From Java 22

eazy bytes

Launch Multi-File Source-Code Programs

Starting from Java 11, it's possible to run Java programs directly from a single file without the need for compilation. This feature is referred to as "single-file source-code programs" or simply "single-file programs" in Java.

The same feature is extended in the form of “multi-file source-code programs” which will allow developers to run a Java program that has logic spread across multiple source code files.

Before Java 11

Hello

```
public class  Hello {

  public static void main(String[] args) {
    System.out.println("Hello World...");
  }

}
```

Compilation -> javac Hello.java
Execution -> java Hello

From Java 11

Hello

```
public class  Hello {

  public static void main(String[] args) {
    System.out.println("Hello World...");
  }

}
```

Compilation & Execution -> java Hello.java

But this feature works only if you have all your code with in a single source file

From Java 22

Hello

```
public class  Hello {

  public static void main(String[] args) {
    Greetings.sayHello();
  }

}
```

Greetings

```
public class  Greetings {

  public static String sayHello() {
    System.out.println("Hello World...");
  }

}
```

Compilation & Execution -> java Hello.java

Though Hello has dependency on Greetings, from Java 22, we can execute the code as it supports multi-file source-code programs

MARCH 2024

From Java 22

eazy bytes

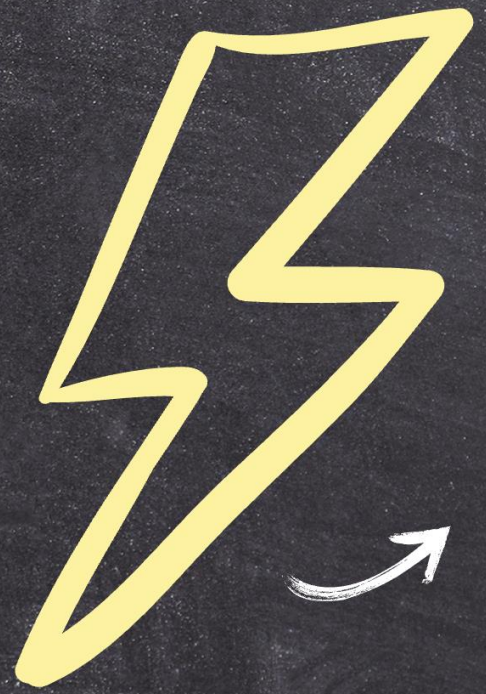
Unnamed Variables & Patterns

1

In certain code snippets, local variables may remain unused, yet deleting them might not be feasible. This could occur due to language constraints or reliance on the side effects of those variables.

2

In situations where deleting an unused variable is either impossible or ill-advised, replacing them with an unnamed variable, denoted by _, can serve as a clear and obvious solution.



3

Unnamed variables & patterns would free maintainers from having to understand irrelevant names, and would avoid false positives on non-use from static analysis tools.

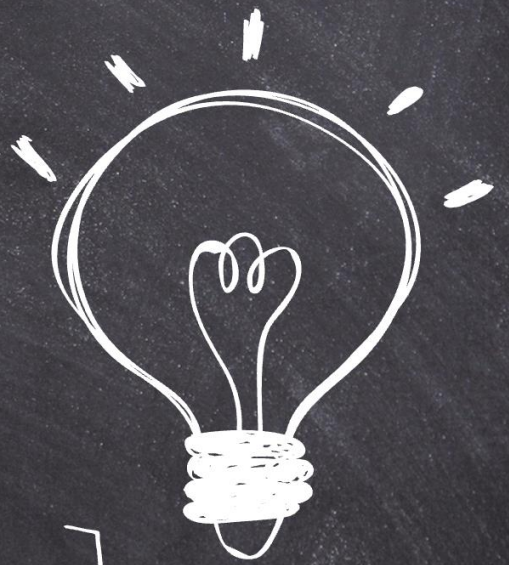
MARCH 2024

From Java 22

eazy bytes

Unnamed Variables & Patterns

Starting from Java 22, you can mark unused local variables, patterns, and pattern variables to be ignored by replacing their names (or their types and names) with an underscore, '_'.



These variables and patterns, known as Unnamed variables and patterns, no longer have a name. This feature is beneficial as it reduces the time and effort required to understand a code snippet and could potentially prevent errors in the future.

