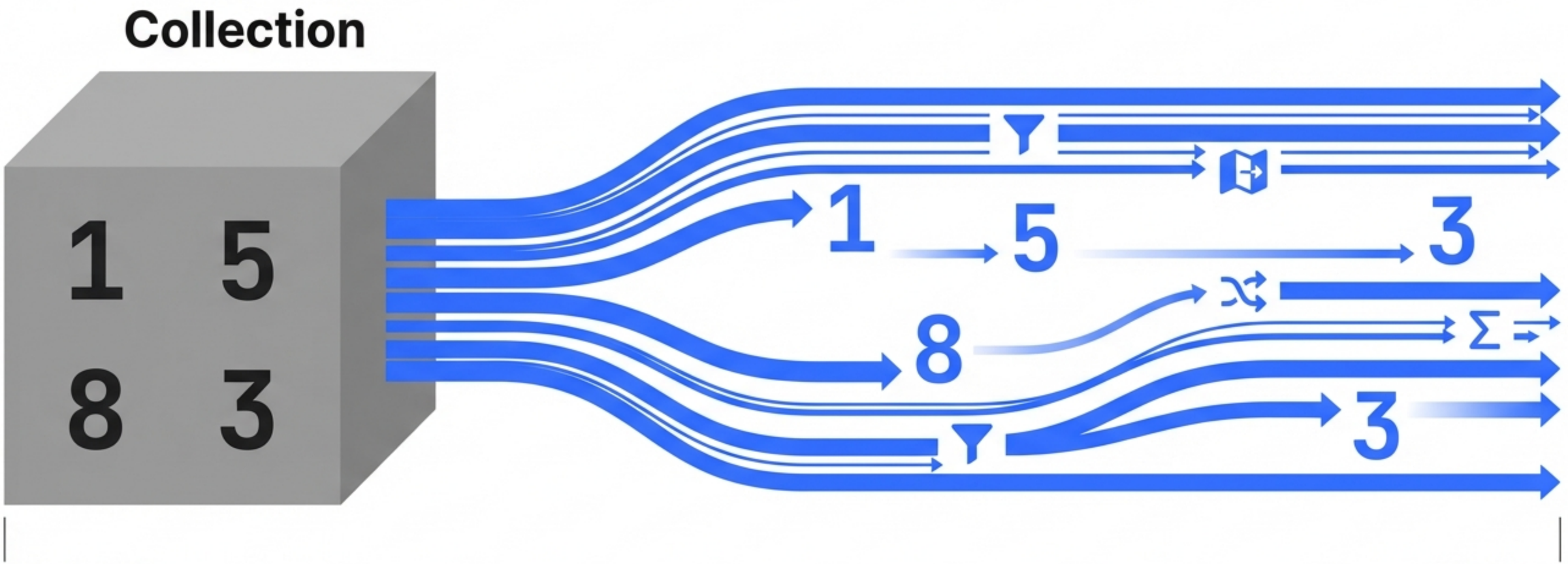
Processing Data, Not Just Storing It.

Mastering the Java Stream API



Inter Regular
COLLECTIONS: Holding State (Storage)

Inter Regular
STREAMS: Processing Flow (Computation)

External Iteration vs. Internal Processing

The Old Way: External Iteration

```
List<Integer> nums = Arrays.asList(4, 5, 7);
for(int n : nums) {
    System.out.println(n * 2);
}
```

Warning
Verbose. Mutable State.
Manual Control.

The Stream Way: Internal Iteration

```
nums.stream()
    .map(n -> n * 2)
    .forEach(System.out::println);
```

Check
Declarative. Immutable.
Pipeline Logic.

Insight: Streams allow you to process data without changing the original source material, enabling safer and more readable code.

