

Collections in Java

1. Wrapper Classes in Java

❑ What are Wrapper Classes?

In Java, **Wrapper Classes** are used to **wrap primitive data types** into objects. Java is an **object-oriented language**, and sometimes we need to **treat primitives like objects**—this is where wrapper classes come in.

❑ Primitive vs Wrapper

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

❑ Example:

```
int a = 10;           // primitive
Integer b = Integer.valueOf(a); // manual boxing
int c = b.intValue(); // manual unboxing
```

❑ Java 1.5+ supports Autoboxing and Unboxing:

```
Integer x = 100; // autoboxing (primitive to object)
int y = x;       // unboxing (object to primitive)
```

2. Why Wrapper Classes?

□ Reasons Why We Need Wrapper Classes:

1. To Work with Java Collections (like ArrayList, HashMap)

Java **Collections only work with objects** (not primitives). So to store int, float, etc., we need to use Integer, Float, etc.

```
ArrayList<Integer> list = new ArrayList<>();  
list.add(10); // int gets autoboxed to Integer
```

2. Utility Methods

Wrapper classes provide **many utility methods** for parsing and converting values.

```
int x = Integer.parseInt("123");  
double d = Double.parseDouble("3.14");
```

3. Null Support

Wrapper classes **can store null**, but primitives cannot.

```
Integer x = null; // valid  
int y = null;    // Compile-time error
```

4. Type Conversion

Wrapper classes help with **conversion between types**, like string to number and vice versa.

```
String str = "45";  
int num = Integer.valueOf(str); // String to int  
String s = Integer.toString(num); // int to String
```

5. Use in Generics

Java **Generics** do not support primitives. So if you're writing a generic class or method, you need wrapper classes.

3. Autoboxing & Unboxing in Java

□ What is Autoboxing?

Autoboxing is the **automatic conversion** of a **primitive type** to its **corresponding wrapper class** object.

- Introduced in Java 1.5
- Makes code cleaner and more readable
- Useful when working with Collections (which only accept objects)

□ Example:

```
int a = 10;  
Integer obj = a; // Autoboxing: int → Integer
```

Behind the scenes:

```
Integer obj = Integer.valueOf(a);
```

□ What is Unboxing?

Unboxing is the **reverse process**: converting a **wrapper class object** back to a **primitive**.

```
Integer obj = new Integer(20);  
int b = obj; // Unboxing: Integer → int
```

Behind the scenes:

```
int b = obj.intValue();
```

Real-life Use Case in Collections:

```
ArrayList<Integer> list = new ArrayList<>();
```

```
// Autoboxing - primitive to object  
list.add(5);  
list.add(10);
```

```
// Unboxing - object to primitive
int sum = list.get(0) + list.get(1);
System.out.println("Sum = " + sum);
```

□ Things to Remember:

Point	Explanation
Supported from Java 1.5	Before that, manual boxing/unboxing was needed
NullPointerException	Unboxing a null wrapper object will throw an exception

```
Integer obj = null;
int a = obj; // Runtime Error: NullPointerException
```

4.What is Collection in Java?

□ Definition:

The **Collection** in Java is a **framework** that provides **architecture to store and manipulate a group of objects**. It is a part of the **Java Collections Framework** and is **located in the java.util package**.

- It is a **root interface** in the Collection hierarchy.
- It represents a group of **individual objects** (not key-value pairs).

Collection Interface:

```
public interface Collection<E> extends Iterable<E> {
    // Core method definitions like add(), remove(), size(), etc.
}
```

□ Collection Hierarchy:

Each of these subinterfaces (List, Set, Queue) have multiple implementations:

- List → ArrayList, LinkedList, Vector, Stack
- Set → HashSet, LinkedHashSet, TreeSet
- Queue → PriorityQueue, LinkedList (as Queue)

❑ **Common Methods in Collection Interface:**

Method	Description
add(E e)	Adds element to the collection
remove(Object o)	Removes element
size()	Returns number of elements
clear()	Empties the collection
contains(E e)	Checks if element is present
isEmpty()	Checks if collection is empty
iterator()	Returns an iterator to traverse

5. Why Collection?

Advantages of using Collection Framework:

1. Dynamic Memory Management

Collections like ArrayList, HashSet **grow/shrink automatically**. No need to manually resize arrays.

```
ArrayList<String> names = new ArrayList<>();  
names.add("Alice"); // dynamically resizes
```

2. Built-in Data Structures

Provides ready-made **data structures** like:

- List
- Stack

- Queue
- Set
- Map (part of the framework but not under Collection interface)

3. Reduces Code Complexity

Simplifies tasks like sorting, searching, insertion, deletion using **ready methods**.