It's important to note that this feature doesn't apply to instance or class variables. Unnamed variables cannot be passed to methods or assigned values

MARCH 2024

From Java 22

eazy bytes

Unnamed Variables & Patterns

```java
public static int countWords(Iterable<String> words) {
    int totalWords = 0;
    for (String word : words) {
        totalWords++;
    }
    return totalWords;
}
```

In the provided example, the unused variable “word” can’t be deleted. So it can be replaced with the unnamed variable syntax which is _

```java
public static int countWords(Iterable<String> words) {
    int totalWords = 0;
    for (String _ : words) {
        totalWords++;
    }
    return totalWords;
}
```

From Java 23

Markdown Documentation Comments

eazy bytes

Are you a Java Developer who finds joy in crafting perfect documentation comments?
Do you get a warm fuzzy feeling from writing detailed /** */ comments, explaining every single variable like it's your baby? If you're nodding right now, then hold onto your IDE because you're about to fall in love with this new feature! 🚀

- ✓ JavaDoc comments have always relied on HTML and JavaDoc tags—until now. With Java 23, you can use Markdown to write cleaner, simpler documentation. And yes, it's production-ready!

- ✓ Curious why this change was made? One reason is that HTML, once popular in the 1990s, isn't the go-to choice for new developers. It's harder to write manually, while Markdown is simpler, easier to read, and easily converts to HTML. Many developers already use Markdown for documenting code, writing articles, blogs, and more.

Markdown documentation comments begin with ///

From Java 23

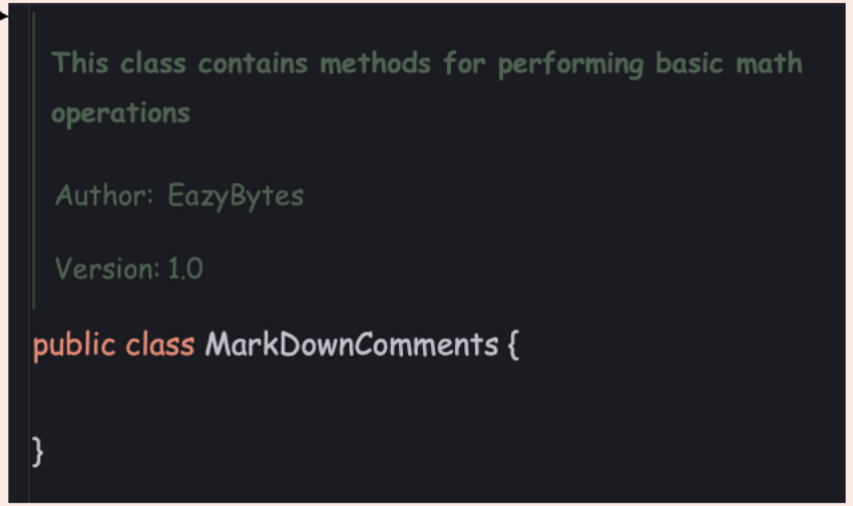
Markdown Documentation Comments

eazy bytes

```
/// **This class contains methods for performing basic math operations**
/// @author EazyBytes
 /// @version 1.0
public class MarkDownComments {

}
```

Class level comments using markdown syntax



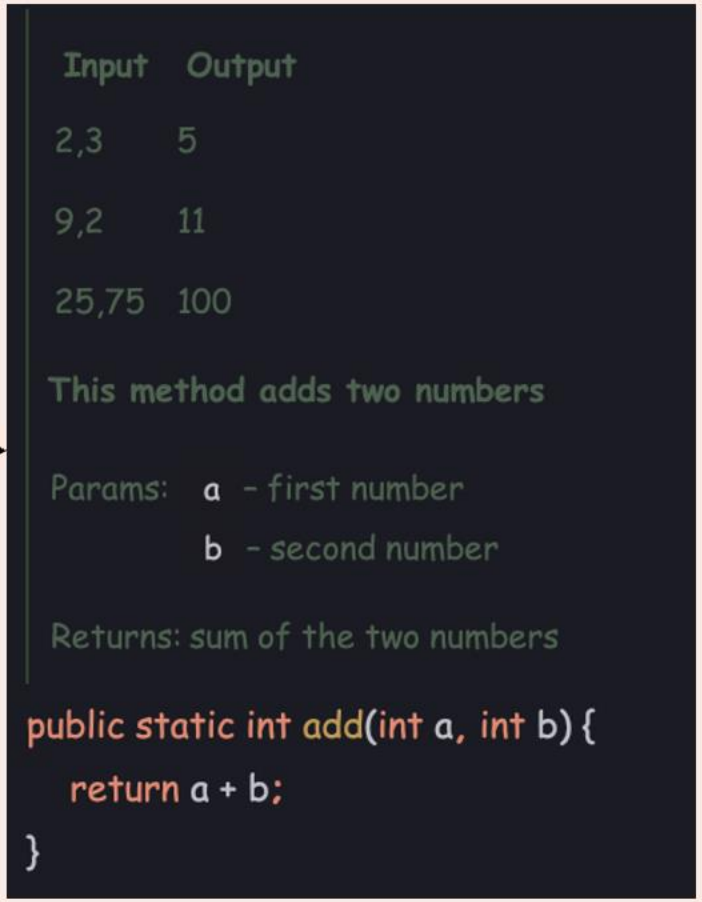
From Java 23

Markdown Documentation Comments

eazy bytes

```
/// | Input | Output |
/// |-------|-------|
/// | 2,3   | 5 |
/// | 9,2   | 11  |
/// | 25,75   | 100 |
/// **This method adds two numbers**
/// @param a first number
/// @param b second number
/// @return sum of the two numbers
public static int add(int a, int b) {
    return a + b;
}
```

