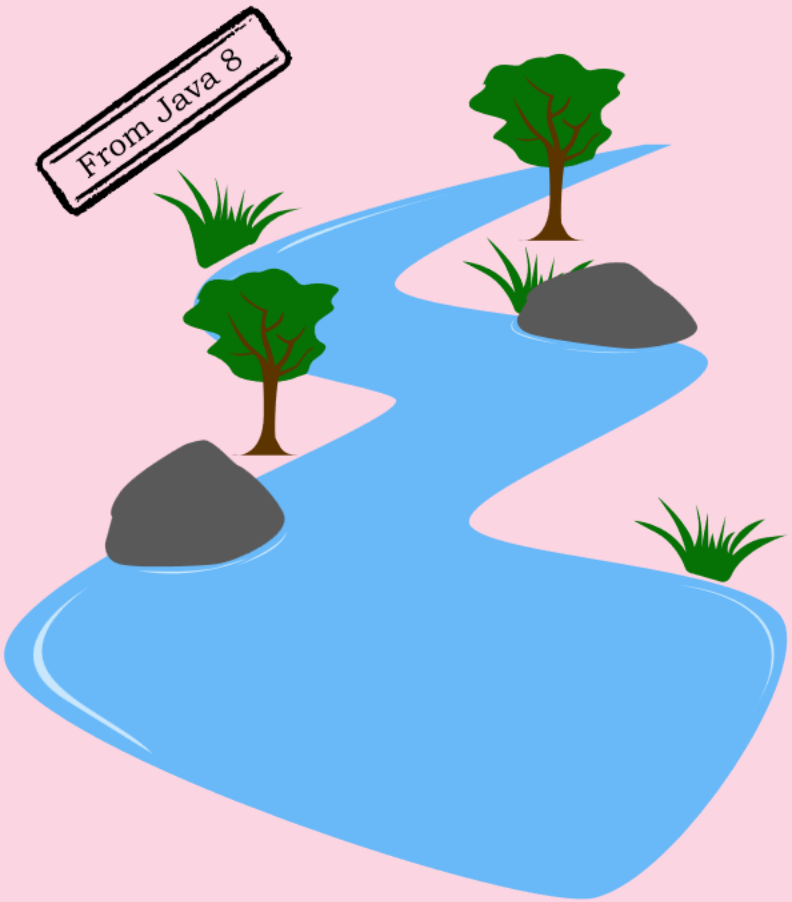
Streams API

eazy bytes

- ✓ Java 8 introduced **java.util.stream** API which has classes for processing collections of objects that we usually store inside the collections. The central API class is the **Stream<T>**



- ✓ Don't get confused with the java.io streams which are meant for processing the binary data to/from the files.

java.io streams != java.util streams

- ✓ Collections like List, Set will be used if we want to represent the group of similar objects as a single entity where as Streams will be used to process a group of objects present inside a collection.

- ✓ You can create streams from collections, arrays or iterators.

- ✓ In Streams the code is written in a declarative way: you specify what you want to achieve like in query style as opposed to specifying how to implement an operation.

Creating a Stream from collection or list of elements

eazy bytes

- ✓ We can create a stream using either by calling stream() default method introduced in all the collections to support streams or with the help of Stream.of()
- ✓ Processing the elements inside streams parallelly is very simple. We just need to call parallelStream() default method instead of stream()
- ✓ A stream does not store its elements. They may be stored in an underlying collection or generated on demand. Stream operations don't mutate their source. Instead, they return new streams that hold the result.

```
List<String> departmentList = new ArrayList<>();
departmentList.add("Supply");
departmentList.add("HR");
departmentList.add("Sales");
departmentList.add("Marketing");

Stream<String> depStream = departmentList.stream();
depStream.forEach(System.out::println);

Stream<String> inStream = Stream.of("Eazy", "Bytes", "Java");
inStream.forEach(System.out::println);

Stream<String> parallelStream = departmentList.parallelStream();
parallelStream.forEach(System.out::println);
```

Creating a Stream from collection or list of elements

eazy bytes

- ✓ Inside java.util.Arrays new static methods were added to convert an array into a stream,

`Arrays.stream(array)` – to create a stream from an array
`Arrays.stream(array, from, to)` – to create a stream from a part of an array

```
String[] arrayOfWords = { "Eazy", "Bytes" };
Stream<String> streamOfwords = Arrays.stream(arrayOfWords);
```

- ✓ To make an empty stream with no elements, we can use empty() method inside Stream class.

```
Stream<String> emptyStream = Stream.empty();
```

- ✓ To generate an infinite stream of elements which is suitable for stream of random elements, Stream has 2 static methods called `Stream.generate()` & `Stream.iterate()`

```
Stream.generate(new Random()::nextInt).forEach(System.out::println);

Stream.iterate(1, n->n+1).forEach(System.out::println);
```

Streams have no storage

eazy bytes

- A collection is an in-memory data structure that stores all its elements. Examples of collections include lists, sets, and maps.
- When you add elements to a collection, they are stored in the collection's memory space. All elements must exist in memory at the time of addition.
- Collections are designed to hold a fixed set of elements, and these elements reside in memory until removed from the collection.

