Java Collection Framework

The Java Collection Framework is a powerful library in Java that provides a set of classes and interfaces for managing and manipulating groups of objects. It simplifies the handling of data structures such as lists, sets, queues, and maps. Here's an overview to help you get started:

Core Concepts

1. **Collection Interface**: The root interface of the framework, representing a group of objects (elements).
2. **Map Interface**: A separate hierarchy to store key-value pairs.

Hierarchy Overview

Interfaces:

1. **Collection Interface**:
    - **List**: Ordered collection (can contain duplicates).
        - Implementations: ArrayList, LinkedList, Vector, Stack
    - **Set**: Unordered collection (no duplicates).
        - Implementations: HashSet, LinkedHashSet, TreeSet
    - **Queue**: Follows FIFO or LIFO order.
        - Implementations: PriorityQueue, Deque (via LinkedList)
2. **Map Interface**:
    - Implementations: HashMap, LinkedHashMap, TreeMap, Hashtable

When to Use

- **ArrayList**: Fast access, rarely modifies the middle.
- **LinkedList**: Frequent insertions/deletions.
- **HashSet**: Fast search, no duplicates.
- **TreeSet**: Sorted unique elements.
- **HashMap**: Key-value pairs with no specific order.
- **TreeMap**: Sorted key-value pairs.

Array List

1. Definition

ArrayList is a resizable array implementation of the List interface in Java. Unlike arrays, it can grow or shrink dynamically.

- **Key Features**:
  - Maintains the insertion order.
  - Allows duplicate elements.
  - Allows random access (via index).
  - Not synchronized (not thread-safe).

2. Initialization

Basic Initialization

```
import java.util.ArrayList;

public class Main {
   public static void main(String[] args) {
      // Create an ArrayList that can store different types
      ArrayList mixedList = new ArrayList<>();

      // Add primitive data types using their wrapper classes
      mixedList.add(10);      // int -> Integer (autoboxing)
      mixedList.add(20.5);    // double -> Double
      mixedList.add('A');     // char -> Character
      mixedList.add(true);    // boolean -> Boolean

      // Print elements
      for (Object element : mixedList) {
         System.out.println("Element: " + element);
      }

      // Access and manipulate elements
      int num = (int) mixedList.get(0);        // Unboxing Integer to int
      double decimal = (double) mixedList.get(1); // Unboxing Double to double
      char letter = (char) mixedList.get(2);     // Unboxing Character to char
      boolean flag = (boolean) mixedList.get(3);  // Unboxing Boolean to boolean

      // Print values
      System.out.println("Integer: " + num);
      System.out.println("Double: " + decimal);
      System.out.println("Character: " + letter);
      System.out.println("Boolean: " + flag);

      ArrayList<String> list = new ArrayList<>(); // Default capacity = 10
      list.add("Apple");
      list.add("Banana");
      list.add("Cherry");

      System.out.println(list); // Output: [Apple, Banana, Cherry]
   }
//iteration using for-each
      for(String fruit: list) {
        System.out.println(fruit);
      } }
```

Initialization with Capacity

```
// Sets initial capacity to 20

ArrayList<Integer> numbers = new ArrayList<>(20);
```

Using Arrays.asList() for Initialization

```
ArrayList<String> cities = new ArrayList<>(List.of("New York", "Paris", "Tokyo"));
System.out.println(cities); // Output: [New York, Paris, Tokyo]
```

3. Type Conversions

Convert ArrayList to Array

```
ArrayList<String> list = new ArrayList<>(List.of("A", "B", "C"));
String[] array = list.toArray(new String[0]); // Converts to array
 System.out.println(Arrays.toString(array)); // Output: [A, B, C]
```

Convert Array to ArrayList

```
String[] array = {"X", "Y", "Z"};
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
System.out.println(list); // Output: [X, Y, Z]
```

Convert ArrayList to LinkedList

```
ArrayList<String> arrayList = new ArrayList<>(List.of("1", "2", "3"));
LinkedList<String> linkedList = new LinkedList<>(arrayList);
System.out.println(linkedList); // Output: [1, 2, 3]
```

4. Common In-Built Methods

add()

```
list.add("Element"); // Adds to the end
list.add(1, "Inserted"); // Adds at index 1
```

get()

```
System.out.println(list.get(0)); // Retrieves element at index 0
```

set()

```
list.set(1, "Updated"); // Updates element at index 1
```

remove()

```
list.remove(0); // Removes element at index 0
list.remove("Cherry"); // Removes by value
```

contains()

System.out.println(list.contains("Apple")); // Checks if "Apple" is in the list

size()

System.out.println(list.size()); // Returns the size of the list

isEmpty()

System.out.println(list.isEmpty()); // Checks if the list is empty

clear()

list.clear(); // Removes all elements

forEach()

list.forEach(System.out::println); // Prints each element

5. Example: Real-World Usage

Scenario 1: Managing a To-Do List

```
import java.util.ArrayList;

public class ToDoList {
   public static void main(String[] args) {
      ArrayList<String> tasks = new ArrayList<>();

      // Add tasks
      tasks.add("Complete Java assignment");
      tasks.add("Buy groceries");
      tasks.add("Pay bills");

      // Update a task
      tasks.set(1, "Buy groceries and fruits");

      // Remove a completed task
      tasks.remove("Pay bills");

      // Display all tasks
      tasks.forEach(System.out::println);
   }
}
```

Scenario 2: Processing Dynamic User Input

```
import java.util.ArrayList;
import java.util.Scanner;

public class UserInput {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      ArrayList<String> users = new ArrayList<>();

      System.out.println("Enter user names (type 'exit' to stop):");

      while (true) {
         String input = scanner.nextLine();
         if (input.equalsIgnoreCase("exit")) break;
         users.add(input);
      }

      System.out.println("Registered Users: " + users);
   }
}
```

Scenario 3: Storing and Sorting Data

```
import java.util.ArrayList;
import java.util.Collections;

public class SortingExample {
   public static void main(String[] args) {
      ArrayList<Integer> numbers = new ArrayList<>();
      numbers.add(5);
      numbers.add(1);
      numbers.add(8);
      numbers.add(3);

      // Sort the list
      Collections.sort(numbers);
      System.out.println("Sorted List: " + numbers);

      // Reverse the list
      Collections.reverse(numbers);
      System.out.println("Reversed List: " + numbers);
   }
}
```

6. Performance Notes

- **Time Complexity**:
  - Access (get/set): O(1)O(1)
  - Insert/Remove (at end): O(1)O(1) amortized
  - Insert/Remove (at index): O(n)O(n)
- **When to Use**:
  - Use ArrayList when:
    - Frequent random access is needed.
    - Insertions and deletions are rare or mostly at the end.

7. Advanced use cases and techniques

1. Bulk Operations

Add, Remove or Retain Multiple Elements

You can add or remove a collection of elements in one go.

- **Add All Elements**

```
ArrayList<String> list1 = new ArrayList<>(List.of("A", "B", "C"));
ArrayList<String> list2 = new ArrayList<>(List.of("D", "E"));

list1.addAll(list2); // Adds all elements from list2 to list1
System.out.println(list1); // Output: [A, B, C, D, E]
```

- **Remove All Matching Elements**

```
ArrayList<String> list = new ArrayList<>(List.of("A", "B", "C", "A"));
list.removeAll(List.of("A")); // Removes all occurrences of "A"
System.out.println(list); // Output: [B, C]
```

- **Retain Only Matching Elements**

```
ArrayList<String> list = new ArrayList<>(List.of("A", "B", "C"));
list.retainAll(List.of("A", "C")); // Retains only "A" and "C"
System.out.println(list); // Output: [A, C]
```

2. Sorting with Custom Logic

You can sort ArrayList elements using a custom comparator.

Example 1:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class CustomSorting {
   public static void main(String[] args) {
      ArrayList<String> names = new ArrayList<>(List.of("Darwin Divya Dinesh", "Dineshkumar", "Divya Dineshkumar"));

      // Sort alphabetically (ascending)
      Collections.sort(names);
      System.out.println("Alphabetical: " + names);

      // Sort by length (descending)
      names.sort(Comparator.comparingInt(String::length).reversed());
      System.out.println("By length: " + names);
   }
}
```

Example 2:

```
import java.util.*;

class Person {
   String name;
   int age;

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

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

```
public class PersonComparator {
   public static void main(String[] args) {
      List<Person> people = Arrays.asList(
         new Person("Dineshkumar", 30),
         new Person("Divya Dineshkumar", 25),
         new Person("Darwin Divya Dinesh", 28)
      );

      // Sort by age using Comparator.comparing
      people.sort(Comparator.comparingInt(person -> person.age)); // Sorting based on age

      System.out.println(people);  // Output: [Divya Dineshkumar (25), Darwin Divya Dinesh (28), Dineshkumar (30)]
   }
}
```

3. Synchronizing an ArrayList

`ArrayList` is not thread-safe by default. Use `Collections.synchronizedList()` to make it thread-safe.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SynchronizedExample {
   public static void main(String[] args) {
      List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>());

      synchronizedList.add("Thread 1");
      synchronizedList.add("Thread 2");

      synchronized (synchronizedList) { // Synchronize during iteration
         for (String s : synchronizedList) {
            System.out.println(s);
         }
      }
   } }
```

4. Using Streams for Complex Operations

Java Streams make it easy to perform complex operations like filtering, mapping, and reducing.

Filtering

```
ArrayList<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 4, 5));
numbers.stream()
      .filter(n -> n % 2 == 0) // Keep only even numbers
      .forEach(System.out::println); // Output: 2, 4
```

Mapping

```
ArrayList<String> names = new ArrayList<>(List.of("Dineshkumar", "Divya Dineshkumar", "Charlie"));
names.stream()
    .map(String::toUpperCase) // Convert to uppercase
    .forEach(System.out::println); // Output: DINESHKUMAR, DIVYA DINESHKUMAR, CHARLIE
```

Reduction

```
ArrayList<Integer> nums = new ArrayList<>(List.of(1, 2, 3));
int sum = nums.stream().reduce(0, Integer::sum); // Sum of all elements
System.out.println(sum); // Output: 6
```

---

5. Custom ArrayList Implementation

Create your custom `ArrayList` by extending the default one. This is useful when you want to add specific behaviors.

```
import java.util.ArrayList;

public class CustomArrayList<E> extends ArrayList<E> {
   @Override
   public boolean add(E e) {
      if (this.contains(e)) {
         System.out.println("Duplicate element: " + e);
         return false;
      }
      return super.add(e);
   }
}
```

Usage

```
CustomArrayList<String> list = new CustomArrayList<>();
list.add("A"); // Adds successfully
list.add("A"); // Prints: Duplicate element: A
```

6. Large Data Handling

Efficiently manage large datasets by tuning ArrayList parameters.

Set Initial Capacity

If you know the approximate size of the data, initialize the ArrayList with a larger capacity to minimize resizing operations.