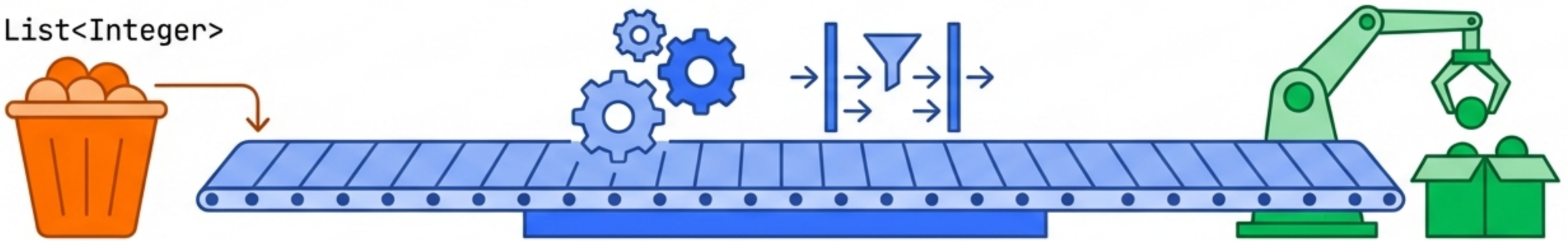
Anatomy of a Stream Pipeline

Lazy Evaluation: Nothing executes until the end.

SOURCE

INTERMEDIATE OPERATIONS

TERMINAL OPERATION



Creation:
.stream()

Configuration:
.filter()
.map()
.sorted()

Execution:
.forEach()
.reduce()

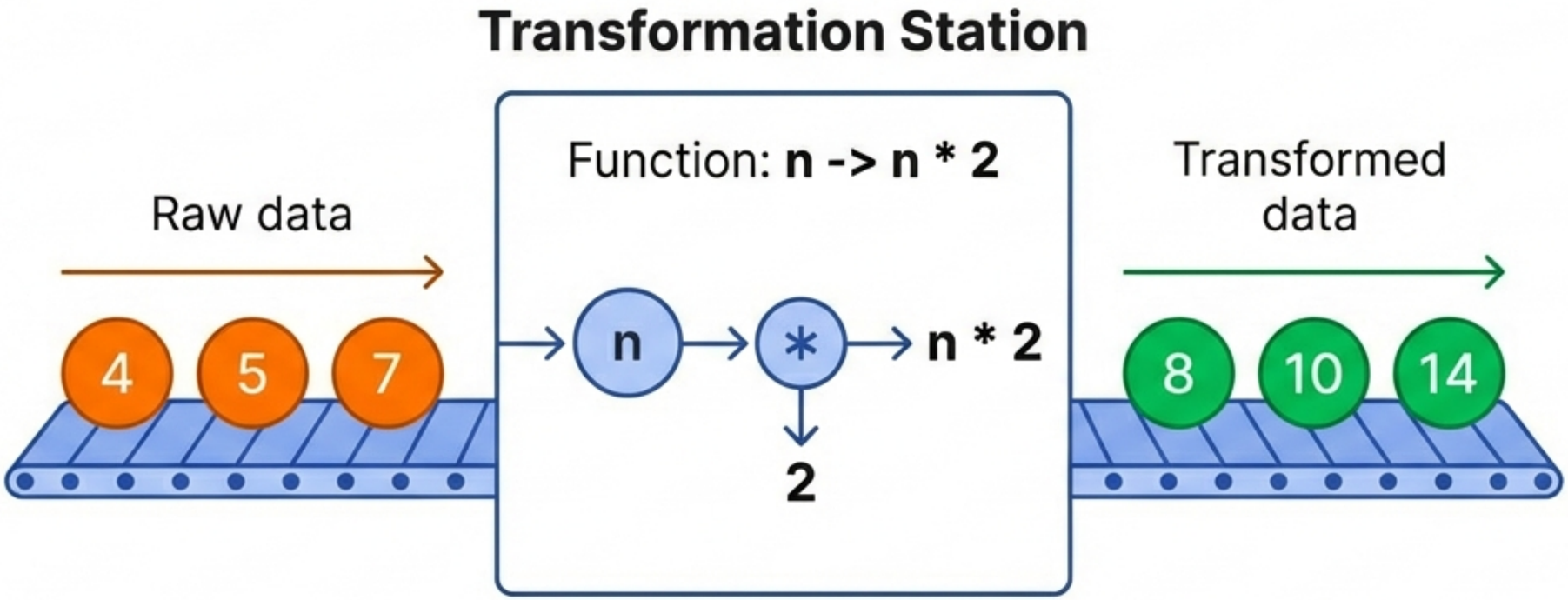
```
nums.stream()
```

```
.filter(...).map(...)
```

```
.forEach(...)
```