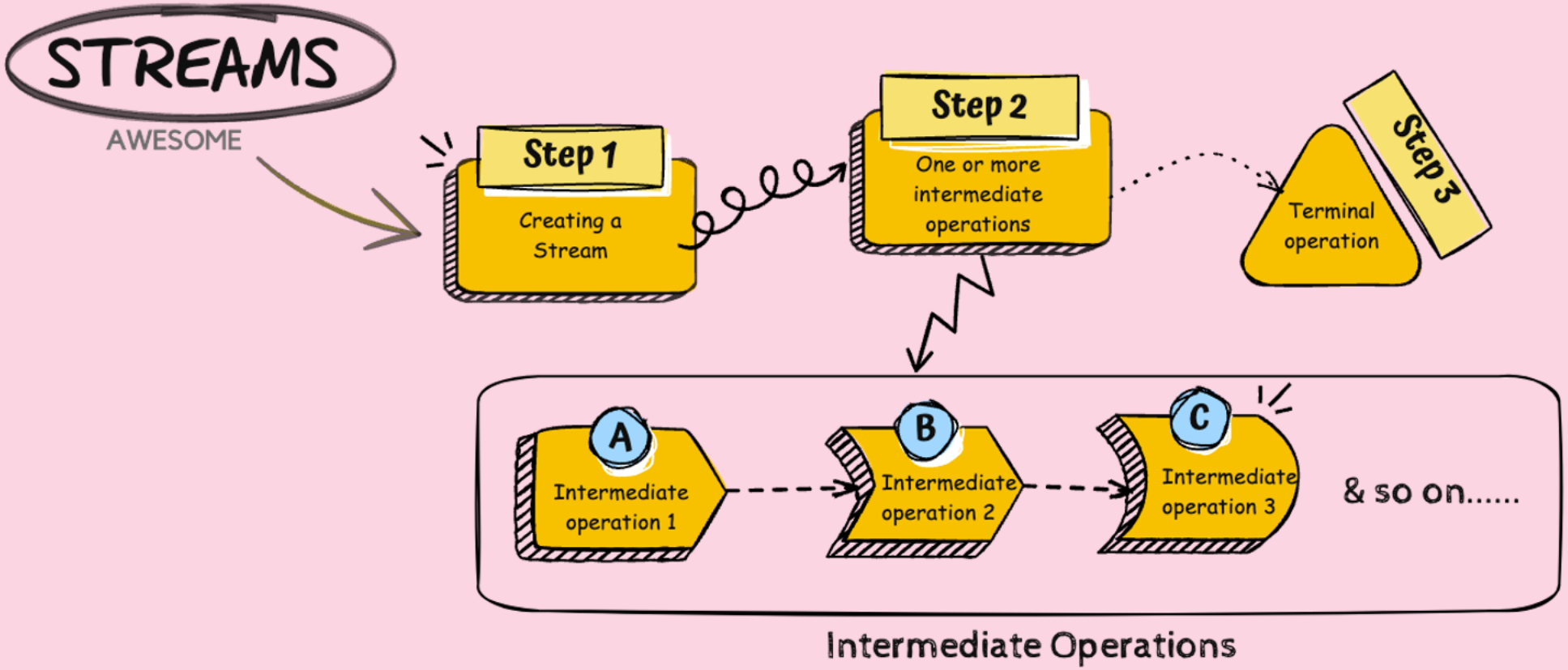
- A stream, on the other hand, does not have storage. It doesn't maintain a collection of elements in memory.
- Instead of storing elements, a stream pulls elements from a data source on demand. The data source can be an array, a collection, or any other source of data.
- Elements are processed in a stream through a series of operations (pipeline of operations) that can include filtering, mapping, sorting, and more.
- The stream operates lazily, meaning it only processes the elements as needed. It doesn't eagerly load all elements into memory at once.

Streams Pipeline

eazy bytes

When we work with streams, we set up a pipeline of operations in different stages as mentioned below.

1. Creating a stream using stream(), parallelStream() or Streams.of() etc.
2. One or more intermediate operations for transforming the initial stream into others or filtering etc.
3. Applying a terminal operation to produce a result.



Streams map() method (Intermediate operation)

eazy bytes

- ✓ If we have a scenario where we need to apply a business logic or transform each element inside a collection, we use map() method inside streams to process them
- ✓ In simple words, the map() is used to transform one object into other by applying a function. Stream map method takes lambda function as argument which is an implementation of a functional interface.
- ✓ Here in our example for each element inside our list, we need to transform them into uppercase letters before printing them on the console.

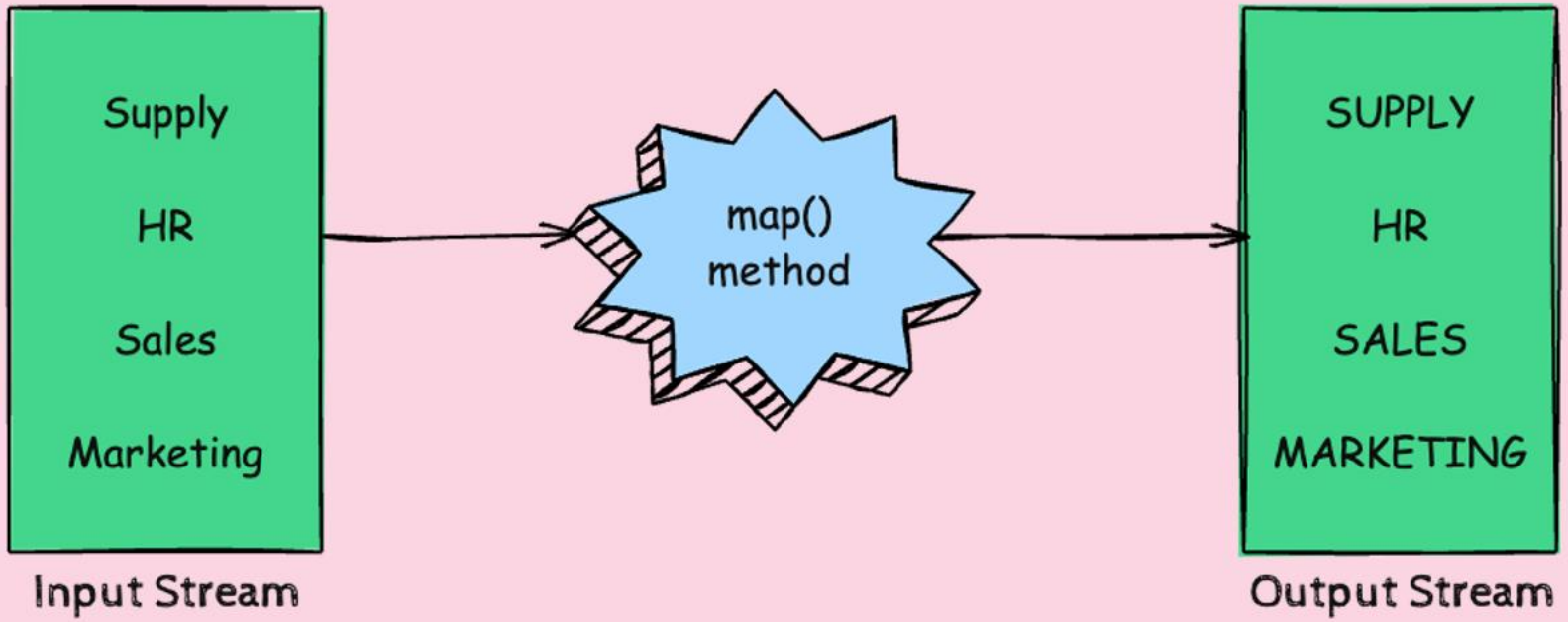
```
List<String> departmentList = new ArrayList<>();
departmentList.add("Supply");
departmentList.add("HR");
departmentList.add("Sales");
departmentList.add("Marketing");

departmentList.stream() // Stream creation
        .map(word -> word.toUpperCase()) // Intermediate operation
            .forEach(word->System.out.println(word)); // Terminal operation
```

