**Review of Java Collections Framework with Code Snippets**

In this session, we will review the Java Collections Framework in detail, focusing on the different types of collections, their characteristics, and the time complexities of common operations. You will also learn how to select the appropriate collection for different scenarios. This foundational understanding will help you solve real-world problems, such as managing an online ticket booking system, which we will explore in the upcoming lab tasks.

**1. Overview of the Java Collections Framework**

The Java Collections Framework (JCF) provides a set of interfaces, implementations, and algorithms to handle collections of data. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

**Key Components of the Framework:**

**Interfaces**: Define common behavior for all collection types. The most commonly used interfaces are Collection, List, Set, Queue, and Map.

**Implementations**: These are the concrete classes that provide implementations of the collection interfaces. Examples include ArrayList, LinkedList, HashSet, TreeSet, HashMap, etc.

**Algorithms**: These are static methods provided by the Collections class to perform operations like sorting, searching, reversing, etc.

**2. Core Collection Interfaces**

The primary interfaces in the Java Collections Framework include:

**a. Collection Interface**

The root interface in the framework, representing a group of objects.

Sub-interfaces: Set, List, Queue.

**b. List Interface**

Characteristics: Ordered collection that allows duplicate elements. Elements can be accessed by index.

Key Implementations: ArrayList, LinkedList, Vector.

**Code Example**: Using ArrayList to store seats:

|  |
| --- |
| import java.util.ArrayList;  public class MovieSeats {  public static void main(String[] args) {  ArrayList<String> seats = new ArrayList<>();  seats.add("Seat 1");  seats.add("Seat 2");  seats.add("Seat 3");  seats.add("Seat 4");  // Print available seats  System.out.println("Available seats: " + seats);  // Reserve a seat  seats.set(1, "Reserved");  System.out.println("Updated seats: " + seats);  }  } |

**c. Set Interface**

**Characteristics**: A collection that does not allow duplicate elements. It models the mathematical set abstraction.

Key Implementations: HashSet, LinkedHashSet, TreeSet.

**Code Example**: Using HashSet to store unique ticket types:

|  |
| --- |
| import java.util.HashSet;  public class TicketTypes {  public static void main(String[] args) {  HashSet<String> ticketTypes = new HashSet<>();  ticketTypes.add("VIP");  ticketTypes.add("Regular");  ticketTypes.add("VIP"); // Duplicate, won't be added    // Print unique ticket types  System.out.println("Unique ticket types: " + ticketTypes);  }  } |

**d. Queue Interface**

**Characteristics**: A collection used to hold elements before processing, typically in FIFO (First In, First Out) order.

Key Implementations: LinkedList, PriorityQueue, ArrayDeque.

**Code Example**: Using LinkedList as a queue:

|  |
| --- |
| import java.util.LinkedList;  import java.util.Queue;  public class CustomerQueue {  public static void main(String[] args) {  Queue<String> queue = new LinkedList<>();  queue.add("Customer 1");  queue.add("Customer 2");  queue.add("Customer 3");  // Process the first customer (FIFO)  System.out.println("Processing: " + queue.poll());  System.out.println("Remaining queue: " + queue);  }  } |

**e. Map Interface**

**Characteristics**: A collection of key-value pairs. It maps unique keys to values.

Key Implementations: HashMap, TreeMap, LinkedHashMap.

**Code Example**: Using HashMap to store customer booking information:

|  |
| --- |
| import java.util.HashMap;  import java.util.Map;  public class TicketBooking {  public static void main(String[] args) {  Map<Integer, String> bookings = new HashMap<>();  bookings.put(101, "VIP");  bookings.put(102, "Regular");  bookings.put(103, "VIP");  // Retrieve ticket type for customer 102  System.out.println("Customer 102's ticket type: " + bookings.get(102));  // Remove a booking  bookings.remove(101);  System.out.println("Updated bookings: " + bookings);  }  } |

**3. Common Java Collection Implementations**

**a. ArrayList**

A resizable array implementation of the List interface.

**Key Features:**

Supports fast random access to elements (O(1)).

Insertion and removal operations at the middle or beginning of the list are O(n) due to the need to shift elements.

