Collection Framework[Important only]

Collections Framework

1. What is the Collections Framework?

Concept

The **Collections Framework** in Java is a unified architecture for storing, retrieving, and manipulating collections of data. It consists of:

1. **Interfaces**: Define operations (e.g., `List`, `Set`, `Map`).
2. **Classes**: Implement the interfaces (e.g., `ArrayList`, `HashMap`).
3. **Algorithms**: Provide utility methods for collections (e.g., sorting, searching).

Real-World Example

- A **List** can represent a queue of people.
- A **Map** can store student IDs and their names.

Core Interfaces and Classes

2. List, Set, SortedSet, Queue, Deque, and Map

Concept

- **List**: Ordered collection (e.g., `ArrayList`, `LinkedList`).
- **Set**: Unordered collection of unique elements (e.g., `HashSet`, `TreeSet`).
- **SortedSet**: A `Set` that maintains ascending order (`TreeSet`).
- **Queue**: FIFO (First In, First Out) data structure (e.g., `PriorityQueue`).

- **Deque**: Double-ended queue allowing insertions/removals from both ends (`ArrayDeque`).
- **Map**: Key-value pairs (e.g., `HashMap`, `TreeMap`).

Example

```java
import java.util.*;

public class CoreInterfacesDemo {
    public static void main(String[] args) {
        // List example
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        System.out.println("List: " + names);

        // Set example
        Set<Integer> uniqueNumbers = new HashSet<>();
        uniqueNumbers.add(10);
        uniqueNumbers.add(20);
        uniqueNumbers.add(10); // Duplicate, ignored
        System.out.println("Set: " + uniqueNumbers);

        // Map example
        Map<Integer, String> idToName = new HashMap<>();
        idToName.put(1, "Alice");
        idToName.put(2, "Bob");
        System.out.println("Map: " + idToName);
    }
}
```

---

3. ArrayList and LinkedList

Concept

- **ArrayList**: Dynamic array; fast for access but slower for insertions.
- **LinkedList**: Doubly-linked list; fast for insertions but slower for access.

Example

```
import java.util.*;

public class ListDemo {
    public static void main(String[] args) {
        // ArrayList
        List<String> arrayList = new ArrayList<>();
        arrayList.add("A");
        arrayList.add("B");
        System.out.println("ArrayList: " + arrayList);

        // LinkedList
        List<String> linkedList = new LinkedList<>();
        linkedList.add("X");
        linkedList.add("Y");
        System.out.println("LinkedList: " + linkedList);
    }
}
```

Explanation

- **ArrayList**: Elements are stored in a resizable array.
- **LinkedList**: Each element points to the next and previous elements.

4. HashSet, LinkedHashSet, TreeSet

Concept

- **HashSet**: Unordered, unique elements.
- **LinkedHashSet**: Ordered insertion, unique elements.
- **TreeSet**: Sorted, unique elements.

Example

```
import java.util.*;

public class SetDemo {
    public static void main(String[] args) {
        // HashSet
        Set<String> hashSet = new HashSet<>();
        hashSet.add("A");
        hashSet.add("B");
        hashSet.add("A"); // Duplicate ignored
        System.out.println("HashSet: " + hashSet);

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

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

5. Queue and Deque

Concept

- **Queue**: FIFO data structure.
- **Deque**: Allows operations at both ends.

Example

```java
import java.util.*;

public class QueueDemo {
    public static void main(String[] args) {
        // Queue
        Queue<Integer> queue = new LinkedList<>();
        queue.add(1);
        queue.add(2);
        System.out.println("Queue: " + queue);

        // Deque
        Deque<Integer> deque = new ArrayDeque<>();
        deque.addFirst(10);
        deque.addLast(20);
        System.out.println("Deque: " + deque);
    }
}
```

---

6. Map and Related Classes

Concept

- **HashMap**: Unordered key-value pairs.
- **LinkedHashMap**: Ordered by insertion.

- **TreeMap**: Sorted by keys.

Example