Streams map() method (Intermediate operation)

eazy bytes

- ✓ Stream map(Function mapper) is an intermediate operation and it returns a new Stream as return value. These operations are always lazy.
- ✓ Stream operations don't mutate their source. Instead, they return new streams that hold the result.



Streams flatMap() method (Intermediate operation)

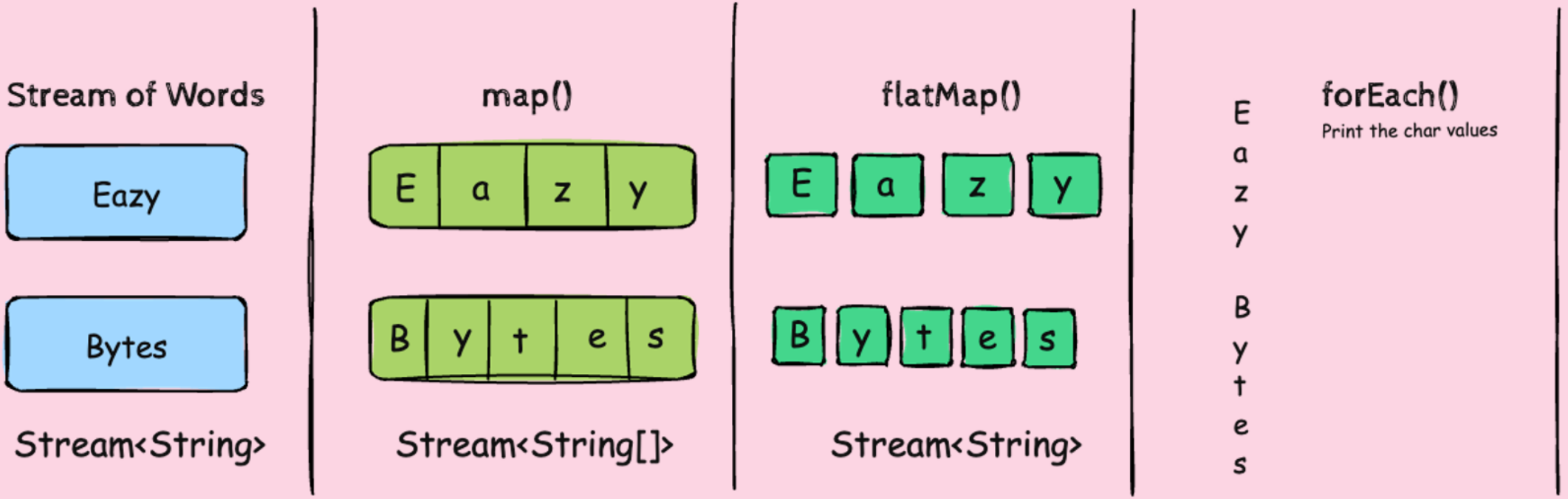
eazy bytes

- ✓ The flatMap method in Java Streams is used to handle scenarios where you have a stream of elements, and each element contains multiple sub-elements or has a one-to-many mapping. The primary purpose of flatMap is to flatten the stream of elements into a single stream by applying a mapping function that returns a stream for each element.

```
String[] arrayOfWords = { "Eazy", "Bytes" };
Stream<String> streamOfwords = Arrays.stream(arrayOfWords);

Stream<String[]> streamOfLetters = streamOfwords.map(word ->word.split(""));
streamOfLetters.flatMap(Arrays::stream).forEach(System.out::println);
```

flatMap is the combination of a map and a flat operation i.e. it applies a map function to elements as well as flatten them



Streams flatMap() method (Intermediate operation)

eazy bytes

- ✓ Suppose you have a class Person with a list of phone numbers, and you want to create a stream of all the phone numbers from a list of persons.

```
class Person {
    private String name;
    private List<String> phoneNumbers;

    public Person(String name, List<String> phoneNumbers) {
        this.name = name;
        this.phoneNumbers = phoneNumbers;
    }

    public String getName() {
        return name;
    }

    public List<String> getPhoneNumbers() {
        return phoneNumbers;
    }
}
```

```
public class FlatMapExample {

    public static void main(String[] args) {
        List<Person> persons = Arrays.asList(
                new Person("Alice", Arrays.asList("123", "456")),
                new Person("Bob", Arrays.asList("789", "101", "112"))
        );

        // Using map to extract phone numbers
        List<List<String>> mapResult = persons.stream()
                .map(Person::getPhoneNumbers)
                .collect(Collectors.toList());

        System.out.println("Using map: " + mapResult);

        // Using flatMap to flatten the stream of phone numbers
        List<String> flatMapResult = persons.stream()
                .flatMap(person -> person.getPhoneNumbers().stream())
                .collect(Collectors.toList());

        System.out.println("Using flatMap: " + flatMapResult);
    }

}
```

- With map, we get a stream of lists of phone numbers for each person.
- With flatMap, we get a single stream of phone numbers by flattening the lists.
- Using flatMap is particularly useful when you have a one-to-many relationship between elements, and you want to flatten the nested structure into a flat stream of elements.

