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.