Day13_OOPJ_Sanket_Shalukar

Thursday, September 11, 2025 10:18 AM

Topics that are in the day 13

Access Modifiers
Non access modifiers
Singleton
Collection – Comparable Comparative

Access Modifiers

Access modifiers in Java are used to define the **visibility/scope** of classes, variables, constructors, and methods.

Types of Access Modifiers

1. **public**
- Accessible from **anywhere** in the program.
- Highest level of visibility.
- Can be used with classes, methods, variables, constructors.
- Example:
  public void display() {}

2. **protected**
- Accessible within the **same package**.
- Also accessible by **subclasses in different packages**.
- Often used for **inheritance**.
- Example:
  protected int rollNo;

3. **default (no modifier)**
- If no modifier is specified, it is **package-private**.
- Accessible only within the **same package**.
- Example:
  String name;

4. **private**
- Accessible **only within the same class**.
- Cannot be inherited.
- Example:
  private double salary;

Rules

- At class level → only **public** and **default** are allowed.
- Methods/variables/constructors can use all four.
- Used to implement **encapsulation**.

Non-Access Modifiers:

- static
- final
- abstract
- synchronized
- volatile
- transient
- native
- strictfp

- **static**
  - Belongs to the class instead of an object.
  - Shared among all instances.
  - Can be applied to variables, methods, blocks, and nested classes.

- **Final**
  - Used to declare constants.
  - Prevents method overriding.
  - Prevents class inheritance.

- **Abstract**
  - Applied to classes and methods.
  - Abstract class cannot be instantiated.
  - Abstract method must be implemented by child classes.

- **synchronized**
  - Ensures only one thread can access a method/block at a time.
  - Used to achieve thread safety in multithreaded applications.
- **volatile**
  - Variable is always read from main memory.
  - Ensures visibility of variable changes across threads.
  - Useful in multithreading scenarios.
- **transient**
  - Used in serialization.
  - Skips saving specific fields when an object is serialized.
  - Helps protect sensitive information.
- **native**
  - Used with methods implemented in non-Java languages (like C/C++).
  - Indicates that the method body is provided externally.
- **strictfp**
  - Stands for *strict floating point*.
  - Ensures floating-point calculations are consistent across platforms.
  - Guarantees IEEE 754 compliance.

Singleton Class

A **Singleton class** is a class that allows only **one instance/object** to exist in the JVM.

Steps to Create Singleton

- Make **constructor private**.
- Create a **private static object** of the class.
- Provide a **public static method** to return that object.

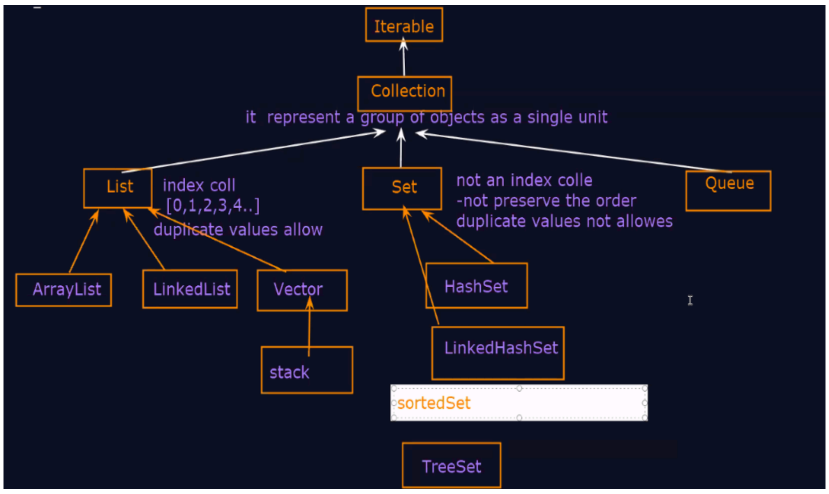
Why Singleton?

- Memory efficient.
- Used for **Database connection, Logger system, Configuration manager**.

  Note
- In multithreading → use **synchronized** or **double-checked locking** to avoid multiple objects.

Collections Framework.

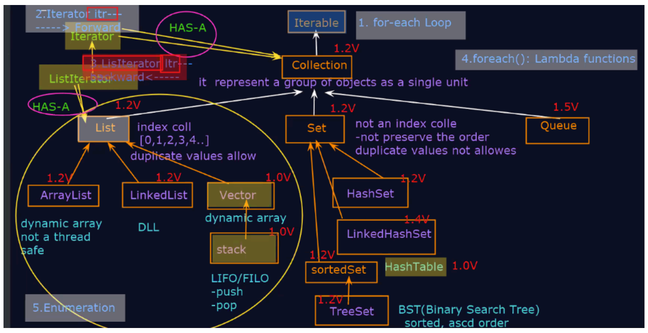
The Collection Framework provides **ready-made classes and interfaces** for storing/managing data.



