# **Collections continued**

**🔷 What is a Set in Java?**

* A **Set** is a **collection that doesn't allow duplicate elements**.
* It's part of the **Java Collection Framework**.
* Common implementations: HashSet, LinkedHashSet, and TreeSet.

**🔹 1. HashSet**

**📌 Features:**

* **No duplicates allowed**.
* **No guaranteed order** of elements.
* **Backed by a hash table**.
* Fast add, remove, contains operations.

✅ Real-World Example: **Storing unique email addresses**import java.util.HashSet;

public class EmailDirectory {

public static void main(String[] args) {

HashSet<String> emails = new HashSet<>();

// Add email addresses

emails.add("alice@example.com");

emails.add("bob@example.com");

emails.add("charlie@example.com");

emails.add("alice@example.com"); // Duplicate

System.out.println("Email directory:");

for (String email : emails) {

System.out.println(email);

}

// Check if email exists

if (emails.contains("bob@example.com")) {

System.out.println("Bob's email is present.");

}

// Remove an email

emails.remove("charlie@example.com");

System.out.println("Updated email directory: " + emails);

}

}  
**🧠 Common Methods:**

* add(E e)
* remove(Object o)
* contains(Object o)
* isEmpty()
* size()
* clear()

**🔹 2. LinkedHashSet  
📌 Features:**

* **Maintains insertion order.**
* **Still no duplicates.**
* **Backed by a hash table + linked list.**

**✅ Real-World Example: User’s recent search history**

import java.util.LinkedHashSet;

public class SearchHistory {

public static void main(String[] args) {

LinkedHashSet<String> history = new LinkedHashSet<>();

// Add search terms

history.add("Java Collections");

history.add("Spring Boot Tutorial");

history.add("Data Structures");

history.add("Java Collections"); // Duplicate

System.out.println("Search History:");

for (String term : history) {

System.out.println(term);

}

// Remove a term

history.remove("Data Structures");

System.out.println("After deletion: " + history);

}

}  
**Common Methods (same as HashSet):**

* Preserves **insertion order** during iteration.

**🔹 3. TreeSet**

**📌 Features:**

* **Sorted in natural order** (e.g., alphabetical for strings, ascending for numbers).
* **No duplicates**.
* Backed by a **Red-Black Tree** (self-balancing binary search tree).
* Slower than HashSet for insertions and deletions.

**✅ Real-World Example: Maintaining a sorted list of student roll numbers**

import java.util.TreeSet;

public class StudentRolls {

public static void main(String[] args) {

TreeSet<Integer> rollNumbers = new TreeSet<>();

// Add roll numbers

rollNumbers.add(105);

rollNumbers.add(101);

rollNumbers.add(103);

rollNumbers.add(101); // Duplicate

System.out.println("Sorted Roll Numbers: " + rollNumbers);

// First and last roll number

System.out.println("First: " + rollNumbers.first());

System.out.println("Last: " + rollNumbers.last());

// Higher and lower than a specific number

System.out.println("Roll higher than 101: " + rollNumbers.higher(101));

System.out.println("Roll lower than 105: " + rollNumbers.lower(105));

}

}  
**🧠 TreeSet Specific Methods:**

* first(), last()
* higher(E e), lower(E e)
* ceiling(E e), floor(E e)
* subSet(E from, E to)
* descendingSet()

🔸 Comparison Table

| **Feature** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| --- | --- | --- | --- |
| Order | No order | Insertion order | Sorted order |
| Duplicates | ❌ Not allowed | ❌ Not allowed | ❌ Not allowed |
| Nulls | ✅ One null | ✅ One null | ✅ No nulls |
| Performance | Fastest | Slower than HashSet | Slowest |
| Thread-safe | ❌ No | ❌ No | ❌ No |
| Use Case | Fast lookup | Ordered history/logs | Sorted unique data |

**🔚 Summary:**

* Use \*\*HashSet\*\* for fastest performance and you don’t care about order.
* Use \*\*LinkedHashSet\*\* when insertion order matters.
* Use \*\*TreeSet\*\* when you need automatic sorting.

**Overview of the Map Interface**

In Java, a **Map** is an object that maps keys to values. It does not allow duplicate keys, and each key can map to at most one value. The most common implementations are:

* **HashMap:** Unordered, fast, and allows one null key and multiple null values.
* **LinkedHashMap:** Maintains insertion (or access) order.
* **TreeMap:** Implements SortedMap, keeps the keys sorted, and does not allow null as a key.

