E-BOOK

UNIT -4

PDF

OOPS
WITH JAVA

written By : Abhay Kumar Singh

Collection in Java

- A collection in Java is a framework that provides an architecture to store and manipulate a group of objects.
- Java Collections can achieve all the operations that you perform on data such as **searching**, **sorting**, **insertion**, **manipulation**, and **deletion**.
- **Key Points:**
  - Collections are used to **store**, **retrieve**, **manipulate**, and communicate aggregate data.
  - Collections can hold both homogeneous and heterogeneous data.
  - They can dynamically grow and shrink in size.

Collection Framework in Java

- The Collection Framework provides a unified architecture for representing and manipulating collections.
- All the collections frameworks contain the following:
  - **Interfaces**: These are abstract data types that represent collections. The interfaces allow collections to be manipulated independently of the details of their representation.

    1.Collection 3.Set 5.Map
    2.List 4.Queue
  - **Implementations**: These are the concrete implementations of the collection interfaces. Essentially, they are reusable data structures.

    1.ArrayList 3. LinkedList 5.HashSet
    2.TreeSet 4.HashMap 6.TreeMap
  - **Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Advantages of Collection Framework

- **Consistent API:** The collection interfaces have a basic set of operations (such as **adding** and **removing**) that are extended by all implementations.
- **Reduces Programming Effort:** By providing useful data structures and algorithms, the Collections Framework reduces the programming effort.

Java Collections Framework Hierarchy

- The Java Collections Framework is structured as a unified architecture for representing and manipulating collections.
- **The hierarchy is broadly divided into three major groups:**
  - **List, Set,** and **Queue**, which extend the **Collection interface**,
  - **Map**, which is a separate hierarchy.
  - **Here are the important points for each component:**
  - **Collection Interface**
    - The root interface of the collections framework.
    - It provides common methods like **add(), remove(), size(), clear(), contains()**, and **iterator()**
  - **Map Interface**
    - Represents a mapping between a **key** and a **value**.
    - Does not extend the Collection interface.
    - Provides methods to **put**, **get**, **remove** elements based on a key.

```
Collection Interface                    Map Interface
 |                                       |
 +---List Interface                      +---HashMap
 |     +---ArrayList                     +---LinkedHashMap
 |     +---LinkedList                    +---TreeMap
 |     +---Vector                        +---Hashtable
 |     |     +---Stack
 |
 +---Set Interface
 |     +---HashSet
 |     +---LinkedHashSet
 |     +---TreeSet
 |
 +---Queue Interface
       +---PriorityQueue
       +---Deque Interface
             +---ArrayDeque
             +---LinkedList Deque
```

Iterator Interface

- The Iterator interface provides a way to access elements of a collection sequentially without exposing the underlying structure.
- It is part of the **java.util package** and is a universal iterator for all collections.
- **Key Points:**
  - **Methods**:
    - **boolean hasNext()**: Returns true if there are more elements to iterate over.
    - **E next()**: Returns the next element in the iteration.
    - **void remove():** Removes the last element returned by the iterator (optional operation).
  - **Usage**:
    - The Iterator interface is used to traverse collections such as **List**, **Set**, and **Map**.
    - It supports both **read** and **remove** operations.

```java
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");

Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
   String element = iterator.next();
   System.out.println(element);
}
```

Collection Interface

- The Collection interface is the root of the collection hierarchy.
- It represents a group of objects known as elements.
- The Collection interface is part of the **java.util package**.
- **Key Points:**
  - **Methods**:
    - **boolean add(E e):** Ensures that this collection contains the specified element.
    - **boolean remove(Object o)**: Removes a single instance of the specified element from this collection.
    - **int size()**: Returns the number of elements in this collection.
    - **boolean isEmpty():** Returns true if this collection contains no elements.
    - **boolean contains(Object o):** Returns true if this collection contains the specified element.
    - **Iterator<E> iterator():** Returns an iterator over the elements in this collection.
    - **boolean addAll(Collection<? extends E> c):** Adds all of the elements in the specified collection to this collection.
    - **void clear()**: Removes all of the elements from this collection.
  - **Subinterfaces**:

    1.List 2.Set 3.Queue
  - **Usage**:
    - The Collection interface provides the base functionality for all collections.

```java
Collection<String> collection = new ArrayList<>();
collection.add("A");
collection.add("B");
collection.add("C");

for (String element : collection) {
   System.out.println(element);
}
```

List Interface

- The List interface extends the Collection interface and represents an ordered collection (also known as a sequence).
- The user can access elements by their integer index (position in the list) and search for elements in the list.
- **Key Points:**
  - **Methods**:
    - **void add(int index, E element):** Inserts the specified element at the specified position in this list.
    - **E get(int index):** Returns the element at the specified position in this list.
    - **E set(int index, E element)**: Replaces the element at the specified position in this list with the specified element.
    - **E remove(int index):** Removes the element at the specified position in this list.
    - **int indexOf(Object o):** Returns the index of the first occurrence of the specified element in this list.
    - **ListIterator<E> listIterator()**: Returns a list iterator over the elements in this list.
  - **Implementations**:
    1.ArrayList 2. LinkedList 3. Vector 4.Stack
  - **Usage**:
    - The List interface allows for ordered collections that can contain duplicate elements.

```
  List<String> list = new ArrayList<>();
 // List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 3); method 1
// method 2
  list.add("A");
  list.add("B");
  list.add("C");

  for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
  }
```

Iterator vs ListIterator

Feature	Iterator	ListIterator
Applicable to	Collection	List
Traversal Direction	Forward only	Both forward and backward
Obtaining Iterator	`collection.iterator()`	`list.listIterator()`
`hasNext()`	Yes	Yes
`next()`	Yes	Yes
`remove()`	Yes	Yes
`hasPrevious()`	No	Yes
`previous()`	No	Yes
`nextIndex()`	No	Yes
`previousIndex()`	No	Yes
`add(E e)`	No	Yes
`set(E e)`	No	Yes

ArrayList

- ArrayList is a resizable array implementation of the List interface.
- It is part of the Java Collections Framework and is found in the **java.util** package.
- ArrayList allows for dynamic arrays that can grow as needed,
- which means it can change its size during runtime.
- **Key Points**:
  - Unlike arrays in Java, ArrayList can grow and shrink in size dynamically.
  - Allows duplicate elements.
  - Provides fast random access to elements.
  - Initial capacity is 10, but it grows automatically as elements are added.
  -

- **Basic Operations**

1.boolean add(e)	6.void add(int index, element)
2. get(int index)	7. set(int index, element)
3. remove(int index)	8.boolean remove(e)
4. int size()	9. void clear()
5. boolean contains(e)	

```
import java.util.*;

public class ArrayListExample {
   public static void main(String[] args) {
      // Creating an ArrayList
      List<String> arrayList = new ArrayList<>();

      // Adding elements
      arrayList.add("A");
      arrayList.add("B");
      arrayList.add("C");
      arrayList.add("D");

      // Accessing elements
      System.out.println("Element at index 2: " + arrayList.get(2));
      // Iterating elements
        for (String element : arrayList) {
           System.out.println(element);
        }
      // Modifying elements
      arrayList.set(1, "E");
      System.out.println("After modification: " + arrayList);
      // Removing elements
      arrayList.remove("C");
      System.out.println("After removal: " + arrayList);
      // Checking size
      System.out.println("Size of ArrayList: " + arrayList.size());
      // Checking if ArrayList contains an element
      System.out.println("Does ArrayList contain 'A'? " +
arrayList.contains("A"));

      // Clearing the ArrayList
      arrayList.clear();
      System.out.println("After clearing: " + arrayList);
   }
}
```