method level comments using markdown syntax



From Java 23

Markdown Documentation Comments

eazy bytes

```java
/// # This method subtract two numbers
///   - 5,3 = 2
///   - 9,2 = 7
/// @param a first number
/// @param b second number
/// @return ***subtraction of the two numbers***
public static int subtract(int a, int b) {
   return a - b;
}
```

method level comments using markdown syntax

This method subtract two numbers

- 5,3 = 2
- 9,2 = 7

Params: a - first number
b - second number

Returns: *subtraction of the two numbers*

```java
public static int subtract(int a, int b) {
   return a - b;
}
```

From Java 23

Markdown Documentation Comments

eazy bytes

```
/// ---
/// # This method multiply two numbers
/// @param a first number
/// @param b second number
/// @return ***multiplication of the two numbers***
public static int multiply(int a, int b) {
    return a * b;
}
```

method level comments using markdown syntax

This method multiply two numbers

Params: a - first number
b - second number

Returns: *multiplication of the two numbers*

```
public static int multiply(int a, int b) {
    return a * b;
}
```

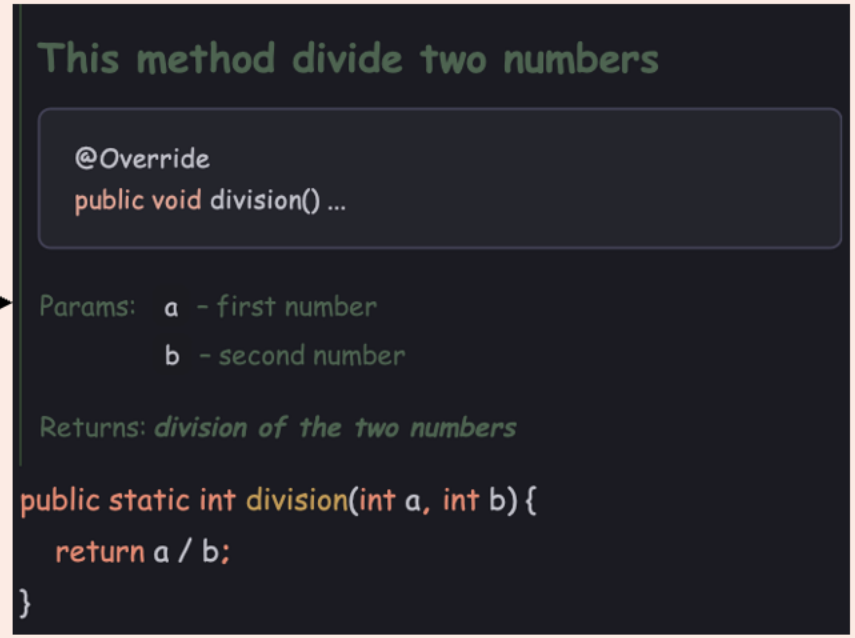
From Java 23

Markdown Documentation Comments

eazy bytes

```
/// # This method divide two numbers
/// ```
 /// @Override
 /// public void division() ...
 /// ```
/// @param a first number
 /// @param b second number
 /// @return ***division of the two numbers***
public static int division(int a, int b) {
   return a / b;
}
```

