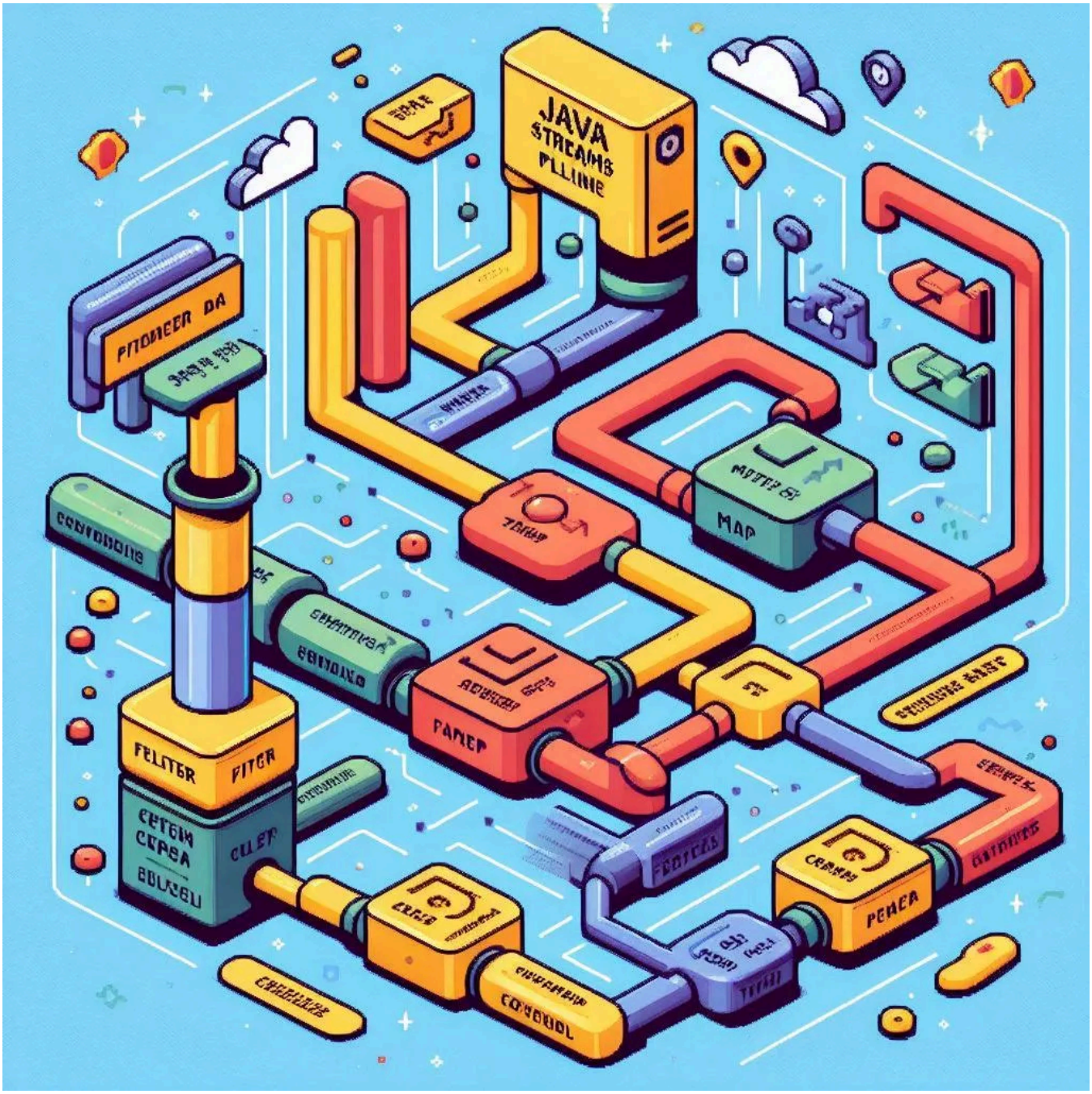
Streamline Your Java Code: A Cheat Sheet for Mastering Streams



Java Streams

Java Streams, introduced in Java 8, revolutionized how we work with collections. They offer a declarative and functional approach to processing data, making code

more readable and maintainable. But with so many operations available, it's easy to feel overwhelmed.

