13.5 Comparator and Comparable

The Java Collections Framework provides several ways to sort collections, making it easier for developers to manage data. Sorting can be performed using built-in methods or by implementing custom sorting logic. Let's explore how sorting is achieved using the Comparator and Comparable interfaces in Java.

👉1. Sorting in Collections

The Collections API simplifies many tasks for developers, including sorting. It provides utility methods for sorting, making the process straightforward and efficient. Let’s start with a simple example of sorting a list of integers:

```
public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = new ArrayList<>();
        // Since Java 7, specifying the generic type for the
implementing class is optional
        nums.add(4);
        nums.add(3);
        nums.add(7);
        nums.add(9);

        System.out.println(nums); // Output: [4, 3, 7, 9]
        Collections.sort(nums);   // Sorting using built-in
Collections class
        System.out.println(nums); // Output: [3, 4, 7, 9]
    }
}
```

Explanation:

The Collections.sort() method sorts the list in ascending order according to the natural ordering of the elements.

👉 2. Collections Class in Java

The Collections class in Java, part of the java.util package, is a utility class that operates on or returns collections. It provides various static methods that can be used to manipulate collections, such as sorting and searching.

Common Method for Sorting:

- sort(List<T> list):

Sorts the specified list into ascending order, according to the natural ordering of its elements.

👉 3. Custom Sorting Using Comparator

The Comparator interface is used to define a custom sorting order. It is a functional interface, meaning it has a single abstract method compare(), which is implemented to provide sorting logic.

Example of Custom Sorting:

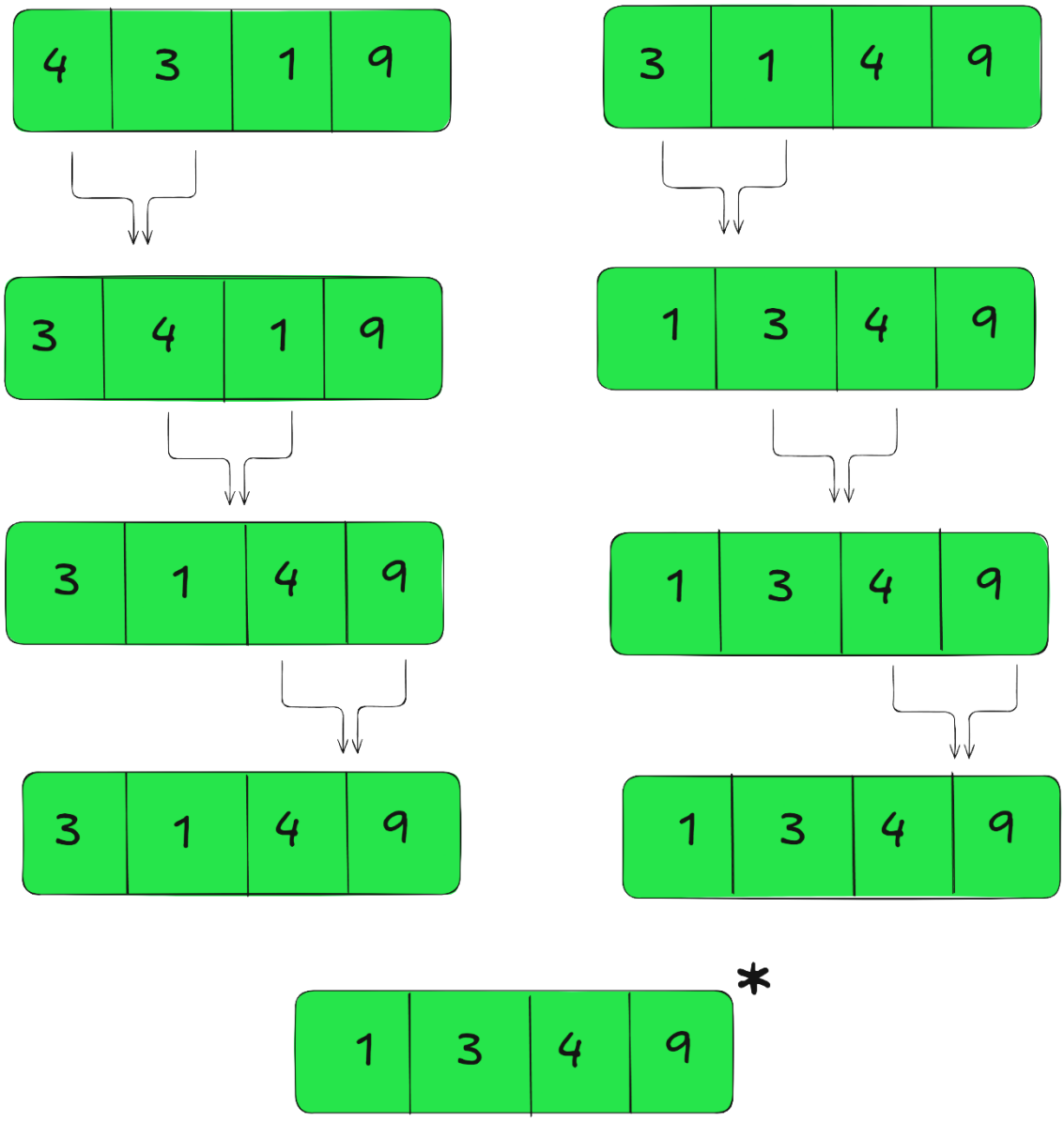
```
public class Demo {
    public static void main(String[] args) {
        // Creating a custom Comparator to sort numbers based
on the last digit
        Comparator<Integer> com = new Comparator<Integer>() {
            public int compare(Integer i, Integer j) {
                if (i % 10 > j % 10)
                    return 1;
                else
                    return -1;
            }
        };

        List<Integer> nums = new ArrayList<>();
        nums.add(54);
        nums.add(33);
        nums.add(41);
        nums.add(69);

        Collections.sort(nums, com); // Sorting using custom
Comparator
        System.out.println(nums);    // Output: [41, 33, 54,
69]
    }
}
```

