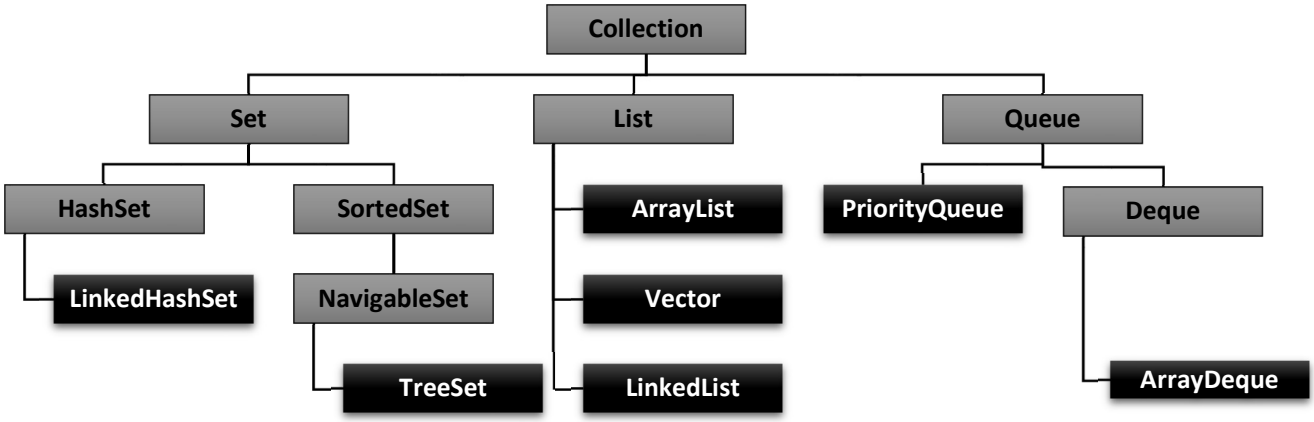
Day 10: Java Collections Framework

- Queue
- Streams API



The Queue Interface

- A Queue is a collection that holds elements before performing any operation
- Apart from collection, Queue provides operations, such as insertion, extraction, inspection.
- Principle used is FIFO
- Each queue shows the following nature: throws Exception when an operation fails

The PriorityQueue Class

- Introduced with Java 5, implements the Queue interface
- Arranged in natural order or Comparator interface(custom)
- The PriorityQueue class provides a Queue implementing the concept of priority-in and priority-out instead of FIFO

The Deque Interface

- Extends Queue and allows elements to be added or removed from both ends.
- Implementations of Deque include ArrayDeque and LinkedList.
- Methods like addFirst(), addLast(), pollFirst(), and pollLast() are available for these double-ended operations.

The ArrayDeque Class

- A resizable array implementation of the Deque interface, which extends the Queue interface.
- It is a double-ended queue and supports fast insertions and removals at both ends.

The LinkedList Class

- The LinkedList class extends the AbstractSequentialList class and implements the List Interface.
- Similar to an ArrayList, a LinkedList is an ordered list based on the index position of elements. Elements of a LinkedList are doubly linked to one another.
- Provides methods like addFirst(Object o), addLast(Object o)
- LinkedList iterate slowly as compared to ArrayList, however, it facilitates fast insertion and deletion of elements.
- With the release of Java 5, the LinkedList class has been enhanced to implement the Queue interface and supports methods like : peek(), poll() and offer()

Methods of Collection interface

public abstract int size()
public abstract boolean isEmpty()
public abstract boolean contains(java.lang.Object)
public abstract java.util.Iterator<E> iterator()
public abstract java.lang.Object[] toArray()
public abstract <T> T[] toArray(T[])
public abstract boolean add(E)
public abstract boolean remove(java.lang.Object)
public abstract boolean containsAll(java.util.Collection<?>)
public abstract boolean addAll(java.util.Collection<? extends E>)
public abstract boolean removeAll(java.util.Collection<?>)
public boolean removeIf(java.util.function.Predicate<? super E>)
public abstract boolean retainAll(java.util.Collection<?>)
public abstract void clear()
public abstract boolean equals(java.lang.Object)
public abstract int hashCode()
public java.util.Spliterator<E> spliterator()
public java.util.stream.Stream<E> stream()
public java.util.stream.Stream<E> parallelStream()