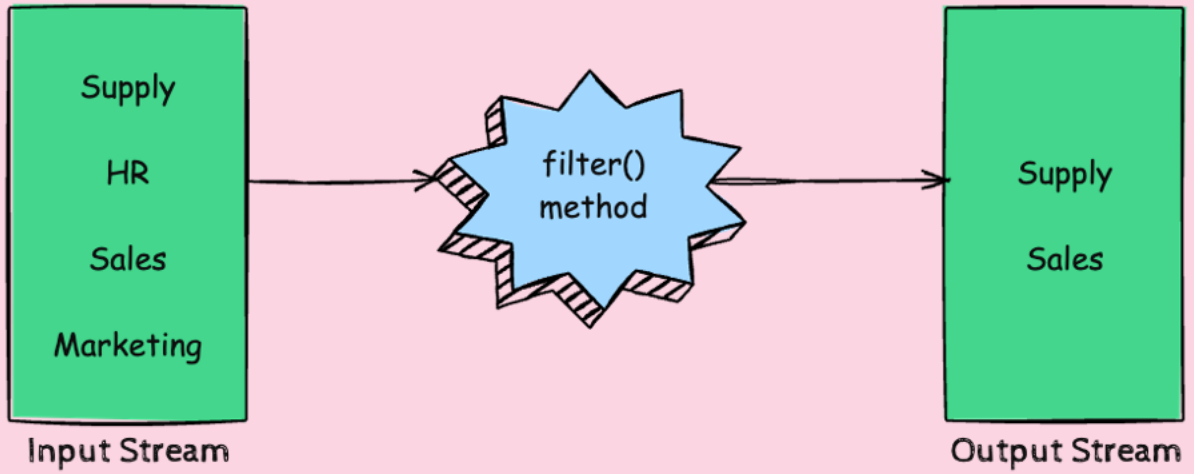
Streams filter() method (Intermediate operation)

eazy bytes

- ✓ If we have a scenario where we need to exclude certain elements inside a collection based on a condition, we can use filter() method inside streams to process them. Here in the below example, our requirement is to filter the departments name that starts with 'S' and print them on to the console
- ✓ Stream filter method takes Predicate as argument that is a functional interface which can act as a boolean to filter the elements based on condition defined. Stream filter(Predicate<T>) is an intermediate operation and it returns a new Stream as return value. These operations are always lazy.

```
List<String> departmentList = new ArrayList<>();
departmentList.add("Supply");
departmentList.add("HR");
departmentList.add("Sales");
departmentList.add("Marketing");

departmentList.stream()
        .filter(word -> word.startsWith("S"))
                .forEach(System.out::println);
```

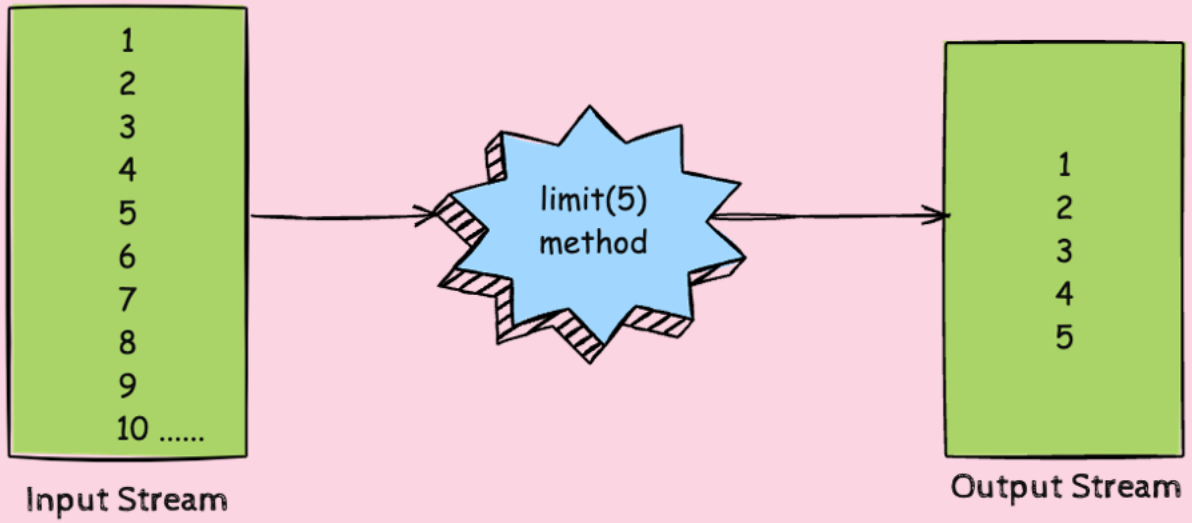


Streams limit() method (Intermediate operation)

eazy bytes

- ✓ If we have a scenario where we need to limit the number of elements inside a stream, we can use limit(n) method inside streams. The limit method takes a number which indicates the size of the elements we want to limit but this limit number should not be greater than the size of elements inside the stream.
- ✓ Note that limit also works on unordered streams (for example, if the source is a Set).In this case we shouldn't assume any order on the result produced by limit.
- ✓ Here in our example, we used generate method to provide random integer numbers. But since generate will provide infinite stream of numbers, we limit it to only first 10 elements.

```
Stream.generate(new Random()::nextInt)
        .limit(10)
                .forEach(System.out::println);
```