LinkedList

- LinkedList is a doubly linked list implementation of the List interface.
- It is part of the Java Collections Framework and is found in the **java.util** package.
- Unlike ArrayList, which uses a dynamic array, LinkedList uses a doubly linked list to store elements.
- **Key Points:**
- **Doubly Linked**: Each element in a LinkedList is stored in a node that contains a reference to the next and previous elements.
- **Dynamic Size:** Can grow and shrink dynamically.
- Allows duplicate elements.
- Efficient for adding or removing elements anywhere in the list.
- Slower than ArrayList for random access (get()), as it requires traversing from the head or tail.
- **Basic Operations:** (Same as ArrayList)

```
import java.util.*;

public class LinkedListExample {
   public static void main(String[] args) {
      // Create a LinkedList
      LinkedList<String> linkedList = new LinkedList<>();
```

```java
    // Adding elements
    linkedList.add("Apple");
    linkedList.add("Banana");
    linkedList.add("Cherry");
     linkedList.set(1, "Orange");      // Modify element
    linkedList.add(2, "Grape");  // Add an element
    linkedList.remove("Apple");   // Remove element

    // Check if "Banana" is present
    boolean containsBanana = linkedList.contains("Banana");
    System.out.println("Does LinkedList contain 'Banana'? " +
containsBanana);

    System.out.println("Print forward order element ");
    ListIterator<String> iterator = linkedList.listIterator();
    while (iterator.hasNext()) {
      System.out.println( iterator.next());
    }

    System.out.println("Print reverse order element ");
    while (iterator.hasPrevious()) {
      System.out.println( iterator.previous());
    }
  }  }
```

ArrayList(Collection<? extends E> c)

- Creates a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
- It is same For Linked List

```java
List<String> existingList = Arrays.asList("A", "B", "C");
ArrayList<String> list = new ArrayList<>(existingList);
System.out.println(list); // Output: [A, B, C]
```

LinkedList(Collection<? extends E> c)

```java
List<String> arrayList = new ArrayList<>();
    arrayList.add("Apple");
    arrayList.add("Banana");
    arrayList.add("Cherry");

    // Create a LinkedList using the ArrayList elements
    LinkedList<String> linkedList = new LinkedList<>(arrayList);
```

LinkedList vs ArrayList

- LinkedList provides methods like **addFirst, addLast, getFirst, getLast, removeFirst, removeLast, offerFirst, offerLast, peekFirst, peekLast, pollFirst, pollLast, descendingIterator** for efficient **insertion**, **removal**, and **traversal** operations from both ends.
- **ArrayList** is optimized for random access and efficient element **insertion/removal** in the middle.
- **Choosing between them depends on the use case:**
  - Use LinkedList for frequent **insertions/removals** at both ends or sequential traversal.
  - Use ArrayList for scenarios requiring random access or faster access by **index**.

Vector

- Vector is a part of the Java Collections Framework and implements a growable array of objects.
- It is synchronized, making it thread-safe, but can have performance overhead due to synchronization.
- **Methods**:
  - **add(e):** Adds an element to the end of the vector.
  - **get(int index)**: Returns the element at the specified position.
  - **remove(int index)**: Removes the element at the specified position.
  - **size()**: Returns the number of elements in the vector.

```java
import java.util.Vector;

public class VectorExample {
  public static void main(String[] args) {
    Vector<String> vector = new Vector<>();

    vector.add("A");  // Adding elements
    vector.add("B");
    vector.add("C");

    // Accessing elements
    System.out.println("Element at index 1: " + vector.get(1));
    // Removing element
    vector.remove(2);
    // Size of vector
    System.out.println("Size of vector: " + vector.size());
  }
}
```

Stack

- Stack is a subclass of Vector that implements a last-in, first-out (LIFO) stack of objects.
- **Methods**:
  - **push(E item):** Pushes an item onto the top of the stack.
  - **pop():** Removes the object at the top of the stack and returns it.
  - **peek()**: Looks at the object at the top of the stack without removing it.
  - **isEmpty()**: Checks if the stack is empty.

```java
import java.util.Stack;

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

    // Pushing elements
    stack.push("A");
    stack.push("B");
    stack.push("C");

    // Peeking the top element
    System.out.println("Top element: " + stack.peek());
    // Popping elements
    System.out.println("Popped element: " + stack.pop());
    // Checking if stack is empty
    System.out.println("Is stack empty? " + stack.isEmpty());
  }
}
```