Each implementation serves a different need based on performance, ordering, and sorting requirements.

**HashMap**

**🔹 Characteristics:**

* **Ordering:** Does not guarantee any order of the keys.
* **Performance:** Provides constant-time performance (O(1)) for basic operations like get() and put() on average.
* **Nulls:** Allows one null key and multiple null values.
* **Use Case:** Ideal when order is not important, such as maintaining a lookup table.

**✅ Real-World Example: Employee ID Directory**

Imagine an employee directory where employee IDs are mapped to employee names.

import java.util.HashMap;

import java.util.Map;

public class EmployeeDirectory {

public static void main(String[] args) {

// Create a HashMap to store employee ID as key and employee name as value.

Map<Integer, String> employeeMap = new HashMap<>();

// Adding entries

employeeMap.put(1001, "Alice Johnson");

employeeMap.put(1002, "Bob Smith");

employeeMap.put(1003, "Carol White");

// Access an element

System.out.println("Employee with ID 1002: " + employeeMap.get(1002));

// Check if a key or value exists

if (employeeMap.containsKey(1003)) {

System.out.println("Employee 1003 exists.");

}

if (employeeMap.containsValue("Alice Johnson")) {

System.out.println("Alice Johnson is in the directory.");

}

// Iterate over keys and values

System.out.println("Complete Employee Directory:");

for (Map.Entry<Integer, String> entry : employeeMap.entrySet()) {

System.out.println("ID: " + entry.getKey() + " | Name: " + entry.getValue());

}

}

}  
**🧠 Common Methods for HashMap:**

* **put(K key, V value)**: Inserts or updates the key–value pair.
* **get(Object key)**: Retrieves the value for the given key.
* **remove(Object key)**: Removes the mapping for the specified key.
* **containsKey(Object key)**, **containsValue(Object value)**: Checks for presence.
* **keySet()**, **values()**, **entrySet()**: Returns collections of keys, values, or key–value pairs.
* **size()**, **clear()**: Returns the number of mappings; clears the map.

**LinkedHashMap  
🔹 Characteristics:**

* **Ordering:** Maintains the order in which entries are inserted (or accessed, if configured).
* **Performance:** Slightly slower than HashMap due to the added overhead for maintaining order, but still provides O(1) performance for basic operations.
* **Nulls:** Allows one null key and multiple null values.
* **Use Case:** Useful for caches, access order tracking, or any scenario where the order of iteration is important.

**✅ Real-World Example: Recent User Activity Log**

Suppose you want to record user activities (e.g., page visits) while preserving the order in which activities were recorded.

import java.util.LinkedHashMap;

import java.util.Map;

public class UserActivityLog {

public static void main(String[] args) {

// Create a LinkedHashMap to maintain insertion order.

Map<String, String> activityLog = new LinkedHashMap<>();

// Adding user activities (key could be a unique identifier, value describes the action)

activityLog.put("2025-04-15 10:00", "Logged In");

activityLog.put("2025-04-15 10:05", "Viewed Dashboard");

activityLog.put("2025-04-15 10:10", "Updated Profile");

// Iterate in the order entries were added

System.out.println("User Activity Log:");

for (Map.Entry<String, String> entry : activityLog.entrySet()) {

System.out.println(entry.getKey() + " -> " + entry.getValue());

}

// Remove an activity

activityLog.remove("2025-04-15 10:05");

System.out.println("\nAfter removal:");

activityLog.forEach((time, action) -> System.out.println(time + " -> " + action));

}

}  
**🧠 Common Methods for LinkedHashMap:**

* All methods inherited from HashMap (e.g., put(), get(), remove(), etc.).
* Maintains a predictable iteration order because of its linked list structure.

**TreeMap**

**🔹 Characteristics:**

* **Ordering:** Stores keys in sorted (natural or customized) order, as it implements the NavigableMap interface.
* **Performance:** Offers O(log n) performance for get(), put(), and remove() operations, due to its underlying red-black tree structure.
* **Nulls:** **Does not allow** null keys, though values can be null.
* **Use Case:** Ideal for applications where keys must be in a sorted order, such as directories, sorted records, or range queries.