```
ArrayList<Integer> largeList = new ArrayList<>(1000000); // Preallocate capacity
```

Batch Processing

Divide large datasets into smaller chunks for processing.

```
ArrayList<Integer> numbers = new ArrayList<>();
for (int i = 1; i <= 100; i++) {
    numbers.add(i);
}

int batchSize = 10;
for (int i = 0; i < numbers.size(); i += batchSize) {
    List<Integer> batch = numbers.subList(i, Math.min(i + batchSize, numbers.size()));
    System.out.println("Processing batch: " + batch);
}
```

7. Memory Optimization

Trimming Excess Capacity

Use trimToSize() to reduce memory usage if the ArrayList capacity is much larger than the number of elements.

```
ArrayList<Integer> list = new ArrayList<>(100);
list.add(1);
list.add(2);

list.trimToSize(); // Adjusts capacity to fit the current size
```

8. Real-Time Use Case: Employee Management System

Scenario

Store and manage employee data, including operations like searching, sorting, and filtering.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class Employee {
   int id;
   String name;
   double salary;

   public Employee(int id, String name, double salary) {
      this.id = id;
      this.name = name;
      this.salary = salary;
   }

   @Override
   public String toString() {
      return "Employee{" + "id=" + id + ", name='" + name + '\'' + ", salary=" + salary + '}';
   }
}

public class EmployeeManagement {
   public static void main(String[] args) {
      ArrayList<Employee> employees = new ArrayList<>();

      employees.add(new Employee(101, "Dineshkumar", 60000));
      employees.add(new Employee(102, "Divya Dineshkumar", 50000));
      employees.add(new Employee(103, "Charlie", 70000));

      // Sort by salary
      employees.sort(Comparator.comparingDouble(emp -> emp.salary));
      System.out.println("Sorted by salary: " + employees);

      // Filter employees earning above 55,000
      employees.stream()
            .filter(emp -> emp.salary > 55000)
            .forEach(System.out::println);
   }
}
```

LinkedList

1. Definition

LinkedList is a part of the Java Collection Framework and is implemented by the java.util.LinkedList class. It is a doubly linked list, meaning each element (node) points to both its previous and next element. LinkedList implements the List, Deque, and Queue interfaces, which means it can function as a dynamic array, a queue, or a stack. It provides constant-time insertion and removal of elements from both ends, making it ideal for scenarios where such operations are frequent.

- **Key Characteristics:**
  - **Dynamic Size**: The size of a LinkedList is not fixed and can grow or shrink dynamically.
  - **Efficient Insertion and Deletion**: Insertion and deletion operations are faster compared to ArrayList when done at the beginning or middle of the list.
  - **Memory Overhead**: Each element in a LinkedList requires extra memory to store references to the next and previous elements.

2. Initialization

You can initialize a LinkedList in several ways. Here are the common approaches:

Empty LinkedList Initialization:

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

- This creates an empty LinkedList that can later have elements added to it.

Initialization with Elements:

```
LinkedList<String> list = new LinkedList<>(List.of("Apple", "Banana", "Cherry"));
```

- This initializes a LinkedList with a pre-defined list of elements.

Using Constructor with Another Collection:

```
List<String> otherList = new ArrayList<>(List.of("Red", "Green", "Blue"));
LinkedList<String> list = new LinkedList<>(otherList);
```

- This initializes a LinkedList from another collection (like ArrayList).

3. Type Conversion

You can convert between LinkedList and arrays or other collections in Java.

Converting LinkedList to an Array:

```
LinkedList<String> list = new LinkedList<>(List.of("A", "B", "C"));
String[] array = list.toArray(new String[0]);
System.out.println(Arrays.toString(array));  // Output: [A, B, C]
```

- toArray() converts the LinkedList to an array of the same type.

Converting LinkedList to ArrayList:

```
LinkedList<String> list = new LinkedList<>(List.of("Apple", "Banana"));
ArrayList<String> arrayList = new ArrayList<>(list);
```

- You can convert a LinkedList to an ArrayList by passing it to the ArrayList constructor.

Converting Array to LinkedList:

```
String[] array = {"X", "Y", "Z"};
LinkedList<String> list = new LinkedList<>(Arrays.asList(array));
```

- Arrays.asList() can be used to convert an array to a List, and then you can initialize a LinkedList with that list.

4. Common In-Built Methods

Adding Elements:

- add(E e): Adds the element at the end.
- addFirst(E e): Adds the element at the beginning.
- addLast(E e): Adds the element at the end (same as add()).
- offerFirst(E e): Adds the element at the front (similar to addFirst()).

Example:

```
list.add("Item");  // Adds element at the end
list.addFirst("Start");  // Adds element at the beginning
list.addLast("End");  // Adds element at the end (same as add())
```

Removing Elements:

- removeFirst(): Removes the first element.
- removeLast(): Removes the last element.
- remove(): Removes the first occurrence of a specified element.
- pollFirst(): Removes the first element and returns it (null if empty).
- pollLast(): Removes the last element and returns it (null if empty).

Example:

```
list.remove("Item");  // Removes the first occurrence of the element
list.removeFirst();  // Removes the first element
list.removeLast();  // Removes the last element
```

Accessing Elements:

```
list.get(0);  // Retrieves the element at the specified index
list.peek();  // Retrieves the first element without removal (returns null if empty)
list.peekFirst();  // Retrieves the first element without removal
list.peekLast();  // Retrieves the last element without removal
```

Other Useful Methods:

```
list.size();  // Returns the number of elements in the LinkedList
list.contains("Item");  // Checks if the list contains the element
list.isEmpty();  // Checks if the list is empty
```

5. Real-World Usage (Include Sorting)

In real-world applications, LinkedList is used in scenarios where dynamic data structures are needed, particularly when frequent insertions and deletions are required.

Use Case: Implementing a Queue (FIFO)

```
LinkedList<String> queue = new LinkedList<>();
queue.add("Task 1");
queue.add("Task 2");
queue.add("Task 3");
System.out.println(queue.poll());  // Output: Task 1 (removes the first element)
```

- **Scenario**: A LinkedList can be used to implement a queue in a task scheduler.

Use Case: Sorting (using Comparator)

```
LinkedList<String> list = new LinkedList<>(List.of("Banana", "Apple", "Cherry"));
list.sort(Comparator.comparing(String::length));  // Sorts by length
System.out.println(list);  // Output: [Apple, Banana, Cherry]
```

- **Scenario**: Sorting LinkedList based on certain criteria like length, alphabetically, or custom logic.

Use Case: Implementing a Stack (LIFO)

```
LinkedList<String> stack = new LinkedList<>();
stack.push("Task 1");
stack.push("Task 2");
System.out.println(stack.pop());  // Output: Task 2 (removes the last element)
```

- **Scenario**: A LinkedList can be used as a stack for depth-first traversal in algorithms.

6. Advanced Use Cases and Techniques

Multi-threading and Synchronization

- A LinkedList can be used in multi-threaded environments when synchronized access is needed for operations like adding and removing elements. Use Collections.synchronizedList() for thread safety.

```
List<String> synchronizedList = Collections.synchronizedList(new LinkedList<>());
```

Reverse Traversal

- You can traverse the list in reverse order using ListIterator.

```
ListIterator<String> iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
   System.out.println(iterator.previous());
}
```

Splitting a LinkedList

- In certain scenarios, you may need to split a LinkedList into two parts.

```
LinkedList<String> subList = new LinkedList<>(list.subList(0, 3));  // Creates a sublist from the first 3 elements
```

LinkedList as a Deque for Efficient Operations

- LinkedList can be used as a deque (double-ended queue), allowing for efficient insertions and deletions at both ends.

```
LinkedList<String> deque = new LinkedList<>();
deque.addFirst("Start");
deque.addLast("End");
deque.removeFirst();
deque.removeLast();
```

7. Real-World Usage with Advanced Use Cases

Real-World Example 1: Task Scheduler (Queue and Stack)

```
class TaskScheduler {
   private LinkedList<String> tasks = new LinkedList<>();

   // Add task
   public void addTask(String task) {
      tasks.addLast(task);  // Add task to the end of the queue
   }

   // Complete task
   public String completeTask() {
      return tasks.pollFirst();  // Complete the first task (FIFO)
   } }
```

Real-World Example 2: Undo/Redo Functionality (Stack)

```
class UndoRedo {
   private LinkedList<String> history = new LinkedList<>();
   private LinkedList<String> redoStack = new LinkedList<>();

   // Save a state
   public void saveState(String state) {
      history.push(state);  // Push to the stack for undo
      redoStack.clear();  // Clear redo stack after a new action
   }

   // Undo the last action
   public String undo() {
      if (!history.isEmpty()) {
         String state = history.pop();
         redoStack.push(state);
         return state;
      }
      return null;  // No action to undo
   }

   // Redo the last undone action
   public String redo() {
      if (!redoStack.isEmpty()) {
         String state = redoStack.pop();
         history.push(state);
         return state;
      }
      return null;  // No action to redo
   }
}
```

Stack

1. Definition

A **Stack** is a collection that follows the **LIFO (Last In, First Out)** principle, where elements are added and removed from the same end (the top of the stack). It is conceptually similar to a real-world stack, such as a stack of plates, where the last plate placed on the stack is the first one to be removed.

In Java, the `Stack` class is a part of the `java.util` package, and it extends `Vector`. The class implements the `List` interface and provides methods to perform stack operations like pushing, popping, and peeking elements.

Key Characteristics:

- **LIFO Principle**: The last element added is the first to be removed.
- **Operations**: The primary operations are `push()`, `pop()`, `peek()`, and `empty()`.
- **Thread Safety**: The `Stack` class is synchronized, meaning it is thread-safe by default. However, `Deque` (implemented by `LinkedList`) is a more efficient alternative for stack operations.

---

2. Initialization

You can initialize a `Stack` in the following ways:

Default Initialization:

```
Stack<String> stack = new Stack<>();
```

- Creates an empty stack.

Initialization with Elements (Optional):

```
Stack<String> stack = new Stack<>();
stack.push("A");
stack.push("B");
stack.push("C");
```

- Adds elements to the stack after creation.

Alternative with `Deque` (Preferred for performance):

```
Deque<String> stack = new ArrayDeque<>();
```

- `ArrayDeque` is generally recommended over `Stack` in modern Java due to better performance and more flexible functionality.

---

3. Type Conversion

You can convert a stack to other collections or arrays, or convert an array into a stack.

