JAVA UNIT 5

1. Explain Java Collection Framework. Explain advantages of Collection Framework

- The Java Collection Framework is a set of classes and interfaces present in the java.util package. It provides a ready-made architecture to store and manipulate groups of objects.
- It allows operations like insertion, deletion, searching, sorting, and manipulation of data. These operations are performed efficiently using predefined data structures.
- Before Java 1.2, data handling was done using arrays and vectors which had many limitations. The Collection Framework was introduced to overcome these problems.

Advantages of Collection Framework:

- It reduces programming effort because developers do not need to write data structures from scratch. Ready-made classes like ArrayList and HashSet are available.
- It increases performance as collections use optimized algorithms internally.
- It provides a unified architecture where common methods like add(), remove(), and size() are used across collections.
- It supports type safety using generics, which avoids runtime errors.
- It improves code readability and maintainability, making programs easier to understand.

2. Explain Collection interface along with all methods of Collection with examples

- The Collection interface is the root interface of the Java Collection Framework. It represents a group of objects known as elements.
- It extends the Iterable interface, so all collections can be traversed using a for-each loop.
- The add(Object o) method adds an element to the collection and returns true if successful. For example, list.add("Java").
- The addAll(Collection c) method adds all elements of one collection to another. It is useful when merging collections.
- The remove(Object o) method removes a specific element from the collection.
- The clear() method removes all elements and makes the collection empty.
- The contains(Object o) method checks whether an element exists in the collection.
- The isEmpty() method returns true if the collection has no elements.
- The size() method returns the total number of elements present.
- The iterator() method returns an iterator object to traverse elements.

3. Explain the role of the Iterator interface. How is it used to traverse collections?

- The Iterator interface is used to traverse elements of a collection one by one. It provides a uniform way to access elements.
- It supports only forward direction traversal. This means elements are accessed from start to end.
- The iterator() method of the Collection interface is used to create an Iterator object.
- The hasNext() method checks whether the next element exists. It returns true if elements are remaining.
- The next() method returns the next element and moves the cursor forward.
- The remove() method removes the current element from the collection.
- Iterator helps avoid errors like ConcurrentModificationException during traversal.
- Example usage includes creating an iterator using Iterator it = list.iterator().
- Iterator is commonly used when removing elements during traversal.
- It improves safe and controlled access to collection elements.

4. Explain List interface along with its methods and example of these methods

- The List interface is a sub-interface of Collection. It stores elements in an ordered sequence.
- It allows duplicate elements and maintains insertion order.
- The add(E e) method adds an element at the end of the list. Example: list.add("A").
- The add(int index, E element) method inserts an element at a specific position.
- The get(int index) method retrieves an element from a given index.
- The remove(int index) method removes an element from a specific index.
- The set(int index, E element) method replaces an element at a given position.
- The indexOf(Object o) method returns the index of the first occurrence of an element.
- The listIterator() method returns a ListIterator for traversal.
- Popular implementations of List are ArrayList, LinkedList, and Vector.

5. Differentiate iterator and listIterator along with example program

- Iterator is used for all Collection implementations. ListIterator is used only for List implementations.
- Iterator supports only forward traversal. ListIterator supports both forward and backward traversal.
- Iterator allows read and remove operations. ListIterator allows create, read, update, and delete operations.
- Iterator does not provide index information. ListIterator provides nextIndex() and previousIndex() methods.
- Iterator is created using collection.iterator(). ListIterator is created using list.listIterator().

Example explanation:

- Using Iterator, elements are accessed using hasNext() and next() in one direction.
- Using ListIterator, elements can be traversed forward using next() and backward using previous().
- ListIterator is preferred when modification during traversal is required.

6. Differentiate ArrayList and LinkedList. Also explain when to use each

- ArrayList uses a dynamic array to store elements, while LinkedList uses a doubly linked list data structure. This affects how elements are stored in memory.
- ArrayList provides fast random access because elements are accessed using index values. LinkedList access is slower because elements must be traversed one by one.
- In ArrayList, insertion and deletion are slow because shifting of elements is required. In LinkedList, insertion and deletion are faster as only node links are updated.
- ArrayList consumes less memory since it stores only data. LinkedList consumes more memory because it stores data along with previous and next references.