Map: Transforming Data Shape

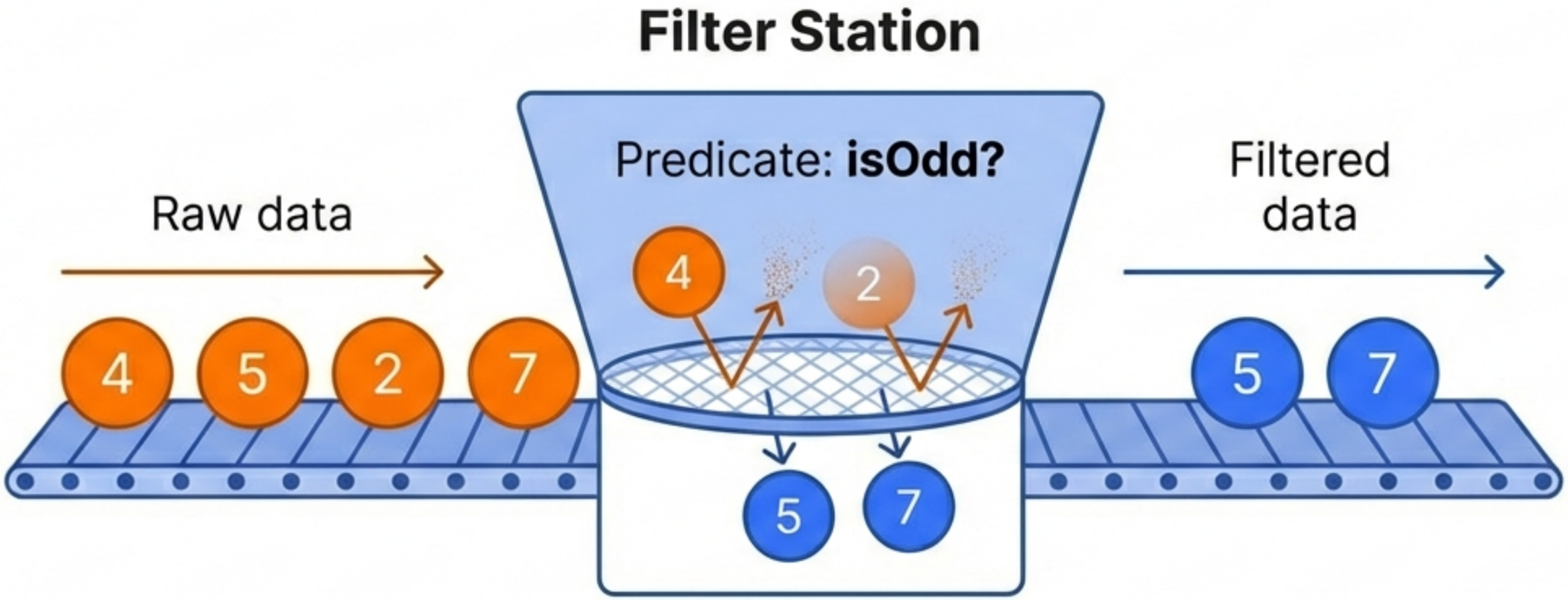
```
data.map(n -> n * 2)
```



Key Concept: **Map** returns a new stream. The original list remains 4, 5, 7 (Immutable).

