

UNIT 10

HOLDING COLLECTION OF DATA

(3 Hours) — Object Oriented Programming in Java

Course Code: CACS204 | **Year/Semester:** II/III

Unit syllabus: Arrays and Collection Classes/Interfaces; Map, List and Set implementations; ArrayList, LinkedList, HashSet and TreeSet; accessing collections; Iterator; Comparator.

1. Learning Objectives

- Understand why Java provides the Collections Framework in addition to arrays.
- Differentiate Collection, List, Set and Map interfaces.
- Use ArrayList, LinkedList, HashSet and TreeSet for storing and processing data.
- Traverse collections using loops and Iterator.
- Sort objects using Comparable and Comparator.

2. Arrays and Collections

2.1 Array

An array is a fixed-size data structure that stores multiple values of the same declared type. Array elements are accessed using an index starting from 0.

```
int[] marks = {78, 85, 91};  
System.out.println(marks[0]); // 78
```

- Size is fixed after creation.
- Can store primitives (int, double, char) and object references.
- Access by index is fast, but inserting or deleting in the middle requires shifting elements.
- Arrays use the length field: marks.length.

2.2 Collection and Collections Framework

The Java Collections Framework (JCF) is a set of interfaces, implementation classes and algorithms used to store, retrieve, manipulate and process groups of objects. It is mainly in the java.util package.

Collection is an interface representing a group of individual objects. **Collections** (plural) is a utility class containing static methods such as sort(), reverse(), shuffle() and max().

Term	Meaning	Example
Collection	Root interface for groups of individual elements	List, Set, Queue

Term	Meaning	Example
Collections	Utility class with algorithms for collections	Collections.sort(list)
Map	Stores key-value pairs; not a subtype of Collection	HashMap, TreeMap
Arrays	Utility class for array operations	Arrays.sort(array)

Simple memory trick

Collection = a group of items. Collections = toolbox of methods. Map = key paired with value.

3. Collection Framework Hierarchy

The main interfaces organize data according to how elements are stored and accessed. A simplified hierarchy is shown below.

Interface / Class	Main property
Iterable	Allows an object to be used in an enhanced for loop.
Collection	Common operations such as add, remove, size and contains.
List	Ordered sequence; allows duplicates and index-based access.
Set	Does not allow duplicate elements.
Queue	Typically processes elements in a particular order, often FIFO.
Map	Maps unique keys to values; separate from Collection.

Simplified hierarchy: Iterable → Collection → List / Set / Queue. Map is a separate interface. Classes such as ArrayList and LinkedList implement List; HashSet and TreeSet implement Set.

4. List Interface

List is an ordered collection (sequence). Each element has an index, starting at 0. A List can contain duplicate elements and usually allows multiple null values, depending on implementation.

4.1 Common List methods

Method	Purpose
add(element)	Adds an element to the end.
add(index, element)	Inserts at a specified index.
get(index)	Returns the element at the index.
set(index, element)	Replaces an element.
remove(index) / remove(object)	Removes by index or matching object.
indexOf(object)	Returns first matching index, or -1.

Method	Purpose
size()	Returns number of elements.
contains(object)	Checks whether an element exists.

4.2 ArrayList

ArrayList is a resizable-array implementation of List. It grows automatically as elements are added and preserves insertion order.

- Fast random access using get(index).
- Adding at the end is usually efficient; inserting/removing near the beginning or middle can require shifting elements.
- Allows duplicates and null values.
- Not synchronized by default; concurrent modification requires appropriate coordination.

Example: ArrayList

```
import java.util.ArrayList;
ArrayList names = new ArrayList<>();
names.add("Asha"); names.add("Ram"); names.add("Asha");
System.out.println(names); // [Asha, Ram, Asha]
System.out.println(names.get(1)); // Ram
names.set(1, "Sita");
names.remove("Asha");
```

What to remember

ArrayList is suitable when frequent reading by index and iteration are needed, and insertions/deletions in the middle are relatively less frequent.

4.3 LinkedList

LinkedList is a doubly linked list implementation. Each node stores the element and links to the previous and next nodes. It implements both List and Deque.