Convert Stack to an Array:

```
Stack<String> stack = new Stack<>();
stack.push("A");
stack.push("B");

String[] array = stack.toArray(new String[0]);
System.out.println(Arrays.toString(array));  // Output: [A, B]
```

Convert a Stack to a List:

```
Stack<String> stack = new Stack<>();
stack.push("Apple");
stack.push("Banana");

List<String> list = new ArrayList<>(stack);
System.out.println(list);  // Output: [Apple, Banana]
```

Convert Array to Stack:

```
String[] array = {"X", "Y", "Z"};
Stack<String> stack = new Stack<>();
for (String element : array) {
   stack.push(element);
}
```

4. Common In-Built Methods

`Stack` comes with several built-in methods for stack operations:

Push (Adding an Element):

```
stack.push("Item");
```

- Adds an element to the top of the stack.

Pop (Removing an Element):

```
String item = stack.pop();
```

- Removes and returns the top element from the stack. Throws `EmptyStackException` if the stack is empty.

Peek (Viewing the Top Element):

```
String topItem = stack.peek();
```

- Returns the top element without removing it. Throws `EmptyStackException` if the stack is empty.

Empty (Checking if the Stack is Empty):

```
boolean isEmpty = stack.empty();
```

- Returns `true` if the stack is empty, otherwise returns `false`.

Search (Finding the Position of an Element):

```
int position = stack.search("Item");
```

- Returns the 1-based position of the element from the top of the stack. Returns `-1` if the element is not found.

---

5. Real-World Usage (Include Sorting)

Stacks are useful in many real-world applications, such as undo mechanisms, expression evaluation, and depth-first search in graphs.

Use Case 1: Undo/Redo Mechanism

A stack can be used to store actions, allowing you to "undo" the last operation.

```
class UndoManager {
   private Stack<String> actions = new Stack<>();

   public void addAction(String action) {
      actions.push(action);
   }

   public String undo() {
      if (!actions.isEmpty()) {
         return actions.pop();
      }
      return "No actions to undo";
   }
}
```

- **Scenario**: When a user performs actions (e.g., typing in a text editor), those actions can be pushed onto a stack, and when they click "undo", the most recent action is popped off the stack.

Use Case 2: Expression Evaluation (Parentheses Matching)

Stacks are commonly used to check balanced parentheses or evaluate expressions.

```
import java.util.*;

public class ParenthesesMatcher {
   public static boolean isBalanced(String expression) {
      Stack<Character> stack = new Stack<>();
```

```
        for (char ch : expression.toCharArray()) {
            if (ch == '(') {
                stack.push(ch);
            } else if (ch == ')') {
                if (stack.isEmpty()) return false;
                stack.pop();
            }
        }
        return stack.isEmpty();
    }

    public static void main(String[] args) {
        System.out.println(isBalanced("(A + B) * (C + D)"));  // Output: true
        System.out.println(isBalanced("((A + B) * (C + D)")); // Output: false
    }
}
```

- **Scenario**: A stack is used to ensure that each opening parenthesis has a matching closing parenthesis.

---

6. Advanced Use Cases and Techniques

Use Case 1: Depth-First Search (DFS) in Graphs

In DFS, a stack is used to explore nodes. You can implement DFS iteratively using a stack to track the nodes to visit.

```
import java.util.*;

class Graph {
    private Map<Integer, List<Integer>> adjacencyList;

    public Graph() {
        adjacencyList = new HashMap<>();
    }

    public void addEdge(int source, int destination) {
        adjacencyList.putIfAbsent(source, new ArrayList<>());
        adjacencyList.get(source).add(destination);
    }

    public void dfs(int start) {
        Set<Integer> visited = new HashSet<>();
        Stack<Integer> stack = new Stack<>();
        stack.push(start);

        while (!stack.isEmpty()) {
            int node = stack.pop();
            if (!visited.contains(node)) {
                visited.add(node);
                System.out.print(node + " ");
                for (int neighbor : adjacencyList.get(node)) {
```

```
                stack.push(neighbor);
            }
        }
    }
  }
}
```

- **Scenario**: DFS explores nodes in a graph by pushing nodes onto a stack and visiting them in a LIFO order.

Use Case 2: Evaluating Postfix Expressions

Postfix notation is often used in calculators, where operands are followed by operators.

```
import java.util.*;

public class PostfixEvaluator {
  public static int evaluate(String expression) {
    Stack<Integer> stack = new Stack<>();
    for (String token : expression.split(" ")) {
      if (token.matches("-?\\d+")) {
        stack.push(Integer.parseInt(token));
      } else {
        int operand2 = stack.pop();
        int operand1 = stack.pop();
        switch (token) {
          case "+":
            stack.push(operand1 + operand2);
            break;
          case "-":
            stack.push(operand1 - operand2);
            break;
          case "*":
            stack.push(operand1 * operand2);
            break;
          case "/":
            stack.push(operand1 / operand2);
            break;
        }
      }
    }
    return stack.pop();
  }

  public static void main(String[] args) {
    String expression = "3 4 + 2 * 7 /";
    System.out.println(evaluate(expression));  // Output: 2
  }
}
```

- **Scenario**: Postfix expressions can be evaluated using a stack, where operands are pushed onto the stack and operators pop operands to perform calculations.

7. Real-World Usage with Advanced Use Cases

Real-World Example 1: Browser Navigation (Back/Forward Stack)

A browser's back and forward functionality can be implemented using two stacks.

```
import java.util.*;

class BrowserHistory {
   private Stack<String> backStack = new Stack<>();
   private Stack<String> forwardStack = new Stack<>();

   public void visit(String url) {
      backStack.push(url);
      forwardStack.clear();  // Clear forward history when a new page is visited
   }

   public String goBack() {
      if (!backStack.isEmpty()) {
         forwardStack.push(backStack.pop());
         return backStack.peek();
      }
      return "No more pages to go back to";
   }

   public String goForward() {
      if (!forwardStack.isEmpty()) {
         backStack.push(forwardStack.pop());
         return backStack.peek();
      }
      return "No more pages to go forward to";
   }
}
```

- **Scenario**: This simulates browser navigation using two stacks: one for the back history and one for the forward history.

Queue

1. Definition

A **Queue** is a collection that follows the **FIFO (First In, First Out)** principle, where elements are added to the rear and removed from the front. It is like a real-world queue (e.g., a line at a store), where the first person to join the line is the first one to be served.

In Java, the `Queue` interface is part of the `java.util` package, and it provides various implementations such as `LinkedList`, `PriorityQueue`, and `ArrayDeque`. The `Queue` interface extends the `Collection` interface and provides methods for inserting, removing, and inspecting elements.

Key Characteristics:

- **FIFO Principle**: The first element added is the first to be removed.
- **Queue Operations**: Key operations are `offer()`, `poll()`, `peek()`, and `remove()`.
- **Thread Safety**: `Queue` itself is not thread-safe, but some implementations like `ConcurrentLinkedQueue` provide thread-safe operations.

2. Initialization

You can initialize a queue in several ways, using different classes that implement the `Queue` interface:

Using LinkedList:

```
Queue<String> queue = new LinkedList<>();
```

- `LinkedList` is the most common implementation of the `Queue` interface. It provides both FIFO and other useful features such as dynamic resizing.

Using ArrayDeque (Recommended):

```
Queue<String> queue = new ArrayDeque<>();
```

- `ArrayDeque` is preferred over `LinkedList` because it offers better performance for most queue operations and is non-synchronized.

Using PriorityQueue (for priority ordering):

```
Queue<Integer> queue = new PriorityQueue<>();
```

- `PriorityQueue` implements the `Queue` interface, but it orders elements based on their natural ordering or a custom comparator, not by the insertion order.

3. Type Conversion

You can convert between Queue and other collections or arrays in Java.

Converting Queue to an Array:

```
Queue<String> queue = new LinkedList<>(List.of("A", "B", "C"));
String[] array = queue.toArray(new String[0]);
System.out.println(Arrays.toString(array));  // Output: [A, B, C]
```

- toArray() converts the queue into an array of the same type.

Converting Queue to List:

```
Queue<String> queue = new LinkedList<>(List.of("Apple", "Banana"));
List<String> list = new ArrayList<>(queue);
System.out.println(list);  // Output: [Apple, Banana]
```

- You can convert a queue into a list by passing it to the ArrayList constructor.

Convert Array to Queue:

```
String[] array = {"X", "Y", "Z"};
Queue<String> queue = new LinkedList<>(Arrays.asList(array));
```

- You can convert an array to a queue by using Arrays.asList() and then initializing the queue with the list.

4. Common In-Built Methods

Here are some common methods of the Queue interface and its implementations:

Offer (Adding an Element):

```
queue.offer("Item");
```

- Adds an element to the queue. Returns true if successful, or false if the queue is full (in case of bounded queues).

Poll (Removing an Element):

```
String item = queue.poll();
```

- Removes and returns the front element of the queue. Returns null if the queue is empty.

Peek (Viewing the Front Element):

```
String item = queue.peek();
```

- Retrieves, but does not remove, the front element of the queue. Returns null if the queue is empty.

Remove (Removing an Element):

```
String item = queue.remove();
```

- Removes and returns the front element of the queue. Throws `NoSuchElementException` if the queue is empty.

Size (Checking the Queue Size):

```
int size = queue.size();
```

- Returns the number of elements in the queue.

5. Real-World Usage (Include Sorting)

Queues are commonly used in real-world scenarios such as task scheduling, message processing, and managing resources.

Use Case 1: Task Scheduling (FIFO)

```
Queue<String> taskQueue = new LinkedList<>();
taskQueue.offer("Task 1");
taskQueue.offer("Task 2");

while (!taskQueue.isEmpty()) {
   String task = taskQueue.poll();  // Process tasks in FIFO order
   System.out.println("Processing " + task);
}
```

- **Scenario**: In task scheduling systems, a queue can be used to manage tasks that need to be processed in the order they arrive.

Use Case 2: Print Queue

```
Queue<String> printQueue = new LinkedList<>();
printQueue.offer("Document 1");
printQueue.offer("Document 2");

while (!printQueue.isEmpty()) {
   String document = printQueue.poll();
   System.out.println("Printing " + document);  // Print documents in order
}
```

- **Scenario**: A print spooler can use a queue to manage documents that need to be printed in the order they were submitted.

Use Case 3: Priority Queue (Sorting by Priority)

```
Queue<String> priorityQueue = new PriorityQueue<>(Comparator.reverseOrder());
priorityQueue.offer("Low Priority");
priorityQueue.offer("High Priority");
priorityQueue.offer("Medium Priority");

while (!priorityQueue.isEmpty()) {
   System.out.println(priorityQueue.poll());  // Output will be sorted: High > Medium > Low
}
```

- **Scenario**: A priority queue is used when elements need to be processed based on priority rather than their arrival order. Elements with higher priority are dequeued first.

6. Advanced Use Cases and Techniques

Use Case 1: Multithreading (Concurrent Queue)

In multi-threaded applications, you may need thread-safe queues to manage tasks. Java provides `ConcurrentLinkedQueue` for such scenarios.