When to use ArrayList

- Use ArrayList when frequent read operations are needed. It is best when random access using index is required.

When to use LinkedList

- Use LinkedList when frequent insertions and deletions are required. It is suitable when working as a queue or stack.

Example / Syntax

ArrayList<String> al = new ArrayList<>();

LinkedList<String> ll = new LinkedList<>();

7. Differentiate ArrayList and Traditional Array

- A **traditional array** has a **fixed size**, which means its size must be decided at the time of creation and cannot be changed later. An **ArrayList** has a **dynamic size** and can automatically grow or shrink as elements are added or removed.
- An array can store **both primitive data types** (like int, double) and **objects** directly. ArrayList can store **only objects**, so primitive values are stored using **wrapper classes** through autoboxing.
- In arrays, the **length** keyword is used to find the total size. In ArrayList, the **size()** method is used to get the number of elements currently present.
- Arrays do **not support generics**, so they are less type-safe and may cause runtime errors. ArrayList supports **generics**, which provide compile-time type safety and avoid ClassCastException.
- Elements in an array are added using **direct assignment** with index positions. In ArrayList, elements are added or removed using built-in methods like **add()**, **remove()**, and **clear()**, making operations easier.
- Arrays offer **better performance** because of fixed size and direct memory access. ArrayList is slightly slower due to resizing and dynamic behavior.

8. Advantages and Disadvantages of ArrayList and Array with Usage Scenarios

Advantages of ArrayList

- ArrayList is **dynamic and resizable**, so its size can increase or decrease during program execution. This makes it suitable when the number of elements is not known in advance.
- It provides many **built-in methods** such as add(), remove(), contains(), and clear() which make data manipulation easy. This reduces manual coding effort.

Disadvantages of ArrayList

- ArrayList is **slower than arrays** because resizing and shifting of elements takes extra time. This affects performance in time-critical applications.
- It **cannot store primitive data types directly**. Primitive values must be converted into wrapper objects, which adds memory overhead.

Advantages of Array

- Arrays are **faster** because they have a fixed size and allow direct access to memory locations. This makes them efficient for performance-critical programs.
- Arrays can store **both primitive data types and objects** directly. This avoids the overhead of wrapper classes.

Disadvantages of Array

- Arrays are **not dynamic**, so their size cannot be changed once created. This can lead to memory wastage or shortage.
- **Insertion and deletion** operations require manual shifting of elements. This increases coding complexity.

Usage Scenarios

- Use **ArrayList** when the size is unknown and frequent modifications are required.
- Use **arrays** when the size is fixed and high performance is required.

9. Explain Set interface with methods and example program

- The **Set interface** extends the Collection interface and is used to store **unique elements only**. Duplicate values are not allowed in a Set.
- Set is **not index-based**, so elements cannot be accessed using index positions.
- It allows **only one null value** and follows **mathematical set behavior**.
- The add() method adds an element to the set only if it is not already present.
- The remove() method removes a specified element from the set.
- The contains() method checks whether a particular element exists in the set.
- The size() method returns the total number of elements present in the set.
- Set elements can be traversed using a **for-each loop or Iterator**.

Example Program using all Set methods (HashSet)

```
import java.util.HashSet;

public class SetAllMethodsExample {

  public static void main(String[] args) {


    HashSet<Integer> set = new HashSet<>();

    // add() method

    set.add(10);

    set.add(20);

    set.add(30);

    set.add(10);   // duplicate element, ignored
```

```
    // contains() method
    System.out.println("Contains 20? " + set.contains(20));

    // size() method
    System.out.println("Size of set: " + set.size());

    // remove() method
    set.remove(30);

    System.out.println("Set elements after removal:");
    // Iteration using for-each loop
    for (int i : set) {
      System.out.println(i);
    }
  }}
```

10. Differentiate List and Set with example programs

- **List** allows **duplicate elements**, so the same value can be stored more than once. **Set** does **not allow duplicate elements**, and repeated values are automatically ignored.
- List **maintains insertion order**, which means elements are stored in the order they are added. Set **does not maintain insertion order by default** (except LinkedHashSet).
- List is **index-based**, so elements can be accessed using index numbers like get(0). Set is **non-indexed**, so elements cannot be accessed using positions.
- List allows **multiple null values** to be stored. Set allows **only one null value**.
- List is suitable when **order and duplicates matter**. Set is suitable when **unique values are required**.

