Java Collections, Streams, Comparable & Comparator - Comprehensive Note

Java Collections Framework Overview

- Collection: Root interface for List, Set, and Queue.
- Map: Separate hierarchy, key-value pair storage.

Main Interfaces:

- List: Ordered, allows duplicates. Implementations: ArrayList, LinkedList.
- Set: Unordered, no duplicates. Implementations: HashSet, LinkedHashSet, TreeSet.
- Queue: Ordered with specific insertion/removal rules. Example: PriorityQueue, LinkedList.
- Map: Key-value pairs. Implementations: HashMap, TreeMap, LinkedHashMap.

Stream API (Java 8+)

- Used to process collections in a functional style (pipeline).
- Does not modify original collection.

Stream Operations:

1. Intermediate Operations:
- .filter(predicate)
- .map(function)
- .sorted() or .sorted(Comparator)
- .distinct()

- .limit(n) / .skip(n)

2. Terminal Operations:

- .collect(Collectors.toList())
- .forEach(consumer)
- .reduce(identity, accumulator)
- .count()
- .anyMatch(predicate) / .allMatch(predicate)
- .findFirst() / .findAny()

Specialized Streams:

- IntStream, LongStream, DoubleStream for primitives.
- Use .mapToInt(), .mapToDouble() etc.

Example:

```
List<Integer> numbers = List.of(1, 2, 3, 4);
int sum = numbers.stream().reduce(0, Integer::sum);
```

Comparable Interface

- Used for natural ordering.
- You implement this in the class you want to sort.

```
class Student implements Comparable<Student> {
    int rollNum;
    public int compareTo(Student other) {
        return Integer.compare(this.rollNum, other.rollNum);
    }
}
```

Use:

```
Collections.sort(studentList); // uses compareTo()
```

Comparator Interface

- Used for custom sorting.
- Separate class or lambda function.

```
Comparator<Student> byName = (s1, s2) -> s1.name.compareTo(s2.name);
```

Chaining Comparators:

```
list.stream()
    .sorted(Comparator.comparing(Student::getName)
                      .thenComparing(Student::getRollNum))
    .toList();
```

Difference Between Comparable and Comparator

Feature	Comparable	Comparator
Belongs to	java.lang.Comparable	java.util.Comparator
Method name	compareTo()	compare()
Sorting logic	Defined inside class	Defined outside (can reuse)
Used for	Natural/default sorting	Custom or multiple sort options

Functional? | No | Yes (can use lambdas)

Reduce Examples

1. Sum of elements:

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

2. Average using reduce:

```
double avg = list.stream().mapToDouble(i -> i).reduce(0, Double::sum) / list.size();
```

3. Longest String:

```
String longest = strings.stream().reduce("", (s1, s2) -> s1.length() > s2.length() ? s1 : s2);
```

4. Merging strings:

```
String merged = list.stream().reduce("", (s1, s2) -> s1 + "," + s2);
```

5. Custom Object Reduce (total age):

```
int totalAge = people.stream().map(p -> p.age).reduce(0, Integer::sum);
```

Printing with Format

Print double with 2 decimal places:

```
System.out.printf("%.2f\n", value);
```

Tips:

- Use Comparable for default sort when class is used in sorted collections.

- Use Comparator when:

  - You need to sort in multiple ways.
  - You can't modify the class.
- reduce() is powerful but use collect() for mutable reduction.
- Prefer method references where possible (Integer::sum, Student::getName).
- Don't forget .orElse(0) or .orElseThrow() when using optional values.

"Practice writing small examples often. Even a month later, muscle memory and clarity will stick."