method level comments using markdown syntax



Stream Gatherers

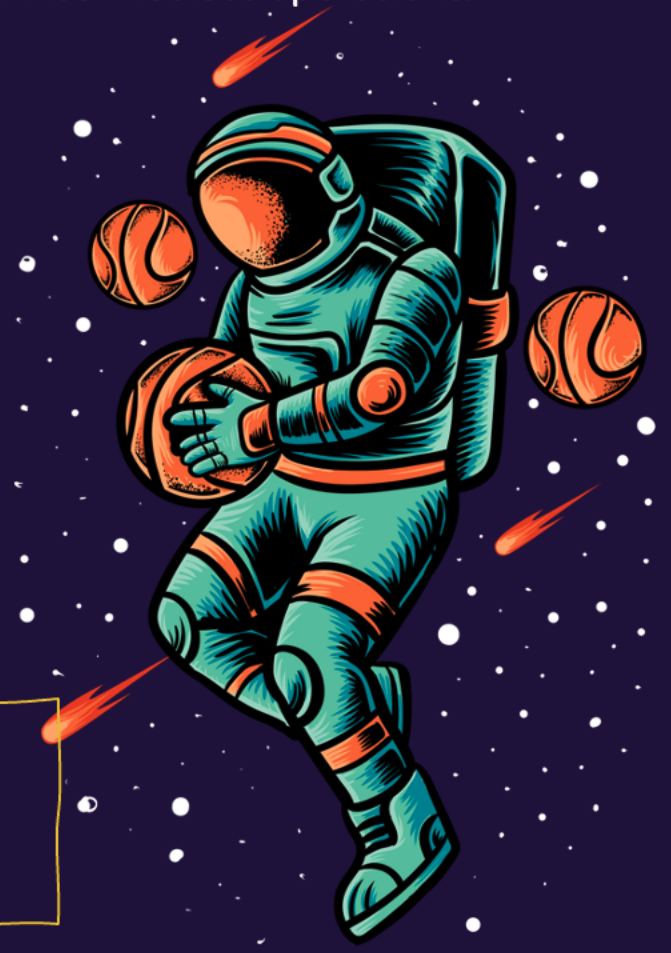
eazy bytes

What are Stream Gatherers in Java?

In Java, we use Streams to process data, like filtering, mapping, collecting. But until now, we could only use predefined operations like .map(), .filter(), .flatMap(), etc.

Java 24 introduced Gatherers — a new tool that lets us create our own custom intermediate operations.

```
Think of a Gatherer like a custom .map() or .filter() that YOU define.
```



What is a Gatherer?

A Gatherer is a component that:

- Takes items from a stream
- Transforms or filters them (like .map() or .filter())
- Passes them to the next step in the pipeline

```
Analogy:
Imagine a "middleman" in a factory assembly line — he checks or
transforms every item before passing it to the next worker.
```

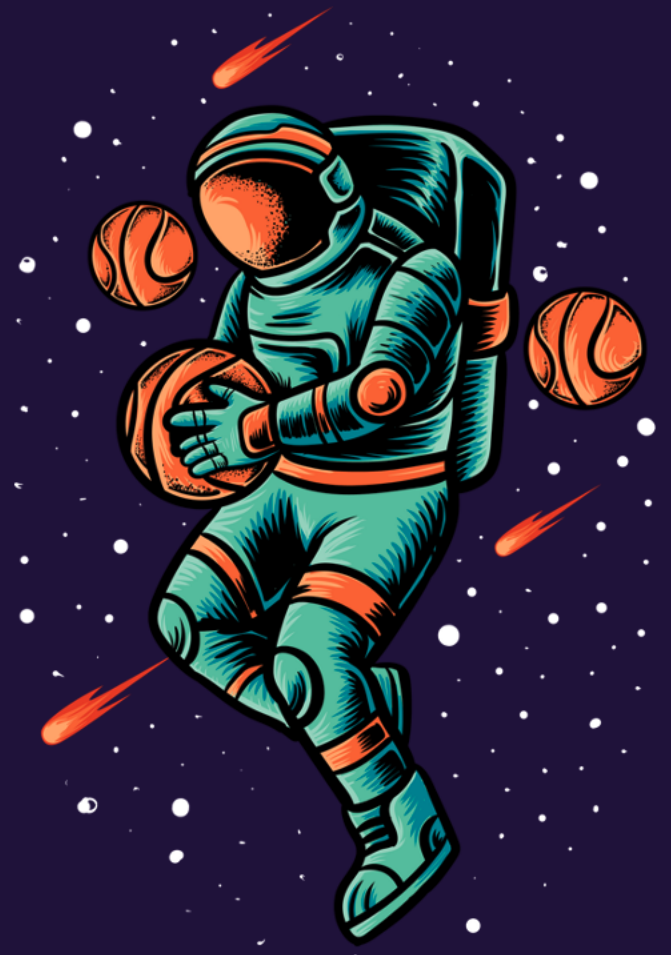
Stream Gatherers

eazy bytes

The Gatherer API is built on two main elements: a Gatherer interface and a Gatherers factory class.

A Gatherer is an object that you can pass to a method of the Stream interface: gather(). This gather() method is an intermediate operation of the Stream API, and this object models what this intermediate operation is doing.

```
Stream.of("java", "python")
                .gather(Gatherer.of((_,element, downstream) ->
                        downstream.push(element.toUpperCase())))
                .toList(); // [JAVA, PYTHON]
```