Example Program – List (ArrayList)

```
import java.util.ArrayList;

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

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

        list.add("Java");
        list.add("Python");
        list.add("Java");   // duplicate allowed
        list.add(null);
        list.add(null);     // multiple nulls allowed

        System.out.println(list);
        System.out.println("Element at index 1: " + list.get(1));
    }
}
```

- Output stores **duplicate values and multiple nulls**.
- Elements can be accessed using **index values**.

Example Program – Set (HashSet)

```
import java.util.HashSet;

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

        set.add("Java");
        set.add("Python");
        set.add("Java");   // duplicate ignored
        set.add(null);
        set.add(null);     // only one null allowed

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

11. Explain HashSet and explain how elements are stored in HashSet

- **HashSet** is a class that implements the **Set interface**. It is used to store **unique elements only**.
- It does **not allow duplicate values**, and duplicate insertion is ignored automatically.
- HashSet is **not index-based**, so elements cannot be accessed using index numbers.
- It allows **only one null value** in the collection.
- HashSet does **not maintain insertion order**, so elements may appear in any order.
- Internally, HashSet uses a **HashMap** to store elements.
- Elements are stored using a **hashing technique**, where a hash code decides the storage location.
- HashSet provides **constant time performance O(1)** for add, remove, and contains operations.

12. Explain SortedSet interface with methods and example program

- **SortedSet** is an interface that extends the **Set interface**. It stores elements in **sorted (ascending) order**.
- SortedSet does not allow duplicate elements.
- It provides methods to access elements based on their sorted position.

Important Methods of SortedSet

- first() returns the **first (lowest)** element.
- last() returns the **last (highest)** element.
- headSet(E e) returns elements **less than e**.
- tailSet(E e) returns elements **greater than or equal to e**.
- subSet(E e1, E e2) returns elements between e1 and e2.

Example Program (TreeSet – SortedSet implementation)

```
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetExample {
  public static void main(String[] args) {
    SortedSet<String> ss = new TreeSet<>();
```

```
    ss.add("Banana");
    ss.add("Apple");
    ss.add("Mango");

    System.out.println(ss);
    System.out.println("First: " + ss.first());
    System.out.println("Last: " + ss.last());
  }
}
```

13. Explain TreeSet interface with methods and example program. Why TreeSet does not allow null values?

- **TreeSet** is a class that implements the **NavigableSet interface** in Java. It is used to store **unique elements in sorted (ascending) order**.
- TreeSet does **not allow duplicate elements**, and any duplicate value is ignored automatically.
- It provides **fast searching and retrieval** operations because elements are stored in a structured form.
- Internally, TreeSet uses a **Red-Black Tree** data structure, which keeps elements balanced and sorted.
- TreeSet does **not allow null values**, because comparison is required for sorting.

Important Methods of TreeSet

- add() is used to insert elements and they are stored in sorted order.
- first() returns the **smallest element** in the TreeSet.
- last() returns the **largest element** in the TreeSet.
- headSet() returns elements **less than a given value**.
- tailSet() returns elements **greater than or equal to a given value**.

Why TreeSet does not allow null values

- TreeSet uses the **compareTo() method** to compare elements for sorting.
- A null value cannot be compared with other objects.
- Therefore, inserting null causes a **NullPointerException**.

Example Program with TreeSet methods

```
import java.util.TreeSet;

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

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

    // add() method
    ts.add(10);
    ts.add(5);
    ts.add(20);

    System.out.println("TreeSet elements: " + ts);

    // first() and last()
    System.out.println("First element: " + ts.first());
    System.out.println("Last element: " + ts.last());

