**Java Stream Operations**

Intermediate Operations

1. \*\*filter\*\*: Filters elements.

- `lst.stream().filter(u -> u.getAge() == 26).forEach(System.out::println);`

- Output: `Users{name='Prateek', age=26}, Users{name='Muskan', age=26}`

2. \*\*map\*\*: Transforms elements.

- `lst.stream().map(Users::getName).forEach(System.out::println);`

- Output: `Prateek, Muskan, Ankit, Siraj`

3. \*\*flatMap\*\*: Flattens nested streams.

- `lst.stream().flatMap(u -> List.of(u.getName(), u.getAge()).stream()).forEach(System.out::println);`

- Output: `Prateek, 26, Muskan, 26, Ankit, 28, Siraj, 28`

4. \*\*distinct\*\*: Removes duplicates.

- `lst.stream().map(Users::getAge).distinct().forEach(System.out::println);`

- Output: `26, 28`

5. \*\*sorted\*\*: Sorts elements.

- `lst.stream().sorted((u1, u2) -> u1.getAge() - u2.getAge()).forEach(System.out::println);`

- Output: `Users{name='Prateek', age=26}, ...`

6. \*\*peek\*\*: Performs action without change.

- `lst.stream().peek(System.out::println).count();`

- Output: Prints all users, returns `4`

7. \*\*limit\*\*: Limits to n elements.

- `lst.stream().limit(2).forEach(System.out::println);`

- Output: `Users{name='Prateek', age=26}, Users{name='Muskan', age=26}`

8. \*\*skip\*\*: Skips n elements.

- `lst.stream().skip(2).forEach(System.out::println);`

- Output: `Users{name='Ankit', age=28}, Users{name='Siraj', age=28}`

9. \*\*takeWhile\*\*: Takes until condition fails.

- `lst.stream().takeWhile(u -> u.getAge() == 26).forEach(System.out::println);`

- Output: `Users{name='Prateek', age=26}, Users{name='Muskan', age=26}`

10. \*\*dropWhile\*\*: Drops until condition fails.

- `lst.stream().dropWhile(u -> u.getAge() == 26).forEach(System.out::println);`

- Output: `Users{name='Ankit', age=28}, Users{name='Siraj', age=28}`

Terminal Operations

1. \*\*forEach\*\*: Applies action to each.

- `lst.stream().forEach(System.out::println);`

- Output: Prints all users

2. \*\*toArray\*\*: Collects to array.

- `Object[] arr = lst.stream().toArray(); System.out.println(arr.length);`

- Output: `4`

3. \*\*reduce\*\*: Reduces to single value.

- `int sum = lst.stream().map(Users::getAge).reduce(0, Integer::sum); System.out.println(sum);`

- Output: `108`

4. \*\*collect\*\*: Collects to collection.

- `List<String> names = lst.stream().map(Users::getName).collect(Collectors.toList()); System.out.println(names);`

- Output: `[Prateek, Muskan, Ankit, Siraj]`

5. \*\*min\*\*: Finds minimum.

- `lst.stream().min((u1, u2) -> u1.getAge() - u2.getAge()).ifPresent(System.out::println);`

- Output: `Users{name='Prateek', age=26}`

6. \*\*max\*\*: Finds maximum.

- `lst.stream().max((u1, u2) -> u1.getAge() - u2.getAge()).ifPresent(System.out::println);`

- Output: `Users{name='Ankit', age=28}`

7. \*\*count\*\*: Counts elements.

- `long count = lst.stream().count(); System.out.println(count);`

- Output: `4`

8. \*\*anyMatch\*\*: Checks if any match.

- `boolean match = lst.stream().anyMatch(u -> u.getAge() == 28); System.out.println(match);`

- Output: `true`

9. \*\*allMatch\*\*: Checks if all match.

- `boolean all = lst.stream().allMatch(u -> u.getAge() > 25); System.out.println(all);`

- Output: `true`

10. \*\*noneMatch\*\*: Checks if none match.