**Code Example**: Adding/removing elements in ArrayList:

|  |
| --- |
| import java.util.ArrayList;  public class ArrayListExample {  public static void main(String[] args) {  ArrayList<String> list = new ArrayList<>();  list.add("Apple");  list.add("Banana");  list.add("Cherry");  System.out.println("List: " + list);  // Remove the second element  list.remove(1);  System.out.println("List after removal: " + list);  }  } |

**b. LinkedList**

A doubly linked list implementation of the List and Queue interfaces.

**Key Features**:

Insertion and removal operations are O(1) at the beginning or end of the list.

Random access is O(n), making it slower than ArrayList for accessing elements by index.

**Code Example**: Using LinkedList as both a list and a queue:

|  |
| --- |
| import java.util.LinkedList;  public class LinkedListExample {  public static void main(String[] args) {  // Using LinkedList as a List  LinkedList<String> list = new LinkedList<>();  list.add("A");  list.add("B");  list.add("C");  System.out.println("List: " + list);  // Using LinkedList as a Queue  list.add("D"); // Enqueue  System.out.println("Queue: " + list);  list.poll(); // Dequeue  System.out.println("Queue after dequeue: " + list);  }  } |

**c. HashSet**

A hash table-based implementation of the Set interface.

**Key Features:**

Guarantees no duplicate elements.

Offers O(1) time complexity for add(), remove(), and contains() operations.

**Code Example**: Storing unique elements in HashSet:

|  |
| --- |
| import java.util.HashSet;  public class HashSetExample {  public static void main(String[] args) {  HashSet<String> set = new HashSet<>();  set.add("Dog");  set.add("Cat");  set.add("Dog"); // Duplicate element, won't be added  System.out.println("Set: " + set);  }  } |

**d. HashMap**

A hash table-based implementation of the Map interface.

**Key Features**:

Allows fast lookups (O(1) average case).

No ordering guarantees unless you use a LinkedHashMap.

**Code Example**: Storing key-value pairs in HashMap:

|  |
| --- |
| import java.util.HashMap;  public class HashMapExample {  public static void main(String[] args) {  HashMap<Integer, String> map = new HashMap<>();  map.put(1, "Apple");  map.put(2, "Banana");  map.put(3, "Cherry");  System.out.println("Value for key 2: " + map.get(2));  map.remove(1);  System.out.println("Map after removal: " + map);  }  } |

**e. PriorityQueue**

A queue implementation where each element is associated with a priority. Elements are processed in order of priority.

**Code Example**: Using PriorityQueue to prioritize elements:

|  |
| --- |
| import java.util.PriorityQueue;  public class PriorityQueueExample {  public static void main(String[] args) {  PriorityQueue<Integer> queue = new PriorityQueue<>();  queue.add(3);  queue.add(1);  queue.add(2);  // The elements are processed in the natural order (ascending)  System.out.println("Priority Queue: " + queue);    System.out.println("Processing: " + queue.poll()); // Processes 1 (lowest)  }  } |

**4. Big-O Notation and Time Complexity of Common Operations**

|  |  |  |
| --- | --- | --- |
| **Collection Type** | **Operation** | **Time Complexity** |
| ArrayList | Access (get) | O(1) |
| Insert (at end) | O(1) |
| Insert (at middle) | O(n) |
| Remove (at middle) | O(n) |
| LinkedList | Access (get) | O(n) |
| Insert (at end/beginning) | O(1) |
| Remove (at end/beginning) | O(1) |
| HashSet | Add/Remove/Search | O(1) |
| HashMap | Put/Get/Remove | O(1) |
| PriorityQueue | Insert/Remove (poll) | O(log n) |

**Practice Tasks**

**Task 1: ArrayList Implementation**

Create an ArrayList to represent a list of available movie seats in a theater. Implement methods to reserve a seat, cancel a reservation, and check the availability of a specific seat.

Code Snippet: Refer to the example for ArrayList above.

**Task 2: LinkedList Queue**

Implement a queue of customers using LinkedList. Simulate a customer service system where customers are processed in the order they arrive (FIFO).

Code Snippet: Refer to the example for LinkedList as a queue above.

**Task 3: HashMap for Ticket Booking**

Use a HashMap to store customer booking information. Each customer has a unique ID and a ticket type (e.g., VIP or Regular).

Code Snippet: Refer to the example for HashMap above.

**Task 4: PriorityQueue for VIP Customers**

Simulate a ticket booking system where VIP customers are prioritized over regular customers using PriorityQueue.

Code Snippet: Refer to the example for PriorityQueue above.