**✅ Real-World Example: Student Score Board Sorted by Roll Number**

Imagine a scenario where student roll numbers (as keys) are always kept in sorted order to easily find ranges or the highest/lowest roll number.

import java.util.Map;

import java.util.TreeMap;

public class StudentScoreBoard {

public static void main(String[] args) {

// Create a TreeMap to automatically sort keys (student roll numbers)

Map<Integer, String> scoreBoard = new TreeMap<>();

// Adding entries to the score board

scoreBoard.put(102, "Alice - 85");

scoreBoard.put(101, "Bob - 92");

scoreBoard.put(104, "Charlie - 78");

scoreBoard.put(103, "Diana - 89");

// Iterating; the keys will be in natural ascending order (sorted by roll number)

System.out.println("Student Score Board:");

for (Map.Entry<Integer, String> entry : scoreBoard.entrySet()) {

System.out.println("Roll No. " + entry.getKey() + ": " + entry.getValue());

}

// Example of retrieving the first (lowest) and last (highest) key entries

System.out.println("\nFirst Entry: " + ((TreeMap<Integer, String>)scoreBoard).firstEntry());

System.out.println("Last Entry: " + ((TreeMap<Integer, String>)scoreBoard).lastEntry());

}

}  
**🧠 Common Methods for TreeMap:**

* **put(K key, V value)**, **get(Object key)**, **remove(Object key)**: Similar to other maps.
* **firstKey()**, **lastKey()**: Returns the lowest and highest key.
* **firstEntry()**, **lastEntry()**: Returns the first and last key–value pair.
* **subMap(K fromKey, K toKey)**: Returns a view of the portion of the map whose keys range from fromKey to toKey.
* **headMap(K toKey)**, **tailMap(K fromKey)**: Views of the map for keys less than or greater than a certain key.

**Comparison Summary**

| **Feature** | **HashMap** | **LinkedHashMap** | **TreeMap** |
| --- | --- | --- | --- |
| **Ordering** | Unordered | Insertion (or access) order | Sorted (natural or custom) |
| **Performance** | O(1) for basic operations | O(1) (slightly slower overhead) | O(log n) due to tree structure |
| **Allow Nulls** | One null key, multiple null values | One null key, multiple null values | **No null keys allowed** |
| **Use Cases** | Fast lookup, caching | Maintaining insertion order | Sorted data, range queries |

**Final Remarks**

* **HashMap** is best when you need fast operations and the order of elements is unimportant.
* **LinkedHashMap** is ideal for cases where the order in which entries are added must be preserved, such as logging or caching recent actions.
* **TreeMap** is the go-to implementation when sorted order (and, frequently, range queries) on keys is required, such as in directories or ranked data sets.

**The Object Class in Java**

**Key Points:**

* **Single Root:**  
  Every Java class is a descendant of Object. This means that methods defined in Object are available to all objects.
* **Core Methods:**  
  Some core methods defined in Object include:
  + **toString()**: Returns a string representation of the object.
  + **equals(Object obj)**: Determines whether another object is “equal to” this one.
  + **hashCode()**: Returns an integer hash code representation of the object.
  + **clone()**: Creates and returns a copy of the object.
  + **getClass()**: Returns the runtime class of the object.
  + **finalize()**: Called by the garbage collector before object is reclaimed (deprecated in recent versions).

These methods form a foundation for object behavior and allow custom classes to be used in collections and frameworks that depend on consistent behavior across different object types.

**equals() and hashCode() Methods**

**equals() Method:**

* **Purpose:**  
  The equals() method checks whether two objects are logically "equal" (i.e., their state is the same), not necessarily the same instance in memory.
* **Default Behavior:**  
  The default implementation in the Object class checks for reference equality (i.e., whether both references point to the same object).
* **Overriding equals():**  
  When creating your own classes, you often override equals() to compare key fields within objects. For example, if you are creating a Person class, you might consider two Person objects equal if they have the same social security number or other unique identifier.

**hashCode() Method:**

* **Purpose:**  
  The hashCode() method returns an integer value, which is used in hashing-based collections (e.g., HashMap, HashSet). The idea is to use this hash code to determine the bucket where an object should be placed.
* **Contract with equals():**  
  The Java contract between equals() and hashCode() requires:
  + If two objects are equal according to equals(), they **must** return the same hash code.
  + It is not required that two objects with the same hash code are equal.
* **Overriding hashCode():**  
  When you override equals(), you should also override hashCode() so that they work together properly. Not following this rule can lead to incorrect behavior when objects are stored in collections that use hashing.