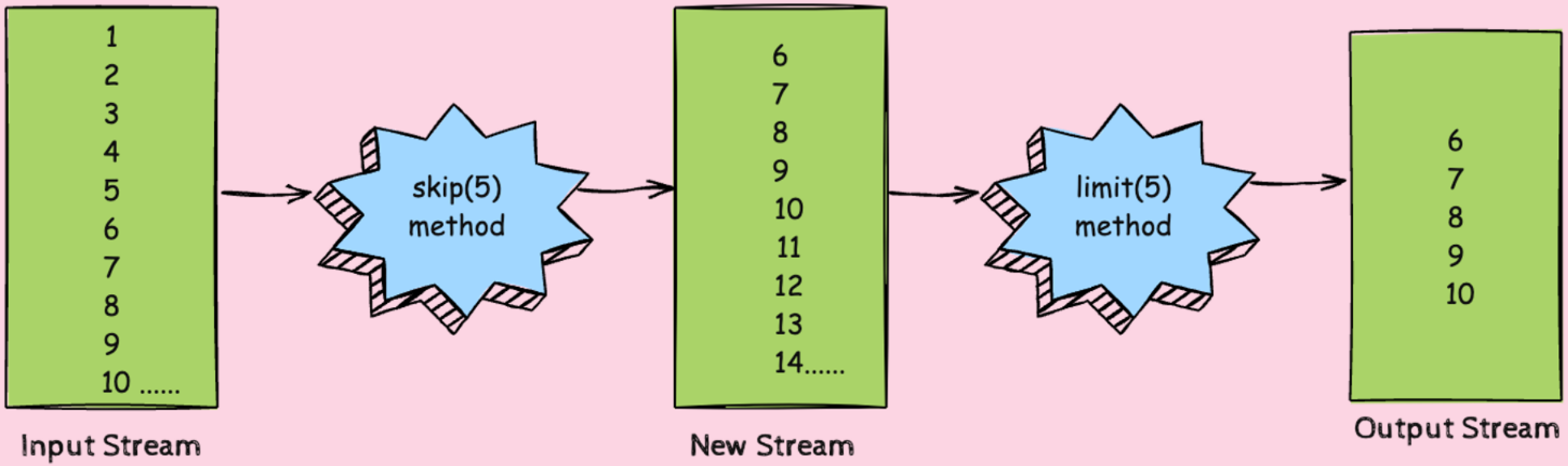


Streams skip() method (Intermediate operation)

eazy bytes

- ✓ Streams support the skip(n) method to return a stream that discards the first n elements. If the stream has fewer than n elements, an empty stream is returned. Note that limit(n) and skip(n) are complementary.
- ✓ Here in our example we used iterate method to provide integer numbers from 1. But since iterate will provide infinite stream of numbers, we skipped first 5 numbers and limit it to only 5 numbers. The output of this method will be the numbers from 6....10

```
Stream.iterate(1, n -> n + 1)
        .skip(5)
        .limit(5)
                .forEach(System.out::println);
```



Streams are traversable only once

eazy bytes

- ✓ Streams in Java are not reusable; they are designed to be one-shot objects. Once a terminal operation is invoked on a stream, it cannot be reused. If there's a need to perform computations on the same elements from the same data source again, it's necessary to recreate the entire stream pipeline. Attempting to reuse a stream may result in an IllegalStateException, as stream implementations are designed to detect and prevent such reuse.
- ✓ For example like in the below example, if we try to traverse the stream again after consuming all the elements inside it, we will get an runtime java.lang.IllegalStateException with a message 'stream has already been operated upon or closed'

```
List<String> departmentList = new ArrayList<>();
departmentList.add("Supply");
departmentList.add("HR");
departmentList.add("Sales");
departmentList.add("Marketing");
Stream<String> depStream = departmentList.stream();
depStream.forEach(System.out::println);
depStream.forEach(System.out::println); // IllegalStateException
```

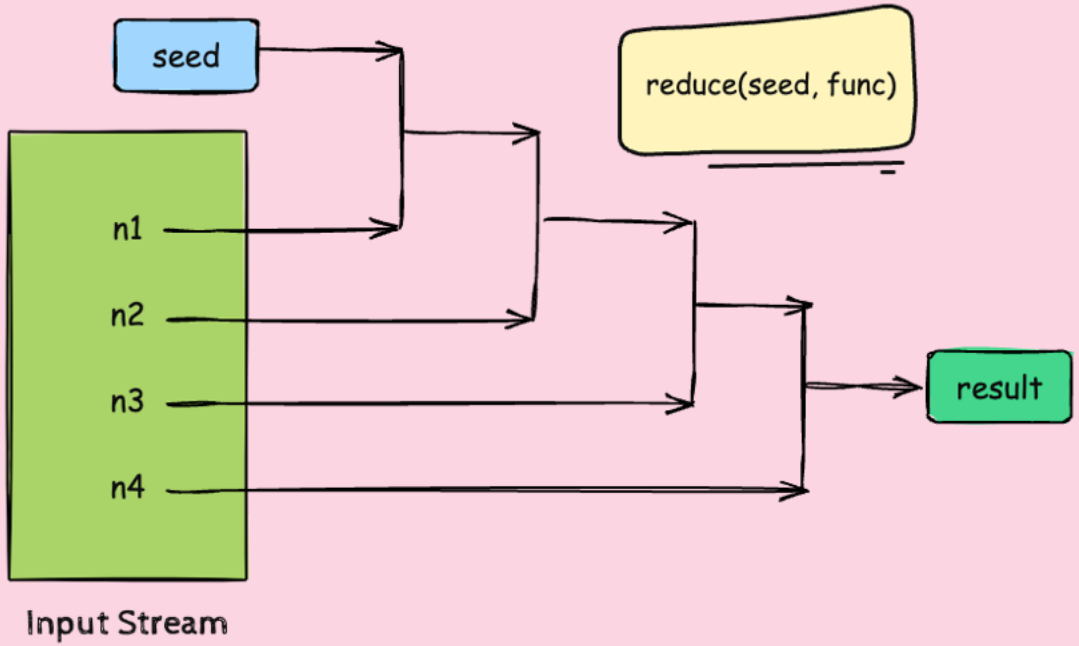
Streams reduce() method (Terminal operation)

eazy bytes

- ✓ The reduce operation, also known as a reduction operation or a fold, involves combining all elements of a stream to generate a single value. This is achieved by iteratively applying a combining function. Examples of reduce operations include computing the sum, maximum, average, count, and similar aggregates for a stream of integers.