Example: TestCollection.java

Methods of Queue interface

E peek() //returns head element or null, returns null if Queue is Empty (safe, no exception)
E element() // returns head element, throws NoSuchElementException if Queue is Empty
E poll() //removes and returns head element or null, returns null if Queue is Empty
E remove() // removes and returns head element, throws NoSuchElementException if Queue is Empty
boolean offer(E element) //The offer method adds an element if possible, otherwise returning false
boolean add(E element) //The offer method adds an element if possible, otherwise throws Exception
int size()

Example: TestQueue.java

Methods of Deque interface

void addFirst(E)
void addLast(E)
boolean offerFirst(E)
boolean offerLast(E)
E removeFirst()
E removeLast()
E pollFirst()
E pollLast()
E getFirst()
E getLast()
E peekFirst()
E peekLast()
boolean removeFirstOccurrence(java.lang.Object)
boolean removeLastOccurrence(java.lang.Object)

default java.util.Deque<E> reversed()

Example: TestDeque.java

PriorityQueue

PriorityQueue()
PriorityQueue(int size)
PriorityQueue(java.util.Comparator<? super E>);
PriorityQueue(int size, java.util.Comparator<? super E>);

Example: PriorityQueueRep.java

Queue 1: Factory Task Scheduling System

You are developing a Factory Task Scheduling System to manage factory tasks in a queue. The system should use a Queue to ensure First-In-First-Out (FIFO) processing. Use ArrayDeque (a double-ended queue) because:

- It provides fast FIFO operations (offer(), poll(), and peek()).
- It is faster than LinkedList for queue operations.
- It does not allow null elements, ensuring better data integrity.

Task Class Definition:

Each task has the following attributes:

1. **Task ID** (int) → A unique identifier for each task.
2. **Task Name** (String) → A brief name/description of the task.
3. **Priority Level** (int) → A number representing task priority (higher means more urgent).

Menu Options & Expected Behaviour

1. Add a Task

- User enters Task ID, Task Name, and Priority.
- The task is added to the queue.
- Output: "Task added successfully."

2. Process a Task (Dequeue)

- Removes and displays the task at the front of the queue.
- Output:
    - "Processing task: [Task Name] (ID: [Task ID], Priority: [Priority])"
    - "No tasks to process." (if empty)

3. Peek at the Next Task

- Shows details of the task at the front without removing it.
- Output:
    - "Next task: [Task Name] (ID: [Task ID], Priority: [Priority])"

- "No tasks in the queue."

4. Display All Pending Tasks

- Lists all tasks in the queue in the order they were added.
- Output:

Pending Tasks:

1. ID: 101, Name: Assemble Parts, Priority: 2

2. ID: 102, Name: Quality Check, Priority: 3

- "No pending tasks." (if empty)

5. Count Total Pending Tasks

- Output: "Total pending tasks: [count]"

6. Exit the Program

- Output: "Exiting the Factory Task Scheduling System. Goodbye!"

Queue 2: Factory Emergency Response System

You are developing a **Factory Emergency Response System** to handle emergency incidents in a factory. The system should use a **PriorityQueue** to ensure that the most critical emergencies are handled first (higher priority first).

Why PriorityQueue?

- It processes **higher-priority elements before lower-priority ones**.
- Uses **natural ordering (or a custom comparator)** for sorting.
- Ensures that the most critical incident is **always processed first**.