**Real-World Example: Employee Records**

Consider a real-world scenario where you have an Employee class. You want to store employee records in a HashSet to avoid duplicates. To do this reliably, you need to override equals() and hashCode() so that two Employee objects with the same employee ID are considered equal.  
  
import java.util.HashSet;

import java.util.Objects;

import java.util.Set;

class Employee {

private int id;

private String name;

private String department;

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

// Overriding equals() to compare employees based on their id

@Override

public boolean equals(Object obj) {

// Check if the object is compared with itself

if (this == obj) {

return true;

}

// Check if the passed object is an instance of Employee

if (!(obj instanceof Employee)) {

return false;

}

// Typecast and compare relevant fields (id)

Employee other = (Employee) obj;

return this.id == other.id;

}

// Overriding hashCode() to be consistent with equals()

@Override

public int hashCode() {

return Objects.hash(id);

}

@Override

public String toString() {

return "Employee{id=" + id + ", name='" + name + "', department='" + department + "'}";

}

}

public class EmployeeRecordDemo {

public static void main(String[] args) {

Set<Employee> employees = new HashSet<>();

// Adding employees

Employee emp1 = new Employee(101, "Alice Johnson", "Finance");

Employee emp2 = new Employee(102, "Bob Smith", "Engineering");

Employee emp3 = new Employee(101, "Alice J.", "Finance"); // Same id as emp1, should be considered duplicate

employees.add(emp1);

employees.add(emp2);

employees.add(emp3); // Because of overriding equals() and hashCode(), this won't be added

System.out.println("Employee Records in HashSet:");

for (Employee emp : employees) {

System.out.println(emp);

}

}

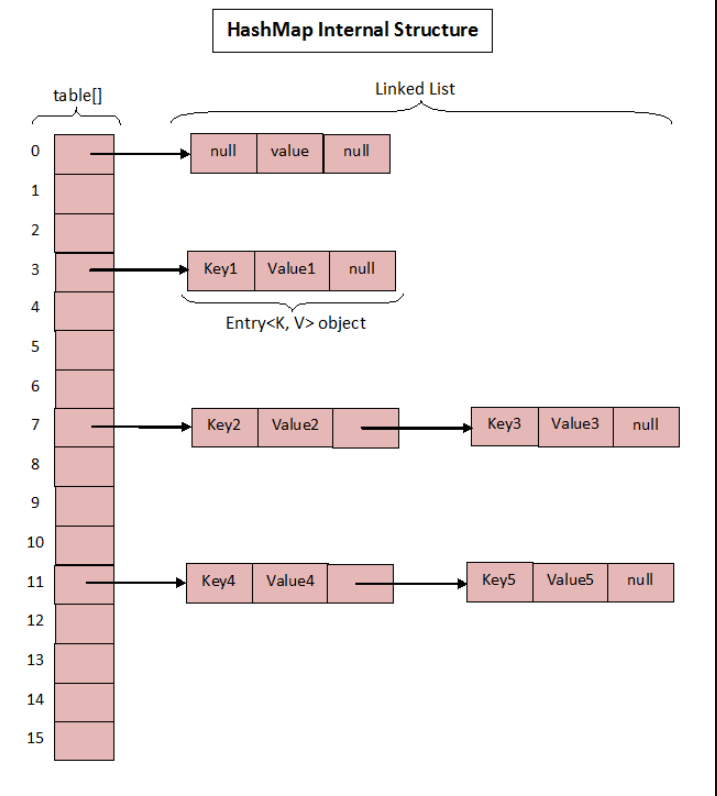
}  
**Explanation:**

1. **Overriding equals():**
   * The method first checks if the current instance and the passed object are the same.
   * Then, it checks whether the passed object is an instance of Employee.
   * Finally, it compares the id field. If two employees have the same id, they are considered equal.
2. **Overriding hashCode():**
   * We use Objects.hash(id) which returns a hash code based on the id field. Since equals() compares employees using id, the hash code is computed solely on id ensuring that equal employees get the same hash code.
3. **Real-World Use (HashSet):**
   * The HashSet uses hashCode() to determine the bucket in which to place an object, and equals() to check for equality.
   * In the example, even though emp1 and emp3 have different names, because they share the same id, they are considered equal. Therefore, adding both results in only one being stored in the HashSet.

**Why This Matters**

* **Data Consistency:**  
  For applications like employee record management, it is crucial to ensure that duplicate entries are not created. Overriding equals() and hashCode() correctly ensures that collections like HashSet or keys in a HashMap work as expected.
* **Performance in Collections:**  
  Proper implementation of hashCode() greatly affects the performance of hash-based collections. A good hash function distributes objects uniformly, reducing the chance of collisions and maintaining efficient access.
* **Logical Equality:**  
  In many real-world applications, two objects might represent the same real-world entity even if their internal state (like a name variation) is slightly different. Defining logical equality via equals() helps capture that notion.