- ✓ In the reduce operation, two parameters are involved: a seed (also known as an initial value) and an accumulator, which is a function. When the stream is empty, the result is the seed. However, if the stream contains elements, the seed signifies a partial result. The accumulator takes the partial result and an element, producing another partial result. This process continues until all elements have been processed by the accumulator. The final value returned from the accumulator represents the result of the reduce operation.

- ✓ Specialized reduce operations like sum(), max(), min(), count(), etc., are provided by certain stream interfaces.



- ✓ It's important to note that these specialized methods are not universally available for all types of streams. For instance, including a sum() method in the general Stream<T> interface doesn't have meaningful application, as adding reference-type elements like two people isn't meaningful. Consequently, methods such as sum() are specific to interfaces like IntStream, LongStream, and DoubleStream. On the other hand, the count() method, which calculates the number of elements in a stream, is applicable to all types of streams.

Streams reduce() method (Terminal operation)

eazy bytes

- ✓ The below code uses the Stream API's reduce operation to find the sum of the elements in the list of integers, and the result is then printed. We provided the seed as 0 along with the function logic that needs to be executed. In this example, it prints the sum of 1 + 2 + 3 + 4 + 5, which is 15.

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

int sum = numbers.stream()
                 .reduce(0, Integer::sum);

System.out.println(sum);
```

- ✓ Below code snippet calculates the total sum of incomes for a list of Person objects using the Stream API. It assumes that each Person object has an associated income, and the getIncome method is used to extract these incomes for the reduction operation. The result is then printed to the console.

```
double sum = persons.stream()
                    .map(Person::getIncome)
                    .reduce(0.0, Double::sum);
System.out.println(sum);
```

Streams collect() method (Terminal operation)

eazy bytes

- ✓ The collect() method in the Java Stream API is used to transform the elements of a stream into a different form, usually a collection like a List, Set, or Map. It allows you to accumulate the elements of a stream into a mutable result container, specified by a Collector parameter.

```
List<String> departmentList = new ArrayList<>();
departmentList.add("Supply");
departmentList.add("HR");
departmentList.add("Sales");
departmentList.add("Marketing");

Stream<String> depStream = departmentList.stream();
List<String> newDepartmentList = depStream
                .filter(word -> word.startsWith("S"))
                        .collect(Collectors.toList());
newDepartmentList.forEach(System.out::println);
```

- ✓ Here in our example we first used filter() method to identify the elements that start with 'S', post that we used collect() method to convert the stream into a list of objects.

- ✓ Recognizing the complexity of creating custom collectors, the designers of the Streams API introduced a convenient utility class named `Collectors`. This class offers ready-made implementations for commonly used collectors. Among the frequently employed methods in the `Collectors` class are `toList()`, `toSet()`, and `toCollection()`. The `toList()` method produces a Collector for gathering data into a List, while `toSet()` creates a Collector for collecting data into a Set. The `toCollection()` method allows customization by taking a Supplier that produces a Collection for data aggregation.

Streams collect() method (Terminal operation)

eazy bytes

✓ The following example of code collects all names of persons in a List<String> :

```
List<String> personNames = persons.stream()
                                  .map(Person::getName)
                                      .collect(Collectors.toList());
System.out.println(personNames);
```

✓ Other important methods inside Collector class are,

- toSet() – Convert stream into a set
- toCollection() - Convert stream into a collection
- toMap() – Convert stream into a Map after applying key/value determination function.
- counting() – Counting number of stream elements
- joining() - For concatenation of stream elements into a single String
- minBy() - To find minimum of all stream elements based on given Comparator
- maxBy() - To find maximum of all stream elements based on given Comparator
- reducing() - Reducing elements of stream based on BinaryOperator function provided

Streams collectingAndThen() method (Terminal operation)

eazy bytes

- ✓ The collectingAndThen() method lets you modify the results of a collector after the collector has collected all elements. Its first argument is a collector that collects the data. The second argument is a finisher that is a function. The finisher is passed a result, and it is free to modify the result, including its type. The return type of such a collector is the return type of the finisher.

- ✓ It accepts 2 parameters,

- 1st input parameter is downstream which is an instance of a Collector<T,A,R> i.e. the standard definition of a collector. In other words, any collector can be used here.
- 2nd input parameter is finisher which needs to be an instance of a Function<R,RR> functional interface.

- ✓ Below is the sample code snippet that prints the name of the Product which has maximum price,

```
List<Product> productList = Arrays.asList(new Product("Apple", 1200), new Product("Samsung",
        1000), new Product("Nokia", 800), new Product("BlackBerry", 1000), new Product("
        Apple Pro Max", 1500), new Product("Mi", 800), new Product("OnePlus", 1000));

String maxPriceProduct = productList.stream()
        .collect(Collectors.collectingAndThen(Collectors.maxBy(Comparator.comparing(Product::
                getPrice)), (Optional<Product> product)-> product.isPresent()? product.get().
                getName():"None"));

System.out.println("The product with max price tag is: " + maxPriceProduct);
```

Streams groupingBy() method (Terminal operation)

eazy bytes

- ✓ The groupingBy() method of Collectors class in Java are used for grouping objects by some property and storing results in a Map instance. In order to use it, we always need to specify a property by which the grouping would be performed. This method provides similar functionality to SQL's GROUP BY clause.

- ✓ Below is the sample code where we pass to the groupingBy method a Function (expressed in the form of a method reference) extracting the corresponding Product.getPrice for each Product in the stream. We call this Function a classification function specifically because it's used to classify the elements of the stream into different groups.

```
List<Product> productList = Arrays.asList(new Product("Apple", 1200), new Product("
        Samsung", 1000), new Product("Nokia", 800), new Product("BlackBerry", 1000),
        new Product("Apple Pro Max", 1500), new Product("Mi", 800), new Product("
        OnePlus", 1000));

Map<Integer, List<Product>> groupByPriceMap = productList.stream()
                        .collect(Collectors.groupingBy(Product::getPrice));

System.out.println("The list of products grouped by price is: " + groupByPriceMap);
```

Streams partitioningBy() method (Terminal operation)

eazy bytes

- ✓ Collectors partitioningBy() method is used to partition a stream of objects(or a set of elements) based on a given predicate. The fact that the partitioning function returns a boolean means the resulting grouping Map will have a Boolean as a key type, and therefore,there can be at most two different groups—one for true and one for false.

- ✓ Below is the sample code where we pass to the partitioningBy method a predicate function to partition all the products into >$1000 and <=$1000.