- Maintains insertion order and permits duplicates and null elements.
- Does not provide fast random access; get(index) requires traversing nodes.
- Adding/removing at the ends is efficient.
- Can be used as a queue or deque with methods such as addFirst(), addLast(), removeFirst() and removeLast().

Example: LinkedList

```
import java.util.LinkedList;

LinkedList tasks = new LinkedList<>();
tasks.add("Study"); tasks.add("Practice");
tasks.addFirst("Wake up");
System.out.println(tasks);
tasks.removeLast();
```

Feature	ArrayList	LinkedList
Underlying structure	Resizable array	Doubly linked nodes
Index access	Generally fast	Sequential traversal
Add/remove at ends	End append efficient; front shifts	Efficient at ends
Memory	Usually lower per element	Extra links per node
Typical use	Frequent access/iteration	Deque operations or changes at known node positions

5. Set Interface

Set is a collection that does not contain duplicate elements. It has no index-based access. The order depends on the implementation.

5.1 HashSet

HashSet stores unique elements using a hash table. It does not guarantee insertion order or sorted order.

- Duplicate values are ignored according to equals() and hashCode().
- Allows one null element.
- Offers generally fast add, remove and contains operations when hashes are well distributed.
- Use it when uniqueness matters and order is not important.

Example: HashSet

```
import java.util.HashSet;

HashSet cities = new HashSet<>();
cities.add("Kathmandu"); cities.add("Pokhara");
cities.add("Kathmandu");
System.out.println(cities); // only unique values; order unspecified
```

5.2 TreeSet

TreeSet stores unique elements in sorted order, using their natural ordering or a Comparator supplied at construction. It is based on a balanced tree structure.

- Elements must be mutually comparable, or a suitable Comparator must be supplied.
- Common operations add, remove and contains take $O(\log n)$ time.
- Does not allow null with natural ordering in modern Java implementations.
- Useful when unique elements must remain sorted.

Example: TreeSet

```
import java.util.TreeSet;
TreeSet nums = new TreeSet<>();
nums.add(30); nums.add(10); nums.add(20); nums.add(10);
System.out.println(nums); // [10, 20, 30]
```

Feature	HashSet	TreeSet
Ordering	No guaranteed order	Sorted order
Typical performance	Average $O(1)$ basic operations	$O(\log n)$ basic operations
Duplicates	Not allowed	Not allowed
Null	One null permitted	Generally no null with natural ordering
Implementation	Hash table	Tree-based balanced structure

6. Map Interface

Map stores data as key-value pairs. Each key is unique, but values may be duplicated. Map is not a subtype of Collection because it stores associations rather than individual elements.

6.1 Important Map methods

Method	Meaning
put(key, value)	Adds a mapping or replaces the value for an existing key.
get(key)	Returns mapped value, or null if no mapping (unless mapped value itself is null).
remove(key)	Removes the mapping for a key.
containsKey(key)	Checks whether a key is present.
containsValue(value)	Checks whether a value is present.
keySet()	Returns a Set view of keys.
values()	Returns a Collection view of values.
entrySet()	Returns a Set view of key-value entries.

Method	Purpose
remove()	Removes the last element returned by next(), where supported; may be called once per next().

Example: Iterator

```
import java.util.*;
ArrayList items = new ArrayList<>();
items.add("Pen"); items.add("Book");
Iterator it = items.iterator();
while (it.hasNext()) {
    String item = it.next();
    System.out.println(item);
}
```

7.3 Removing safely during iteration

When removing elements while traversing, use `Iterator.remove()` rather than directly calling `collection.remove()` inside a for-each loop. Direct structural changes can cause `ConcurrentModificationException` in fail-fast iterators.

Safe removal example

```
Iterator it = numbers.iterator();
while (it.hasNext()) {
    if (it.next() < 0) {
        it.remove();
    }
}
```

For a `List`, `ListIterator` also supports traversal in both directions and methods such as `add()` and `set()`.

8. Comparator and Sorting

`Comparator` is a functional interface used to define a custom ordering for objects. It is in `java.util` and provides the `compare(a, b)` method.

`compare(a, b)` returns a negative value if `a` should come before `b`, zero if they are equal in this ordering, and a positive value if `a` should come after `b`.

8.1 Comparable vs Comparator

Comparable	Comparator
Defines natural ordering inside the class.	Defines an external or custom ordering.