Queue Interface

- Queue is a collection designed for holding elements prior to processing.
- It typically orders elements in a **FIFO (first-in, first-out)** manner.
- Implementations include **LinkedList** and **PriorityQueue**.
- **Methods**:
  - **add(e):** Inserts the specified element into the queue (throws an exception if it fails).
  - **offer(e)**: Inserts the specified element into the queue (returns false if it fails).
  - **remove()**: Retrieves and removes the **head** of the queue (throws NoSuchElementException an exception if the queue is empty).
  - **poll()**: Retrieves and removes the **head** of the queue (returns null if the queue is empty).
  - **peek():** Retrieves, but does not remove, the **head** of the queue (returns null if the queue is empty).

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
   public static void main(String[] args) {
      Queue<String> queue = new LinkedList<>();

      // Adding elements
      queue.add("A");
      queue.add("B");
      queue.add("C");

      // Peeking the head element
      System.out.println("Head element: " + queue.peek());

      // Polling elements
      System.out.println("Polled element: " + queue.poll());

      // Checking the size
      System.out.println("Size of queue: " + queue.size());
   }    }
```

Set Interface

- ➔ Represents a collection that cannot contain duplicate elements.
- ➔ It models the mathematical set abstraction.
- ➔ Does not guarantee the order of elements.
- ➔ Allows at most one null element.
- ➔ Implementations include **HashSet**, **LinkedHashSet**, and **TreeSet**.

HashSet

- ➔ Implements the Set interface using a hash table. It does not guarantee the order of elements.
- ➔ Offers constant-time performance for basic operations (**add**, **remove**, **contains**).
- ➔ Does not maintain the insertion order.
- ➔ Allows null elements.

LinkedHashSet

- ➔ Extends HashSet and maintains a doubly-linked list of entries to preserve the insertion order.
- ➔ Iterates over elements in insertion order.
- ➔ Slower than **HashSet** for basic operations due to maintaining order.
- ➔ Allows null elements.

SortedSet Interface

- ➔ Extends Set and maintains elements in sorted order defined by their natural ordering or a comparator.
- ➔ Provides methods for accessing elements by **their position** in the sorted set.
- ➔ Implementations include **TreeSet**.

TreeSet

- ➔ Implements SortedSet using a tree structure **(red-black tree).**
- ➔ Maintains elements in sorted order (ascending by default or based on a custom comparator).
- ➔ Slower performance for basic operations compared to **HashSet** and **LinkedHashSet**.
- ➔ Does not allow null elements.

Example Of All Sets interface

```java
import java.util.*;
public class SetExamples {
   public static void main(String[] args) {
     // HashSet example
     Set<String> hashSet = new HashSet<>();
     hashSet.add("B");
     hashSet.add("A");
     hashSet.add("C");
     System.out.println("HashSet: " + hashSet); // C B  A

     // LinkedHashSet example
     Set<String> linkedHashSet = new LinkedHashSet<>();
     linkedHashSet.add("B");
     linkedHashSet.add("A");
     linkedHashSet.add("C");
     System.out.println("LinkedHashSet: " + linkedHashSet);  //  B A C

     // TreeSet example
     Set<String> treeSet = new TreeSet<>();
     treeSet.add("B");
     treeSet.add("A");
     treeSet.add("C");
     System.out.println("TreeSet: " + treeSet);   // A B C
   }
}
```

""" HashSet: Fastest for basic operations, no guaranteed order.
LinkedHashSet: Maintains insertion order, slower than HashSet.
TreeSet: Maintains elements in sorted order, slower than HashSet and LinkedHashSet. """

Map Interface

- ➔ Represents a collection of key-value pairs where each key is unique.
- ➔ Maps keys to values and does not allow duplicate keys.
- ➔ methods for adding, accessing, removing, and checking for key-value pairs.

HashMap Class

- ➔ Implements the **Map interface** using a **hash table**.
- ➔ Does not guarantee the order of key-value pairs.
- ➔ Provides **constant-time** performance for basic operations (**put, get, remove**) on average.
- ➔ Allows null values and one null key.