**HashMap**

A HashMap in Java is a widely used data structure that stores key–value pairs and allows fast retrieval, insertion, and deletion operations. Internally, it uses the principles of hashing to distribute data across an array of buckets. Let’s dive into the internal mechanics and then illustrate with a real-world analogy.  
  
  
  
**1. Internal Structure and Mechanics**

**A. Buckets and Hashing**

* **Buckets:**  
  A HashMap maintains an array (often called the “table”) where each element is a bucket. Each bucket can hold multiple entries (nodes) if necessary.
* **Hash Function:**  
  When a key–value pair is added, the key’s hashCode() is calculated. Java then typically applies an additional process (bitwise operations) to distribute the hash codes uniformly across the table. The resulting value is used to determine the index in the array (usually using a modulus operation, e.g., index = hash & (n - 1) where n is the capacity of the table).

**B. Collision Resolution**

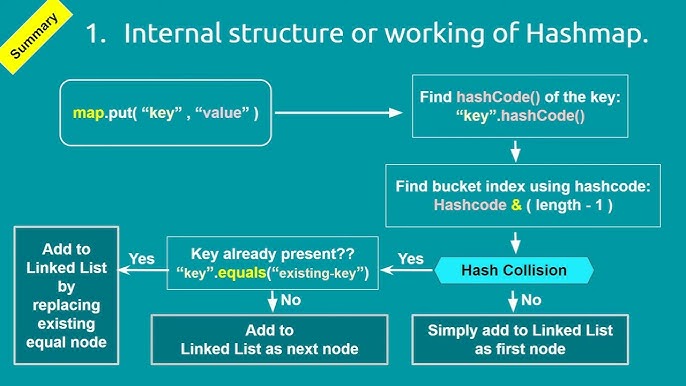
* **Chaining (Linked List):**  
  If two different keys produce the same hash index, they are stored in the same bucket. Originally, this was handled by maintaining a linked list for each bucket where multiple entries could reside.
* **Treeification:**  
  Starting with Java 8, if the number of elements in a single bucket exceeds a certain threshold (typically 8), the linked list is transformed into a balanced tree (a red-black tree). This makes lookups, insertions, and deletions within that bucket faster (O(log n) rather than O(n)).

**C. Resizing**

* **Load Factor:**  
  HashMap uses a load factor (by default 0.75). When the number of key–value pairs exceeds capacity \* loadFactor, the HashMap automatically resizes (usually doubling the number of buckets) and rehashes existing entries.
* **Rehashing:**  
  During resizing, each entry is recalculated for its new position in the larger table. This ensures the keys are still distributed uniformly.

**D. Insertion and Retrieval Process**

1. **Insertion (put method):**
   * The key’s hashCode() is computed.
   * The hash is compressed to find the bucket index.
   * If the bucket is empty, the new entry is inserted.
   * If there’s an existing entry (or several), the new key is compared (using equals()) with keys in the chain:
     + If a matching key is found, the value is updated.
     + If not, the new entry is appended to the bucket’s chain (or inserted in the appropriate order if the bucket is treeified).
2. **Retrieval (get method):**
   * The key’s hashCode() is computed and compressed to the corresponding bucket index.
   * The bucket is scanned (either as a linked list or via a tree search) to find an entry whose key matches the given key using equals().
   * The value associated with the key is returned.



**Real-World Analogy: Library Book Management System**

Imagine a **library** with thousands of books. Instead of having all books on a single shelf, the librarian organizes them by a simple hash-based system.

**A. Organizing Books (Insertion)**

1. **Book Identification:**
   * Every book has a unique ISBN number. This ISBN serves as the key in our system.
   * A hash function is applied to the ISBN to determine which shelf (bucket) the book should go on.
2. **Storing on Shelves:**
   * Each shelf (bucket) can hold multiple books. If two books hash to the same shelf, they are placed on that shelf together.
   * If too many books accumulate on one shelf (a long chain), the librarian reorganizes that shelf into a more refined system (like arranging books alphabetically by title or author on that shelf)—this is analogous to treeification in HashMap.

**B. Finding a Book (Retrieval)**

1. **Lookup Process:**
   * When you request a book by its ISBN, the hash function quickly directs the librarian to a specific shelf.
   * The librarian then scans that shelf (or looks through an alphabetically sorted list if the shelf has been reorganized) to find your book.
2. **Efficiency:**
   * Because of the hash-based organization, even if the library has thousands of books, the librarian only needs to look at a small subset on a specific shelf—ensuring a fast retrieval process.