```
List<Product> productList = Arrays.asList(new Product("Apple", 1200), new Product("
        Samsung", 1000), new Product("Nokia", 800), new Product("BlackBerry", 1000),
        new Product("Apple Pro Max", 1500), new Product("Mi", 800), new Product("
        OnePlus", 1000));

Map<Boolean, List<Product>> products = productList.stream()
                .collect(Collectors.partitioningBy(product -> product.getPrice() >
                        1000));
System.out.println("The list of products partitioned by price is: " + products);
```

- ✓ Compared to filters, partitioning has the advantage of keeping both lists of the stream elements, for which the application of the partitioning function returns true or false.

Finding and Matching methods in Streams (Terminal operation)

eazy bytes

- ✓ Below are the methods in the Stream interface that are used to perform find and match operations. All find and match operations are terminal operations.

    - boolean allMatch(Predicate<? super T> predicate)
    - boolean anyMatch(Predicate<? super T> predicate)
    - boolean noneMatch(Predicate<? super T> predicate)
    - Optional<T> findAny()
    - Optional<T> findFirst()

- ✓ The operations mentioned are also classified as short-circuiting operations. Short-circuiting operations have the ability to return a result without processing the entire stream. For instance, consider the `allMatch()` method, which verifies if a specified predicate holds true for all elements in the stream. In this case, the method can promptly return false upon encountering a single element where the predicate evaluates to false. Once a false result is determined, further processing (short-circuiting) of elements is halted, and the final result is returned as false.

- ✓ The same principle applies to other methods. For example, `findAny()` and `findFirst()` return an `Optional<T>` as their result type. This is because these methods may not produce a result if the stream is empty. The short-circuiting behavior ensures that as soon as a matching element is found, or the first element is encountered, the processing is terminated, and the result is provided.

Finding and Matching methods in Streams (Terminal operation)

eazy bytes

- ✓ Let's consider a scenario where we have a list of Item objects, and we want to perform various checks using stream operations:

```
public class Item {

    private String name;
    private double price;
    private boolean inStock;
    private boolean onSale;

    public Item(String name, double price, boolean inStock, boolean onSale) {
        this.name = name;
        this.price = price;
        this.inStock = inStock;
        this.onSale = onSale;
    }

    // Getter, ToString

}
```

```
// Get the list of Items
private static List<Item> getItems() {
    return List.of(
            new Item("Laptop", 1200, true, false),
            new Item("Smartphone", 800, true, true),
            new Item("Headphones", 150, false, true),
            new Item("Camera", 2000, true, true),
            new Item("Smartwatch", 300, true, false)
    );
}
```

```
// Check if all items in stock
boolean allInStock = items.stream().allMatch(Item::isInStock);
System.out.println(allInStock); // false
```

```
// Check if any item is on sale
boolean anyOnSale = items.stream().anyMatch(Item::isOnSale);
System.out.println(anyOnSale); // true
```

Finding and Matching methods in Streams (Terminal operation)

eazy bytes

```
// Check if any item is out of stock
boolean anyOutOfStock = items.stream().anyMatch(item -> !item.isInStock());
System.out.println(anyOutOfStock); // true
```

```
// Check if all items are less than or equal to 5000
boolean isAnyCostlyProduct = items.stream().noneMatch(item -> item.getPrice()>5000);
System.out.println(isAnyCostlyProduct); // true
```

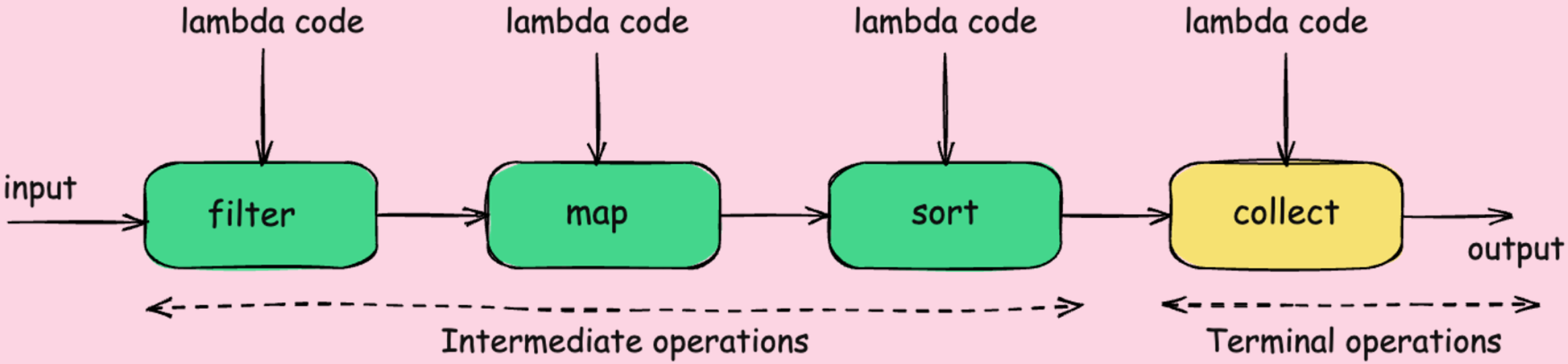
```
// Fetch a random item that has price greater than 1000
Optional<Item> randomItemOptional = items.stream().filter(item -> item.getPrice()>1000).findAny();
System.out.println(randomItemOptional);
```

