Coding Cheat Sheet

This reading provides a reference list of code that you'll encounter as you work with object-oriented coding in Java. Understanding these concepts will help you write and debug your Java programs. Let's explore the following Java coding concepts:

- Java Collections Framework (JCF)
- Working with Lists
- HashSet and TreeSet
- Implementing queues in Java
- Using HashMap and TreeMap
- Using Java collections in the real world

Keep this summary reading available as a reference as you progress through your course, and refer to this reading as you begin object-oriented coding with Java after this course!

Java Collections Framework (JCF)

Using an ArrayList array

Description	Example
Import `ArrayList` and `List` from the java.util package to use dynamic lists.	`import java.util.ArrayList;` `import java.util.List;`

Define a class `ListExample` that contains the Java `main` method. Create a `List` of type `String` using the `ArrayList` implementation. This list will store fruit names as string elements. Add elements "Apple", "Banana", and "Cherry" to the list. Print the entire list, showing its elements in the order they were added. Retrieve the first element `Apple` from the list using index 0. Print the retrieved element.	`public class ListExample {`
	`public static void main(String[] args) {`
	`List<String> fruits = new ArrayList<>();`
	`fruits.add("Apple");`
	`fruits.add("Banana");`
	`fruits.add("Cherry");`
	`System.out.println("Fruits: " + fruits);`
	`String firstFruit = fruits.get(0);`
	`System.out.println("First fruit: " + firstFruit);`

Close curly braces to end the `ListExample` class definition.	`}` `}`

Explanation: This Java program demonstrates how to use the `List` interface with an `ArrayList` implementation to store and manipulate a list of fruit names. `ArrayList` is a dynamic array-based implementation of `List`, allowing for flexible resizing. Elements are added in order and accessed using a zero-based index. The `get(index)` method retrieves elements at specific positions.

Using a LinkedList array

Description	Example
Import the `LinkedList` class from the java.util package to use a linked list.	`import java.util.LinkedList;`

Define a class `LinkedListExample` that contains the Java `main` method. Create a `LinkedList` of type `String` to store animal names. Add elements "Dog", "Cat", and "Elephant" to the list. Print the contents of the `LinkedList`, displaying all elements.	`public class LinkedListExample {`
	`public static void main(String[] args) {`
	`LinkedList<String> animals = new LinkedList<>();`
	`animals.add("Dog");`
	`animals.add("Cat");`

Description	Example
	`animals.add("Elephant");` `System.out.println("Animals: " + animals);`
Close curly braces to end the `LinkedListExample` class definition.	`}` `}`

Explanation: This Java program demonstrates how to use a `LinkedList` to store and manipulate a list of animal names. `LinkedList` is a doubly linked list implementation in Java, meaning that elements are linked using pointers. In `LinkedList`, insertions and deletions are faster compared to `ArrayList` (especially for large lists).

Using a HashSet collection

Description	Example
Import the `HashSet` class from the java.util package to store a collection of unique elements.	`import java.util.HashSet;`

Define a class `HashSetExample` that contains the Java `main` method. Create a `HashSet` of type `String` to store color names. Add elements "Red", "Green", and "Blue" to the HashSet. Add "Red" again to the HashSet. HashSet does not allow duplicate values. If a duplicate is added, it is ignored. Print the contents of the `HashSet`, displaying all elements.	`public class HashSetExample {`
	`public static void main(String[] args) {`
	`HashSet<String> colors = new HashSet<>();`
	`colors.add("Red");`
	`colors.add("Green");`
	`animals.add("Blue");`
	`colors.add("Red");`
	`System.out.println("Colors: " + colors);`

Close curly braces to end the `HashSetExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `HashSet`, which is a part of the Java Collections Framework and is used to store a collection of unique elements. `HashSet` does not maintain any specific order and ignores duplicates. It is useful when you need a collection of distinct elements with fast lookup times.