4. Interoperability and Flexibility

Supports **generics**, **iterators**, and **algorithms**, making the framework powerful and reusable.

```
Collections.sort(list);
```

5. Unified Architecture

All collections share a **common set of interfaces and methods**, ensuring **consistency** and **polymorphism**.

Collection vs Collections in Java

One of the most common points of confusion for beginners is the difference between **Collection** and **Collections** in Java.

❑ Collection (Interface)

Feature	Description
Type	Interface
Package	java.util.Collection
Purpose	Root of the Collection Hierarchy (List, Set, Queue)
Represents	A group of individual elements

Feature	Description
Extends	Iterable<E> interface
Key Subinterfaces	List, Set, Queue
Implementations	ArrayList, HashSet, LinkedList, etc.

❑ **Example:**

```
Collection<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Mango");
System.out.println(fruits);
```

❑ **Collections (Utility Class)**

Feature	Description
Type	Final class
Package	java.util.Collections
Purpose	Provides static utility methods for operations on collections
Common Methods	sort(), reverse(), shuffle(), min(), max(), frequency()
Cannot be instantiated	Only contains static methods

❑ **Example:**

```
List<Integer> nums = new ArrayList<>();
nums.add(10);
nums.add(5);
nums.add(20);
```

```
Collections.sort(nums); // Sorts the list in ascending order
System.out.println(nums); // Output: [5, 10, 20]
```

❑ Key Differences

Feature	Collection	Collections
Type	Interface	Class (final)
Role	Represents a group of objects	Provides utility methods for collections
Located in	java.util.Collection	java.util.Collections
Can be instantiated?	No, it's an interface	No, it's a utility class with static methods
Examples	List, Set, Queue	Collections.sort(), Collections.reverse()

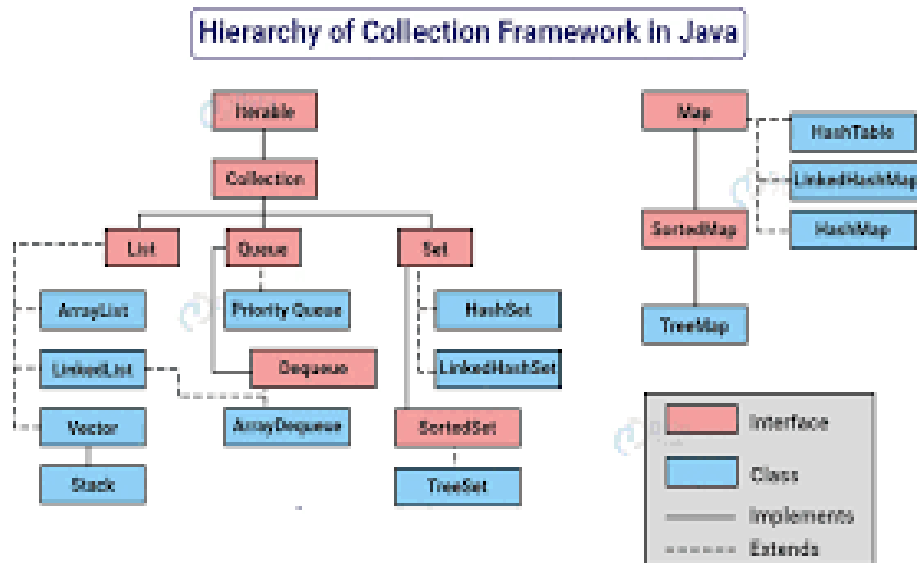
Collection Hierarchy in Java

❑ What is Collection Hierarchy?

The **Java Collection Framework** is a unified architecture for representing and manipulating groups of objects. It contains **interfaces, abstract classes, and concrete implementations** organized in a **hierarchical structure**.