- `boolean none = lst.stream().noneMatch(u -> u.getAge() < 25); System.out.println(none);`

- Output: `true`

11. \*\*findFirst\*\*: Returns first element.

- `lst.stream().findFirst().ifPresent(System.out::println);`

- Output: `Users{name='Prateek', age=26}`

12. \*\*findAny\*\*: Returns any element.

- `lst.stream().findAny().ifPresent(System.out::println);`

- Output: `Users{name='Prateek', age=26}`

**Notes**

- Chain intermediate operations before a terminal operation.

- Use `parallelStream()` for parallel processing where applicable.

Here’s a **cheat sheet** of the main Java Stream operations grouped by type:

**1️⃣ Creating Streams**

* Stream.of(T...)
* Arrays.stream(array)
* list.stream()
* Stream.empty()
* Stream.generate()
* Stream.iterate()

**2️⃣ Intermediate Operations *(return a new Stream)***

**Filtering & Slicing**

* filter(Predicate) → keep elements matching a condition
* distinct() → remove duplicates
* limit(n) → keep first n elements
* skip(n) → skip first n elements

**Mapping & Transforming**

* map(Function) → transform elements
* mapToInt, mapToLong, mapToDouble → primitive specializations
* flatMap(Function) → flatten nested streams

**Sorting**

* sorted() → natural order
* sorted(Comparator) → custom order

**Peeking**

* peek(Consumer) → debug/log elements in the pipeline

**3️⃣ Terminal Operations *(produce a result and close the stream)***

**Matching** *(short-circuiting)*

* anyMatch(Predicate) → true if at least one matches
* allMatch(Predicate) → true if all match
* noneMatch(Predicate) → true if none match

**Finding** *(short-circuiting)*

* findFirst() → Optional first element
* findAny() → Optional element (useful in parallel streams)

**Reduction**

* forEach(Consumer) / forEachOrdered() → iterate elements
* toArray() → collect into array
* reduce() → combine into a single value
* collect(Collectors...) → collect into list, set, map, etc.

**Statistics**

* count() → number of elements
* min(Comparator) → smallest element
* max(Comparator) → largest element

**4️⃣ Short-Circuiting Operations**

These can end the stream early:

* limit(n)
* anyMatch, allMatch, noneMatch
* findFirst, findAny

**can we join 2 streams into 1 ?**

Yes ✅ — you can join two streams in Java using **Stream.concat()** or by combining them into a collection first.

**Example with Stream.concat():**

Stream<String> s1 = Stream.of("A", "B");

Stream<String> s2 = Stream.of("C", "D");

Stream<String> combined = Stream.concat(s1, s2);

combined.forEach(System.out::println);

// Output: A B C D

**Using Stream.of() + flatMap():**

Stream<String> merged = Stream.of(s1, s2)

.flatMap(s -> s);

⚠ **Note:** A stream can be consumed only once — you can’t reuse s1 or s2 after they’ve been joined.

**What is “groupingBy” in streams ?**

groupingBy in Java Streams is a **collector** that groups elements of a stream into a Map based on a **classification function**.

It comes from Collectors.groupingBy() in java.util.stream.Collectors.

**Basic example – group strings by length:**

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

Map<Integer, List<String>> grouped =

words.stream()

.collect(Collectors.groupingBy(String::length));

System.out.println(grouped);

// {3=[dog], 5=[apple], 6=[banana, cherry]}

* **Key** → length of the string
* **Value** → list of words with that length

**Advanced example – group and count:**

Map<Integer, Long> countByLength =

words.stream()

.collect(Collectors.groupingBy(

String::length,

Collectors.counting()