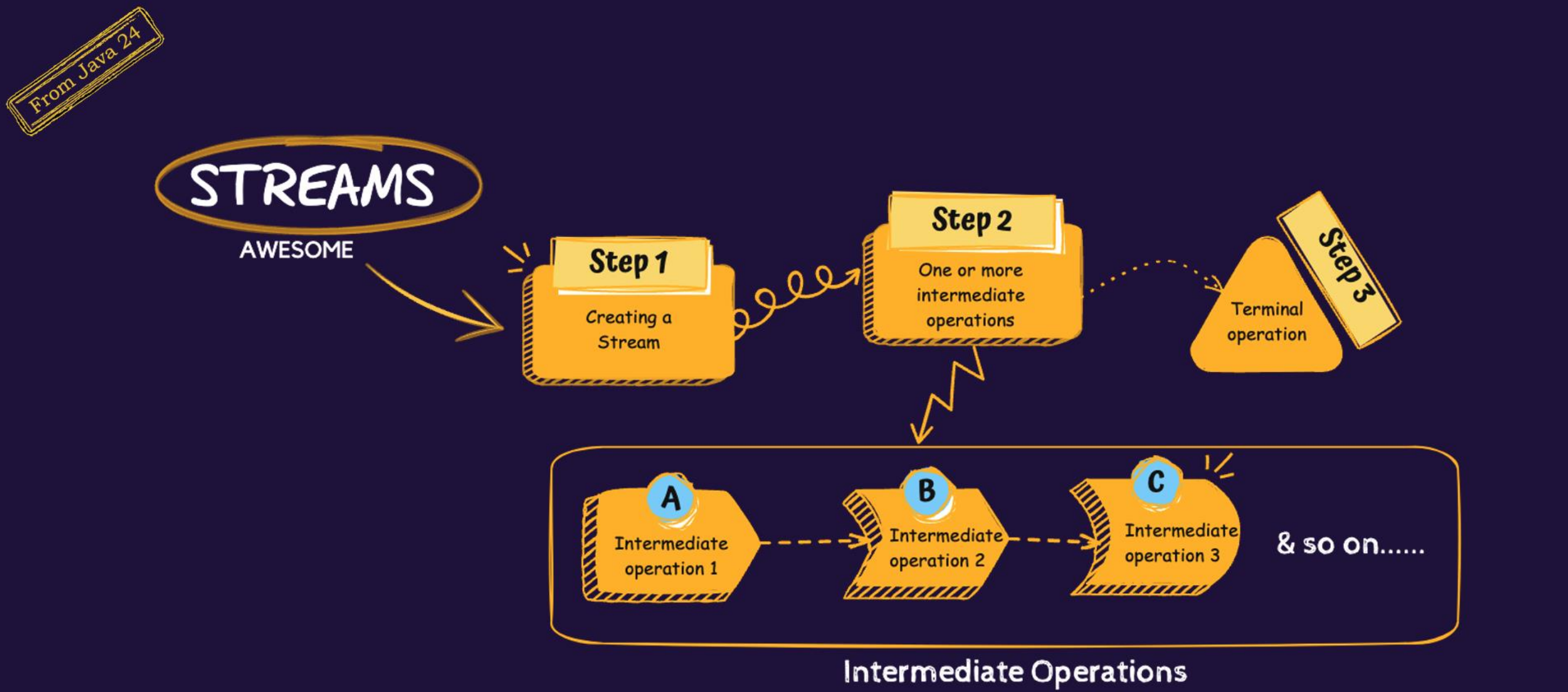


Vocabulary Fix – Three Key Terms

Term	Meaning
Upstream	The stream or data source that produces elements
Gatherer	The logic that takes input, processes it, and decides what to pass next
Downstream	The next step in the stream that receives elements from the Gatherer. It can be another intermediate operation or a terminal operation

Stream Gatherers

eazy bytes



The Four Parts of a Stream Gatherers

eazy bytes

A Gatherer is defined by four functions that work together:

Initializer (Optional):

Creates a "state" object to track information (e.g., a list to store seen elements). Example: Initialize an empty ArrayList to store the largest number seen so far.

Integrator (Mandatory):

Processes each stream element, updates the state, and decides what to send to the output. Example: Compare the current number with the largest so far and update if needed.

Combiner (Optional):

Merges states when processing in parallel (for parallel streams). Example: Combine two lists of largest numbers from different threads.

Finisher (Optional):

Performs final actions after all elements are processed, like sending a final result. Example: Output the largest number after checking all elements.

Gatherer Integrator

eazy bytes

An Integrator is the part of the Gatherer that:

- Receives elements from the upstream
- Has access to a state variable
- Uses that state to process the element and possibly push results to downstream

💡 The state variable is what makes the integrator stateful.

Integrator is a functional interface with a single abstract method named integrate. Below is the signature of the method,

```
@FunctionalInterface
interface Integrator<A, T, R> {

    boolean integrate(A state, T element, Downstream<? super R> downstream);

}
```

state - The mutable state that can be used in the integrator logic
element - Each element of the upstream to aplly integrator logic
downstream - The downstream object where the elements need to be pushed
return true if subsequent integration is desired

Creating a gatherer implementation

eazy bytes

There are two ways to write a Gatherer:

Using a Lambda – because Gatherer is a functional interface (Less Readable)
Using Factory Methods – like Gatherer.of()

The Integrator is the main part of a gatherer. It is where you write what to do with each element.