This cheat sheet is your one-stop guide to conquering Java Streams! We'll break down the essential operations, from filtering and mapping to sorting and reducing. Along the way, we'll provide clear examples using a list of integers to solidify your understanding. Whether you're a seasoned Java developer or just getting started with streams, this cheat sheet will equip you with the knowledge to unlock their full potential. Let's dive in and streamline your Java code!

Stream Creation

Here's a deeper dive into the different ways you can create streams:

1. From Collections:

Most collection classes like `List`, `Set`, and `Map` offer a convenient `stream()` method. This method returns a stream containing all the elements of the collection, allowing you to perform various operations on them. **Example:**

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

2. From Arrays:

Java arrays can also be converted into streams using the static `stream(arrayName)` method of the `Arrays` class. This is particularly useful when you're working with data pre-populated in arrays. **Example:**

```
int[] numbers = {1, 3, 4, 2, 5};
IntStream numberStream = Arrays.stream(numbers);
```

3. Of Primitives:

For primitive data types like `int`, `long`, and `double`, Java provides the `Stream.of(element1, element2, ... )` method. This allows you to create a stream

directly from a list of primitive values. **Example:**

```
DoubleStream doubleStream = Stream.of(2.5, 3.14, 1.0);
```

4. Stream Builder:

The `Stream.builder()` method offers a more flexible approach to creating streams. You can add elements one by one using methods like `add(element)` or build a stream from another collection using `addAll(collection)`. This is useful when you need to construct a stream dynamically or perform transformations before creating the final stream. **Example:**

```
Stream<String> customStream = Stream.builder()
  .add("Apple")
  .add("Banana")
  .filter(fruit -> fruit.startsWith("A"))  // Apply a filter
                                           //before building the stream
  .build();
```

By understanding these different ways to create streams, you can choose the most appropriate method based on your data source and processing needs.

Intermediate Stream Operations:

Intermediate operations in Java Streams provide powerful ways to transform and filter data within a stream. These operations don't actually execute until a terminal operation is called, allowing for efficient processing. Here's a closer look at some key intermediate operations:

1. **filter(predicate):** The `filter(predicate)` operation in Java Streams acts as a gatekeeper, allowing you to select only specific elements that meet a certain criteria. For example, let's filter even numbers from a list of integers:

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

```
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());  // evenNumbers = [2, 4]
```

2. map(mapperFunction): The `map(mapperFunction)` operation is a cornerstone of Java Streams, allowing you to transform each element within a stream into a new element. For example, let's try to convert names in a list to uppercase:

```
List<String> names = Arrays.asList("alice", "bob", "charlie");
List<String> upperCaseNames = names.stream()
  .map(String::toUpperCase)
  .collect(Collectors.toList());
// upperCaseNames = ["ALICE", "BOB", "CHARLIE"]
```

3. flatMap(mapperFunction): The `flatMap(mapperFunction)` operation in Java Streams is a powerful tool for dealing with nested collections. It allows you to flatten multiple nested streams into a single, level stream, making it easier to process the elements within.
For example : Let's combine nested lists of products (assuming a `Product` class with a `getCategories()` method returning a list of strings):

```
List<Product> products = Arrays.asList(
  new Product("Laptop", Arrays.asList("electronics", "computers")),
  new Product("Shirt", Arrays.asList("clothing", "apparel"))
);
List<String> allCategories = products.stream()
  .flatMap(product -> product.getCategories().stream())
  .distinct()  // Remove duplicates if needed
  .collect(Collectors.toList());
// allCategories = ["electronics", "computers", "clothing", "apparel"]
```

4. peek(consumer): The `peek(consumer)` operation in Java Streams might seem straightforward at first glance. As the name suggest it allows us to peek into and perform side effects (like printing) on each element within the stream. **This should be used very cautiously!**