Incident Class Definition:

Each incident has the following attributes:

1. **Incident ID** (int) → Unique identifier for the incident.
2. **Incident Type** (String) → Description of the emergency (e.g., Fire, Electrical Failure).
3. **Severity Level** (int) → A number representing the severity (higher means more severe).

Menu Options & Expected Behaviour:

1. Report an Incident

- User enters **Incident ID, Incident Type, and Severity Level**.
- The incident is added to the queue based on severity.
- Output: "Incident reported successfully."

2. Handle the Most Severe Incident

- Removes and displays the most severe incident.
- Output:

- "Handling incident: [Incident Type] (ID: [Incident ID], Severity: [Severity Level])"
- "No incidents to handle." (if empty)

3. View the Most Severe Incident

- Displays the most critical incident without removing it.
- Output:
  - "Most severe incident: [Incident Type] (ID: [Incident ID], Severity: [Severity Level])"
  - "No incidents in the queue."

4. Display All Reported Incidents

- Lists all pending incidents sorted by severity.
- Output:

Reported Incidents (Sorted by Severity):

1. ID: 202, Type: Fire, Severity: 5

2. ID: 203, Type: Gas Leak, Severity: 4

- "No incidents reported." (if empty)

5. Count Total Pending Incidents

- Output: "Total pending incidents: [count]"

6. Exit the Program

- Output: "Exiting the Factory Emergency Response System. Stay Safe!"

Queue 3: Factory Package Dispatch System

You are developing a **Factory Package Dispatch System** to manage package deliveries. The system should use an **ArrayDeque** to ensure **FIFO processing** of package dispatches.

Why ArrayDeque?

- It provides **fast queue operations** (offer(), poll(), and peek()).
- It **does not allow null elements**, ensuring data integrity.
- Faster than LinkedList for queue-based operations.

Package Class Definition:

Each package has the following attributes:

1. **Package ID** (int) → Unique identifier for the package.
2. **Recipient Name** (String) → The name of the recipient.
3. **Weight (kg)** (double) → The weight of the package.

Menu Options & Expected Behavior:

1. Add a Package to the Dispatch Queue

- User enters **Package ID, Recipient Name, and Weight**.

- The package is added to the queue.
- Output: "Package added successfully."

2. Dispatch a Package (Dequeue)

- Removes and displays the package at the front of the queue.
- Output:
    - "Dispatching package: [Recipient Name] (ID: [Package ID], Weight: [Weight] kg)"
    - "No packages to dispatch." (if empty)

3. View the Next Package for Dispatch

- Displays the package at the front without removing it.
- Output:
    - "Next package: [Recipient Name] (ID: [Package ID], Weight: [Weight] kg)"
    - "No packages in the queue."

4. Display All Pending Packages

- Lists all packages in FIFO order.
- Output:

Pending Packages:

1. ID: 301, Recipient: John Doe, Weight: 2.5 kg

2. ID: 302, Recipient: Jane Smith, Weight: 5.0 kg

    - "No pending packages." (if empty)

5. Count Total Pending Packages

- Output: "Total pending packages: [count]"

6. Exit the Program

- Output: "Exiting the Factory Package Dispatch System. Safe deliveries!"

Stream API

In Java Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels. Streams don't change the original data structure; they only provide the result as per the pipelined methods. Each intermediate operation is lazily executed and returns a stream as a result; hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Fetching a Stream from Arrays, Collections, and Maps in Java

Before diving into Stream API methods, let's first understand how to create a **Stream** from different data structures:

1. Stream from Arrays

Use Arrays.stream() or Stream.of() to convert an array into a stream.

Example:

```
import java.util.Arrays;

import java.util.stream.Stream;


public class StreamFromArray {

   public static void main(String[] args) {

      Integer[] numbers = {10, 20, 30, 40, 50};


      // Using Arrays.stream()

      Stream<Integer> stream1 = Arrays.stream(numbers);

      stream1.forEach(System.out::println);


      // Using Stream.of()

      Stream<Integer> stream2 = Stream.of(numbers);

      stream2.forEach(System.out::println);

   }

}
```

2. Stream from Collections (List, Set)

Use the .stream() method on any Collection (List, Set, etc.).