It works with:

- Optionally: A state variable
- The input element from upstream
- A downstream object to push results to

This integrator doesn't change anything — it just passes elements along.

```
Gatherer.Integrator<Void, String, String> integrator =
        (_, element, downstream) -> downstream.push(element);
```

Use Gatherer.of() to create the gatherer:

```
Gatherer<String, ?, String> identityGatherer = Gatherer.of(integrator);
```

```
First String: input type, Second ?: no state (Void), Third String: output type
```

Creating a gatherer implementation

eazy bytes

Below is the simple stream pipeline logic using gatherer as a intermediate operation. This gatherer behaves like the identity function – passes each item as-is.

```
Gatherer.Integrator<Void, String, String> integrator =
    (_, element, downstream) -> downstream.push(element);

Gatherer<String, ?, String> identityGatherer = Gatherer.of(integrator);

var result = Stream.of(”java", ”python", ”javascript")
                   .gather(identityGatherer)
                   .toList();

System.out.println("result = " + result);
```

Gatherer with Filter + Map Logic

eazy bytes

Gatherer can do filtering + mapping in one step. Saves performance and improves code clarity. The integrator controls both behaviors. Below is the sample logic,

```
Gatherer<String, ?, String> filterAndMapGatherer =
    Gatherer.of((_, element, downstream) -> {
        if (element.startsWith("a")) {
            return downstream.push(element.toUpperCase());
        }
        return true; // Skip and continue
    });

var result = Stream.of("apple", "banana", "avocado")
    .gather(filterAndMapGatherer)
    .toList();

System.out.println("result = " + result); // [APPLE, AVOCADO]
```

Gatherers with Mutable State & Initializer

eazy bytes

Some stream operations need to:

- Remember things between elements (e.g. running count, previous item)
- Track accumulations (e.g. sum, group size, last item seen)
- In traditional streams, this was hard without Collectors (terminal operation)
- With Gatherers, we can do this as part of the intermediate steps!

Mutable State is a variable that the integrator can use and update across elements. The Initializer is a supplier function that creates a new instance of the state. This state is then passed to the integrator, which updates it as the stream is processed.

Cumulative Total Gatherer

```
Gatherer<Integer, ?, Integer> cumulativeSum =
                Gatherer.ofSequential(
                  AtomicInteger::new, // state instantiated using Initializer
                  (state, number, downstream) -> {
                     int updated = state.addAndGet(number); // update state
                     return downstream.push(updated);
                  }
                );
var result = Stream.of(10, 20, 30, 40).gather(cumulativeSum).toList();
```

Gatherers with Mutable State & Initializer

eazy bytes

What's happening behind the scenes ?

Element	State Before	State After	Output pushed
10	0	10	10
20	10	30	30
30	30	60	60
40	60	100	100

Gatherer.ofSequential(...) is a factory method that helps you create a stateful gatherer for sequential streams.

This method is specifically for use with sequential (non-parallel) streams, because:

- It creates a single state object used throughout the entire stream pipeline
- It avoids the need to merge/combine states from parallel threads (which would require a combiner)

Finisher in a Stream Gatherer

eazy bytes

The Finisher is an optional part of a Gatherer that is called after all elements have been processed by the stream.

- It lets you flush out any remaining data from the state
- It's useful when:
  - You buffer elements using a greedy integrator
  - The final output depends on end-of-stream logic

When is the Finisher Called?

- After the stream is done consuming elements
- Only once per stream run
- Used to emit anything left in memory/state that wasn't yet pushed

Finisher in a Stream Gatherer

eazy bytes

Example – Grouping Every 3 Elements

```
Gatherer.ofSequential(
    () -> new ArrayList<>(), // state: buffer
    (buffer, item, downstream) -> {
        buffer.add(item);
        if (buffer.size() == 3) {
            downstream.push(new ArrayList<>(buffer));
            buffer.clear();
        }
        return true;
    },
    (buffer, downstream) -> {
        if (!buffer.isEmpty()) {
            downstream.push(new ArrayList<>(buffer)); // flush leftovers
        }
    }
);
```

Input → Output

[1, 2, 3, 4, 5, 6, 7] → [[1, 2, 3], [4, 5, 6], [7]]

Greedy Integrator

eazy bytes

The integrator in the before example is called Greedy Integrator

A Greedy Integrator is an integrator that may consume multiple elements from the upstream before pushing anything to downstream. They are called so, because they greedily pull elements from the upstream without immediately sending them to downstream.