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
numbers.stream()
  .peek(System.out::println)  // Prints each number on a new line
  .collect(Collectors.toList());
// This terminal operation triggers the stream processing
```

5. distinct(): The `distinct()` operation in Java Streams acts as a filter, ensuring that only unique elements remain in the resulting stream. For example,

```
List<String> fruits = Arrays.asList("apple", "banana", "apple", "orange");
List<String> uniqueFruits = fruits.stream()
  .distinct()
  .collect(Collectors.toList());
// uniqueFruits = ["apple", "banana", "orange"]
```

6. sorted(comparator): The `sorted(comparator)` operation in Java Streams empowers you to efficiently sort elements within a stream. It provides flexibility for both natural ordering and custom sorting logic based on your specific requirements. For example, we can sort list of products based on it's price

```
List<Product> products = Arrays.asList(
  new Product("Laptop", 1000),
  new Product("Shirt", 20),
  new Product("Phone", 500)
);
List<Product> sortedByPrice = products.stream()
  .sorted(Comparator.comparingInt(Product::getPrice))// Sorts in ascending
// order by price
  .collect(Collectors.toList());
```

7. limit(n): The `limit(n)` operation in Java Streams acts as a gatekeeper, allowing you to restrict the number of elements processed from the stream to a specific maximum (`n`). For example, get the first 3 elements from a list of strings:

```
List<String> words = Arrays.asList("hello", "world", "how", "are", "you");
List<String> firstThreeWords = words.stream()
  .limit(3)
  .collect(Collectors.toList());
// firstThreeWords = ["hello", "world", "how"]
```

8. skip(n): The `skip(n)` operation in Java Streams allows you to bypass a specific number of elements at the beginning of the stream, essentially starting the processing from a later point. For example, skip the first 2 elements from a list of integers and get the remaining ones:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> allExceptFirstTwo = numbers.stream()
  .skip(2)
  .collect(Collectors.toList());
// allExceptFirstTwo = [3, 4, 5]
```

Terminal Operations: Collect & Analyze

Terminal operations mark the end of a stream processing pipeline in Java Streams. They consume the elements in the stream and produce a result or perform a side effect.

1. **collect(Collector):** The `collect(Collector)` operation in Java Streams reigns supreme as the most versatile terminal operation. It allows you to transform a stream of elements into a variety of collection types, custom data structures, or perform aggregations.
   Common Built-in Collectors:
   Java provides a rich set of pre-defined collectors for common collection types:

- `Collectors.toList()` : Collects elements into a `List` . For example,

```
List<String> uniqueProductNames = products.stream()
  .map(Product::getName)
  .collect(Collectors.toList());
```

- `Collectors.toSet()` : Collects elements into a `Set` (removes duplicates). For example,

```
List<String> uniqueProductNames = products.stream()
  .map(Product::getName)
  .collect(Collectors.toSet());
```

- `Collectors.toMap(keyMapper, valueMapper)` : Collects elements into a `Map` based on provided key and value mappers. For example,

```
Map<String, List<Product>> productsByCategory = products.stream()
  .collect(Collectors.groupingBy(Product::getCategory));
```

- `Collectors.joining(delimiter)` : Concatenates elements into a single String with a specified delimiter. For example,

```
String allNames = names.stream()
  .collect(Collectors.joining(", "));
```

- `Collectors.counting()` : Counts the number of elements in the stream. For example,

```
// Count the number of even elements in the stream
long totalElements = numbers.stream()
                    .filter(x -> x % 2 == 0)
                    .collect(Collectors.counting());
```

- `Collectors.averagingInt/Double/Long(mapper)` : Calculates the average of a numeric property extracted using a mapper function. For example,

```
// Calculate the average of the integers (converts to double)
double average = numbers.stream()
    .collect(Collectors.averagingInt(Integer::intValue));
```