Example:

```
import java.util.Arrays;

import java.util.List;
```

```
public class StreamFromCollection {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Jane", "Jake");

        // Fetching stream from List
        names.stream().forEach(System.out::println);
    }
}
```

For a Set, the process is the same:

```
import java.util.HashSet;
import java.util.Set;

public class StreamFromSet {
    public static void main(String[] args) {
        Set<Integer> numbers = new HashSet<>(Set.of(5, 10, 15, 20));

        // Fetching stream from Set
        numbers.stream().forEach(System.out::println);
    }
}
```

3. Stream from Map (Keys, Values, Entries)

Maps are not directly iterable as streams. Instead, use:

- map.keySet().stream() for keys.
- map.values().stream() for values.
- map.entrySet().stream() for key-value pairs.

Example:

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

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

```
        studentMap.put(1, "Alice");
        studentMap.put(2, "Bob");
        studentMap.put(3, "Charlie");

        // Stream of keys
        studentMap.keySet().stream().forEach(System.out::println);

        // Stream of values
        studentMap.values().stream().forEach(System.out::println);

        // Stream of key-value pairs
        studentMap.entrySet().stream().forEach(entry ->
            System.out.println(entry.getKey() + " -> " + entry.getValue()));
    }
}
```

4. Creating a Stream

Method: Stream.of()

- **Syntax:** Stream<T> stream = Stream.of(T... values);

Example:

```
import java.util.stream.Stream;

public class StreamOfExample {
    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
        stream.forEach(System.out::println);
    }
}
```

1. Stream API Methods with Wrapper or String Class

1.1 Filtering Elements

Method: filter(Predicate<T>)

- **Syntax:**

  stream.filter(element -> condition).collect(Collectors.toList());

- **Logic:**
  Filters elements based on a condition.

- **Example:**

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamFilterExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Jane", "Jack", "Jill");
        List<String> filteredNames = names.stream()
                                .filter(name -> name.startsWith("J"))
                                .collect(Collectors.toList());
        System.out.println(filteredNames);
    }
}
```

1.2 Transforming Data

Method: map(Function<T, R>)

- **Syntax:**

  stream.map(element -> transformation).collect(Collectors.toList());

- **Logic:**
  Applies a function to each element.

- **Example:**

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamMapExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("apple", "banana", "cherry");
        List<String> upperNames = names.stream()
                                .map(String::toUpperCase)
                                .collect(Collectors.toList());
        System.out.println(upperNames);
    }
```

```
}
```

1.3 Sorting Data

Method: sorted(Comparator<T>)

- **Syntax:**

  stream.sorted().collect(Collectors.toList());

- **Logic:**
  Sorts elements in natural or custom order.

- **Example:**

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;


public class StreamSortExample {

   public static void main(String[] args) {

      List<Integer> numbers = Arrays.asList(5, 2, 8, 3, 1);

      List<Integer> sortedNumbers = numbers.stream()

                                    .sorted()

                                    .collect(Collectors.toList());

      System.out.println(sortedNumbers);

   }

}
```

1.4 Reducing Data

Method: reduce(BinaryOperator<T>)

- **Syntax:**

  stream.reduce(initial_value, (acc, element) -> operation);

- **Logic:**
  Reduces elements to a single result.

- **Example:**

```
import java.util.Arrays;

import java.util.List;


public class StreamReduceExample {
```

```
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        int sum = numbers.stream()
                .reduce(0, Integer::sum);
        System.out.println(sum);
    }
}
```

2. Stream API Methods with Custom Class

2.1 Filtering Custom Objects

```
List<Employee> filteredEmployees = employees.stream()
    .filter(e -> e.age > 28)
    .collect(Collectors.toList());
System.out.println(filteredEmployees);
```

2.2 Mapping Custom Objects