It allows you to:

- Hold onto elements in internal state
- Push only when you're ready (e.g. after collecting N items)
- Implement complex patterns like windowing, batching, etc.
- Delay pushing until a certain condition is met

Caution with Greedy Integrators

- You must manually handle the last remaining elements (e.g., flush remaining items in finisher)
- You may want to define a finisher in your gatherer to push remaining buffered elements when stream ends

Parallel Gatherers

eazy bytes

Parallel Gatherers process elements in multiple threads. Java provides different ways to define them, depending on whether they use internal state.

🧠 Tip: If you don't use internal state, it's very simple. If you use internal state, merging becomes important.

Pattern 1 – Gatherer.of(integrator) - Use this when you only need to inspect or transform elements without remembering anything.

- No mutable state
- No finisher
- Simple and safe for parallel execution

```
Gatherer<Integer, ?, Integer> evenFilter =
    Gatherer.of((_,item, downstream) ->
        item % 2 == 0 && downstream.push(item)
    );
```

Pattern 2 – Gatherer.of(integrator, finisher) - Use this when you don't need memory, but you want to say something at the end.

- No mutable state
- Has a finisher to perform final actions

```
Gatherer<Integer, ?, String> gathererWithMessage =
    Gatherer.of(
        (_,item, downstream) -> downstream.push(item),
        (_,downstream) -> downstream.push("All items
processed")
    );
```

Parallel Gatherers

eazy bytes

Pattern 3 – Gatherer.of(initializer , integrator, combiner, finisher) - Use this when you need to inspect or transform elements paralley by using a mutable state value

- Need to keep track of state (e.g., sum, count, etc.)
- Each thread gets its own copy of the state
- State must be combined later

Each thread sums part of the numbers, then all sums are combined into one.

```
Gatherer<Integer, ? , Integer> parallelSum =
    Gatherer.of(
        AtomicInteger::new, // initializer
        (sum, number, downstream) -> {
           sum.addAndGet(number);
           return true;
        },
        (s1, s2) -> {
            s1.addAndGet(s2.get()); // combiner
            return s1;
        },
        (sum, downstream) -> downstream.push(sum.get())
    );
```

What if you can't combine state?

❌ You can't run the gatherer in parallel.
Use sequential gatherer (Gatherer.ofSequential(...) instead

Summary of Gatherers

eazy bytes

Pattern	Mutable State	Finisher	Combiner	Parallel Support
of(integrator)	❌ No	❌ No	❌ No	✅ Yes
of(integrator, finisher)	❌ No	✅ Yes	❌ No	✅ Yes
of(init, integ, comb, fin)	✅ Yes	✅ Yes	✅ Yes	✅ Yes
ofSequential(...)	✅ Yes or ❌ No	✅ Yes or ❌ No	❌ No	❌ No

Parallel Streams + Gatherers

eazy bytes

When you use a parallel stream with a gatherer, two things can happen:

✅ 1. Parallel-Friendly Gatherer

- Created using Gatherer.of(...)
- Runs fully in parallel, including the gatherer logic.

🚫 2. Sequential Gatherer

- Created using Gatherer.ofSequential(...)
- Before gather() → runs in parallel
- gather() itself → runs in a single thread
- After gather() → continues in parallel

🧠 Key Point:
Even sequential gatherers still benefit from parallel execution before and after gather() — a unique feature of the Gatherer API!

Interrupting a Stream while using Gatherer API

eazy bytes

The Gatherer API lets you:

- Track how many items you've processed
- Stop when you've had enough
- Let the upstream know it's done

You do this using downstream.push() and return false to stop early

Example of a interrupting streamuUsing Gatherer

```
List<Integer> largeList = Stream.iterate(1, i -> i + 1)
                .limit(1000)
                .toList(); // simulate a large list

Gatherer<Integer, ?, Integer> limitGatherer = Gatherer.of(
        (_, element, downstream) -> {
            System.out.println(element);
            return downstream.push(element);
        });

var res = largeList.stream().gather(limitGatherer).limit(10).toList();
```

Chaining Gatherers

eazy bytes

What is andThen() in Gatherers?

- andThen() lets you chain two gatherers together.
- Output of the first gatherer becomes input to the next one.

📦 Think of it like combining two filters or processors in a pipeline.

When Can You Chain?

You can use .andThen() if:

✅ The output type of the first gatherer matches the input type of the second gatherer

```
Gatherer<A, ?, B> g1;
Gatherer<B, ?, C> g2;

Gatherer<A, ?, C> combined = g1.andThen(g2);
```

Chaining Gatherers

eazy bytes

Simple Example of chaning gatherers – Uppercase & Limit

📦 Think of it like combining two filters or processors in a pipeline.

```
Gatherer<String, ?, String> upper = Gatherer.of((_, element, downstream) ->
downstream.push(element.toUpperCase()));

Gatherer<String, ?, String> filter = Gatherer.of((_, element, downstream) -> {
    if (element.startsWith("J")) {
        downstream.push(element);
    }
    return true;
});

Gatherer<String, ?, String> upperThenFilter = upper.andThen(filter);

var result = Stream.of("java", "spring", "react").gather(upperThenFilter).toList();
```

Factory methods in Gatherers class

eazy bytes

Below are the five factory methods present in Gatherers class, that can be used by developers in various scenarios.