```java
import java.util.HashMap;
import java.util.Map;

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

      hashMap.put("One", 1);
      hashMap.put("Two", 2);
      hashMap.put("Three", 3);

      System.out.println("HashMap: " + hashMap);
   }
}
```

LinkedHashMap Class

- ➔ Extends **HashMap** and maintains insertion order of keys.
- ➔ Iterates over elements in the order they were inserted.
- ➔ Slower performance for basic operations compared to HashMap due to maintaining order.
- ➔ Allows null values and one null key.

```java
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapExample {
   public static void main(String[] args) {
      Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

      linkedHashMap.put("One", 1);
      linkedHashMap.put("Two", 2);
      linkedHashMap.put("Three", 3);

      System.out.println("LinkedHashMap: " + linkedHashMap);
   }
}
```

TreeMap Class

- Implements the SortedMap interface using a Red-Black tree.
- Maintains keys in ascending order (natural order or custom Comparator).
- Slower performance for basic operations compared to HashMap and LinkedHashMap due to sorting.
- Does not allow null keys but allows null values.

```
import java.util.Map;
  import java.util.TreeMap;

  public class TreeMapExample {
    public static void main(String[] args) {
      Map<String, Integer> treeMap = new TreeMap<>();


      treeMap.put("Three", 3);
      treeMap.put("One", 1);
      treeMap.put("Two", 2);

      System.out.println("TreeMap: " + treeMap);
    }
  }
```

""" HashMap: Fastest for basic operations, no order guarantee.
LinkedHashMap: Maintains insertion order, inherits from HashMap.
TreeMap: Maintains keys in sorted order, slower for basic operations due to sorting."" "

Hashtable Class

- The Hashtable class in Java provides a basic implementation of a hash table, which maps keys to values.
- It inherits from the Dictionary class and implements the Map interface, making it similar to HashMap but with some differences :
  - **Hashtable is synchronized,**
  - **Neither keys nor values can be null**

```
import java.util.Hashtable;

public class HashtableExample {
  public static void main(String[] args) {
    // Using Hashtable
    Hashtable<String, Integer> hashtable = new Hashtable<>();
    hashtable.put("One", 1);
    hashtable.put("Two", 2);
    // hashtable.put(null, 3); // Throws NullPointerException
    System.out.println("Hashtable: " + hashtable);

  }
}
```

Iterators used in Map and Set

```
import java.util.*;

public class SetMapIterationExample {
  public static void main(String[] args) {
    // Create a HashSet
    Set<String> set = new HashSet<>();
    // Add elements to the Set
    set.add("Apple");
    set.add("Banana");
    set.add("Orange");

     // Iterating over the Set using for-each loop
     System.out.println("Elements in the Set:");
     for (String element : set) {
       System.out.println(element);
     }

    // Create a HashMap
    Map<String, Integer> map = new HashMap<>();

    // Add key-value pairs to the Map
    map.put("One", 1);
    map.put("Two", 2);
    map.put("Three", 3);


 // Iterating over the Map using for-each loop
    System.out.println("\nKey-Value pairs in the Map:");
    for (Map.Entry<String, Integer> entry : map.entrySet()) {
      System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
    }
  }
}
```

Sorting

- Sorting in Java refers to arranging elements in a collection in a specific order, typically ascending or descending based on certain criteria.
- Java provides several ways to achieve sorting depending on the data structure and requirements:
- **Sorting Arrays** ⇒ Arrays.sort()

```
int[] nums = {5, 2, 8, 1, 3};
Arrays.sort(nums); // Sorts nums array in ascending order
```

- **Sorting Collections**

  1.**Collections.sort():**Sorts collections such as lists using natural ordering (if elements implement Comparable) or a specified Comparator.

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
Collections.sort(names); // Sorts alphabetically (natural order)
```