**Example Code for a Simple HashMap Use Case**

Imagine you’re building a simple directory for a company’s employees using their employee ID (an integer) as the key and their name as the value:

import java.util.HashMap;

import java.util.Map;

public class EmployeeDirectory {

public static void main(String[] args) {

// Creating a HashMap to store employee records

Map<Integer, String> employeeDirectory = new HashMap<>();

// Inserting employee records

employeeDirectory.put(1001, "Alice Johnson");

employeeDirectory.put(1002, "Bob Smith");

employeeDirectory.put(1003, "Carol White");

// Inserting a record with a key that collides (hypothetically, if 1002 collided with another key)

// HashMap handles collisions internally as described above.

employeeDirectory.put(1002, "Robert Smith"); // This updates the existing record for key 1002

// Retrieving and printing an employee name based on ID

String employeeName = employeeDirectory.get(1002);

System.out.println("Employee with ID 1002: " + employeeName);

}

}  
**Explanation of the Code Example:**

* **Insertion:**
  + The put() method computes the hash code for each employee ID and places the entries into an array of buckets.
  + If there’s a collision (i.e., two keys map to the same bucket), HashMap will use a linked list (or a tree if many collisions occur) within that bucket.
* **Retrieval:**
  + The get() method uses the hash code of the key (employee ID) to locate the correct bucket and then searches within that bucket to return the corresponding employee name.

In Java, both **Comparable** and **Comparator** are used to compare objects, but they serve different purposes and are used in different scenarios. Below is an in-depth explanation of each, along with examples to demonstrate how and when to use them.  
  
**1. Comparable Interface**

**What It Is:**

* **Comparable** is a built-in interface in Java.
* It is used for defining the *natural ordering* of objects.
* A class that implements the Comparable interface must override the compareTo() method.
* This method compares the current object with another object of the same type and returns:
  + A **negative integer** if the current object is less than the other.
  + **Zero** if they are equal.
  + A **positive integer** if the current object is greater than the other.

**When to Use:**

* Use Comparable when a single, natural sorting order is defined for the objects.
* It is common to implement Comparable if you expect the objects to be sorted often in a consistent manner (e.g., sorting student records by roll number or name).

**Example: Sorting a List of Books by Title**

// Book.java

public class Book implements Comparable<Book> {

private String title;

private String author;

private int publicationYear;

public Book(String title, String author, int publicationYear) {

this.title = title;

this.author = author;

this.publicationYear = publicationYear;

}

// Getters (for demonstration purposes)

public String getTitle() { return title; }

public String getAuthor() { return author; }

public int getPublicationYear() { return publicationYear; }

// Define natural ordering based on title (alphabetical order)

@Override

public int compareTo(Book other) {

return this.title.compareTo(other.title);

}

@Override

public String toString() {

return title + " by " + author + " (" + publicationYear + ")";

}

}

// Main.java

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Book> library = new ArrayList<>();

library.add(new Book("Effective Java", "Joshua Bloch", 2008));

library.add(new Book("Clean Code", "Robert C. Martin", 2008));

library.add(new Book("Java Concurrency in Practice", "Brian Goetz", 2006));

// Sort using natural ordering defined in Book.compareTo()

Collections.sort(library);

System.out.println("Books sorted by title:");

for (Book book : library) {

System.out.println(book);

}

}

}  
**Explanation:**

* The Book class implements **Comparable** and overrides compareTo(Book other).
* Natural order is defined as alphabetical order of the book’s title.
* In the main program, Collections.sort(library) sorts the books based on their titles without needing any additional comparator.

**2. Comparator Interface**

**What It Is:**

* Comparator is a separate interface used to define multiple ways of ordering objects.
* It is useful when you want to sort objects in different ways at different times.
* A class that implements Comparator must override the compare() method, which takes two objects and returns:
  + A negative integer if the first object is less than the second.
  + Zero if they are equal.
  + A positive integer if the first object is greater than the second.

**When to Use:**

* Use Comparator when:
  + You want to sort objects in an order other than their natural ordering.
  + You need to implement multiple sort criteria for a class.
  + The class does not implement Comparable, or you do not wish to modify the class.