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

Queue<String> concurrentQueue = new ConcurrentLinkedQueue<>();
concurrentQueue.offer("Task 1");
concurrentQueue.offer("Task 2");

System.out.println(concurrentQueue.poll());  // Output: Task 1 (thread-safe)
```

- **Scenario**: In a producer-consumer model, where multiple threads add to and remove from a queue, thread-safe implementations are necessary to avoid race conditions.

Use Case 2: Implementing a Circular Queue

A circular queue can be implemented using a Queue and an array. It helps in scenarios where a fixed-size buffer is required.

```
class CircularQueue {
   private int[] queue;
   private int front, rear, size, capacity;

   public CircularQueue(int capacity) {
      this.capacity = capacity;
      queue = new int[capacity];
      front = rear = size = 0;
   }

   public void enqueue(int item) {
      if (size == capacity) {
         System.out.println("Queue is full");
         return;
      }
      queue[rear] = item;
      rear = (rear + 1) % capacity;
      size++;
   }

   public int dequeue() {
      if (size == 0) {
         System.out.println("Queue is empty");
         return -1;
      }
      int item = queue[front];
      front = (front + 1) % capacity;
      size--;
      return item;
   }
}
```

- **Scenario**: A circular queue is useful in applications like network buffers and memory management, where the space is reused in a circular manner.

7. Real-World Usage with Advanced Use Cases

Real-World Example 1: Message Queue in Distributed Systems

A message queue is often used in distributed systems for decoupling services. It allows one service to produce messages that can be consumed by another service.

```
Queue<String> messageQueue = new LinkedList<>();
messageQueue.offer("Message 1");
messageQueue.offer("Message 2");

// Consumer consumes messages from the queue
while (!messageQueue.isEmpty()) {
   System.out.println("Processing: " + messageQueue.poll());
}
```

- **Scenario**: In microservices architecture, a message queue helps in ensuring that messages are processed in the order they arrive, enabling asynchronous communication between services.

Real-World Example 2: Ticketing System

A queue can be used to simulate a ticketing system where customers are served in the order of their arrival.

```
Queue<String> ticketQueue = new LinkedList<>();
ticketQueue.offer("Customer 1");
ticketQueue.offer("Customer 2");

while (!ticketQueue.isEmpty()) {
   String customer = ticketQueue.poll();
   System.out.println("Serving " + customer);
}
```

- **Scenario**: In a customer service center, tickets are processed in a FIFO order, ensuring fairness.

PriorityQueue

1. Definition

A **PriorityQueue** in Java is a type of queue where elements are ordered based on their **natural ordering** (as defined by Comparable) or a custom **Comparator** provided during the creation of the queue. Unlike a normal queue (FIFO), elements with the highest priority are dequeued first.

Key Characteristics:

- Implements the Queue interface.
- Maintains a **heap-based priority structure** internally (a min-heap by default).
- Does not allow null elements.
- Duplicate elements are allowed.

2. Initialization

You can initialize a PriorityQueue with or without a custom comparator.

Default Initialization (Natural Ordering):

```
PriorityQueue<Integer> queue = new PriorityQueue<>();
```

- In this case, elements are ordered in their **natural ascending order** (e.g., for integers, smallest to largest).

Initialization with a Comparator:

```
PriorityQueue<Integer> queue = new PriorityQueue<>(Comparator.reverseOrder());
```

- The custom comparator orders elements in descending order (or any other order you define).

Initializing with Initial Capacity:

```
PriorityQueue<Integer> queue = new PriorityQueue<>(50); // Capacity 50
```

- Specifies an initial capacity for the queue.

Initialization Using Collection:

```
List<Integer> numbers = List.of(10, 5, 20);
PriorityQueue<Integer> queue = new PriorityQueue<>(numbers);
```

- Initializes the priority queue with elements from a collection.

3. Type Conversion

Convert PriorityQueue to Array:

```
PriorityQueue<Integer> queue = new PriorityQueue<>(List.of(10, 20, 5));
Integer[] array = queue.toArray(new Integer[0]);
System.out.println(Arrays.toString(array)); // Output: [5, 20, 10] (heap structure, not sorted)
```

Convert PriorityQueue to List:

```
List<Integer> list = new ArrayList<>(queue);
System.out.println(list); // Output: [5, 20, 10]
```

4. Common In-Built Methods

Here are some key methods provided by the PriorityQueue class:

Method	Description
offer(E e)	Inserts the specified element into the priority queue.
poll()	Retrieves and removes the head of the queue (highest-priority element). Returns null if empty.
peek()	Retrieves the head of the queue without removing it. Returns null if empty.
remove(Object o)	Removes a single instance of the specified element, if present.
contains(Object o)	Checks if the queue contains the specified element.
toArray()	Converts the priority queue into an array.
size()	Returns the number of elements in the queue.
isEmpty()	Checks if the queue is empty.

5. Real-World Usage (Include Sorting)

Use Case 1: Sorting Elements

A PriorityQueue can be used to sort elements in ascending or descending order.

```
PriorityQueue<Integer> queue = new PriorityQueue<>(List.of(20, 10, 5, 30));
while (!queue.isEmpty()) {
   System.out.print(queue.poll() + " "); // Output: 5 10 20 30
}
```

Use Case 2: Descending Order Sorting

```
PriorityQueue<Integer> queue = new PriorityQueue<>(Comparator.reverseOrder());
queue.addAll(List.of(20, 10, 5, 30));

while (!queue.isEmpty()) {
   System.out.print(queue.poll() + " "); // Output: 30 20 10 5
}
```

6. Advanced Use Cases and Techniques

Use Case 1: Custom Objects with Priority

You can prioritize custom objects using a Comparator.

```
class Task {
   String name;
   int priority;

   public Task(String name, int priority) {
      this.name = name;
      this.priority = priority;
   }

   @Override
   public String toString() {
      return name + " (Priority: " + priority + ")";
   }
}

PriorityQueue<Task> taskQueue = new PriorityQueue<>(Comparator.comparingInt(task -> task.priority));

taskQueue.offer(new Task("Write report", 2));
taskQueue.offer(new Task("Fix bugs", 1));
taskQueue.offer(new Task("Code review", 3));

while (!taskQueue.isEmpty()) {
   System.out.println(taskQueue.poll()); // Output: Fix bugs, Write report, Code review
}
```

- **Scenario**: In task scheduling, higher-priority tasks are executed before lower-priority ones.

Use Case 2: Merging Sorted Arrays

You can use a PriorityQueue to merge multiple sorted arrays.

```
int[][] arrays = {
   {1, 5, 9},
   {2, 6, 8},
   {3, 4, 7}
};

PriorityQueue<int[]> queue = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
for (int[] array : arrays) {
   queue.offer(new int[]{array[0], 0, array}); // Element, Index, Array
}

List<Integer> result = new ArrayList<>();
while (!queue.isEmpty()) {
   int[] current = queue.poll();
   result.add(current[0]);
   if (current[1] + 1 < current[2].length) {
```

```
        queue.offer(new int[]{current[2][current[1] + 1], current[1] + 1, current[2]});
    }
}

System.out.println(result); // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Scenario**: Merging k sorted arrays is common in search engines and distributed data systems.

Use Case 3: Top K Elements

A PriorityQueue can efficiently find the top k largest or smallest elements in a collection.

```
int[] nums = {3, 2, 1, 5, 6, 4};
int k = 3;
PriorityQueue<Integer> queue = new PriorityQueue<>();

for (int num : nums) {
    queue.offer(num);
    if (queue.size() > k) {
        queue.poll();
    }
}

System.out.println(queue); // Output: [4, 5, 6] (Top 3 elements)
```

---

7. Real-World Usage with Advanced Use Cases

Use Case 1: Event Scheduling

```
class Event {
    String name;
    int time; // Earlier time has higher priority

    public Event(String name, int time) {
        this.name = name;
        this.time = time;
    }

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

PriorityQueue<Event> eventQueue = new PriorityQueue<>(Comparator.comparingInt(e -> e.time));
eventQueue.offer(new Event("Conference", 10));
eventQueue.offer(new Event("Team Meeting", 5));
eventQueue.offer(new Event("Lunch", 12));

while (!eventQueue.isEmpty()) {
    System.out.println(eventQueue.poll()); // Output: Team Meeting, Conference, Lunch
}
```

- **Scenario**: Managing scheduled events or appointments.

Use Case 2: Dijkstra's Algorithm

A PriorityQueue is used in Dijkstra's shortest path algorithm to efficiently pick the next vertex with the smallest distance.

Deque

1. Definition

A **Deque (Double-Ended Queue)** is a data structure that allows elements to be added or removed from both ends (front and rear). It is a versatile data structure that supports **FIFO (First In, First Out)** and **LIFO (Last In, First Out)** operations.

In Java, the `Deque` interface is part of the `java.util` package and extends the `Queue` interface. The most common implementations of `Deque` are `ArrayDeque` and `LinkedList`.

Key Characteristics:

- Allows insertion and removal from both ends.
- Can be used as a queue (FIFO) or a stack (LIFO).
- Does not allow `null` elements.
- Provides better performance than `Stack` or `LinkedList` for stack and queue operations.

2. Initialization

You can initialize a `Deque` using different implementations like `ArrayDeque` or `LinkedList`.

Using ArrayDeque (Recommended):

```
Deque<String> deque = new ArrayDeque<>();
```

- Preferred for most use cases due to better performance in amortized time complexity for insertions and removals.

Using LinkedList:

```
Deque<String> deque = new LinkedList<>();
```

- Useful if you need frequent access to the middle elements, though slightly slower than `ArrayDeque` for most operations.

Initialization with Elements:

```
Deque<String> deque = new ArrayDeque<>(List.of("A", "B", "C"));
```

- Populates the deque with initial elements.

3. Type Conversion

Deque to Array:

```
Deque<String> deque = new ArrayDeque<>(List.of("A", "B", "C"));
String[] array = deque.toArray(new String[0]);
System.out.println(Arrays.toString(array)); // Output: [A, B, C]
```

Deque to List:

```
Deque<String> deque = new ArrayDeque<>(List.of("Apple", "Banana", "Cherry"));
List<String> list = new ArrayList<>(deque);
System.out.println(list); // Output: [Apple, Banana, Cherry]
```

Array to Deque:

```
String[] array = {"X", "Y", "Z"};
Deque<String> deque = new ArrayDeque<>(Arrays.asList(array));
```

4. Common In-Built Methods

The Deque interface provides methods for both ends (head and tail):

Method	Description
addFirst(E e)	Inserts an element at the front of the deque. Throws an exception if the deque is full.
addLast(E e)	Inserts an element at the rear of the deque. Throws an exception if the deque is full.
offerFirst(E e)	Inserts an element at the front of the deque. Returns false if the deque is full.
offerLast(E e)	Inserts an element at the rear of the deque. Returns false if the deque is full.
removeFirst()	Removes and returns the element at the front. Throws NoSuchElementException if empty.
removeLast()	Removes and returns the element at the rear. Throws NoSuchElementException if empty.
pollFirst()	Retrieves and removes the front element, or null if the deque is empty.
pollLast()	Retrieves and removes the rear element, or null if the deque is empty.
getFirst()	Retrieves the front element without removing it. Throws NoSuchElementException if empty.
getLast()	Retrieves the rear element without removing it. Throws NoSuchElementException if empty.
peekFirst()	Retrieves the front element without removing it, or null if empty.
peekLast()	Retrieves the rear element without removing it, or null if empty.
size()	Returns the number of elements in the deque.
isEmpty()	Checks if the deque is empty.

