1Z0-808 Exam Topic Reviewer

TopicId: 1027
Topic: Sorting and Searching Collections (Comparable, Comparator)

August 5, 2025

Sorting and Searching: Bringing Order to Collections

Now that we have our collections, how do we sort them? If you have a list of `Strings` or `Integers`, `Collections.sort()` works out of the box. But what about a list of your own custom objects, like `Employee`? Java needs you to define the comparison logic. This is done using one of two interfaces: `Comparable` or `Comparator`. Mastering these is essential.

0.1 `Comparable`: The Natural Order

Use the `Comparable` interface to define the **single, natural ordering** for an object. This is implemented *inside* the class itself.

- A class must `implements Comparable<Type>`.
- It must implement one method: `public int compareTo(Type other)`.
- **The Contract:**
  - returns < 0 if `this` object comes before `other`.
  - returns 0 if `this` object is equal to `other`.
  - returns > 0 if `this` object comes after `other`.

```
public class Product implements Comparable<Product> {
    private int id;
    private String name;
    // constructor, getters...
    @Override
    public int compareTo(Product other) {
        // Natural order is by id
        return this.id - other.id;
    }
}
// Usage:
List<Product> products = ...;
Collections.sort(products); // Sorts using the compareTo method.
```

0.2 `Comparator`: Custom and Multiple Orders

Use the `Comparator` interface when you need **multiple different sort orders**, or when you cannot modify the source code of the class you want to sort. This logic is defined in a *separate class* or, more commonly in Java 8, a **lambda expression**.

- A class `implements Comparator<Type>`.
- It must implement one method: `public int compare(Type o1, Type o2)`.
- The return value contract is the same as `compareTo`.

```
// Using a separate class (pre-Java 8 style)
```

```
public class SortProductByName implements Comparator<Product> {
    @Override
    public int compare(Product p1, Product p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
// Usage:
Collections.sort(products, new SortProductByName());

// Using a lambda (Java 8 style - PREFERRED)
Comparator<Product> byName = (p1, p2) -> p1.getName().compareTo(p2.getName());
Collections.sort(products, byName);
// Or even more concisely:
products.sort((p1, p2) -> p1.getName().compareTo(p2.getName()));
```

Comparable vs. Comparator: The Final Verdict

Feature	Comparable	Comparator
Package	`java.lang`	`java.util`
Method	`compareTo(T obj)`	`compare(T o1, T o2)`
Implementation	Inside the domain class	In a separate class or lambda
Purpose	Defines one natural order	Defines multiple custom orders

0.3 Searching with `Collections.binarySearch()`

Binary search is a fast way to find an element in a list, but it has one absolute requirement: **the list must be sorted first!**

- If the list is not sorted, the result is undefined. Don't trust it.
- If the element is found, it returns its index.
- If the element is not found, it returns `(-(insertion point) - 1)`. The insertion point is the index where the element would be inserted to keep the list sorted. The exam loves to test this.

```
List<Integer> list = Arrays.asList(2, 4, 6, 8); // Sorted!
Collections.binarySearch(list, 6); // returns 2 (index of 6)
Collections.binarySearch(list, 5); // returns -3. Insertion point is 2. -(2)-1 =
```

Key Takeaways for the 1Z0-808 Exam

- Use `Comparable` for a single, natural sort order defined inside your class.
- Use `Comparator` for multiple, custom sort orders defined outside your class. Embrace lambdas for this in Java 8!
- For `Collections.binarySearch()` to work correctly, the list **must be sorted**.
- Know how to interpret the negative return value of `binarySearch()`.