Filter: Selective Processing

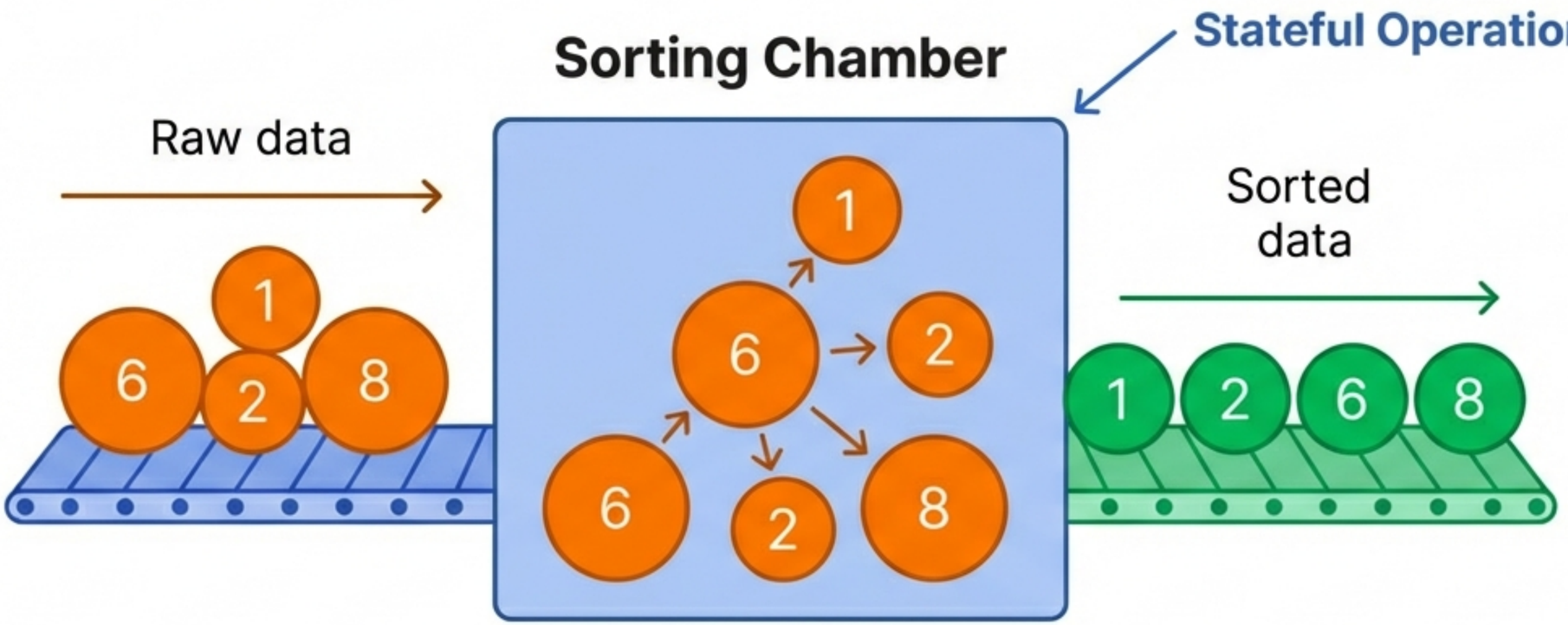
```
.filter(n -> n % 2 == 1)
```



Logic: Expects a **True/False** return. True passes through; False is discarded.