Iterator & Iterable

- **Iterable** – It is the root interface of the Java Collection Framework. Any class that implements Iterable can be traversed using a for-each loop or by creating an Iterator.
- **Purpose of Iterable** – It provides a way for collections to return an Iterator, which allows sequential access to elements without exposing internal details of the collection.
- **Iterator** – It is an object that allows traversing through a collection one element at a time. It is called a **"cursor"** because it moves through elements sequentially.

- **Important Methods of Iterator**
- `hasNext()` – This method returns true if there are more elements to traverse.
- `next()` – This method returns the next element in the collection.
- `remove()` – This method removes the last element returned by the `next()` method.
- **Usage** – Iterator can only move in a forward direction. It works with all types of collections such as List, Set, and Queue.



List Iterator

- **Definition**
  ListIterator is a special type of iterator available only for classes that implement the List interface, such as ArrayList and LinkedList.
  Unlike Iterator, ListIterator allows bidirectional traversal, meaning we can move both forward and backward through a list.
  It is obtained by calling the `listIterator()` method on a List object.

- **Important Methods of ListIterator**
- `hasNext()` – Returns true if there is a next element when moving forward.
- `next()` – Returns the next element and moves the cursor forward.
- `hasPrevious()` – Returns true if there is a previous element when moving backward.
- `previous()` – Returns the previous element and moves the cursor backward.
- `add(E e)` – Adds a new element at the current position of the cursor.
- `set(E e)` – Replaces the last element returned with a new value.
- `remove()` – Removes the last element returned by `next()` or `previous()`.
- **Usage** – ListIterator is mainly used when there is a need to traverse a list in both directions or modify elements while traversing.

Ways to Print Collections

- **For-each Loop** – This is the simplest way to print collection elements. It is used only for reading elements and works with any class that implements `Iterable`.
- **Iterator** – Iterator is a universal cursor for all collections. It allows forward-only traversal and provides methods like `hasNext()`, `next()`, and `remove()`.
- **ListIterator** – This is a special iterator that works only with lists such as ArrayList and LinkedList. It allows both forward and backward traversal and also supports adding or updating elements.
- **forEach() Method** – Introduced in Java 8, this method uses a functional style for traversal. It makes printing more compact and can use lambda expressions or method references.
- **Enumeration** – Enumeration is an older technique used with legacy classes like Vector and Stack. It is now mostly replaced by Iterator, but still exists for backward compatibility.
- **Streams** – Streams were introduced in Java 8 and provide an advanced way to process collections. They not only print elements but also allow operations like filtering, mapping, and sorting.

4. Set

- **Definition**
- A Set in Java is a collection that represents a group of unique elements. It does not allow duplicate values, and at most one `null` element is permitted depending on the implementation.
- **Ordering**

  Set does not guarantee order of elements in general. Some implementations maintain insertion order (LinkedHashSet) or sorted order (TreeSet).
- **No Index Access**

  Unlike List, Set does not provide index-based access to elements. Traversal is done using iterators, for-each loop, or streams.
- **HashSet**

This implementation stores elements in a hash table. It does not maintain order, but provides very fast performance for operations like add, remove, and search.

- **LinkedHashSet**

  This implementation maintains elements in the order they were inserted. It is useful when both uniqueness and ordering are required.

- **TreeSet**

  This implementation stores elements in a sorted (ascending) order. It uses a Red-Black Tree internally and does not allow `null` values.

- **Use Cases**

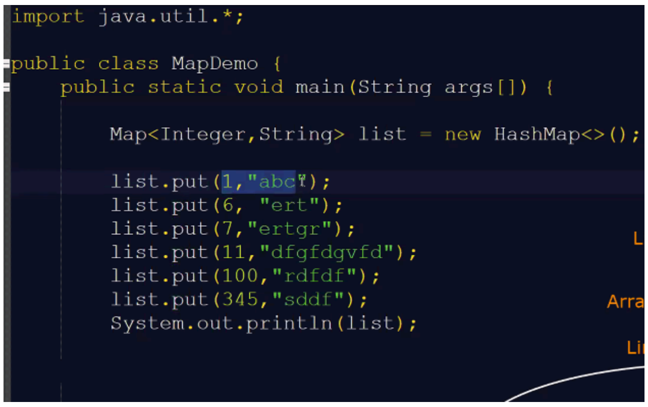
  Set is best used when we need to avoid duplicate entries, such as storing unique IDs, email addresses, or usernames.

Types of Set

HashSet
Unordered, fast.

LinkedHashSet :
Maintains insertion order.