    // headSet() and tailSet()
    System.out.println("HeadSet (<10): " + ts.headSet(10));
    System.out.println("TailSet (>=10): " + ts.tailSet(10));
  }
}
```

14. Differentiate between HashSet and TreeSet (Simple and Short)

- HashSet stores elements using a hashing technique, while TreeSet stores elements using a tree-based structure. HashSet uses HashMap, and TreeSet uses a Red-Black Tree internally.
- HashSet does not maintain any order of elements. TreeSet maintains elements in sorted (ascending) order.
- HashSet allows one null value in the collection. TreeSet does not allow null values and throws a NullPointerException.
- HashSet provides faster performance for add, remove, and search operations. Its average time complexity is O(1).
- TreeSet operations take O(log n) time because elements are stored in sorted order.
- HashSet uses equals() and hashCode() methods to compare elements. TreeSet uses compareTo() or Comparator for comparison.
- Use HashSet when speed is required and order does not matter. Use TreeSet when sorted unique elements are needed.

15. Java program to store five employee names using ArrayList and iterate using for-each, iterator and listIterator

```
import java.util.ArrayList;

import java.util.Iterator;

import java.util.ListIterator;


public class EmployeeList {

  public static void main(String[] args) {


    ArrayList<String> emp = new ArrayList<>();


    emp.add("Amit");

    emp.add("Ravi");

    emp.add("Neha");

    emp.add("Pooja");

    emp.add("Kiran");


    // For-each loop
```

```
    for(String e : emp) {
      System.out.println(e);
    }
    // Iterator
    Iterator<String> it = emp.iterator();
    while(it.hasNext()) {
      System.out.println(it.next());
    }
    // ListIterator
    ListIterator<String> li = emp.listIterator();
    while(li.hasNext()) {
      System.out.println(li.next());
    }
} }
```

16. Differentiate List and Set interfaces in terms of ordering, indexing, efficiency, duplicate handling, and use cases

- **Ordering:** List maintains the **insertion order** of elements, meaning elements appear in the same order as they are added. Set does **not maintain order** by default, so elements may appear randomly.
- **Indexing:** List is **index-based**, so elements can be accessed using index values like get(0). Set is **non-indexed**, so positional access is not allowed.
- **Duplicate Handling:** List **allows duplicate elements**, so the same value can be stored multiple times. Set **does not allow duplicates**, ensuring uniqueness.
- **Efficiency:** List is efficient when **ordered data and indexing** are required. Set is efficient when **fast searching and uniqueness** are required.
- **Null Values:** List allows **multiple null values**. Set allows **only one null value**.
- **Use Cases:**
    - Use **List** when order matters, duplicates are allowed, and index access is needed (e.g., playlist, steps list).
    - Use **Set** when unique elements are required and order is not important (e.g., unique IDs, usernames).

These differences help developers choose the correct collection based on program requirements.

17. Differentiate between List, Set, and SortedSet

- **List** stores elements in an **ordered sequence** and allows **duplicate values**. Elements can be accessed using index positions.
- **Set** stores **unique elements only** and does **not maintain order** by default. Index-based access is not allowed.
- **SortedSet** extends Set and stores elements in **sorted (ascending) order**. It also does not allow duplicates.
- **Indexing:** List supports indexing, while Set and SortedSet do not support indexing.
- **Sorting:** List does not sort elements automatically. Set is unordered, while SortedSet **always keeps elements sorted**.
- **Null Values:** List allows multiple nulls, Set allows one null, and SortedSet **does not allow null values**.
- **Performance:**
    - List is good for ordered access.
    - Set is fast for searching unique data.
    - SortedSet is slower than Set but provides sorted data.

18. Compare List and Set in terms of indexing, sorting, duplicate elements, ordering, and impact on real-world applications

- **Indexing:** List supports **index-based access**, which makes it easy to retrieve elements by position. Set does not support indexing.
- **Sorting:** List does not automatically sort elements. Set does not guarantee sorting, while some implementations like TreeSet provide sorting.
- **Duplicate Elements:** List allows duplicates, which is useful when repeated data is required. Set removes duplicates automatically.
- **Ordering:** List maintains insertion order, making it suitable for sequence-based data. Set does not maintain order by default.
- **Efficiency Impact:**
    - List is useful when **order and indexing** are important but may be slower for searching duplicates.
    - Set is efficient for **fast lookup and uniqueness**, improving performance in large datasets.
- **Real-World Scenarios:**
    - List is used in playlists, ordered records, and task lists.
    - Set is used for storing unique user IDs, tags, or email lists.

19. WAP to implement LinkedList, add elements, delete elements, retrieve elements, update elements, traverse elements and checking number of elements. (apply operations on it.)