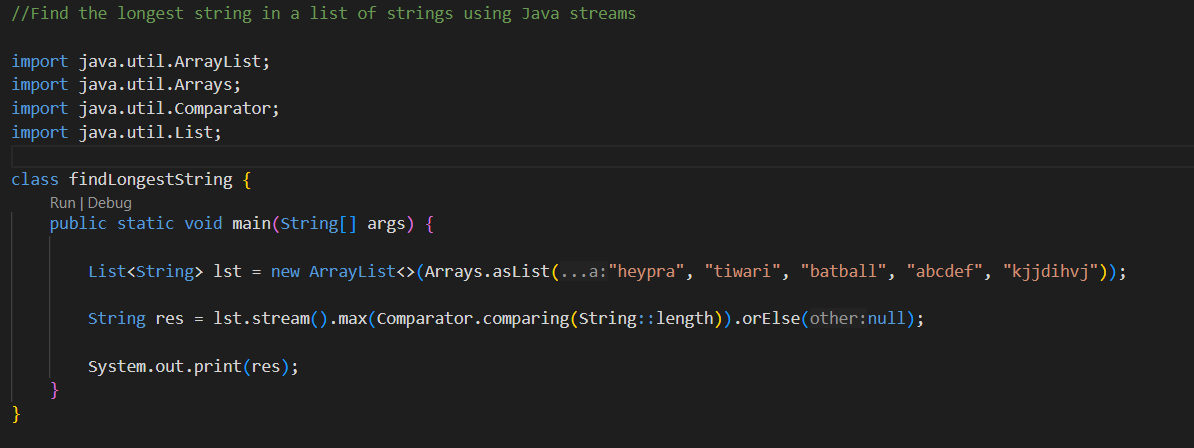
));

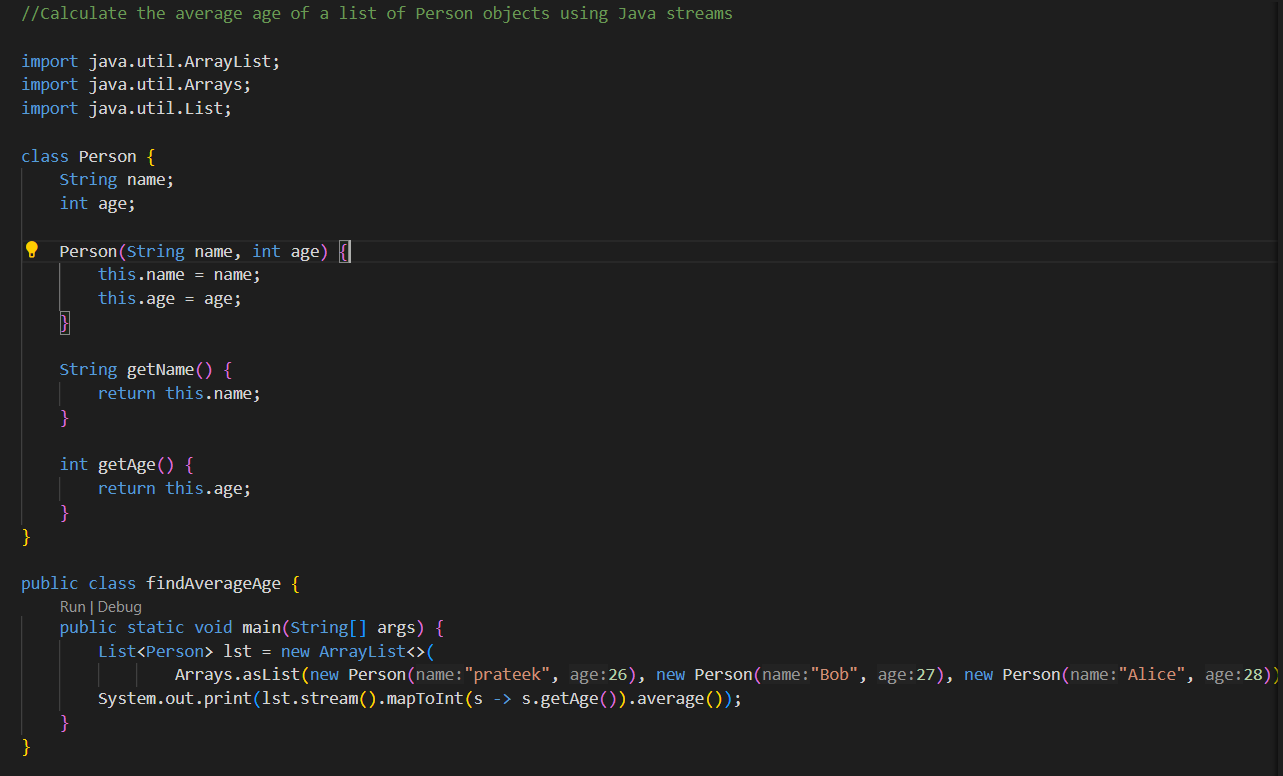
System.out.println(countByLength);

