output: 6

}

}

**Explanation**:

* **filter()**: Filters the elements based on the condition.
* **findFirst()**: Returns the first element that matches the condition, wrapped in an Optional.

**7. Grouping Elements by a Property**

**Question:**

You have a list of Person objects. Write a code to group them by their age using the Stream API.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

public class GroupByAge {

public static void main(String[] args) {

List<Person> people = Arrays.asList(

new Person("Alice", 30),

new Person("Bob", 25),

new Person("Charlie", 30),

new Person("Dave", 25)

);

Map<Integer, List<Person>> groupedByAge = people.stream()

.collect(Collectors.groupingBy(Person::getAge));

groupedByAge.forEach((age, group) -> {

System.out.println("Age " + age + ": " + group.stream()

.map(Person::getName)

.collect(Collectors.joining(", ")));

});

}

}

**Explanation**:

* **groupingBy()**: Groups the Person objects by age.
* **Joining**: Combines the names of people in each age group.

**8. Combine Multiple Streams into One**

**Question:**

You have two lists of integers. Write a program to combine them into one list using streams.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class CombineStreams {

public static void main(String[] args) {

List<Integer> list1 = Arrays.asList(1, 2, 3, 4);

List<Integer> list2 = Arrays.asList(5, 6, 7, 8);

List<Integer> combined = Stream.concat(list1.stream(), list2.stream())

.collect(Collectors.toList());

System.out.println(combined); // Output: [1, 2, 3, 4, 5, 6, 7, 8]

}

}

**Explanation**:

* **Stream.concat()**: Combines two streams into one.
* **collect()**: Collects the combined stream into a list.

**9. Flatten Nested Lists**

**Question:**

Given a list of lists of integers, flatten the list into a single list using the flatMap() method.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

public class FlattenNestedLists {

public static void main(String[] args) {

List<List<Integer>> nestedList = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5, 6),

Arrays.asList(7, 8, 9)

);

List<Integer> flattened = nestedList.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

System.out.println(flattened); // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

}

}

**Explanation**:

* **flatMap()**: Flattens the nested lists into a single stream.

**10. Calculate the Average Salary of Employees by Department**

**Question:**

You are given a list of employees. Calculate the average salary of employees grouped by their department.

**Answer:**

java

Copy

import java.util.\*;

import java.util.stream.\*;

class Employee {

String name;

double salary;

String department;

Employee(String name, double salary, String department) {

this.name = name;

this.salary = salary;

this.department = department;

}

public String getDepartment() {

return department;

}

public double getSalary() {

return salary;

}

}

public class AverageSalaryByDepartment {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", 5000, "HR"),

new Employee("Bob", 6000, "IT"),

new Employee("Charlie", 5500, "HR"),

new Employee("Dave", 7000, "IT")

);

Map<String, Double> avgSalary = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment,

Collectors.averagingDouble(Employee::getSalary)));

System.out.println(avgSalary); // Output: {HR=5250.0, IT=6500.0}

}

}

**Explanation**:

* **groupingBy()**: Groups employees by department.
* **averagingDouble()**: Computes the average salary per department.