- ❑ Think of it like a **tree structure** starting from Collection interface at the root.
-

Java Collection Framework Hierarchy



1. Iterable<E> (Top-most interface)

- Introduced in Java 5
- Enables **for-each loop (for-each)** support
- Method: iterator()

2. Collection<E> (Base Interface)

- Root interface for all collection classes
- Defines basic operations: add(), remove(), size(), clear(), etc.

Subinterfaces of Collection:

List<E> – Ordered, allows duplicates

Implementation	Description
ArrayList	Resizable array, fast access, slow insert/delete
LinkedList	Doubly linked list, good for insert/delete

Implementation	Description
Vector	Synchronized version of ArrayList (legacy)
Stack	LIFO data structure, extends Vector

Set<E> – Unordered, unique elements

Implementation	Description
HashSet	No order, fastest access
LinkedHashSet	Maintains insertion order
TreeSet	Sorted set, uses TreeMap, implements NavigableSet

Queue<E> – FIFO structure

Implementation	Description
LinkedList	Also implements Queue
PriorityQueue	Elements ordered by natural/comparator
ArrayDeque	Fast queue/deque operations

Other Important Interfaces (not under Collection):

Map<K, V> – Key-Value Pairs

Implementation	Description
HashMap	Fast, no order
LinkedHashMap	Maintains insertion order
TreeMap	Sorted map based on keys
Hashtable	Legacy, synchronized
Properties	Used for config files (key-value as Strings)

8.List Interface in Java

□ What is List?

List is a **child interface of Collection**. It is an **ordered collection** that:

- Allows **duplicates**
- Maintains **insertion order**
- Supports **index-based access**

Interface: `java.util.List<E>`

8.1 ArrayList

□ Overview:

- Resizable array implementation
- Fast **random access**
- Slow insert/delete in the middle (shifting needed)

Class: `java.util.ArrayList<E>`

□ Commonly Used Methods:

```
import java.util.*;
```

```
public class ArrayListDemo {  
    public static void main(String[] args) {  
        ArrayList<String> list = new ArrayList<>();  
  
        list.add("Java");    // add element  
        list.add("Python");  
        list.add(1, "C++");  // add at specific index  
  
        list.remove("Python"); // remove by value  
        list.remove(1);        // remove by index  
  
        list.set(0, "Kotlin"); // replace element at index  
        String val = list.get(0); // get element  
    }  
}
```

```
        System.out.println(list); // print list
    }
}
```

❑ Important Methods:

Method	Description
add(E e)	Add element at end
add(int i, E e)	Add element at specific index
remove(E e)	Remove by value
remove(int i)	Remove by index
get(int i)	Access element by index
set(int i, E e)	Update value at index
contains(E e)	Check if list contains element
size()	Get number of elements
clear()	Remove all elements

8.2 LinkedList

❑ Overview:

- Doubly linked list
- Better for **insertions/deletions** at start or middle
- Implements both List and Deque interfaces

Class: java.util.LinkedList<E>

❑ Example & Methods:

```
import java.util.*;

public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<String> ll = new LinkedList<>();

        ll.add("Dog");
```

```
    ll.addFirst("Cat");    // add at beginning
    ll.addLast("Elephant"); // add at end
    ll.removeFirst();      // remove from beginning
    ll.removeLast();       // remove from end

    System.out.println(ll);
}
}
```

❑ Important Methods:

Method	Description
add(E e)	Add at end
addFirst(E e)	Add at start
addLast(E e)	Add at end
removeFirst()	Remove from start
removeLast()	Remove from end
getFirst()	Get first element
getLast()	Get last element
offer(), poll()	Queue-like operations

8.3 Vector

❑ Overview:

- Legacy class (thread-safe)
- Similar to ArrayList but **synchronized**
- Slower in single-threaded apps

Class: java.util.Vector<E>

❑ Example & Methods:

```
import java.util.*;
```

```
public class VectorDemo {
    public static void main(String[] args) {
```

```
Vector<Integer> vec = new Vector<>();

vec.add(10);
vec.add(20);
vec.addElement(30); // legacy add
vec.remove(1);      // remove by index

System.out.println(vec);
}
```

☐ Important Methods:

Method	Description
add(E e)	Add element
addElement(E e)	Legacy method to add
remove(int index)	Remove at index
elementAt(int i)	Get element at index
firstElement()	Get first element
lastElement()	Get last element
capacity()	Get current capacity

8.4 Stack

☐ Overview:

- **LIFO** data structure (Last In, First Out)
- Extends **Vector**
- Used for **backtracking**, **undo**, etc.