5. Real-World Usage (Include Sorting)

Use Case 1: Double-Ended Queue

```
Deque<String> deque = new ArrayDeque<>();
deque.addFirst("First");
deque.addLast("Last");

System.out.println(deque.pollFirst()); // Output: First
System.out.println(deque.pollLast());  // Output: Last
```

- **Scenario**: Managing a deque where elements can be added or removed from both ends.

Use Case 2: Sorting Deque

To sort elements in a deque, you can convert it into a list:

```
Deque<Integer> deque = new ArrayDeque<>(List.of(30, 10, 20));
List<Integer> list = new ArrayList<>(deque);
Collections.sort(list);
deque = new ArrayDeque<>(list);

System.out.println(deque); // Output: [10, 20, 30]
```

- **Scenario**: Sorting elements in a deque for ordered processing.

6. Advanced Use Cases and Techniques

Use Case 1: Palindrome Checker

A `Deque` can be used to check if a string is a palindrome.

```
public static boolean isPalindrome(String str) {
   Deque<Character> deque = new ArrayDeque<>();
   for (char ch : str.toCharArray()) {
      deque.addLast(ch);
   }

   while (deque.size() > 1) {
      if (!deque.pollFirst().equals(deque.pollLast())) {
         return false;
      }
   }
   return true;
}

System.out.println(isPalindrome("radar")); // Output: true
System.out.println(isPalindrome("hello")); // Output: false
```

Use Case 2: Sliding Window Maximum

A Deque can efficiently calculate the maximum in every window of size k in an array.

```
public static List<Integer> maxSlidingWindow(int[] nums, int k) {
   Deque<Integer> deque = new ArrayDeque<>();
   List<Integer> result = new ArrayList<>();

   for (int i = 0; i < nums.length; i++) {
      // Remove elements out of the current window
      while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
         deque.pollFirst();
      }

      // Remove smaller elements in k range as they are useless
      while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
         deque.pollLast();
      }

      deque.offerLast(i); // Add the current element at the end of the deque

      // Add the maximum element of the current window to the result
      if (i >= k - 1) {
         result.add(nums[deque.peekFirst()]);
      }
   }
   return result;
}

int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
System.out.println(maxSlidingWindow(nums, 3)); // Output: [3, 3, 5, 5, 6, 7]
```

- **Scenario**: Efficiently finding maximum elements in a sliding window, commonly used in data analysis and signal processing.

7. Real-World Usage with Advanced Use Cases

Real-World Example 1: Browser History Navigation

```
Deque<String> history = new ArrayDeque<>();
history.addLast("Page1");
history.addLast("Page2");
history.addLast("Page3");

System.out.println(history.pollLast()); // Navigate back to Page3
System.out.println(history.peekLast()); // Peek at the current page
```

- **Scenario**: Managing forward and backward navigation in browser history.

Real-World Example 2: Task Scheduling with Priority

```
Deque<String> highPriority = new ArrayDeque<>();
Deque<String> lowPriority = new ArrayDeque<>();

highPriority.addLast("Urgent Task 1");
lowPriority.addLast("Regular Task 1");

while (!highPriority.isEmpty() || !lowPriority.isEmpty()) {
   if (!highPriority.isEmpty()) {
      System.out.println("Processing: " + highPriority.pollFirst());
   } else {
      System.out.println("Processing: " + lowPriority.pollFirst());
   }
}
```

- **Scenario**: Scheduling tasks based on priority levels.

HashSet

1. Definition

A **HashSet** in Java is a collection that implements the Set interface. It uses a **hash table** for storage and ensures that:

1. **No duplicate elements** are allowed.
2. Elements are stored in no particular order (unordered).
3. It provides constant-time performance for basic operations like add, remove, and contains.

Key Characteristics:

- Implements Set interface and is backed by a HashMap.
- Allows one null element.
- Not synchronized (thread-unsafe by default but can be synchronized explicitly).

2. Initialization

Default Initialization:

```
HashSet<String> set = new HashSet<>();
```

Initialization with Initial Capacity:

```
HashSet<String> set = new HashSet<>(100); // Initial capacity of 100
```

Initialization with Initial Capacity and Load Factor:

```
HashSet<String> set = new HashSet<>(100, 0.75f);
```

- **Load factor** determines how full the set can be before resizing.

Initialization Using a Collection:

```
List<String> list = List.of("Apple", "Banana", "Cherry");
HashSet<String> set = new HashSet<>(list);
```

3. Type Conversion

Convert HashSet to Array:

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
String[] array = set.toArray(new String[0]);
System.out.println(Arrays.toString(array)); // Output: [A, B, C]
```

Convert HashSet to List:

```
HashSet<String> set = new HashSet<>(Set.of("X", "Y", "Z"));
List<String> list = new ArrayList<>(set);
System.out.println(list); // Output: [X, Y, Z]
```

Array to HashSet:

```
String[] array = {"A", "B", "C"};
HashSet<String> set = new HashSet<>(Arrays.asList(array));
```

4. Common In-Built Methods

add(E e)

Adds the specified element to the set if it is not already present.

```
HashSet<String> set = new HashSet<>();
set.add("A"); // Adds "A" to the set
set.add("B"); // Adds "B" to the set
set.add("A"); // Duplicate ignored
System.out.println(set); // Output: [A, B]
```

remove(Object o)

Removes the specified element from the set if it exists.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
set.remove("B"); // Removes "B"
System.out.println(set); // Output: [A, C]
```

contains(Object o)

Checks if the set contains the specified element.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B"));
System.out.println(set.contains("A")); // Output: true
System.out.println(set.contains("C")); // Output: false
```

isEmpty()

Checks if the set is empty.

```
HashSet<String> set = new HashSet<>();
System.out.println(set.isEmpty()); // Output: true
set.add("A");
System.out.println(set.isEmpty()); // Output: false
```

size()

Returns the number of elements in the set.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
System.out.println(set.size()); // Output: 3
```

clear()

Removes all elements from the set.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
set.clear(); // Clears the set
System.out.println(set); // Output: []
```

toArray()

Converts the set into an array.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
String[] array = set.toArray(new String[0]);
System.out.println(Arrays.toString(array)); // Output: [A, B, C]
```

iterator()

Returns an iterator over the elements in the set.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
Iterator<String> iterator = set.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next()); // Output: A B C (order may vary)
}
```

forEach(Consumer<? super E> action)

Performs the specified action for each element in the set.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
set.forEach(System.out::println); // Output: A B C (order may vary)
```

Additional In-Built Methods

retainAll(Collection<?> c)

Retains only the elements in this set that are also contained in the specified collection. **Usage:** Used for finding the intersection of two sets.

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2, 3, 4));
HashSet<Integer> set2 = new HashSet<>(Set.of(3, 4, 5, 6));

set1.retainAll(set2); // Retains only elements present in both sets
System.out.println(set1); // Output: [3, 4]
```

removeAll(Collection<?> c)

Removes all elements in the set that are contained in the specified collection. **Usage:** Used for finding the difference between two sets.

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2, 3, 4));
HashSet<Integer> set2 = new HashSet<>(Set.of(3, 4, 5, 6));

set1.removeAll(set2); // Removes all elements present in set2
System.out.println(set1); // Output: [1, 2]
```

addAll(Collection<? extends E> c)

Adds all elements from the specified collection to this set (union operation). **Usage:** Used for combining two sets.

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2));
HashSet<Integer> set2 = new HashSet<>(Set.of(3, 4));

set1.addAll(set2); // Adds all elements from set2 to set1
System.out.println(set1); // Output: [1, 2, 3, 4]
```

containsAll(Collection<?> c)

Returns true if the set contains all elements in the specified collection. **Usage:** Used for checking if one set is a subset of another.

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2, 3, 4));
HashSet<Integer> set2 = new HashSet<>(Set.of(2, 3));

System.out.println(set1.containsAll(set2)); // Output: true
```

spliterator()

Creates a Spliterator for this set, which can be used for parallel iteration. **Usage:** Used in advanced scenarios like parallel stream processing.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
Spliterator<String> spliterator = set.spliterator();

spliterator.forEachRemaining(System.out::println); // Output: A B C (order may vary)
```

equals(Object o)

Compares the specified object with this set for equality. **Usage:** Used to compare two sets.

```
HashSet<String> set1 = new HashSet<>(Set.of("A", "B", "C"));
HashSet<String> set2 = new HashSet<>(Set.of("A", "B", "C"));

System.out.println(set1.equals(set2)); // Output: true
```

hashCode()

Returns the hash code value for this set. **Usage:** Useful when using HashSet in a data structure like a HashMap or for debugging.

```
HashSet<String> set = new HashSet<>(Set.of("A", "B", "C"));
System.out.println(set.hashCode()); // Output: A hash code integer
```

Set Operations: Intersection, Difference, and Union with HashSet

Here's how you can perform **intersection**, **difference**, and **union** using methods from the HashSet class. These operations are essential in real-world applications like filtering, data comparison, and merging datasets.

1. Union of Two Sets

The union of two sets contains all unique elements from both sets.

Method Used: addAll()

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2, 3));
HashSet<Integer> set2 = new HashSet<>(Set.of(3, 4, 5));

// Union operation
set1.addAll(set2);
System.out.println("Union: " + set1); // Output: Union: [1, 2, 3, 4, 5]
```

2. Intersection of Two Sets

The intersection of two sets contains only the elements that are present in both sets.