Sorted: Reorganizing the Flow

`.sorted()`



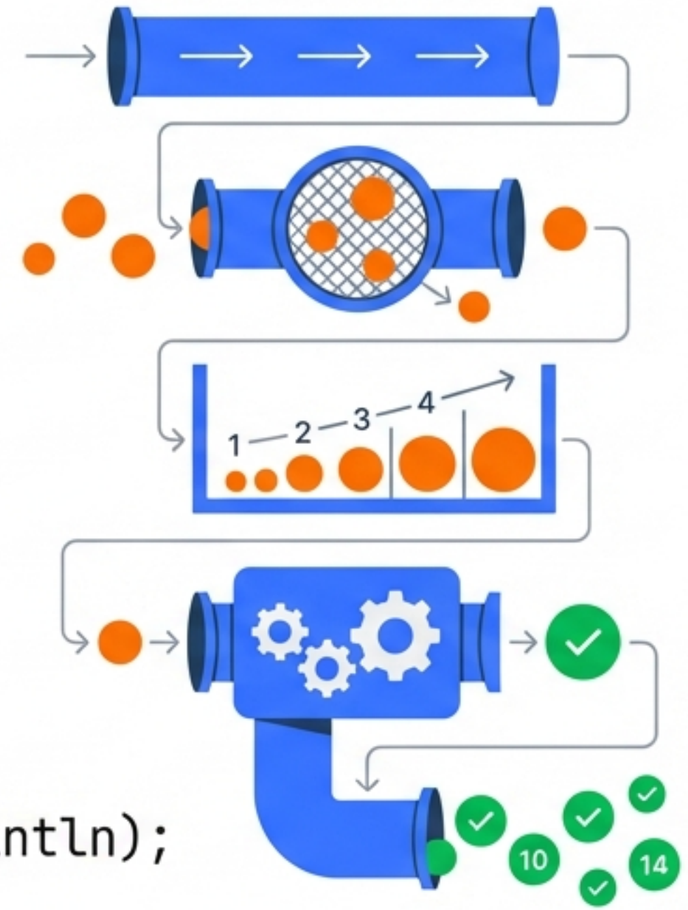
Annotator:
Unlike map or filter which handle one, **sorted()** must see the data to organize it. It creates a new instance of the stream with the correct order.

The Builder Pattern: Chaining Logic

```
nums.stream().filter(n -> n%2==1).sorted().map(n -> n*2).forEach(System.out::println);
```

Hard to read

```
nums.stream()
    .filter(n -> n % 2 == 1)
      .sorted()
       .map(n -> n * 2)
        .forEach(System.out::println);
```



Open stream pipe

Sieve filter module

Sorting alignment module

Transformation gears module

Output nozzle

We are constructing a complex processing pipeline in a single, fluent sentence.

This describes WHAT to do, not HOW to loop.

Under the Hood: The Predicate Interface

Stage 1
The Boilerplate
(Anonymous Inner Class)

```
Predicate<Integer> p = new Predicate<Integer>() {
    public boolean test(Integer n) {
        return n % 2 == 1;
    }
};
```