Class: java.util.Stack<E>

☐ Example & Methods:

```
import java.util.*;

public class StackDemo {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();

        stack.push("A");    // add
        stack.push("B");
        stack.pop();        // remove top
        String top = stack.peek(); // view top

        System.out.println("Top: " + top);
        System.out.println(stack);
    }
}
```

❑ Important Methods:

Method	Description
push(E e)	Add element to top
pop()	Remove and return top
peek()	View top without removing
empty()	Check if stack is empty
search(E e)	Return position from top

Set Interface in Java

❑ What is Set?

The Set interface in Java is a subtype of the Collection interface. It is used to store a **collection of unique elements**, meaning **duplicates are not allowed**.

Interface: java.util.Set<E>

❑ Key Characteristics of Set:

- No duplicate elements
 - Can be unordered or ordered depending on implementation
 - Allows null (depends on implementation)
-

9.1 HashSet

❑ Overview:

- Stores elements in a **hash table**
- **Unordered**, no guarantee of insertion order
- **Allows null**

Class: java.util.HashSet<E>

❑ Example:

```
import java.util.*;

public class HashSetDemo {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();

        set.add("Apple");
        set.add("Banana");
        set.add("Apple"); // Duplicate, ignored
        set.remove("Banana");

        System.out.println(set);
    }
}
```

❑ Important Methods:

Method	Description
add(E e)	Adds element if not already present
remove(E e)	Removes specified element

Method	Description
contains(E e)	Checks if element exists
isEmpty()	Checks if set is empty
size()	Returns number of elements
clear()	Removes all elements
iterator()	Returns iterator for traversal

9.2 LinkedHashSet

□ Overview:

- Stores elements in a **LinkedHashMap**
- Maintains **insertion order**
- **Allows null**

Class: java.util.LinkedHashSet<E>

□ Example:

```
import java.util.*;

public class LinkedHashSetDemo {
    public static void main(String[] args) {
        LinkedHashSet<String> lhs = new LinkedHashSet<>();

        lhs.add("Java");
        lhs.add("Python");
        lhs.add("C++");
        lhs.remove("Python");

        System.out.println(lhs);
    }
}
```

□ Important Methods:

Method	Description
--------	-------------

Method	Description
add(E e)	Adds element, maintains order
remove(E e)	Removes specified element
contains(E e)	Checks if element exists
size()	Returns size
isEmpty()	Checks if set is empty
clear()	Removes all elements
iterator()	Traverse elements in insertion order

9.3 TreeSet

□ Overview:

- Stores elements in a **Red-Black Tree** (self-balancing BST)
- Elements are **sorted in natural order** (or by custom comparator)
- **Does NOT allow null**

Class: java.util.TreeSet<E>

□ Example:

```
import java.util.*;

public class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet<Integer> ts = new TreeSet<>();

        ts.add(30);
        ts.add(10);
        ts.add(20);
        // ts.add(null); // NullPointerException

        System.out.println(ts); // Output: [10, 20, 30]
    }
}
```

❑ Important Methods:

Method	Description
add(E e)	Adds element in sorted order
remove(E e)	Removes specified element
first()	Returns the first (lowest) element
last()	Returns the last (highest) element
higher(E e)	Element strictly greater than given
lower(E e)	Element strictly less than given
headSet(E e)	All elements less than e
tailSet(E e)	All elements greater than or equal to e
subSet(e1, e2)	Elements between e1 and e2

Enumeration, Iterator, and ListIterator in Java

These are **three different cursor types** (also called iterators) used to **traverse Java Collections** like List, Set, and Vector.

They are found in the **java.util package** and are key tools for collection navigation.