- `Collectors.summarizingInt/Double/Long(mapper)` : Provides detailed summary statistics like sum, average, min, max, count for a numeric property. For example,

```
IntSummaryStatistics stats = numbers.stream()
        .collect(Collectors
.summarizingInt(Integer::intValue));  // Can be simplified to
// Collectors.summarizingInt()
    System.out.println("Count: " + stats.getCount());
    System.out.println("Sum: " + stats.getSum());
    System.out.println("Min: " + stats.getMin());
    System.out.println("Max: " + stats.getMax());
    System.out.println("Average: "
+ stats.getAverage());  // Same as Collectors.averagingInt
```

2. count(): The `count()` operation in Java Streams streamlines the process of calculating the total number of elements within a stream. For example,

```
// Count the number of even elements in the stream
    long totalElements = numbers.stream()
                         .filter(x -> x % 2 == 0)
                         .count();
```

3. min(Comparator) & max(Comparator): The `min(Comparator)` and `max(Comparator)` operations in Java Streams empower you to identify the minimum or maximum element within a stream based on based on a provided `Comparator` or if no `Comparator` is specified, the natural ordering of the elements is used. For example,

```
Optional<Product> cheapestProduct = products.stream()
  .min(Comparator.comparingInt(Product::getPrice));
```

```
Optional<String> longestName = names.stream()
  .max(Comparator.comparingInt(String::length));
```

4. forEach(consumer): The `forEach(consumer)` operation in Java Streams allows you to iterate over each element in a stream and perform a specific side effect using a provided `consumer` function.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream().forEach(System.out::println);
// Prints each name on a new line
```

5. anyMatch(predicate) & allMatch(predicate): The `anyMatch(predicate)` and `allMatch(predicate)` operations in Java Streams empower you to efficiently check if elements within a stream satisfy a particular condition.
Java

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
// Check if any number is even boolean
hasEvenNumber = numbers.stream()
          .anyMatch(num -> num % 2 == 0);  // hasEvenNumber will be true
// Check if any number is greater than 10 (all are less than 10 in this case)
boolean hasNumberGreaterThan10 = numbers.stream().anyMatch(num -> num > 10);
 // hasNumberGreaterThan10 will be false
```

6. findFirst() & findAny(): In Java Streams, `findFirst()` and `findAny()` serve the purpose of retrieving elements that meet specific criteria.
For example,

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
// Find the first even number
Optional<Integer> firstEven = numbers.stream()
  .filter(num -> num % 2 == 0)
  .findFirst();// Stops processing the stream elements
// as soon as it finds a matching element,
// making it potentially efficient for large streams.
if (firstEven.isPresent()) {
```

```
  System.out.println(firstEven.get()); // Output: 2
} else {
  System.out.println("No even number found");
}

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Find any name starting with "A" (order doesn't matter)
Optional<String> anyStartingWithA = names.stream()
  .filter(name -> name.startsWith("A"))
  .findAny();

if (anyStartingWithA.isPresent()) {
  System.out.println(anyStartingWithA.get()); // Could
 //print "Alice" or any other matching name
} else {
  System.out.println("No name starting with 'A' found");
}
```

Please note that In contrast to `findFirst()`, `findAny()` does not guarantee the order in which elements are processed, especially in parallel streams. It can return any matching element it encounters first.

7. Custom Collectors : The true power of `collect` lies in its ability to create custom collectors for specific needs. You can implement the `Collector` interface to define tailored accumulation logic for different data structures or transformations. Here's a basic example,

```
public static Collector<Product, List<String>,
          List<String>> collectProductDescriptions() {
  return Collector.of(
    // Supplier to create a new list for descriptions
    () -> new ArrayList<>(),
    // Accumulator to add product description to the list
    (list, product) -> list.add(product.getDescription()),
    // Combiner to merge two description lists
    // (unused for sequential streams)
    (list1, list2) -> { list1.addAll(list2); return list1; },
    // Finisher to return the final list of descriptions
    //(can be identity for List)
    Function.identity()
  );
}
```

```
List<String> allDescriptions = products.stream()
  .collect(collectProductDescriptions());
```

First of all thank you for reaching this far.
This post has unpacked a range of Java Stream terminal operations. Stay tuned for the next part, where we'll delve into more complex operations and explore how to create powerful stream processing pipelines!