```
List<String> employeeNames = employees.stream()
    .map(e -> e.name)
    .collect(Collectors.toList());
System.out.println(employeeNames);
```

2.3 Sorting Custom Objects

```
List<Employee> sortedEmployees = employees.stream()
    .sorted((e1, e2) -> Integer.compare(e1.age, e2.age))
    .collect(Collectors.toList());
System.out.println(sortedEmployees);
```

2.5 Reducing Custom Objects

```
int totalAge = employees.stream()
    .map(e -> e.age)
    .reduce(0, Integer::sum);
System.out.println("Total Age: " + totalAge);
```

Java Stream API- Advanced Examples

1. Find Employee with the Largest Age

We can use the max() function with a comparator to get the employee with the highest age.

Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.Comparator;
import java.util.Optional;

class Employee {
    String name;
    int age;

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

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

public class FindMaxAgeEmployee {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("John", 30),
            new Employee("Jane", 25),
            new Employee("Jake", 35),
            new Employee("Emma", 40)
        );

        Optional<Employee> oldestEmployee = employees.stream()
                .max(Comparator.comparingInt(e -> e.age));
```

```
        oldestEmployee.ifPresent(System.out::println);
    }
}
```

Output:

Emma (40)

2. Find Employee with the Smallest Age

Use min() function similarly.

Example:

```
Optional<Employee> youngestEmployee = employees.stream()
        .min(Comparator.comparingInt(e -> e.age));

youngestEmployee.ifPresent(System.out::println);
```

Output:

Jane (25)

3. Group Employees by Age (e.g., Age above or below 30)

Use Collectors.partitioningBy() to separate employees into two categories.

Example:

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

public class GroupEmployees {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("John", 30),
            new Employee("Jane", 25),
            new Employee("Jake", 35),
            new Employee("Emma", 40)
        );

        Map<Boolean, List<Employee>> partitionedEmployees =
            employees.stream().collect(Collectors.partitioningBy(e -> e.age > 30));
```

```
        System.out.println("Employees older than 30: " + partitionedEmployees.get(true));
        System.out.println("Employees 30 or younger: " + partitionedEmployees.get(false));
    }
}
```

Output:

Employees older than 30: [Jake (35), Emma (40)]

Employees 30 or younger: [John (30), Jane (25)]

4. Find the Average Age of Employees

Use Collectors.averagingInt() to find the average age.

Example:

```
double averageAge = employees.stream()
        .collect(Collectors.averagingInt(e -> e.age));
System.out.println("Average Age: " + averageAge);
```

Output:

Average Age: 32.5

5. Count Occurrences of Words in a List

Use Collectors.groupingBy() to count word occurrences.

Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class WordCount {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "apple", "orange", "banana", "apple");

        Map<String, Long> wordCount = words.stream()
                .collect(Collectors.groupingBy(w -> w, Collectors.counting()));

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

}

Output:

{orange=1, banana=2, apple=3}

6. Find Distinct Elements from a List

Use distinct() to remove duplicates.

Example:

```
List<Integer> numbers = Arrays.asList(10, 20, 30, 20, 10, 40, 50);
List<Integer> uniqueNumbers = numbers.stream().distinct().collect(Collectors.toList());
System.out.println(uniqueNumbers);
```

Output:

[10, 20, 30, 40, 50]

7. Get the Top 3 Oldest Employees

Use sorted() with limit().

Example:

```
List<Employee> topThreeOldest = employees.stream()
    .sorted(Comparator.comparingInt(Employee::age).reversed())
    .limit(3)
    .collect(Collectors.toList());
System.out.println(topThreeOldest);
```

Output:

[Emma (40), Jake (35), John (30)]

8. Concatenating Employee Names

Use Collectors.joining() to concatenate names.

Example:

```
String names = employees.stream()
    .map(e -> e.name)
    .collect(Collectors.joining(", "));
System.out.println("Employees: " + names);
```

Output:

Employees: John, Jane, Jake, Emma

9. Convert List of Employees to a Map

Use Collectors.toMap().

Example:

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

Map<String, Integer> employeeMap = employees.stream()
    .collect(Collectors.toMap(e -> e.name, e -> e.age));


System.out.println(employeeMap);
```

Output:

{John=30, Jane=25, Jake=35, Emma=40}

Java Stream API- Assignment

Instructions:

- Solve all questions using Java Stream API.
- Do not use traditional loops (for, while).
- Use functional programming principles.