1.Enumeration

❑ Overview:

- **Legacy cursor**, used only with **older classes like Vector, Stack, Hashtable**
- Only supports **read-only traversal (forward only)**

Interface: `java.util Enumeration<E>`

❑ Example:

```
import java.util.*;
```

```
public class EnumerationDemo {  
    public static void main(String[] args) {  
        Vector<String> vec = new Vector<>();  
        vec.add("Java");  
        vec.add("Python");  
  
        Enumeration<String> e = vec.elements();  
  
        while (e.hasMoreElements()) {  
            System.out.println(e.nextElement());  
        }  
    }  
}
```

□ Key Methods:

Method	Description
hasMoreElements()	Returns true if more elements
nextElement()	Returns next element

□ Limitations:

- **Forward-only**
 - **Read-only**
 - **Works with legacy classes only**
-

2.Iterator

□ Overview:

- Introduced to **replace Enumeration**
- Works with **all Collection classes**
- Can **traverse and remove** elements
- Still **forward-only**

Interface: java.util.Iterator<E>

□ Example:

```
import java.util.*;

public class IteratorDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");

        Iterator<String> it = list.iterator();

        while (it.hasNext()) {
            String val = it.next();
            if (val.equals("B"))
                it.remove(); // Remove using iterator
            else
                System.out.println(val);
        }

        System.out.println(list); // [A]
    }
}
```

□ **Key Methods:**

Method	Description
hasNext()	Checks if next element exists
next()	Returns next element
remove()	Removes current element (optional)

3.ListIterator

□ **Overview:**

- **Extended version** of Iterator
- Used **only with List implementations**
- Allows **bidirectional traversal**
- Allows **modification** (add, remove, set)

Interface: java.util.ListIterator<E>
Available only via list.listIterator()

❑ **Example:**

```
import java.util.*;

public class ListIteratorDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("X", "Y", "Z"));
        ListIterator<String> li = list.listIterator();

        while (li.hasNext()) {
            String val = li.next();
            if (val.equals("Y")) {
                li.set("Modified"); // Replace Y with Modified
                li.add("New");       // Add after Modified
            }
        }

        System.out.println(list); // Output: [X, Modified, New, Z]
    }
}
```

❑ **Key Methods:**

Method	Description
hasNext()	Checks if next element exists
next()	Moves forward and returns element
hasPrevious()	Checks if previous element exists
previous()	Moves backward and returns element
add(E e)	Adds element at current position
remove()	Removes last element returned
set(E e)	Replaces last element returned
nextIndex()	Returns index of next element
previousIndex()	Returns index of previous element

❑ When to Use What?

Scenario	Cursor Type
Traversing only legacy collections	Enumeration
Simple read + remove (any collection)	Iterator
Full control: forward, backward, modify	ListIterator

Map Interface in Java

❑ What is Map?

The **Map** interface in Java is part of the **Java Collections Framework**, but **not a subtype of Collection**.

A Map stores **key-value pairs**, where **keys are unique** and each key maps to a **single value**.

Interface: `java.util.Map<K, V>`

❑ Key Characteristics:

- Stores (**Key, Value**) pairs
- **Keys are unique**, but values can be duplicate
- **Not iterable using Collection cursor** (like Iterator)
- You access values using keys

1.Hashtable

❑ Overview:

- **Legacy class**, thread-safe (synchronized)

- Doesn't allow null keys or values
- Slower than HashMap

Class: java.util.Hashtable<K, V>

□ **Example:**

```
import java.util.*;
```

```
public class HashtableDemo {  
    public static void main(String[] args) {  
        Hashtable<Integer, String> ht = new Hashtable<>();  
  
        ht.put(1, "Java");  
        ht.put(2, "Python");  
        // ht.put(null, "NullKey"); // Throws NullPointerException  
        // ht.put(3, null);         // Not allowed  
  
        System.out.println(ht);  
    }  
}
```

□ **Important Methods:**

Method	Description
put(K, V)	Add key-value pair
get(K)	Retrieve value for a key
remove(K)	Remove entry by key
containsKey(K)	Checks if key exists
containsValue(V)	Checks if value exists
keySet()	Returns Set of keys
values()	Returns Collection of values
entrySet()	Returns Set of Map.Entry pairs

2.HashMap

□ Overview:

- **Most commonly used Map implementation**
- Allows **one null key** and **multiple null values**
- **Not synchronized** (faster in single-threaded env)
- Uses **hashing** internally