  2.Sorting with Comparator: ascending order use : Comparator.naturalOrder() and descending order use : Comparator.reverseOrder()

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
Collections.sort(names, Comparator.reverseOrder()); // Sorts in
reverse order
```

Comparable Interface

- Comparable is an interface in the java.lang package.
- It declares one method compareTo() which compares the current object (this) with another object of the same type.
- Classes that implement Comparable can be sorted automatically using methods like **Arrays.sort()** or **Collections.sort().**

```

public interface Comparable<T> {
  public int compareTo(T o);
}
```

Example of Comparable InterFace

```
public class Student implements Comparable<Student> {
  private String name;
  private int age;

  // Constructor
  public Student(String name, int age) {
    this.name = name;
    this.age = age;
  }
```

```
    public int compareTo(Student other) {
        // Compare students based on age
        return Integer.compare(this.age, other.age);
    }
    // Example usage in main method
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 20));
        students.add(new Student("Bob", 18));
        students.add(new Student("Charlie", 22));

// Sorting using Collections.sort() (uses Comparable)
        Collections.sort(students);

        // Printing sorted students
        System.out.println("Sorted Students by Age:");
        for (Student student : students) {
            System.out.println(student);
        }
    }
}
```

Comparator Interface

- The Comparator interface in Java is located in the java.util package.
- It defines two methods: **compare(T o1, T o2)** and **equals(Object obj)**.
- You typically create an instance of Comparator either as an anonymous class.
- Comparator is commonly used with sorting methods like **Collections.sort()** for lists or **Arrays.sort()** for arrays to define the order in which elements should be sorted.

Example of Comparator Interface

```
import java.util.*;

public class Student {
    private String name;
    private int age;

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

    int getAge() {return this.age;}

    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 20));
        students.add(new Student("Bob", 18));
        students.add(new Student("Charlie", 22));

        // Using Comparator to sort by age in descending order
        Comparator<Student> ageComparator = new Comparator<Student>() {
            @Override
            public int compare(Student s1, Student s2) {
                return Integer.compare(s2.getAge(), s1.getAge()); // Descending order
            }
        };

        // Sorting students list using ageComparator
        Collections.sort(students, ageComparator);

        // Printing sorted students
        System.out.println("Sorted Students by Age (Descending):");
        for (Student student : students) {
            System.out.println(student);
        }
    }
}
```

Properties Class

- Properties stores key-value pairs where both keys and values are strings.
- It supports loading from and saving to files using **load**() and **store**() methods.
- Default values can be set and queried if a property is not found in the current instance.
- Java system properties (**System.getProperties()**) are accessible through a Properties object.
- Example usage involves setting, saving to file, loading, and accessing properties.
- It provides persistence for application settings like **database connections** and **UI configurations**.
- **methods** :
    - **setProperty(String key, String value):** Sets a key-value pair in the Properties object.
    - **getProperty(String key):** Retrieves the value associated with a specified key.
    - **store(OutputStream out, String comments)**: Saves properties to an output stream, with optional comments.
    - **load(InputStream in)**: Loads properties from an input stream.
    - **stringPropertyNames()**: Returns keys where both the key and value are strings.

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

public class Student {
    public static void main(String[] args) {
        Properties prop = new Properties();

        // Setting properties
        prop.setProperty("database.url", "jdbc:mysql://localhost:3306/mydb");
        prop.setProperty("database.user", "root");
        prop.setProperty("database.password", "password");

        // Saving properties to a file
        try (OutputStream output = new
FileOutputStream("config.properties")) {
            prop.store(output, "Database Configuration");
            System.out.println("Properties saved successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Loading properties from a file
        try (InputStream input = new FileInputStream("config.properties")) {
            prop.load(input);
            System.out.println("Properties loaded successfully.");

            // Display properties
            prop.forEach((key, value) -> {
                System.out.println(key + ": " + value);
            });

            // Accessing individual property
            String dbUrl = prop.getProperty("database.url");
            System.out.println("Database URL: " + dbUrl);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```