Using a HashMap collection

Description	Example
Import the `HashMap` class from the java.util package to store key-value pairs.	`import java.util.HashMap;`

Description	Example

Define a class `HashMapExample` that contains the Java `main` method. Create a `HashMap<String, Integer>` named `ageMap`. The keys are `names` (String), and the values are `ages` (Integer). Add key-value pair to the `HashMap` using the `put()` method. The `System.out.println()` statement prints the entire `HashMap` but does not maintain any order because `HashMap` does not maintain insertion order. The program retrieves Alice's age using `ageMap.get("Alice")` and stores it in `aliceAge`.	`public class HashMapExample {`
	`  public static void main(String[] args) {`
	`  HashMap<String, Integer> ageMap = new HashMap<>();`
	`  ageMap.put("Alice", 30);`
	`  ageMap.put("Bob", 25);`
	`  ageMap.put("Charlie", 35);`
	`  System.out.println("Age Map: " + ageMap);`
	`  int aliceAge = ageMap.get("Alice");`
	`  System.out.println("Alice's Age: " + aliceAge);`

Close curly braces to end the `HashMapExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates the usage of a `HashMap`, which is a part of the Java Collections Framework and is used to store key-value pairs. Keys are unique (if a duplicate key is added, it replaces the old value). Values can be duplicated. `HashMap` does not maintain any specific order. It provides fast access to values using keys.

Working with lists

Creating an ArrayList

Description	Example
Import `ArrayList` from the java.util package to use dynamic lists.	`import java.util.ArrayList;`

Define a class `ArrayListExample` that contains the Java `main` method. Create an `ArrayList<String>` named `fruits` to store a list of fruit names. This list will store fruit names as string elements. Add elements "Apple", "Banana", and "Cherry" to the list. Print the entire list, showing its elements in the order they were added. Retrieve the first element `Apple` from the list using index 0 and print the retrieved element. Call `fruits.remove("Banana")` to remove "Banana" from the list. Print the remaining elements of `ArrayList`.	`public class ArrayListExample {`
	`  public static void main(String[] args) {`
	`  ArrayList<String> fruits = new ArrayList<>();`
	`  fruits.add("Apple");`
	`  fruits.add("Banana");`
	`  fruits.add("Cherry");`
	`  System.out.println("First fruit: " + fruits.get(0));`
	`  fruits.remove("Banana");`
	`  System.out.println("Fruits List: " + fruits);`

Close curly braces to end the `ArrayListExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates the usage of an `ArrayList`, which is a part of the Java Collections Framework and is used to store a resizable list of elements. `ArrayList` elements are added in order and accessed using a zero-based index. The `get(index)` method retrieves elements at specific positions. `ArrayList` allows duplicates and removing elements shifts subsequent elements left (affecting performance for large lists).

Creating a LinkedList

Description	Example
Import the `LinkedList` class from the java.util package to create a linked list.	`import java.util.LinkedList;`

Define a class `LinkedListExample` that contains the Java `main` method. Create a `LinkedList` of type `String` to store a list of color names. Add elements "Red", "Green", and "Blue" to the list using the `add()` method. Retrieve the first element of the list using `colors.get(0)`. Remove the first occurrence of "Green" from the list using `colors.remove("Green")`. Print the remaining elements of the `LinkedList`.	`public class LinkedListExample {`
	`  public static void main(String[] args) {`
	`  LinkedList<String> colors = new LinkedList<>();`
	`  colors.add("Red");`
	`  colors.add("Green");`
	`  animals.add("Blue");`
	`  System.out.println("First color: " + colors.get(0));`
	`  colors.remove("Green");`
	`  System.out.println("Colors List: " + colors);`

Close curly braces to end the LinkedListExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates the usage of a `LinkedList`, which is a part of the Java Collections Framework. `LinkedList` stores elements in nodes, where each node contains a reference to the next node. It allows efficient insertion and removal of elements from both ends: `addFirst()`, `addLast()`, `removeFirst()`, and `removeLast()`. Accessing elements by index `get(index)` is slower than in `ArrayList`, because it requires traversing the list from the beginning. Duplicates are allowed, and order is maintained. Unlike ArrayList, elements are not shifted after removal (only the references are updated), which can improve performance for certain operations.