**Example 1: Sorting Books by Publication Year**import java.util.Comparator;

// Comparator to sort books by publication year

class PublicationYearComparator implements Comparator<Book> {

@Override

public int compare(Book b1, Book b2) {

return Integer.compare(b1.getPublicationYear(), b2.getPublicationYear());

}

}

// Main.java (continued from previous example)

import java.util.Collections;

public class MainWithComparator {

public static void main(String[] args) {

List<Book> library = new ArrayList<>();

library.add(new Book("Effective Java", "Joshua Bloch", 2008));

library.add(new Book("Clean Code", "Robert C. Martin", 2008));

library.add(new Book("Java Concurrency in Practice", "Brian Goetz", 2006));

// Sort using Comparator: by publication year

Collections.sort(library, new PublicationYearComparator());

System.out.println("Books sorted by publication year:");

for (Book book : library) {

System.out.println(book);

}

}

}

**3. Key Differences Between Comparable and Comparator**

| Aspect | Comparable | Comparator |
| --- | --- | --- |
| Location of Code | Implemented in the class of objects that need to be compared. | Implemented in a separate class or using lambda. |
| Method to Override | compareTo(Object o) | compare(Object o1, Object o2) |
| Natural Ordering | Provides a natural, default order. | Provides custom ordering (multiple ways possible). |
| Modifying Class | Requires modifying the class whose objects you want to sort. | Does not require modifying the class. |
| Usage | Used when one single order is natural to the object. | Used to implement different sorting strategies. |

**4. Real-World Analogy**

Imagine a library where books are typically arranged by title (natural order). This is like implementing Comparable in the Book class. However, sometimes a librarian may need to arrange books by publication year or by author's name for special displays—this is where Comparator comes in handy. The librarian can use different sorting methods without changing the inherent order of books when returned to the shelves.

**3. Real-World Example: Task Scheduler**

Imagine you’re building a simple task scheduler for an operating system, where tasks have different priorities. The scheduler always processes the task with the highest priority first. Here is how you might implement this using PriorityQueue.

**Step A: Define a Task Class**

Create a Task class that holds task details and implements the Comparable interface to order tasks by priority (here, a lower value indicates higher priority). You can also use a custom Comparator if you prefer separating comparison logic from the model.

class Task implements Comparable<Task> {

private String name;

private int priority; // Lower value indicates higher priority

public Task(String name, int priority) {

this.name = name;

this.priority = priority;

}

public String getName() {

return name;

}

public int getPriority() {

return priority;

}

@Override

public int compareTo(Task other) {

// Lower priority number means higher precedence

return Integer.compare(this.priority, other.priority);

}

@Override

public String toString() {

return "Task[name=" + name + ", priority=" + priority + "]";

}

}  
  
**Generics**Generics in Java allow you to write flexible, type-safe code by parameterizing classes, interfaces, and methods with type placeholders. This means you can design components that work with any type while catching type mismatches at compile time, instead of at runtime. Below, we’ll explore generics in detail, explain when and where to use them, and provide real-world examples.  
  
**1. What Are Generics?**

Generics enable classes, interfaces, and methods to operate on objects of various types while providing compile-time type safety. Rather than writing code that works on raw types (like Object), you can specify a parameterized type (e.g., Box<T>), reducing the need for casting and minimizing runtime errors.

**Benefits:**

* **Type Safety:** Misplaced types are detected during compilation.
* **Elimination of Casts:** When retrieving objects, no explicit type conversion is required.
* **Code Reusability:** Write a single class or method that works with different data types.
* **Improved Readability and Maintainability:** Code expresses its intent clearly and is easier to maintain.

**2. When and Where to Use Generics**When to Use Generics:

* Collections: Most of the Java Collections Framework uses generics (e.g., ArrayList<T>, HashMap<K, V>) to enforce type safety.
* Utility Classes and Methods: For example, helper classes like a Pair or Box that can hold any type.
* Custom Data Structures: When building your own data structure (e.g., a linked list or tree), generics can ensure that your structure is usable with any data type.
* Framework and API Design: When designing libraries, using generics allows your API to work with a wide range of data types without sacrificing safety.