Method Used: retainAll()

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2, 3));
HashSet<Integer> set2 = new HashSet<>(Set.of(3, 4, 5));

// Intersection operation
set1.retainAll(set2);
System.out.println("Intersection: " + set1); // Output: Intersection: [3]
```

3. Difference of Two Sets

The difference of two sets contains elements that are in the first set but not in the second.

Method Used: removeAll()

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2, 3));
HashSet<Integer> set2 = new HashSet<>(Set.of(3, 4, 5));

// Difference operation
set1.removeAll(set2);
System.out.println("Difference: " + set1); // Output: Difference: [1, 2]
```

5. Real-World Usage

Use Case 1: Removing Duplicates

A `HashSet` is commonly used to remove duplicate elements from a collection.

```
List<String> list = List.of("A", "B", "A", "C", "B");
HashSet<String> uniqueElements = new HashSet<>(list);

System.out.println(uniqueElements); // Output: [A, B, C]
```

Use Case 2: Checking Membership

```
HashSet<String> allowedUsers = new HashSet<>(Set.of("Alice", "Bob", "Charlie"));

if (allowedUsers.contains("Alice")) {
   System.out.println("Access granted!");
} else {
   System.out.println("Access denied!");
}
```

- **Scenario**: Quickly checking if a user is allowed access.

Use Case 3: Union, Intersection, and Difference

```
HashSet<Integer> set1 = new HashSet<>(Set.of(1, 2, 3, 4));
HashSet<Integer> set2 = new HashSet<>(Set.of(3, 4, 5, 6));

// Union
HashSet<Integer> union = new HashSet<>(set1);
union.addAll(set2);
System.out.println(union); // Output: [1, 2, 3, 4, 5, 6]

// Intersection
HashSet<Integer> intersection = new HashSet<>(set1);
intersection.retainAll(set2);
System.out.println(intersection); // Output: [3, 4]

// Difference
HashSet<Integer> difference = new HashSet<>(set1);
difference.removeAll(set2);
System.out.println(difference); // Output: [1, 2]
```

6. Advanced Use Cases and Techniques

Use Case 1: Detecting Duplicates in Streams

```
public static boolean hasDuplicates(int[] arr) {
   HashSet<Integer> seen = new HashSet<>();
   for (int num : arr) {
      if (!seen.add(num)) { // add() returns false if the element is already present
         return true;
      }
   }
   return false;
}

int[] numbers = {1, 2, 3, 4, 2};
System.out.println(hasDuplicates(numbers)); // Output: true
```

Use Case 2: Caching Results

```
class Fibonacci {
   private HashSet<Integer> cache = new HashSet<>();

   public boolean isFibonacci(int num) {
      if (cache.contains(num)) return true;
      int a = 0, b = 1;
      while (b < num) {
         int next = a + b;
         cache.add(next);
         a = b;
         b = next;
      }
      return cache.contains(num);
   }
}

Fibonacci fib = new Fibonacci();
System.out.println(fib.isFibonacci(8)); // Output: true
System.out.println(fib.isFibonacci(10)); // Output: false
```

Use Case 3: Efficient String Matching

```
HashSet<String> dictionary = new HashSet<>(List.of("cat", "dog", "bird"));
String word = "dog";

if (dictionary.contains(word)) {
   System.out.println("Word found in dictionary!");
} else {
   System.out.println("Word not found.");
}
```

7. Real-World Usage with Advanced Use Cases

Example 1: Detecting Cycles in a Graph

A HashSet can be used to detect cycles in a directed graph.

```
class Graph {
    private final Map<Integer, List<Integer>> adjacencyList = new HashMap<>();

    public void addEdge(int source, int target) {
        adjacencyList.computeIfAbsent(source, k -> new ArrayList<>()).add(target);
    }

    public boolean hasCycle() {
        HashSet<Integer> visited = new HashSet<>();
        HashSet<Integer> recursionStack = new HashSet<>();

        for (int node : adjacencyList.keySet()) {
            if (dfs(node, visited, recursionStack)) {
                return true;
            }
        }
        return false;
    }

    private boolean dfs(int node, HashSet<Integer> visited, HashSet<Integer> recursionStack) {
        if (recursionStack.contains(node)) return true;
        if (visited.contains(node)) return false;

        visited.add(node);
        recursionStack.add(node);

        for (int neighbor : adjacencyList.getOrDefault(node, List.of())) {
            if (dfs(neighbor, visited, recursionStack)) {
                return true;
            }
        }
        recursionStack.remove(node);
        return false;
    }
}

Graph graph = new Graph();
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.addEdge(3, 1);

System.out.println(graph.hasCycle()); // Output: true
```

- **Scenario**: Detecting circular dependencies in task scheduling or workflows.

Example 2: Removing Duplicates from a File

```
Set<String> uniqueLines = new HashSet<>();
try (BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
     PrintWriter writer = new PrintWriter(new FileWriter("unique_data.txt"))) {

    String line;
    while ((line = reader.readLine()) != null) {
        if (uniqueLines.add(line)) { // Add only unique lines
            writer.println(line);
        }
    }
}
```

- **Scenario**: Removing duplicate entries from large datasets.

LinkedHashSet

1. Definition

LinkedHashSet is a subclass of HashSet that maintains the **insertion order** of elements. It uses a combination of a hash table and a linked list to store elements, ensuring that elements are retrieved in the order they were added.

- **Key Characteristics:**
  - No duplicate elements (like HashSet).
  - Maintains insertion order (unlike HashSet).
  - Allows null elements.

2. Initialization

Default Constructor

```
LinkedHashSet<String> set = new LinkedHashSet<>();
```

With Initial Capacity and Load Factor

```
LinkedHashSet<Integer> set = new LinkedHashSet<>(16, 0.75f);
```

With Collection

```
LinkedHashSet<Integer> set = new LinkedHashSet<>(Set.of(1, 2, 3));
```

3. Common In-Built Methods with Code Snippets

METHOD	DESCRIPTION	CODE SNIPPET

ADD(E E)	Adds an element to the set.	LinkedHashSet<Integer> set = new LinkedHashSet<>();
		set.add(1);
		set.add(2);
		System.out.println(set); // [1, 2]

REMOVE(OBJECT O)	Removes the specified element.	set.remove(1); System.out.println(set); // [2]
CONTAINS(OBJECT O)	Checks if the set contains the element.	System.out.println(set.contains(1)); // true
SIZE()	Returns the size of the set.	System.out.println(set.size()); // 2
ISEMPTY()	Checks if the set is empty.	System.out.println(set.isEmpty()); // false
CLEAR()	Removes all elements from the set.	set.clear(); System.out.println(set); // []
TOARRAY()	Converts the set into an array.	Integer[] array = set.toArray(new Integer[0]); System.out.println(Arrays.toString(array)); // [1, 2]

ITERATOR()	Returns an iterator over elements in insertion order.	for (Integer i : set) {
		 System.out.println(i);
		}

4. Real-World Usage (Include Sorting)

Scenario: Tracking unique URLs in the order they were visited.

```
import java.util.LinkedHashSet;

public class URLTracker {
    public static void main(String[] args) {
        LinkedHashSet<String> visitedURLs = new LinkedHashSet<>();

        visitedURLs.add("https://example.com");
        visitedURLs.add("https://google.com");
        visitedURLs.add("https://example.com"); // Duplicate ignored
        visitedURLs.add("https://openai.com");

        System.out.println("Visited URLs in order: " + visitedURLs);
        // Output: Visited URLs in order: [https://example.com, https://google.com, https://openai.com]
    }
}
```

5. Advanced Use Cases and Techniques

Preserving Order in Set Operations (Intersection, Union, Difference):

```
import java.util.LinkedHashSet;

public class SetOperations {
    public static void main(String[] args) {
        LinkedHashSet<Integer> set1 = new LinkedHashSet<>(Set.of(1, 2, 3, 4));
        LinkedHashSet<Integer> set2 = new LinkedHashSet<>(Set.of(3, 4, 5, 6));

        // Union
        LinkedHashSet<Integer> unionSet = new LinkedHashSet<>(set1);
        unionSet.addAll(set2);
        System.out.println("Union: " + unionSet); // [1, 2, 3, 4, 5, 6]

        // Intersection
        LinkedHashSet<Integer> intersectionSet = new LinkedHashSet<>(set1);
        intersectionSet.retainAll(set2);
        System.out.println("Intersection: " + intersectionSet); // [3, 4]

        // Difference
        LinkedHashSet<Integer> differenceSet = new LinkedHashSet<>(set1);
        differenceSet.removeAll(set2);
        System.out.println("Difference: " + differenceSet); // [1, 2]
    }
}
```

Accessing Elements in Order

Iterate in insertion order to process elements sequentially:

```
LinkedHashSet<String> tasks = new LinkedHashSet<>();
tasks.add("Task1");
tasks.add("Task2");
tasks.add("Task3");

for (String task : tasks) {
   System.out.println(task); // Task1, Task2, Task3
}
```

6. Real-World Usage with Advanced Use Cases

Caching Recently Used Elements (LRU Cache Implementation):

LinkedHashSet can simulate an LRU cache where the least recently accessed elements are removed.

```
import java.util.LinkedHashSet;

public class LRUCache {
   private LinkedHashSet<Integer> cache;
   private int capacity;

   public LRUCache(int capacity) {
      this.capacity = capacity;
      this.cache = new LinkedHashSet<>();
   }

   public void access(int item) {
      if (cache.contains(item)) {
         cache.remove(item); // Remove to re-insert at the end
      } else if (cache.size() == capacity) {
         int first = cache.iterator().next(); // Get least recently used
         cache.remove(first);
      }
      cache.add(item);
   }

   public void display() {
      System.out.println(cache);
   }

   public static void main(String[] args) {
      LRUCache lru = new LRUCache(3);
      lru.access(1);
      lru.access(2);
      lru.access(3);
      lru.access(2); // Re-access 2
      lru.access(4); // Evicts 1 (LRU)

      lru.display(); // Output: [3, 2, 4]
   }
}
```

TreeSet

1. Definition

TreeSet is a collection class in Java that implements the Set interface and is part of the **Java Collections Framework**. It uses a **TreeMap** internally and stores elements in a **sorted and ascending order** by default.

- **Key Characteristics:**
  - No duplicate elements (like all Set implementations).
  - Automatically sorts elements in **natural order** (e.g., numeric for numbers, lexicographical for strings) or by a custom comparator.
  - Backed by a **Red-Black Tree**, providing logarithmic time complexity (O(log n)) for operations like add, remove, and contains.