1. `fold()`
2. `scan()`
3. `mapConcurrent()`
4. `windowFixed()`
5. `windowSliding()`

Gatherers.fold()

eazy bytes

What is Gatherers.fold()?

- fold() is like reduce() in streams.
- It accumulates values from a stream into a single result.
- It is ordered and runs sequentially (not parallel-friendly).
- Especially useful when combining values cannot be parallelized.

When to Use fold()

Use fold() when:

- You want to combine all elements into one result
- You can't use a combiner, e.g., in order-sensitive operations
- You only need a single output at the end of the stream

How to use fold()

```
Gatherers.fold(initial, folder)
```

initial -> Supplier<R> - Creates the starting value
folder -> BiFunction<R, T, R> - Combines current state with element

Gatherers.fold()

eazy bytes

Count Digits in All Numbers Combined

```
Optional<Integer> totalDigits = Stream.of(10, 200, 3, 45)
    .gather(
        Gatherers.fold(
            () -> 0,
            (count, num) -> count + String.valueOf(num).length()
        )
    )
    .findFirst();

System.out.println(totalDigits); // Output: Optional[8]
```

Gatherers.scan()

eazy bytes

What is Gatherers.scan()?

- scan() is like fold(), but instead of producing only the final result, it emits intermediate results at each step.

🔄 Think of it as a running total, or progressive reduction.

Example – Running Total (Sum)

```
List<Integer> runningSums = Stream.of(1, 2, 3, 4)
    .gather(
        Gatherers.scan(() -> 0, (sum, next) -> sum + next)
    )
    .toList();

System.out.println(runningSums); // Output: [1, 3, 6, 10]
```

Gatherers.mapConcurrent()

eazy bytes

What is Gatherers.mapConcurrent()?

- mapConcurrent() is like map(), but for parallel streams
- It allows concurrent (parallel) transformations
- Perfect when your mapping function is thread-safe and independent

🧠 Think: transform many items at the same time, safely

When to Use mapConcurrent()

Use mapConcurrent() when:

- You are using a parallel stream
- The mapping function is stateless & thread-safe
- You want to boost performance using concurrent processing

Make sure your mapping logic is:

💡 Stateless
🧵 Thread-safe
🔁 Independent per element

❗ Don't use mapConcurrent() if your function depends on shared state or order!

Gatherers.mapConcurrent()

eazy bytes

Word Count using mapConcurrent()

```
List<String> articles = List.of(
                "Java is a high-level, class-based, object-oriented programming language.",
                "Spring Boot makes it easy to create stand-alone, production-grade applications.",
                "Microservices architecture enables better scalability and maintainability.",
                "Text processing is a common use case for functional programming.",
                "Streams in Java provide a modern way to process collections efficiently."
        );

        // Count words concurrently using mapConcurrent
        List<Integer> wordCounts = articles.parallelStream()
                .gather(Gatherers.mapConcurrent(10,article ->
                        article.split("\\s+").length
                ))
                .toList();


        for (int i = 0; i < articles.size(); i++) {
            System.out.printf("Article %d word count: %d%n", i + 1, wordCounts.get(i));
        }
```

Gatherers.windowFixed()

eazy bytes

What is Gatherers.windowFixed()?

- windowFixed() splits a stream into fixed-size chunks (windows).
- Each window is a List of N elements.
- Emits one list for every complete group of elements.

Think of sliding a fixed-size window over a stream.

How Does It Work?

```
var output = Stream.of(1, 2, 3, 4, 5, 6)
                .gather(Gatherers.windowFixed(2))
                .toList();
System.out.println(output);

Output - [[1, 2], [3, 4], [5, 6]]
```

Real Use Cases for windowFixed()

API Batching, Parallel Processing Chunks, Database Insert Batching, Pagination etc

Gatherers.windowSliding()

eazy bytes

What is Gatherers.windowSliding()?

- windowSliding(windowSize) creates overlapping or moving windows of elements.
- Unlike windowFixed(), it doesn't drop elements — it slides the window by step.

How Does It Work?

```
List<List<Integer>> result =
             Stream.of(1,2,3,4,5,6,7,8)
                 .gather(Gatherers.windowSliding(2)).toList();

Output - [[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8]]
```

Real Use Cases for windowSliding()

Moving averages / rolling stats, Detecting anomalies in overlapping data windows