Where to Use Generics:

* Class-Level: When creating a class that can work with multiple types.

public class Box<T> {

private T content;

public void setContent(T content) {

this.content = content;

}

public T getContent() {

return content;

}

@Override

public String toString() {

return "Box contains: " + content;

}

}

**Method-Level:** For utility or helper methods that need to operate on different types.  
public class Utility {

// A generic method that prints array elements

public static <E> void printArray(E[] array) {

for (E element : array) {

System.out.println(element);

}

}

}  
  
**Interface-Level**: When defining an interface that must be implemented for various types.  
public interface Pair<K, V> {

K getKey();

V getValue();

}  
  
**3. Real-World Examples**

Example 1: A Generic Container

Imagine you’re building an e-commerce platform where different types of items are stored and managed. You can create a generic container (or box) that can hold any type of product.  
**Code: Generic Box Example**// A generic container class that can hold any type of product.

public class Box<T> {

private T item;

public Box(T item) {

this.item = item;

}

public T getItem() {

return item;

}

public void setItem(T item) {

this.item = item;

}

@Override

public String toString() {

return "Box contains: " + item;

}

public static void main(String[] args) {

// Box that holds a String representing a product name.

Box<String> stringBox = new Box<>("Laptop");

System.out.println(stringBox);

// Box that holds an Integer representing a product ID.

Box<Integer> integerBox = new Box<>(12345);

System.out.println("Product ID: " + integerBox.getItem());

}

}

# **Assignments for this week**

**1. Library Book Management System**

**Objective:**  
Implement a system to manage books in a library, using collections to store and retrieve book information efficiently.

**Requirements:**

* Create a class Book with attributes like title, author, isbn, and yearPublished.
* Use a HashSet<Book> to store unique books (no duplicates).
* Implement methods to:
  + Add a new book.
  + Remove a book by ISBN.
  + Search for a book by title or author using a HashMap<String, List<Book>> (key is title/author and value is a list of books).
  + Display all books in the library.
* Implement an interface LibraryOperations that defines methods like addBook, removeBook, and searchBook.
* Use TreeSet for storing and sorting books by yearPublished.

**Sample Input:**

LibrarySystem library = new LibrarySystem();

library.addBook(new Book("The Alchemist", "Paulo Coelho", "123456", 1988));

library.addBook(new Book("Clean Code", "Robert C. Martin", "654321", 2008));

library.searchBookByTitle("The Alchemist");

library.removeBook("123456");

**Expected Output:**

Book added: The Alchemist

Book added: Clean Code

Search result for title 'The Alchemist': [The Alchemist, Paulo Coelho, 1988]

Book removed: The Alchemist

**2. Online Shopping Cart System**

**Objective:**  
Simulate an online shopping cart where customers can add products, apply discounts, and view cart details.

**Requirements:**

* Create a Product class with attributes like name, price, and category.
* Use a HashMap<String, Product> to store products by name (key is the product name and value is the product).
* Create a ShoppingCart class that stores a list of products and uses a HashMap<Product, Integer> to store product quantities.
* Implement methods to:
  + Add products to the cart.
  + Remove products from the cart.
  + Apply a discount on the entire cart (using a double to calculate total price after applying a percentage discount).
  + View all products in the cart with quantities and total price.

**Sample Input:**

ShoppingCart cart = new ShoppingCart();

cart.addProduct(new Product("Laptop", 60000, "Electronics"));

cart.addProduct(new Product("Shirt", 1500, "Clothing"));

cart.applyDiscount(10);

cart.viewCart();

**Expected Output:**

Product added to cart: Laptop

Product added to cart: Shirt

Applying discount of 10%. Total price: 67,650

Shopping Cart:

Laptop: 1 x 60,000

Shirt: 1 x 1,500

Total: 67,650

**3. Employee Database System**

**Objective:**  
Create an employee database system that allows you to store employee details and perform operations such as searching, adding, updating, and removing employees.

**Requirements:**

* Create an Employee class with attributes like id, name, department, and salary.
* Use a HashMap<Integer, Employee> to store employees by their id.
* Implement methods to:
  + Add a new employee.
  + Update an employee’s details.
  + Delete an employee by id.
  + Search for an employee by id or name using a HashMap for efficient lookups.
  + Display all employees in the database using a TreeSet sorted by name.
  + Calculate the average salary of employees in each department using a Map<String, List<Employee>>.

**Sample Input:**

EmployeeDatabase db = new EmployeeDatabase();

db.addEmployee(new Employee(101, "John", "HR", 50000));

db.addEmployee(new Employee(102, "Alice", "Finance", 60000));

db.updateEmployee(102, new Employee(102, "Alice", "Finance", 65000));

db.deleteEmployee(101);

db.displayAllEmployees();

db.calculateAverageSalary("Finance");

**Expected Output:**

Employee added: John, HR, 50,000

Employee added: Alice, Finance, 60,000

Employee updated: Alice, Finance, 65,000

Employee deleted: John

All Employees:

Alice, Finance, 65,000

Average Salary in Finance: 65,000