```
import java.util.*;

public class MapDemo {
    public static void main(String[] args) {
        // HashMap
        Map<Integer, String> hashMap = new HashMap<>();
        hashMap.put(1, "A");
        hashMap.put(2, "B");
        System.out.println("HashMap: " + hashMap);

        // TreeMap
        Map<Integer, String> treeMap = new TreeMap<>();
        treeMap.put(2, "B");
        treeMap.put(1, "A");
        System.out.println("TreeMap: " + treeMap); // Sorted
by key
    }
}
```

7. Comparator and RandomAccess Interfaces

Concept

- **Comparator**: Defines custom sorting.
- **RandomAccess**: Marker interface for fast random access in lists.

Example

```
import java.util.*;

public class ComparatorDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Bob", "Alice", "Charlie");

        list.sort((s1, s2) -> s1.length() - s2.length()); // Sort by length
        System.out.println("Sorted by length: " + list);
    }
}
```

8. Abstract Collections

Concept

Abstract collections provide skeletal implementations of collection interfaces (e.g., `AbstractList`, `AbstractSet`).

1. Traversing Collections

Concept

Traversing a collection means iterating through its elements. Java provides multiple ways to traverse collections:

1. **For-each Loop**: Simplest way to iterate over elements.
2. **Iterator**: Provides a generic way to traverse collections.
3. **ListIterator**: A bidirectional iterator for lists.
4. **Enumeration**: Legacy traversal for older classes like `Vector`.
5. **Streams API**: Functional-style traversal introduced in Java 8.

Examples

For-each Loop

```
import java.util.*;

public class ForEachExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

Iterator

```
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");

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

ListIterator

```
import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");

        ListIterator<String> listIterator = names.listIterator();

        // Forward Traversal
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }

        // Backward Traversal
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

Streams API

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

```
public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        names.stream().forEach(name -> System.out.println(name));
    }
}
```

2. Sorting Collections

Concept

Sorting arranges the elements of a collection in a specific order (natural or custom). Java provides:

1. **Natural Sorting**: Uses the natural ordering of elements (e.g., ascending for numbers).
2. **Custom Sorting**: Allows defining custom order using `Comparator`.

Examples

Natural Sorting

`Collections.sort()` is used to sort a list in ascending order by default.

```
import java.util.*;

public class NaturalSortingExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 3, 8, 1);

        Collections.sort(numbers); // Ascending order
        System.out.println("Sorted List: " + numbers);
    }
}
```

Sorting with Comparable

`Comparable` is an interface that allows objects of a class to be compared to one another. It is used to define the **natural order** for custom objects.

```
import java.util.*;

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

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

    @Override
    public int compareTo(Student other) {
        return this.age - other.age; // Ascending by age
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class ComparableExample {
    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));

        Collections.sort(students);
        System.out.println("Sorted by Age: " + students);
    }
}
```

3. Custom Sorting

Concept

Custom sorting is achieved using the `Comparator` interface. This allows defining multiple sorting criteria for a collection.

Example

Sorting students by name in descending order using a `Comparator`.

```
import java.util.*;

class Student {
    String name;
    int age;

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

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class CustomSortingExample {
    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));

        // Custom sorting by name (descending)
        students.sort((s1, s2) -> s2.name.compareTo(s1.name));

        System.out.println("Sorted by Name (Descending): " + students);

        // Custom sorting by age (ascending)
        students.sort(Comparator.comparingInt(s -> s.age));
        System.out.println("Sorted by Age (Ascending): " + students);
    }
}
```

Using Streams for Custom Sorting

With Java 8, the `Stream` API provides an elegant way to sort collections.