Comparable	Comparator
Uses compareTo(Object).	Uses compare(Object, Object).
Typically one natural order.	Can define multiple different orders.
java.lang package.	java.util package.

Example: sort strings by length

```
List words = new ArrayList<>();
words.add("elephant"); words.add("cat"); words.add("bird");
words.sort(Comparator.comparingInt(String::length));
System.out.println(words); // [cat, bird, elephant]
```

Example: descending integer order

```
List values = new ArrayList<>(Arrays.asList(4, 1, 9, 2));
values.sort(Comparator.reverseOrder());
System.out.println(values); // [9, 4, 2, 1]
```

Other useful forms include `Comparator.comparing(Person::getName)`, `thenComparing(...)`, and `reversed()`. Avoid subtracting numeric fields in a comparator because overflow can produce incorrect results; use `Integer.compare(a, b)`.

9. Generics and Type Safety

Generics allow a collection to specify the type of objects it stores. This improves compile-time type checking and reduces explicit casting.

```
ArrayList<String> names = new ArrayList<>();
names.add("Asha");
String first = names.get(0);
```

Without a type parameter, a collection can hold different object types and retrieval may require casting. Prefer parameterized collections such as `List` or `Map`.

10. Practical Program: Demonstrating List, Set, Map and Iterator

```
import java.util.*;
public class CollectionDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Java"); list.add("Python"); list.add("Java");
        System.out.println("List: " + list);
        Set<String> set = new TreeSet<>(list);
        System.out.println("Unique sorted: " + set);
        Map<Integer, String> map = new HashMap<>();
        map.put(101, "Asha"); map.put(102, "Ram");
        System.out.println("Student 101: " + map.get(101));
        Iterator<String> it = list.iterator();
        while (it.hasNext()) { System.out.println(it.next()); }
    }
}
```

Expected behavior

List keeps both occurrences of Java and preserves insertion order. TreeSet removes the duplicate and sorts the strings. Map retrieves a value using its key. Iterator visits each List element.

11. Big-O Quick Revision

Operation (typical)	ArrayList	LinkedList	HashSet	TreeSet
get by index	O(1)	O(n)	Not applicable	Not applicable
add at end	Amortized O(1)	O(1)	Average O(1)	O(log n)
contains/search	O(n)	O(n)	Average O(1)	O(log n)
Remove by value	O(n)	O(n) search + links	Average O(1)	O(log n)

These are typical complexity estimates, not absolute guarantees; performance depends on implementation details, hashing quality and the operation used.

12. Important Exam Questions

Short-answer / viva questions

- **What is a collection?** An object representing a group of individual elements.
- **Difference between Collection and Collections?** Collection is an interface; Collections is a utility class.
- **Can a List contain duplicates?** Yes.
- **Can a Set contain duplicates?** No.
- **Does HashSet maintain insertion order?** No guaranteed order.
- **Which collection stores key-value pairs?** Map.
- **What is an Iterator?** An object used to traverse elements one by one.
- **What is Comparator?** An interface defining custom object ordering.
- **Which is sorted: HashSet or TreeSet?** TreeSet.
- **Why use generics?** To provide type safety and reduce casting.

Long-answer questions to prepare

- Explain the Java Collections Framework and its hierarchy with examples.
- Explain List and compare ArrayList with LinkedList.
- Explain Set and compare HashSet with TreeSet.
- Explain the Map interface and demonstrate HashMap operations.
- Explain Iterator and show how to remove elements safely during iteration.
- Explain Comparator and distinguish it from Comparable with code.

KEY TAKEAWAYS — LAST-MINUTE REVISION

- ✓ Array = fixed size; ArrayList = resizable array.
- ✓ List: ordered, index-based, duplicates allowed.
- ✓ Set: unique elements; HashSet has no guaranteed order; TreeSet sorts.
- ✓ Map: unique keys mapped to values; it is not a Collection.
- ✓ LinkedList is a doubly linked list and also supports Deque operations.
- ✓ Iterator uses hasNext() and next(); use iterator.remove() for safe removal during traversal.
- ✓ Comparator customizes sorting using compare(a, b); Comparable defines a class's natural order.

— *End of Unit 10* —

bcalink.com