Stage 2
The Logic
Extraction

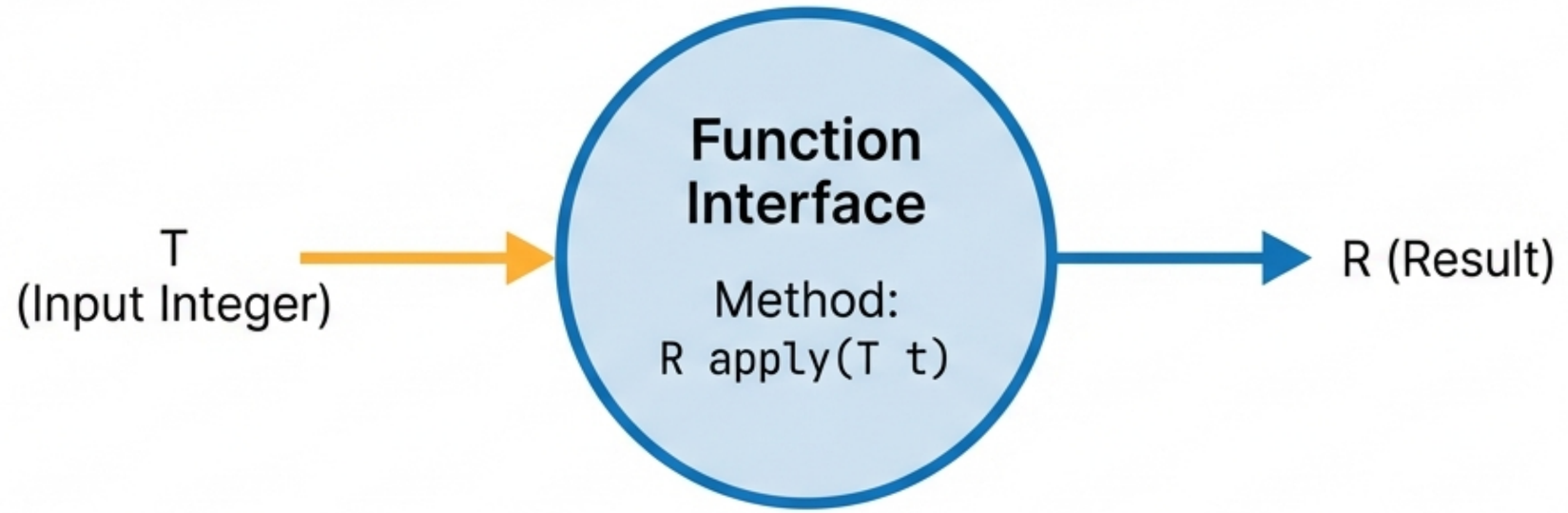
(n) -> n % 2 == 1

Stage 3
The Lambda
(Final Syntax)

.filter(n -> n % 2 == 1)

Technical Detail: Filter relies on the Predicate interface. It has a single abstract method: test(T t) which returns a boolean.

Under the Hood: The Function Interface




```
Function<Integer, Integer> doubler = new Function<Integer, Integer>() {
    public Integer apply(Integer n) {
        return n * 2;
    }
};
```

```
.map(n -> n * 2)
```

The `map` operation uses the `Function` interface to transform type T into type R.

Lambdas make this implementation invisible.

The One-Time Use Rule

Error: `IllegalStateException`

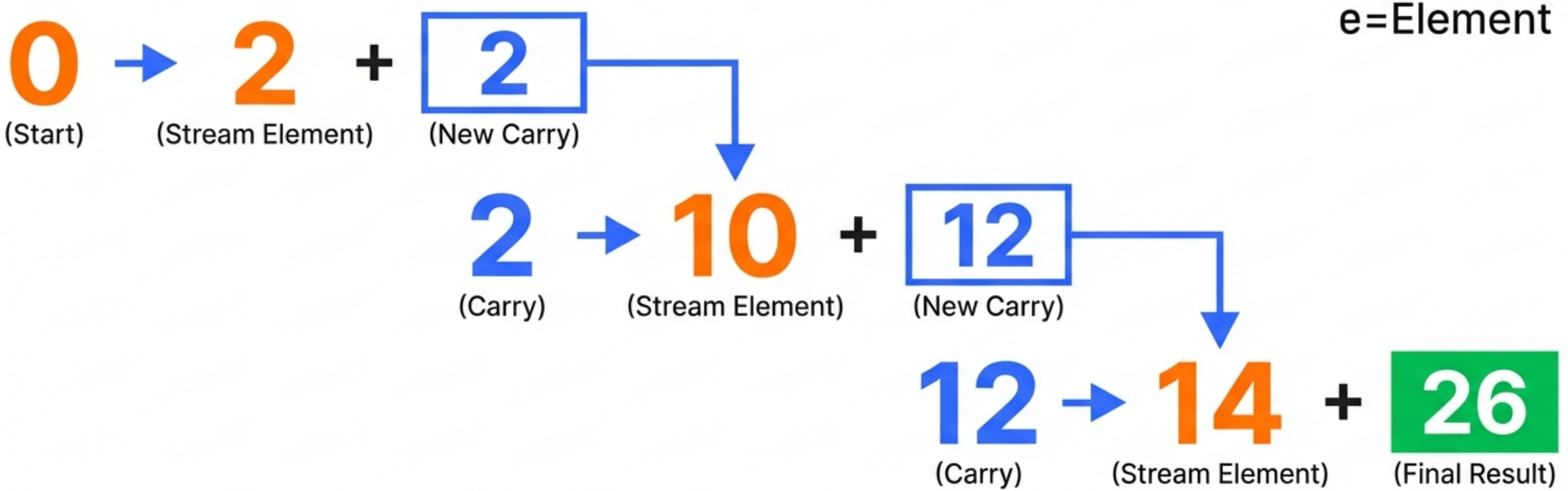
Message: `stream has already been operated upon or closed.`

Streams are not data structures; they are flows. Once a Terminal Operation (like **count** or **forEach**) executes, the stream is consumed. To process the data again, you must create a new stream from the source.