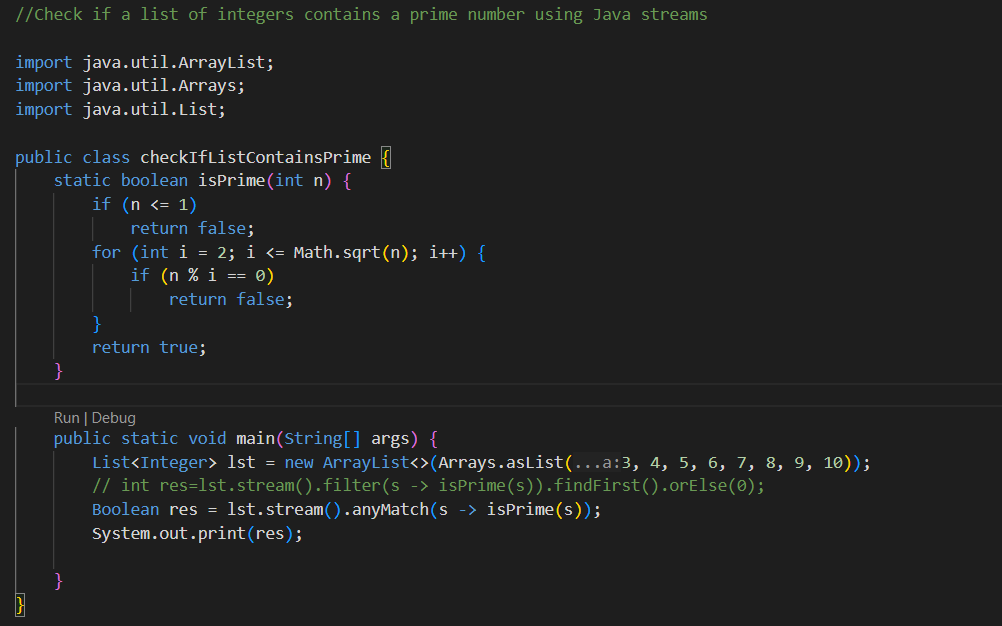
// {3=1, 5=1, 6=2}

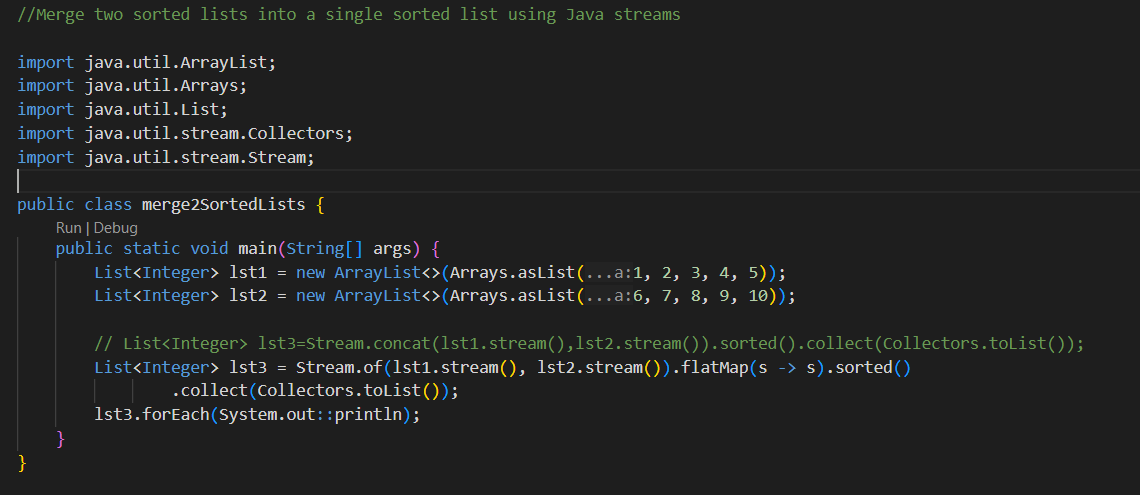
**Key points:**

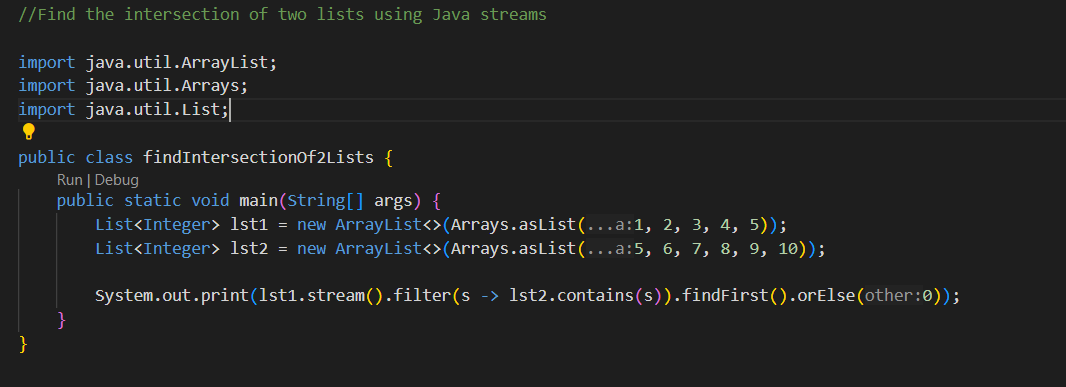
* First argument: classifier function (how to group)