TreeSet
Sorted order, uses Comparable or Comparator.



```
import java.util.*;

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

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

        list.put(1,"abc");
        list.put(6, "ert");
        list.put(7,"ertgr");
        list.put(11,"dfgfdgvfd");
        list.put(100,"rdfdf");
        list.put(345,"sddf");
        System.out.println(list);
```

5. HashMap

- **Definition** :

  A HashMap in Java is a part of the collection framework that stores data in the form of key–value pairs.

- **Key Characteristics** –
  - Each key in a HashMap must be unique.
  - Values can be duplicated.
  - One null key is allowed, and multiple null values are allowed.
  - The order of elements is not guaranteed because HashMap does not maintain insertion order.

- **Important Methods** :
  - `put(key, value)` – Inserts a new key–value pair or updates the value if the key already exists.
  - `get(key)` – Retrieves the value associated with the given key.
  - `remove(key)` – Removes the key–value pair for the specified key.
  - `containsKey(key)` – Checks if a given key exists in the map.
  - `containsValue(value)` – Checks if a given value exists in the map.
  - `keySet()` – Returns a set of all the keys present in the map.
  - `values()` – Returns a collection of all values present in the map.
  - `entrySet()` – Returns a set of all key–value pairs (entries).

- **Advantages** –
  - Provides fast insertion, search, and deletion operations (average O(1) time complexity).
  - Flexible for storing data with unique identifiers.
  - Useful when mapping relationships are needed, like storing roll numbers with student names.

- **Limitations** –
  - Does not maintain order of elements.
  - Not synchronized, so not thread-safe by default (use `Collections.synchronizedMap()`

Compare & CompareTo

Compare():

- Definition: The compare() method is a static method in the Comparator interface used to compare two objects of the same type.
- Functionality: It returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second object.
- Usage: It is used when a separate class or lambda expression defines custom sorting logic.
- Example: Comparator<Integer> c = Integer::compare; int result = c.compare(5, 10);
-

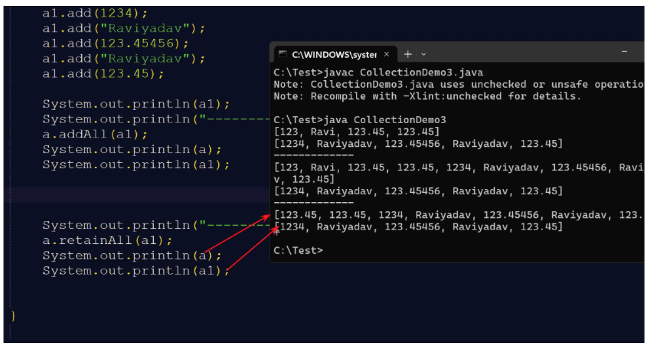
CompareTo():

- **Definition**: The compareTo() method is defined in the Comparable interface and is used to compare the current object with another object.
- **Functionality**: It returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object.
- **Usage**: It is used when the class itself defines a natural ordering.
- **Example**: Integer a = 5; int result = a.compareTo(10);

Comparable & Comparator

Comparable:

- **Definition**:
  `Comparable` is an interface in `java.lang` used to define the natural ordering of objects.
- **Method:** It contains the method `compareTo(Object o)` which compares the current object with the specified object.
- **Usage**:
  It is implemented by the class whose objects need to be compared.
- **Functionality**:
  It returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object.



```
        a1.add(1234);
        a1.add("Raviyadav");
        a1.add(123.45456);
        a1.add("Raviyadav");
        a1.add(123.45);

        System.out.println(a1);
        System.out.println("--------
        a.addAll(a1);
        System.out.println(a);
        System.out.println(a1);


        System.out.println("--------
        a.retainAll(a1);
        System.out.println(a);
        System.out.println(a1);

}
```

```
C:\Test>javac CollectionDemo3.java
Note: CollectionDemo3.java uses unchecked or unsafe operatio
Note: Recompile with -Xlint:unchecked for details.

C:\Test>java CollectionDemo3
[123, Ravi, 123.45, 123.45]
[1234, Raviyadav, 123.45456, Raviyadav, 123.45]
-------------
[123, Ravi, 123.45, 123.45, 1234, Raviyadav, 123.45456, Ravi
v, 123.45]
[1234, Raviyadav, 123.45456, Raviyadav, 123.45]
-------------
[123.45, 123.45, 1234, Raviyadav, 123.45456, Raviyadav, 123.
[1234, Raviyadav, 123.45456, Raviyadav, 123.45]

C:\Test>
```

Comparator:

- Definition:
  `Comparator` is an interface in `java.util` used to define custom ordering of objects externally.
- Methods:
  It contains the method `compare(Object o1, Object o2)` and a default `reversed()` method.
- Usage:
  It is implemented when we want multiple ways to sort objects or cannot modify the class.
- Functionality: It returns a negative integer, zero, or a positive integer based on the comparison of the two objects.