HashSet and TreeSet

Creating a HashSet

Description	Example
Import the `HashSet` class from the java.util package to store a collection of unique elements.	`import java.util.HashSet;`

Define a class `HashSetExample` that contains the Java `main` method. Create a `HashSet` of type `String` to store fruit names. Add elements "Apple", "Banana", and "Cherry" to the HashSet. Add "Banana" again to the `HashSet`. Since `HashSet` does not allow duplicate values, it is ignored. Print the contents of the `HashSet`, displaying all elements. Checks if "Apple" is in the set by calling `fruits.contains("Apple")`. If found, the message "Apple" is present in the set is printed. The method `fruits.remove("Cherry")` removes "Cherry" from the set.	`public class HashSetExample {`
	`  public static void main(String[] args) {`
	`  HashSet<String> fruits = new HashSet<>();`
	`  fruits.add("Apple");`
	`  fruits.add("Banana");`
	`  fruits.add("Cherry");`
	`  fruits.add("Banana");`
	`  System.out.println("Fruits in the HashSet: " + fruits);`
	`  if (fruits.contains("Apple")) {`
	`  System.out.println("Apple is present in the set.");`
	`  }`
	`  fruits.remove("Cherry");`
	`  System.out.println("After removal: " + fruits);`

Description	Example
Close curly braces to end the `HashSetExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `HashSet`, which is a part of the Java Collections Framework and is used to store a collection of unique elements. The `contains()` method provides fast lookup to check if an element exists. The `remove()` method efficiently removes elements. `HashSet` does not maintain any specific order and ignores duplicates. It is useful when you need a collection of distinct elements with fast lookup times.

Creating a TreeSet

Description	Example
Import the `TreeSet` class from the java.util package to store a collection of unique elements.	`import java.util.TreeSet;`

Define a class `TreeSetExample` that contains the Java `main` method. Create a `TreeSet<Integer>` named `numbers` to store a set of integer values. Add the numbers 5, 3, 8, and 1 using the `add()` method. Add 3 again to the `TreeSet`. Since `TreeSet` does not allow duplicate values, it is ignored. Print the contents of the `TreeSet`, displaying all elements. Checks if 5 is in the set by calling `numbers.contains(5)`. If found, the message "5 is present in the set" is printed. The method `numbers.remove(8)` removes 8 from the set.	`public class TreeSetExample {`
	`public static void main(String[] args) {`
	`TreeSet<Integer> numbers = new TreeSet<>();`
	`numbers.add(5);`
	`numbers.add(3);`
	`numbers.add(8);`
	`numbers.add(1);`
	`numbers.add(3);`
	`System.out.println("Numbers in the TreeSet: " + numbers););`
	`if (numbers.contains(5)) {`
	`System.out.println("5 is present in the set.");`
	`}`
	`numbers.remove(8);`
	`System.out.println("After removal: " + numbers);`

Close curly braces to end the `HashSetExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `TreeSet`, which is used to store a collection of unique elements. `TreeSet` elements are always sorted in ascending order. The `contains()` method provides fast lookup (uses a Balanced Tree structure). The `remove()` methokd efficiently deletes elements while maintaining order. `TreeSet` is useful when you need a sorted set with fast operations.

TreeSet versus HashSet: need for order

Description	Example

Use TreeSet: When you need the elements to be sorted in a specific order. For example: If you want to store a list of student grades and display them in ascending order, a TreeSet will automatically sort them.	`TreeSet<Integer> grades = new TreeSet<>();`
	`grades.add(85);`
	`grades.add(90);`
	`grades.add(70);`
	`// Output: [70, 85, 90]`
	`System.out.println(grades);`

Description	Example

Use HashSet: When the order of elements does not matter. For example: If you are storing unique user IDs and do not care about their order.	`HashSet<String> userIds = new HashSet<>();`
	`userIds.add("user1");`
	`userIds.add("user2");`
	`userIds.add("user3");`
	`// Output: Order may vary`
	`System.out.println(userIds);`