* Second (optional) argument: downstream collector (e.g., counting(), mapping())

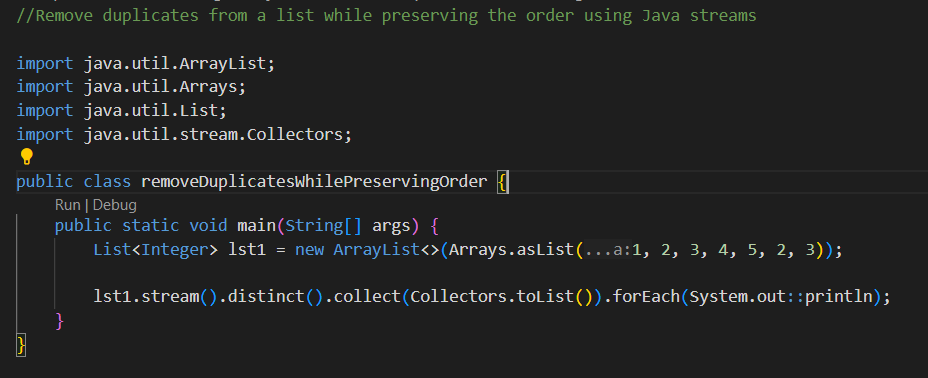












Given a list of transactions, find the sum of transaction amounts for each day using Java streams