```
// Fetch first item that has price greater than 1000
Optional<Item> firstItemOptional = items.stream().filter(item -> item.getPrice()>1000).findFirst();
System.out.println(firstItemOptional);
```

Chaining stream operations to form a stream pipeline

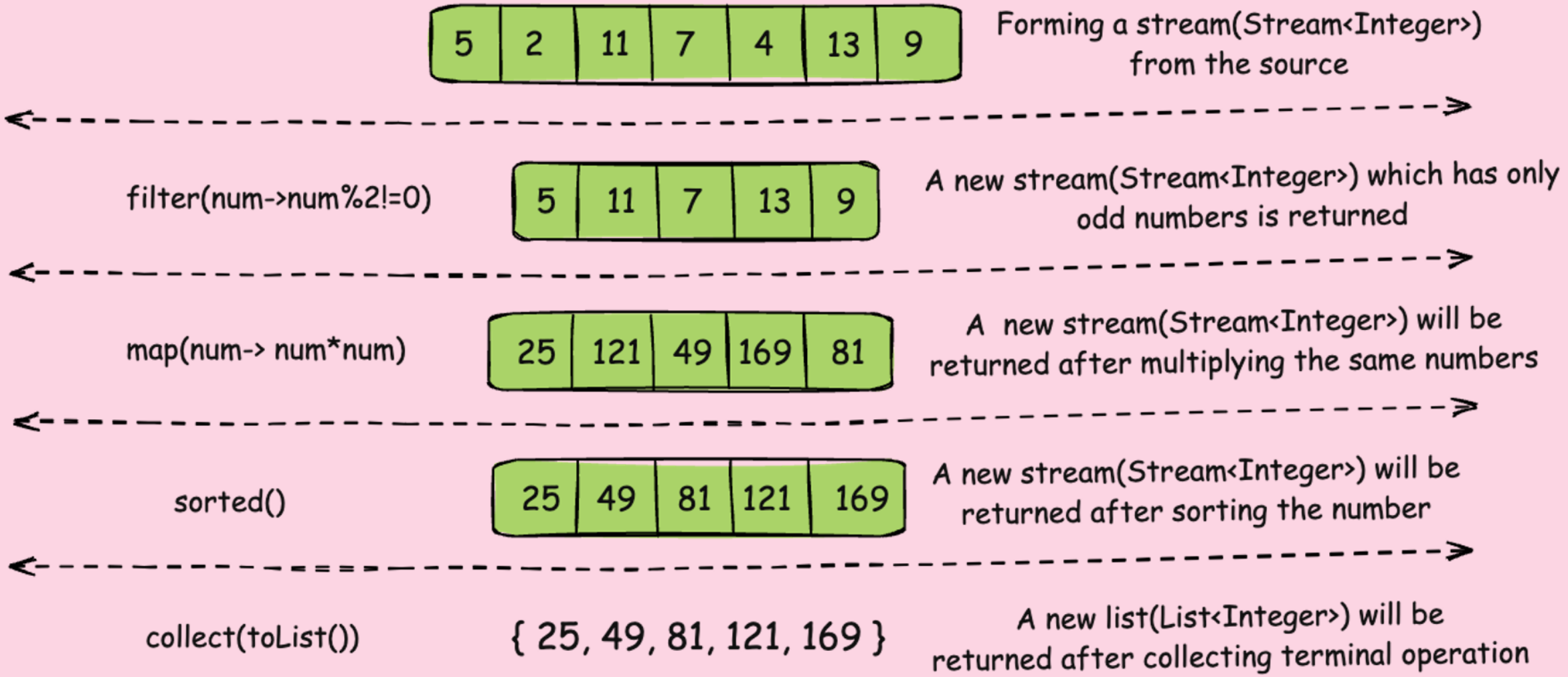
eazy bytes

- ✓ We can form a chain of stream operations using intermediate operations and a single terminal operation to achieve a desire output. This we also call as stream pipeline.
- ✓ Suppose think of an example where I have list of int values inside a list where I want to filter all odd numbers, followed by converting the remaining numbers by multiply themselves, sorting and at last display the output in a new list.



Chaining stream operations to form a stream pipeline

eazy bytes



parallelStream()

eazy bytes

- ✓ Streams in Java can be processed as either sequential or parallel. In a sequential stream, operations are carried out serially using a single thread. On the other hand, a parallel stream allows operations to be processed concurrently using multiple threads. The convenience of working with sequential or parallel streams is notable because you don't have to undertake additional steps for processing.
- ✓ To turn a collection into a parallel stream, we just have to invoke the method parallelStream() on the collection source.
- ✓ A parallel stream is a stream that splits its elements into multiple chunks, processing each chunk with a different thread. Thus, you can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy.
- ✓ Here in the example, we have a list of departments which need to be displayed. For the same we took the parallelStream and print each element. This will happen parallelly and the order of elements displayed will not be guaranteed.

```
List<String> departmentList = new ArrayList<>();
departmentList.add("Supply");
departmentList.add("HR");
departmentList.add("Sales");
departmentList.add("Marketing");
departmentList.add("Insurance");
departmentList.add("Security");
departmentList.add("Finance");

departmentList.parallelStream().forEach(System.out::println);
```

- ✓ If we want to convert a sequential stream into a parallel one, just invoke the method parallel() on the stream.

parallelStream()

eazy bytes

✓ This code snippet first filters major persons sequentially, then switches to parallel processing for extracting and combining their names. The parallelism is introduced by the .parallel() method, and the sequential and parallel portions of the stream are seamlessly integrated by the Streams API.

```
String names = persons.stream()
                .filter(Person::isMajor) // Processed in serial
                .parallel()
                    .map(Person::getName)   // Processed in parallel
                .collect(Collectors.joining(", "));
```

Collections Vs Streams

eazy bytes

Collections

A collection is an in-memory data structure that holds all the values the data structure currently has

A collection is eagerly constructed and elements can be added or removed

Collections doesn't use functional interfaces using lambda expressions

They are non-consumable i.e. can be traversable multiple times without creating it again

Collections are iterated using external loops like for, while

Streams

A stream does not store its elements. They may be stored in an underlying collection or generated on demand

Stream is like a lazily constructed collection and elements inside streams can't be added or removed

Streams uses lot of functional interfaces using lambda expressions

Streams are consumable i.e. to traverse the stream, it needs to be created every time.

Streams are iterated internally based on the operation mentioned like map, filter.