HashSet versus TreeSet: Need for performance

Description	Example

Use HashSet: For faster performance when adding, removing, or searching for elements. For Example: In a game, if you need to quickly check if a player has collected a unique item.	`HashSet<String> collectedItems = new HashSet<>();`
	`collectedItems.add("Sword");`
	`collectedItems.add("Shield");`
	`<boolean>boolean hasSword = collectedItems.contains("Sword"); // Fast check`

Use TreeSet: Use TreeSet: When you can afford slower operations but need the elements sorted. For Example: If you are maintaining a leaderboard that requires sorted scores, a TreeSet is suitable even if it's slightly slower.	`TreeSet<Integer> scores = new TreeSet<>();`
	`scores.add(300);`
	`scores.add(150);`
	`scores.add(200);`
	`System.out.println(scores); // [150, 200, 300]`

HashSet versus TreeSet: Avoidance of duplicates

Description	Example

Using HashSet to avoid duplicates. A `HashSet<String>` named `fruits` is created. "Apple" and "Banana" are added. A duplicate "Apple" is added but ignored because `HashSet` does not allow duplicates. The output may appear as [Banana, Apple] or [Apple, Banana], but the order is NOT guaranteed, since `HashSet` is unordered.	`HashSet<String> fruits = new HashSet<>();`
	`fruits.add("Apple");`
	`fruits.add("Banana");`
	`fruits.add("Apple"); // Duplicate will not be added`
	`System.out.println(fruits); // Output: [Banana, Apple] </String>`

Using TreeSet to avoid duplicates. A `TreeSet<String>` named `sortedFruits` is created. "Apple" and "Banana" are added. A duplicate "Apple" is added but ignored because `TreeSet` also does not allow duplicates. Unlike `HashSet`, `TreeSet` automatically sorts elements in ascending order. The output is always [Apple, Banana], since `TreeSet` maintains sorted order.	`TreeSet<String> sortedFruits = new TreeSet<>();`
	`sortedFruits.add("Apple");`
	`sortedFruits.add("Banana");`
	`sortedFruits.add("Apple"); // Duplicate will not be added`
	`System.out.println(sortedFruits); // Output: [Apple, Banana]`

Implementing queues in Java

Creating a simple queue using LinkedList

Description	Example
Import the java.util.LinkedList and java.util.Queue packages to use the `Queue` interface with a `LinkedList` implementation.	`import java.util.LinkedList;` `import java.util.Queue;`

Create an instance of `Queue<String>` named `queue` using new `LinkedList<>()`. Add three elements ("Apple", "Banana", "Cherry") to the queue using `offer()`, which inserts elements at the end of the queue. Print the queue to show its contents. The `poll()` method removes and returns the front element ("Apple") from the queue. Print the removed element ("Apple") and display the state of the queue again after removing the front element.	`public class QueueExample {`
	`public static void main(String[] args) {`
	`// Creating a Queue`
	`Queue<String> queue = new LinkedList<>();`
	`// Enqueue operation`
	`queue.offer("Apple");`
	`queue.offer("Banana");`
	`queue.offer("Cherry");`
	`// Displaying the Queue`
	`System.out.println("Queue: " + queue);`
	`// Dequeue operation`
	`String removedItem = queue.poll();`
	`System.out.println("Removed Item: " + removedItem);`
	`// Displaying the Queue after Dequeue`
	`System.out.println("Queue after dequeue: " + queue);`

Close curly braces to end the `QueueExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the use of a Queue data structure using the `LinkedList` class. The method `offer()` adds an element to the queue (enqueue), `poll()` removes and returns the front element (dequeue). `LinkedList` as a queue implements FIFO (First-In-First-Out) behavior.