2. Initialization

Default Constructor

Creates a TreeSet that sorts elements in natural order.

```
TreeSet<Integer> set = new TreeSet<>();
```

With Custom Comparator

Allows sorting elements based on a custom logic.

```
TreeSet<String> set = new TreeSet<>(Comparator.reverseOrder()); // Descending order
```

With Existing Collection

Creates a TreeSet containing elements of another collection.

```
TreeSet<Integer> set = new TreeSet<>(List.of(5, 2, 8));
```

3. Common In-Built Methods with Code Snippets

Method	Description	Code Snippet

add(E e)	Adds an element to the set.	TreeSet<Integer> set = new TreeSet<>();
		set.add(10);
		set.add(5);
		System.out.println(set); // [5, 10]

remove(Object o)	Removes the specified element.	set.remove(5); System.out.println(set); // [10]
contains(Object o)	Checks if the set contains the element.	System.out.println(set.contains(10)); // true
size()	Returns the size of the set.	System.out.println(set.size()); // 1
isEmpty()	Checks if the set is empty.	System.out.println(set.isEmpty()); // false
clear()	Removes all elements from the set.	set.clear(); System.out.println(set); // []

Method	Description	Code Snippet
first()	Retrieves the first (smallest) element.	System.out.println(set.first()); // 5
last()	Retrieves the last (largest) element.	System.out.println(set.last()); // 10
pollFirst()	Retrieves and removes the first element.	System.out.println(set.pollFirst()); // 5
pollLast()	Retrieves and removes the last element.	System.out.println(set.pollLast()); // 10
headSet(E toElement)	Returns elements less than toElement.	System.out.println(set.headSet(10)); // [5]
tailSet(E fromElement)	Returns elements greater than or equal to fromElement.	System.out.println(set.tailSet(5)); // [5, 10]
subSet(E fromElement, E toElement)	Returns elements within the range.	System.out.println(set.subSet(5, 10)); // [5]

4. Real-World Usage (Include Sorting)

Scenario: Managing a set of unique student IDs and retrieving them in sorted order.

```
import java.util.TreeSet;

public class StudentIDManager {
   public static void main(String[] args) {
      TreeSet<Integer> studentIDs = new TreeSet<>();
      studentIDs.add(102);
      studentIDs.add(101);
      studentIDs.add(103);

      System.out.println("Student IDs (sorted): " + studentIDs); // [101, 102, 103]
      System.out.println("First ID: " + studentIDs.first()); // 101
      System.out.println("Last ID: " + studentIDs.last()); // 103
   }
}
```

5. Advanced Use Cases and Techniques

Custom Sorting with Comparator:

Sort strings by their length instead of natural order.

```
import java.util.TreeSet;
import java.util.Comparator;

public class CustomSorting {
    public static void main(String[] args) {
        TreeSet<String> set = new TreeSet<>(Comparator.comparingInt(String::length));
        set.add("Apple");
        set.add("Banana");
        set.add("Kiwi");

        System.out.println(set); // [Kiwi, Apple, Banana]
    }
}
```

Efficient Range Queries:

Retrieve all elements within a specific range.

```
import java.util.TreeSet;

public class RangeQuery {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>(Set.of(10, 20, 30, 40, 50));

        System.out.println("Elements < 30: " + set.headSet(30)); // [10, 20]
        System.out.println("Elements >= 30: " + set.tailSet(30)); // [30, 40, 50]
        System.out.println("Elements between 20 and 40: " + set.subSet(20, 40)); // [20, 30]
    }
}
```

6. Real-World Usage with Advanced Use Cases

Real-Time Leaderboard:

Maintain a sorted leaderboard of scores, and efficiently find top scorers.

```
import java.util.TreeSet;

public class Leaderboard {
    public static void main(String[] args) {
        TreeSet<Integer> scores = new TreeSet<>(Comparator.reverseOrder());
        scores.add(300);
        scores.add(500);
        scores.add(400);

        System.out.println("Top Scores: " + scores); // [500, 400, 300]
        System.out.println("Highest Score: " + scores.first()); // 500
    }
}
```

Scheduling Tasks by Priority:

Schedule tasks and always process the smallest (earliest) task first.

```
import java.util.TreeSet;

public class TaskScheduler {
   public static void main(String[] args) {
      TreeSet<String> tasks = new TreeSet<>();
      tasks.add("Task3: Write report");
      tasks.add("Task1: Review code");
      tasks.add("Task2: Prepare slides");

      while (!tasks.isEmpty()) {
         System.out.println("Processing: " + tasks.pollFirst());
      }
      // Output:
      // Processing: Task1: Review code
      // Processing: Task2: Prepare slides
      // Processing: Task3: Write report
   }
}
```

Map Interface

1. Definition

The **Map Interface** in Java represents a collection of **key-value pairs**, where each key maps to a specific value. It is a part of the **Java Collections Framework** and is found in the java.util package.

- **Key Characteristics**:
  - No duplicate keys are allowed.
  - Each key maps to exactly one value.
  - Implementations may or may not maintain the order of the mappings.

---

2. Common Implementations of Map

1. **HashMap**:
   - Unordered, allows null keys and values.
   - Backed by a hash table.
   - Suitable for most general-purpose use cases.
2. **LinkedHashMap**:
   - Maintains the insertion order of keys.
   - Slightly slower than HashMap.
3. **TreeMap**:
   - Maintains keys in sorted (natural or custom) order.
   - Does not allow null keys but allows multiple null values.
4. **Hashtable**:
   - Synchronized and thread-safe, but slower.
   - Does not allow null keys or values.
5. **ConcurrentHashMap**:
   - A thread-safe implementation designed for concurrent access.

HashMap

1. Definition

A **HashMap** is a part of the **Java Collections Framework** that implements the `Map` interface. It is used to store data in **key-value pairs**, where keys are unique, and each key maps to exactly one value.

- **Key Characteristics**:
  - Allows `null` keys and multiple `null` values.
  - Does **not maintain any order** of keys or values.
  - Backed by a **hash table**, providing constant time (`O(1)`) performance for basic operations like `put()` and `get()`, assuming a good hash function.

2. Initialization

Default Constructor

Creates an empty `HashMap`.

```
HashMap<String, Integer> map = new HashMap<>();
```

With Initial Capacity and Load Factor

Specifies the initial size of the hash table and its load factor.

```
HashMap<String, Integer> map = new HashMap<>(16, 0.75f);
```

- **Initial Capacity**: Number of buckets created initially (default is 16).
- **Load Factor**: Determines when to increase the hash table size (default is 0.75).

From Existing Map

Creates a new `HashMap` with mappings from another map.

```
Map<String, Integer> original = Map.of("A", 1, "B", 2);
HashMap<String, Integer> map = new HashMap<>(original);
```

3. Commonly Used Methods (With Examples)

Method	Description	Code Snippet

put(K key, V value)	Adds or updates a key-value pair.	map.put("A", 1);
		map.put("B", 2);

get(Object key)	Retrieves the value for the given key.	System.out.println(map.get("A")); // 1
remove(Object key)	Removes the mapping for the given key.	map.remove("B");
containsKey(Object key)	Checks if the map contains the given key.	System.out.println(map.containsKey("A")); // true
containsValue(Object value)	Checks if the map contains the given value.	System.out.println(map.containsValue(2)); // false
keySet()	Returns a set of all keys in the map.	System.out.println(map.keySet()); // [A]
values()	Returns a collection of all values in the map.	System.out.println(map.values()); // [1]

entrySet()	Returns a set of key-value pairs (Map.Entry).	for (Map.Entry<String, Integer> entry : map.entrySet()) {
		 System.out.println(entry);
		}

size()	Returns the number of key-value pairs.	System.out.println(map.size()); // 1
isEmpty()	Checks if the map is empty.	System.out.println(map.isEmpty()); // false
clear()	Removes all mappings.	map.clear(); System.out.println(map); // {}
putIfAbsent(K key, V value)	Adds the key-value pair if the key is not present.	map.putIfAbsent("C", 3);

4. Real-World Use Cases

1. Storing Student Grades

```
import java.util.HashMap;

public class StudentGrades {
   public static void main(String[] args) {
      HashMap<String, Double> grades = new HashMap<>();
      grades.put("Alice", 85.5);
      grades.put("Bob", 92.0);
      grades.put("Charlie", 77.3);

      System.out.println("Alice's Grade: " + grades.get("Alice")); // 85.5
   }
}
```

2. Counting Word Frequencies

```
import java.util.HashMap;

public class WordCount {
   public static void main(String[] args) {
      String[] words = {"apple", "banana", "apple", "orange", "banana", "apple"};

      HashMap<String, Integer> wordCount = new HashMap<>();
      for (String word : words) {
         wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
      }

      System.out.println(wordCount); // {orange=1, banana=2, apple=3}
   } }
```

5. Advanced Use Cases

Custom HashMap Behavior Using compute() Methods

- Modify values based on custom logic during insertion or update.

```
import java.util.HashMap;

public class ComputeExample {
   public static void main(String[] args) {
      HashMap<String, Integer> map = new HashMap<>();
      map.put("A", 1);
      map.put("B", 2);

      // Update value of A
      map.compute("A", (key, val) -> (val == null) ? 1 : val + 1);

      // Add new key C
      map.computeIfAbsent("C", key -> 3);

      System.out.println(map); // {A=2, B=2, C=3}
   }
}
```

Efficient LRU Cache Using LinkedHashMap

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

public class LRUCache<K, V> extends LinkedHashMap<K, V> {
   private final int capacity;

   public LRUCache(int capacity) {
      super(capacity, 0.75f, true); // Access order
      this.capacity = capacity;
   }

   @Override
   protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
      return size() > capacity;
   }

   public static void main(String[] args) {
      LRUCache<Integer, String> cache = new LRUCache<>(3);
      cache.put(1, "A");
      cache.put(2, "B");
      cache.put(3, "C");
      cache.get(1); // Access 1
      cache.put(4, "D"); // Evicts 2

      System.out.println(cache); // {3=C, 1=A, 4=D}
   }
}
```

6. Best Practices

1. **Use Appropriate Initial Capacity**
    - If you know the approximate size, initialize the HashMap with a custom capacity to avoid resizing overhead.
2. **Avoid Poor Hashing**
    - Use immutable objects like String or custom objects with proper hashCode() and equals() implementations as keys.
3. **Thread-Safe Alternative**
    - Use ConcurrentHashMap for multi-threaded environments.

LinkedHashMap

1. Definition

A **LinkedHashMap** is a **HashMap** that maintains the **insertion order** of its keys. This means the entries in the map are stored in the order in which they were added. It extends the HashMap class and implements the Map interface.

- **Key Characteristics**:
  - **Order of Elements**: It maintains the insertion order of keys.
  - **Performance**: Slightly slower than HashMap due to the additional linked list for maintaining order.
  - **Allows null keys and values**: Like HashMap, it allows one null key and multiple null values.

2. Initialization

Default Constructor

Creates an empty LinkedHashMap with the default capacity and load factor.

```
LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
```

With Initial Capacity and Load Factor

Specifies the initial size and load factor.

```
LinkedHashMap<String, Integer> map = new LinkedHashMap<>(16, 0.75f);
```

With Initial Capacity and Access Order

You can specify whether the map should maintain **insertion order** (default) or **access order** (used in LRU caches).