Example

```
import java.util.*;
import java.util.stream.Collectors;
```

```
class Student {
    String name;
    int age;

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

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

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

        // Sorting by name
        List<Student> sortedByName = students.stream()
            .sorted(Comparator.comparing(s -> s.name))
            .collect(Collectors.toList());
        System.out.println("Sorted by Name: " + sortedByNam
e);

        // Sorting by age (descending)
        List<Student> sortedByAgeDescending = students.stream
()
            .sorted((s1, s2) -> Integer.compare(s2.age, s1.ag
e))
            .collect(Collectors.toList());
        System.out.println("Sorted by Age (Descending): " + sortedByAgeDescending);
    }
}
```

Collection Framework Interfaces

Interface/Class	Description	Key Features	Implementation Classes
Collection	Root interface for all collection types.	Basic operations: `add`, `remove`, `size`, `isEmpty`, `clear`.	-
List	Ordered collection that allows duplicate elements.	- Indexed access to elements- Allows duplicates- Preserves insertion order	`ArrayList`, `LinkedList`, `Vector`, `Stack`
Set	Collection of unique elements.	- Does not allow duplicates- Unordered (except for `LinkedHashSet` and `TreeSet`)	`HashSet`, `LinkedHashSet`, `TreeSet`, `EnumSet`
SortedSet	A `Set` with sorted order.	- Maintains elements in natural or custom order	`TreeSet`
Queue	FIFO (First In, First Out) collection.	- Used for holding elements before processing- May allow duplicates- Elements processed sequentially	`LinkedList`, `PriorityQueue`, `ArrayDeque`
Deque	Double-ended queue, supports insertion and removal from both ends.	- Can act as a queue or stack- Can hold null elements (except `ArrayDeque`)	`ArrayDeque`, `LinkedList`
Map	Key-value pairs; keys must be unique.	- Allows null keys and values (except `TreeMap`)-	`HashMap`, `LinkedHashMap`,

		Efficient retrieval by key	`TreeMap`
SortedMap	A `Map` with sorted keys.	- Maintains natural or custom order for keys	`TreeMap`
NavigableMap	Extends `SortedMap` with navigation methods.	- Additional methods like `floorKey`, `ceilingKey`, `higherKey`, etc.	`TreeMap`

Important Classes in the Collections Framework

Class	Description	Key Features
ArrayList	Resizable array implementation of `List`.	- Fast random access- Slow insertion/removal in the middle- Allows duplicates
LinkedList	Doubly-linked list implementation of `List` and `Deque`.	- Fast insertion and deletion- Slower random access- Can act as `Queue` or `Deque`
HashSet	Implements `Set` using a hash table.	- Unordered- Allows one null element- Fast lookups
LinkedHashSet	Extends `HashSet` with predictable iteration order.	- Maintains insertion order- Slower than `HashSet`
TreeSet	Implements `SortedSet` using a red-black tree.	- Sorted elements- No null elements
PriorityQueue	Implements `Queue` with priority ordering.	- Not necessarily FIFO- Uses natural or custom ordering
ArrayDeque	Implements `Deque`.	- Resizable array- Fast insertion and deletion- Does not allow null elements
HashMap	Implements `Map` using a hash table.	- Unordered- Allows one null key and multiple null values
LinkedHashMap	Extends `HashMap` with predictable iteration order.	- Maintains insertion order
TreeMap	Implements `NavigableMap` using a red-black tree.	- Sorted keys- Does not allow null keys

IdentityHashMap	Implements `Map` using reference equality instead of `equals()`.	- Keys compared using `==`
WeakHashMap	Implements `Map` with keys that are weak references.	- Keys are garbage-collected when no longer in use
EnumMap	Map with keys restricted to an enumeration type.	- Keys must be `enum` constants- Very efficient
Vector	Synchronized resizable array implementation of `List`.	- Legacy class- Thread-safe
Stack	Extends `Vector` to provide a LIFO (Last In, First Out) stack.	- Legacy class- Methods: `push`, `pop`, `peek`

Key Functional Interfaces

Interface	Description	Key Features
Comparator	Used to define custom sorting for objects.	- Functional interface- Method: `compare()`
Iterable	Base interface for traversing collections.	- Method: `iterator()`
Iterator	Allows forward traversal of a collection.	- Methods: `hasNext()`, `next()`, `remove()`
ListIterator	Bi-directional iterator for `List`.	- Methods: `hasPrevious()`, `previous()`, `add()`
RandomAccess	Marker interface for fast random access in `List` implementations.	- Implemented by `ArrayList` and `Vector`