Class: java.util.HashMap<K, V>

□ Example:

```
import java.util.*;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();

        map.put("Apple", 50);
        map.put("Banana", 30);
        map.put("Apple", 60); // Overwrites value
        map.put(null, 0);     // Allows one null key

        System.out.println(map);
    }
}
```

□ Important Methods:

Method	Description
put(K, V)	Insert/Update key-value
get(K)	Get value by key
remove(K)	Remove entry
containsKey(K)	Check if key exists
containsValue(V)	Check if value exists
isEmpty()	Returns true if map is empty
size()	Number of entries

Method	Description
keySet()	Returns Set of keys
entrySet()	Returns key-value pairs

3.TreeMap

□ Overview:

- Stores entries in **sorted order of keys (natural or comparator)**
- **Does not allow null key**, but allows null values
- Backed by a **Red-Black Tree**

Class: java.util.TreeMap<K, V>

□ Example:

```
import java.util.*;
```

```
public class TreeMapDemo {  
    public static void main(String[] args) {  
        TreeMap<Integer, String> tm = new TreeMap<>();  
  
        tm.put(3, "Three");  
        tm.put(1, "One");  
        tm.put(2, "Two");  
        // tm.put(null, "NullKey"); // Throws NullPointerException  
  
        System.out.println(tm); // Output: {1=One, 2=Two, 3=Three}  
    }  
}
```

□ Important Methods:

Method	Description
put(K, V)	Insert entry
get(K)	Get value
firstKey()	Returns the smallest key

Method	Description
lastKey()	Returns the largest key
headMap(K)	Keys less than given key
tailMap(K)	Keys greater than/equal to given key
subMap(K1, K2)	Keys in range [K1, K2)

Comparable vs Comparator in Java

In Java, we use **Comparable** and **Comparator** to sort objects of custom classes like Student, Employee, Product, etc.

1. Comparable Interface

□ Definition:

Comparable is an interface used to **define the natural sorting order** of objects in a class itself.

Package: java.lang

Method: public int compareTo(T o)

□ Example:

```
import java.util.*;

class Student implements Comparable<Student> {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

```
// Sort by id (natural order)
public int compareTo(Student s) {
    return this.id - s.id;
}
}

public class ComparableDemo {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Rahul"));
        list.add(new Student(1, "Amit"));
        list.add(new Student(2, "Zoya"));

        Collections.sort(list); // Uses compareTo()

        for (Student s : list)
            System.out.println(s.id + " " + s.name);
    }
}
```

□ **Key Points:**

Feature	Description
Interface	java.lang.Comparable<T>
Method	compareTo(T o)
Sorting logic location	Inside the class itself
Sorting type	Natural order
Used with	Collections.sort(list)

2.Comparator Interface

□ Definition:

Comparator is used to define **multiple/custom sorting orders** outside the class, often via **lambda or anonymous classes**.

Package: java.util

Method: public int compare(T o1, T o2)

□ Example:

```
import java.util.*;

class Student {
    int id;
    String name;

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

public class ComparatorDemo {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Rahul"));
        list.add(new Student(1, "Amit"));
        list.add(new Student(2, "Zoya"));

        // Sort by name (custom order)
        Collections.sort(list, new Comparator<Student>() {
            public int compare(Student a, Student b) {
                return a.name.compareTo(b.name);
            }
        });
    }
}
```

```
        for (Student s : list)
            System.out.println(s.id + " " + s.name);
    }
}
```

❑ **Or using Lambda Expression (Java 8+):**

```
Collections.sort(list, (a, b) -> a.name.compareTo(b.name));
```

❑ **Key Points:**

Feature	Description
Interface	java.util.Comparator<T>
Method	compare(T o1, T o2)
Sorting logic location	Outside the class
Sorting type	Custom order (can be multiple)
Used with	Collections.sort(list, comparator)

Comparable vs Comparator – Summary Table

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo(Object o)	compare(Object o1, Object o2)
Defined In	Inside the class	Outside the class
Used For	Single, natural ordering	Multiple, custom sorting

Feature	Comparable	Comparator
Modifies original?	Yes (class must implement)	No (can be defined separately)
Use With	Collections.sort(list)	Collections.sort(list, comparator)

☐ **Real-World Use Case:**

Scenario	Use
Sort products by price (default)	Comparable
Sort products by name or rating	Comparator