```
// Access order = true
LinkedHashMap<String, Integer> map = new LinkedHashMap<>(16, 0.75f, true);
```

3. Commonly Used Methods (With Examples)

METHOD	DESCRIPTION	CODE SNIPPET

PUT(K KEY, V VALUE)	Adds or updates a key-value pair.	map.put("A", 1);
		map.put("B", 2);

GET(OBJECT KEY)	Retrieves the value for the given key.	System.out.println(map.get("A")); // 1
REMOVE(OBJECT KEY)	Removes the mapping for the given key.	map.remove("B");
CONTAINSKEY(OBJECT KEY)	Checks if the map contains the given key.	System.out.println(map.containsKey("A")); // true
CONTAINSVALUE(OBJECT VALUE)	Checks if the map contains the given value.	System.out.println(map.containsValue(2)); // false
KEYSET()	Returns a set of all keys in the map.	System.out.println(map.keySet()); // [A]
VALUES()	Returns a collection of all values in the map.	System.out.println(map.values()); // [1]

ENTRYSET()	Returns a set of key-value pairs (Map.Entry).	for (Map.Entry<String, Integer> entry : map.entrySet()) {
		 System.out.println(entry);
		}

SIZE()	Returns the number of key-value pairs in the map.	System.out.println(map.size()); // 1
ISEMPTY()	Checks if the map is empty.	System.out.println(map.isEmpty()); // false
CLEAR()	Removes all mappings.	map.clear(); System.out.println(map); // {}

4. Real-World Use Cases

1. Storing User Sessions in Web Applications

In a web application, you may want to store session data for users while maintaining the order in which sessions were created. A LinkedHashMap can be used to store and manage this session data.

```
import java.util.LinkedHashMap;

public class UserSessions {
   public static void main(String[] args) {
      LinkedHashMap<String, String> sessions = new LinkedHashMap<>();
      sessions.put("session1", "User1");
      sessions.put("session2", "User2");
      sessions.put("session3", "User3");

      // Accessing sessions in the order they were added
      for (String session : sessions.keySet()) {
         System.out.println(session + ": " + sessions.get(session));
      }
   }
}
```

2. Maintaining the Insertion Order of Key-Value Pairs

```
import java.util.LinkedHashMap;

public class MaintainOrder {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
        map.put("One", 1);
        map.put("Two", 2);
        map.put("Three", 3);

        // Iterating to print keys in insertion order
        for (String key : map.keySet()) {
            System.out.println(key + ": " + map.get(key));
        }
    }
}
```

5. Advanced Use Cases

1. Implementing LRU Cache (Least Recently Used)

A `LinkedHashMap` can be configured to maintain access order, which makes it ideal for implementing an **LRU cache**. The least recently accessed entry is removed when the cache exceeds its capacity.

```
import java.util.LinkedHashMap;

public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75f, true); // Access order = true
        this.capacity = capacity;
    }

    @Override
    protected boolean removeEldestEntry(java.util.Map.Entry<K, V> eldest) {
        return size() > capacity; // Removes eldest entry when size exceeds capacity
    }

    public static void main(String[] args) {
        LRUCache<Integer, String> cache = new LRUCache<>(3);
        cache.put(1, "A");
        cache.put(2, "B");
        cache.put(3, "C");
        cache.get(1); // Access 1
        cache.put(4, "D"); // Evicts 2

        System.out.println(cache); // {3=C, 1=A, 4=D}
    }
}
```

2. Implementing Cache with Automatic Expiration

A `LinkedHashMap` can be used with custom expiration logic to implement a cache where entries are automatically removed after a certain period.

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

public class ExpiringCache<K, V> extends LinkedHashMap<K, V> {
    private final long expirationTime;
    private final Map<K, Long> timestamps;

    public ExpiringCache(long expirationTime) {
        super(16, 0.75f, true); // Access order = true
        this.expirationTime = expirationTime;
        this.timestamps = new LinkedHashMap<>();
    }

    @Override
    public V put(K key, V value) {
        timestamps.put(key, System.currentTimeMillis());
        return super.put(key, value);
    }

    @Override
    public V get(Object key) {
        if (isExpired(key)) {
            remove(key);
            return null;
        }
        return super.get(key);
    }

    private boolean isExpired(Object key) {
        Long timestamp = timestamps.get(key);
        if (timestamp == null) {
            return true;
        }
        return System.currentTimeMillis() - timestamp > expirationTime;
    }

    public static void main(String[] args) throws InterruptedException {
        ExpiringCache<Integer, String> cache = new ExpiringCache<>(5000); // 5 seconds expiration
        cache.put(1, "A");
        Thread.sleep(3000); // Wait 3 seconds
        System.out.println(cache.get(1)); // Should print "A"
        Thread.sleep(3000); // Wait another 3 seconds
        System.out.println(cache.get(1)); // Should print null because it expired
    }
}
```

6. Best Practices

1. **Use When Order Matters**
    - LinkedHashMap is best used when you need to maintain the order of insertion of the keys. It's particularly useful in cases like caches, ordered mappings, or tracking the sequence of events.
2. **Use LinkedHashMap for Cache Implementations**
    - When implementing LRU (Least Recently Used) caches or caches where the order of access matters, LinkedHashMap with access order is a natural choice.
3. **Performance Considerations**
    - If order is not important, prefer HashMap as it offers slightly better performance due to the overhead of maintaining the linked list for insertion order.

TreeMap

1. Definition

A **TreeMap** is a **Map** implementation that stores its entries in **sorted order** according to the natural ordering of its keys (or by a specified Comparator). It is part of the java.util package and implements the NavigableMap interface.

- **Key Characteristics**:
  - **Sorted Order**: TreeMap orders the keys in ascending order by default or according to the provided Comparator.
  - **No null keys**: Unlike HashMap or LinkedHashMap, TreeMap does not allow null keys, but it allows null values.
  - **Red-Black Tree**: Internally, TreeMap uses a **Red-Black Tree**, ensuring that the keys are always sorted.

---

2. Initialization

Default Constructor

Creates an empty TreeMap, where the keys are sorted in their natural order.

```
TreeMap<String, Integer> map = new TreeMap<>();
```

With Custom Comparator

Creates a TreeMap with a custom comparator to define a specific sorting order.

```
TreeMap<String, Integer> map = new TreeMap<>(Comparator.reverseOrder());
```

From Another Map

Creates a new TreeMap from another Map (entries are inserted in sorted order).

```
Map<String, Integer> original = Map.of("A", 1, "B", 2);
TreeMap<String, Integer> map = new TreeMap<>(original);
```

3. Commonly Used Methods (With Examples)

METHOD	DESCRIPTION	CODE SNIPPET

PUT(K KEY, V VALUE)	Adds or updates a key-value pair.	map.put("A", 1);
		map.put("B", 2);

GET(OBJECT KEY)	Retrieves the value for the given key.	System.out.println(map.get("A")); // 1
REMOVE(OBJECT KEY)	Removes the mapping for the given key.	map.remove("B");
CONTAINSKEY(OBJECT KEY)	Checks if the map contains the given key.	System.out.println(map.containsKey("A")); // true
CONTAINSVALUE(OBJECT VALUE)	Checks if the map contains the given value.	System.out.println(map.containsValue(2)); // false
KEYSET()	Returns a set of all keys in the map (sorted order).	System.out.println(map.keySet()); // [A, B]
VALUES()	Returns a collection of all values in the map.	System.out.println(map.values()); // [1, 2]

ENTRYSET()	Returns a set of key-value pairs (Map.Entry).	for (Map.Entry<String, Integer> entry : map.entrySet()) {
		 System.out.println(entry);
		}

SIZE()	Returns the number of key-value pairs in the map.	System.out.println(map.size()); // 2
ISEMPTY()	Checks if the map is empty.	System.out.println(map.isEmpty()); // false
CLEAR()	Removes all mappings.	map.clear(); System.out.println(map); // {}

4. Real-World Use Cases

1. Storing Sorted Data

TreeMap can be used to store a collection of data in a sorted manner. For instance, if you are storing student IDs and their corresponding scores, and you want to keep the IDs in ascending order.

```
import java.util.TreeMap;

public class SortedStudentScores {
   public static void main(String[] args) {
      TreeMap<String, Integer> studentScores = new TreeMap<>();
      studentScores.put("Alice", 90);
      studentScores.put("Bob", 80);
      studentScores.put("Charlie", 85);

      // Print the sorted map
      studentScores.forEach((key, value) -> System.out.println(key + ": " + value));
   }
}
```

2. Ranking System

If you're implementing a ranking system, TreeMap can help to store the ranking in sorted order as it automatically arranges the keys.

```
import java.util.TreeMap;

public class RankingSystem {
   public static void main(String[] args) {
      TreeMap<Integer, String> rankings = new TreeMap<>();
      rankings.put(1, "Alice");
      rankings.put(2, "Bob");
      rankings.put(3, "Charlie");

      // Print the rankings in ascending order
      rankings.forEach((rank, name) -> System.out.println("Rank " + rank + ": " + name));
   }
}
```

5. Advanced Use Cases

1. NavigableMap Methods: Since `TreeMap` implements `NavigableMap`, it provides methods to navigate and work with the sorted map.

- `firstKey()`: Returns the first (lowest) key.
- `lastKey()`: Returns the last (highest) key.
- `lowerKey(K key)`: Returns the largest key strictly less than the given key.
- `higherKey(K key)`: Returns the smallest key strictly greater than the given key.

```
import java.util.TreeMap;

public class NavigableMapExample {
   public static void main(String[] args) {
      TreeMap<String, Integer> map = new TreeMap<>();
      map.put("Alice", 90);
      map.put("Bob", 80);
      map.put("Charlie", 85);

      System.out.println("First Key: " + map.firstKey()); // Alice
      System.out.println("Last Key: " + map.lastKey()); // Charlie
      System.out.println("Key just lower than 'Charlie': " + map.lowerKey("Charlie")); // Bob
   }
}
```

2. Implementing a Range Query

You can use subMap(), headMap(), and tailMap() methods to implement range queries.

```
import java.util.TreeMap;

public class RangeQueryExample {
   public static void main(String[] args) {
      TreeMap<Integer, String> map = new TreeMap<>();
      map.put(1, "Alice");
      map.put(2, "Bob");
      map.put(3, "Charlie");
      map.put(4, "David");
      map.put(5, "Eve");

      // Get entries within the range 2 to 4
      System.out.println(map.subMap(2, 4)); // {2=Bob, 3=Charlie, 4=David}
   }
}
```

6. Best Practices

1. **Use when Sorting is Required**:
   Use TreeMap when you need to maintain the natural or custom order of the keys, like in cases where you need data sorted for presentation or further processing.
2. **Avoid null Keys**:
   TreeMap does not allow null keys. If null keys are required, consider using HashMap or LinkedHashMap.
3. **Optimize for Search**:
   For applications that require efficient sorted data lookups, TreeMap is a great choice as it provides logarithmic time (O(log n)) for insertion, deletion, and lookup operations.