Section 1: Basic Stream Operations (10 Questions)

1. **Stream from an Array:**
    - Convert an array of integers {5, 10, 15, 20, 25} into a stream and print all elements.
2. **Stream from a List:**
    - Convert a List<String> containing names ["Alice", "Bob", "Charlie", "David"] into a stream and print each name in uppercase.
3. **Stream from a Set:**
    - Convert a Set<Integer> {3, 6, 9, 12, 15} into a stream and print all elements.
4. **Stream from a Map (Keys):**
    - Create a Map<Integer, String> with student IDs and names. Convert the keys to a stream and print them.
5. **Stream from a Map (Values):**
    - From the same map, extract and print all student names using a stream.
6. **Filtering a List:**
    - Given List<Integer> numbers = [12, 45, 67, 89, 23, 56, 78], filter out numbers greater than 50 and print them.
7. **Filtering a List of Strings:**

- Given List<String> names = ["Mike", "Michael", "John", "Jonathan", "Mona"], filter and print names starting with "M".

8. **Filtering Products (Custom Class):**
    - Given a List<Product> (with name and price fields), filter and print products that cost more than 500.
9. **Mapping Data:**
    - Convert a list of lowercase words ["java", "streams", "lambda"] to uppercase using map() and print them.
10. **Mapping Product Names:**

- Given a List<Product>, extract only product names and print them.

Section 2: Sorting and Aggregation (8 Questions)

11. **Sorting a List of Integers:**

- Sort List<Integer> numbers = [9, 3, 6, 1, 8, 4] in ascending order and print them.

12. **Sorting a List of Strings:**

- Given List<String> words = ["banana", "apple", "cherry", "date"], sort them alphabetically.

13. **Sorting Products by Price (Custom Class):**

- Given a List<Product>, sort products by price in descending order and print them.

14. **Find Maximum Price in Product List:**

- Find and print the product with the highest price.

15. **Find Minimum Price in Product List:**

- Find and print the product with the lowest price.

16. **Find Sum of All Prices in a List:**

- Given List<Product> products, find the sum of all product prices using reduce().

17. **Find Average Price of Products:**

- Given a List<Product>, calculate and print the average price of products.

18. **Find the Total Number of Products:**

- Count the number of products using count() and print the result.

Section 3: Grouping and Collecting Data (7 Questions)

19. **Get Distinct Numbers from a List:**

- Given List<Integer> numbers = [2, 4, 2, 6, 8, 4, 10, 8], remove duplicates and print the unique values.

20. **Find the Top 3 Most Expensive Products:**

- Given a List<Product>, find the three most expensive products using sorted() and limit().

21. **Concatenate Product Names with Collectors.joining():**

- Given List<Product>, concatenate product names into a single string separated by commas.

22. **Convert Product List to Map:**

- Convert a List<Product> into a Map<String, Double> where key = product name, value = product price.

23. **Group Products Based on Price (Above or Below 500):**

- Partition products into two lists: those above 500 and those 500 or cheaper.

24. **Count Occurrences of Words in a List:**

- Given List<String> words = ["apple", "banana", "apple", "orange", "banana", "apple"], count occurrences of each word.

25. **Count Even and Odd Numbers in a List:**

- Given List<Integer> numbers = [10, 21, 34, 47, 56, 63, 78], count how many numbers are even and how many are odd.