```
import java.util.LinkedList;

import java.util.Iterator;


public class LinkedListOperations {

  public static void main(String[] args) {


    // Creating LinkedList

    LinkedList<String> list = new LinkedList<>();


    // Adding elements

    list.add("Apple");

    list.add("Banana");

    list.add("Mango");

    list.add("Orange");


    System.out.println("Initial LinkedList: " + list);


    // Retrieving elements

    System.out.println("First element: " + list.getFirst());

    System.out.println("Element at index 2: " + list.get(2));


    // Updating elements

    list.set(1, "Grapes");

    System.out.println("After updating element: " + list);


    // Deleting elements

    list.remove("Mango");

    list.removeFirst();

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

```
        // Traversing elements using for-each loop
        System.out.println("Traversing using for-each loop:");
        for (String item : list) {
            System.out.println(item);
        }

        // Traversing elements using Iterator
        System.out.println("Traversing using Iterator:");
        Iterator<String> it = list.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }

        // Checking number of elements
        System.out.println("Number of elements: " + list.size());
    }
}
```

20. WAP to implement HashSet and perform all operations

```
import java.util.HashSet;
import java.util.Iterator;
public class HashSetOperations {
    public static void main(String[] args) {
        // Creating HashSet
        HashSet<String> hs = new HashSet<>();

    // Adding elements
        hs.add("Java");
        hs.add("Python");
```

```java
    hs.add("C++");
    hs.add("Java");   // Duplicate ignored

    System.out.println("Initial HashSet: " + hs);

    // Retrieving elements (checking existence)
    System.out.println("Contains Java? " + hs.contains("Java"));
    System.out.println("Contains Ruby? " + hs.contains("Ruby"));

    // Updating elements
    // HashSet does not support direct update, so remove + add is used
    hs.remove("C++");
    hs.add("JavaScript");
    System.out.println("After update: " + hs);

    // Deleting elements
    hs.remove("Python");
    System.out.println("After deletion: " + hs);

    // Traversing using for-each loop
    System.out.println("Traversing using for-each loop:");
    for (String lang : hs) {
      System.out.println(lang);
    }
    // Traversing using Iterator
    System.out.println("Traversing using Iterator:");
    Iterator<String> it = hs.iterator();
    while (it.hasNext()) {
      System.out.println(it.next());
    }
  } }
```

21. WAP to implement TreeSet and perform all operations

```java
import java.util.TreeSet;
import java.util.Iterator;
public class TreeSetOperations {
  public static void main(String[] args) {

    // Creating TreeSet
    TreeSet<Integer> ts = new TreeSet<>();

    // Adding elements
    ts.add(30);
    ts.add(10);
    ts.add(20);
    ts.add(10);   // Duplicate ignored

    System.out.println("Initial TreeSet: " + ts);

    // Retrieving elements
    System.out.println("First element: " + ts.first());
    System.out.println("Last element: " + ts.last());
    System.out.println("Contains 20? " + ts.contains(20));

    // Updating elements
    // TreeSet does not support direct update, so remove + add is used
    ts.remove(30);
    ts.add(40);
    System.out.println("After update: " + ts);
```

```
        // Deleting elements
        ts.remove(10);
        System.out.println("After deletion: " + ts);

        // Traversing using for-each loop
        System.out.println("Traversing using for-each loop:");
        for (int i : ts) {
            System.out.println(i);
        }

        // Traversing using Iterator
        System.out.println("Traversing using Iterator:");
        Iterator<Integer> it = ts.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```