Reduce: Aggregation and Result

```
.reduce(0, (c, e) -> c + e)
```

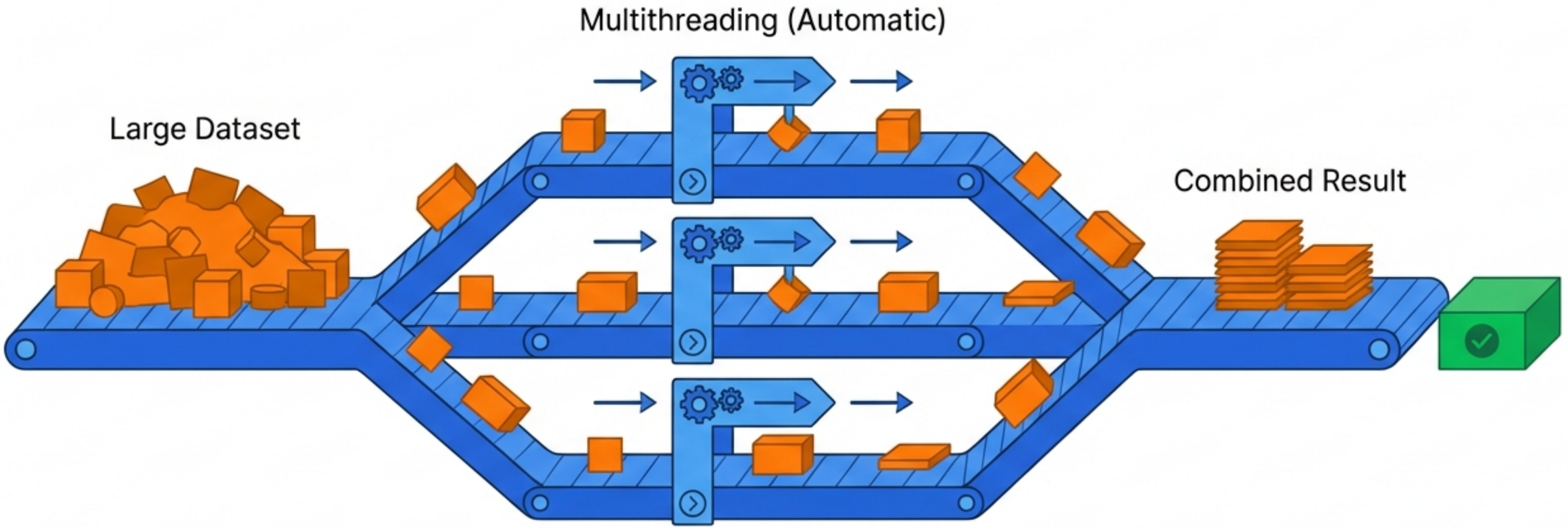
0=Start
c=Carry
e=Element



Like Big Data's MapReduce. It collapses a stream into a single value by repeatedly applying a formula to an accumulator and the next element.

Result: 26

Parallel Streams: Scaling Up




```
Standard: nums.stream()...
Parallel: nums.parallelStream()...
```

Multithreading without the headache. Java handles the thread management automatically, utilizing multiple cores to process large datasets faster. Caution: Avoid using with stateful operations or order-dependent logic.

The Declarative Shift

Immutability - Source data remains untouched; safe for complex apps.

Readability - Code describes *what* to do, not *how* to iterate.

Efficiency - Lazy evaluation and easy parallelism.

“From complex loops to a single, readable pipeline. The Stream API isn't just syntax; it's a better way to think about data.”