Explanation:

This example sorts the list based on the last digit of each number. The custom comparator sorts the elements according to the remainder of division by 10.



👉 4. Optimizing with Lambda Expressions

Since Comparator is a functional interface, we can use a lambda expression to simplify the code.

Example Using Lambda:

Comparator<Integer> com = (i, j) -> (i % 10 > j % 10) ? 1 : -1;

The lambda **expression (i, j) -> (i % 10 > j % 10) ? 1 : -1** provides a more concise way to implement the custom sorting logic.

👉 5. Sorting a List of Objects

To sort a list of custom objects (e.g., Student objects), we can use the Comparator interface to specify the sorting criteria.

Example:

```
class Student {
    int age;
    String name;

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class Demo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(33, "Navin"));
        students.add(new Student(12, "John"));
        students.add(new Student(45, "Edward"));
        students.add(new Student(35, "Michael"));

        // Sorting based on age
        Comparator<Student> com = (s1, s2) -> (s1.age >
s2.age) ? 1 : -1;
        Collections.sort(students, com);

        for (Student s : students) {
            System.out.println(s);
        }
```

```
    }
}
```

Output:

```
John (12)
Navin (33)
Michael (35)
Edward (45)
```

Explanation:

This example sorts the list of Student objects based on age. The Comparator sorts the students in ascending order of age.

👉 6. Using the Comparable Interface

The Comparable interface is a part of java.lang package and it provides a way to define the natural ordering of objects. It is a functional interface with a compareTo() method that must be implemented by any class whose objects are intended to be sorted.

Example of Implementing Comparable:

```
class Student implements Comparable<Student> {
    int age;
    String name;

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public int compareTo(Student that) {
        if (this.age > that.age)
            return 1;
        else
            return -1;
    }
```

```
    @Override public String toString() {
 return "Student{name='" + name + "', age=" + age + "}"; }
}

public class Demo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(33, "Navin"));
        students.add(new Student(12, "John"));
        students.add(new Student(45, "Edward"));
        students.add(new Student(35, "Michael"));

        Collections.sort(students); // Sorting using
Comparable

        for (Student s : students) {
            System.out.println(s);
        }
    }
}
```

Output:

```
Student{name='John', age=12}
Student{name='Navin', age=33}
Student{name='Michael', age=35}
Student{name='Edward', age=45}
```

Explanation:

In this example, the Student class implements Comparable, allowing objects of Student to be sorted directly using Collections.sort().

👉🏻 7. Key Differences Between Comparator and Comparable

- **Comparator:**
  - Used for custom sorting logic.
  - Can sort objects in multiple ways (different comparators for different criteria).
  - Does not require modification of the class whose objects are being sorted.
- **Comparable:**

- Defines the natural ordering of a class.
- Used to compare objects of the same class.
- Requires implementing the compareTo() method in the class.

👉 Conclusion

Both Comparator and Comparable are essential for sorting objects in Java. Comparator is useful for custom sorting, while Comparable provides a natural sorting order for classes. Understanding when to use each interface allows developers to manage sorting tasks efficiently.