Creating a priority queue

Description	Example
Import the java.util.PriorityQueue package to use the PriorityQueue class.	`import java.util.PriorityQueue;`

Create an instance of `PriorityQueue<Integer>` named `priorityQueue` using new `PriorityQueue<>()`. Add three elements: 20, 15, and 30 using the `offer()` method. The `PriorityQueue` maintains a min-heap structure (smallest element has the highest priority). Print the queue, but its order may not be in the exact insertion order due to the heap-based priority structure. Remove elements in priority order (ascending order for integers). A while loop continuously removes and prints the smallest element until the queue is empty.	`public class PriorityQueueExample {`
	`public static void main(String[] args) {`
	`PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();`
	`// Adding elements`
	`priorityQueue.offer(20);`
	`priorityQueue.offer(15);`
	`priorityQueue.offer(30);`
	`// Displaying the Priority Queue`
	`System.out.println("Priority Queue: " + priorityQueue);`
	`// Removing elements in priority order`
	`while (!priorityQueue.isEmpty()) {`
	`System.out.println("Removed Item: " + priorityQueue.poll());`
	`}`

Description	Example
Close curly braces to end the `PriorityQueueExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a PriorityQueue, which is a type of queue where elements are processed based on their priority (natural order by default for numbers). The method `offer()` adds an element to the queue (enqueue), `poll()` removes and returns the element with the highest priority (smallest number in this case). Heap-based Implementation ensures efficient insertions and deletions.

Implementing a queue in the real world

Description	Example
Import the java.util.LinkedList and java.util.Queue packages to create and manage the queue.	`import java.util.LinkedList;` `import java.util.Queue;`

Create an instance of `Queue<String>` named `customerQueue` using new `LinkedList<>()` to store customers. Add "Customer 1", "Customer 2", and "Customer 3" are added to the queue using `offer()`. Prints the queue to show the customers waiting in order. The `poll()` method removes and returns the first customer ("Customer 1") from the queue. Display the remaining customers in the queue. Cal `poll()` again to serve the next customer and print the final state of the queue.	`public class CustomerServiceQueue {`
	`public static void main(String[] args) {`
	`// Creating a queue to represent customers waiting for service`
	`Queue<String> customerQueue = new LinkedList<>();`
	`// Customers arrive and join the queue`
	`customerQueue.offer("Customer 1");`
	`customerQueue.offer("Customer 2");`
	`customerQueue.offer("Customer 3");`
	`// Displaying the current queue`
	`System.out.println("Current Customer Queue: " + customerQueue);`
	`// Serving the first customer in the queue`
	`String servedCustomer = customerQueue.poll();`
	`System.out.println("Serving: " + servedCustomer);`
	`// Displaying the queue after serving one customer`
	`System.out.println("Customer Queue after serving one: " + customerQueue);`
	`// Serving another customer`
	`servedCustomer = customerQueue.poll();`
	`System.out.println("Serving: " + servedCustomer);`
	`// Final state of the queue`
	`System.out.println("Final Customer Queue: " + customerQueue);`

Close curly braces to end the `CustomerServiceQueue` class definition.	`}` `}`

Explanation: This Java program simulates a customer service queue using a Queue (FIFO - First In, First Out) implemented with a `LinkedList`. It models how customers arrive, wait, and are served in order. The method `offer()` adds customers to the queue and `poll()` removes customers in FIFO order. `LinkedList` as a queue mimics a real world waiting line. This approach can be extended to simulate bank queues, call centers, or ticket counters.

Using HashMap and TreeMap

Creating a HashMap

Description	Example
Import the `HashMap` class from the java.util package, which is a part of Java's Collection Framework.	`import java.util.HashMap;`

Initialize a `HashMap<String, Integer>` named `map` to represent `fruit names` as keys and their corresponding `numeric values` as values. Add key-value pairs using the `put` method. "Apple" is mapped to 1, "Banana" to 2, and "Cherry" to 3. Keys are unique: If the same key is added again, its value gets updated. The `map.get("Apple")` method fetches and prints the value associated with "Apple". The `keySet()` method returns all the keys in the `HashMap`, and the `for` loop prints each key-value pair. Order is NOT guaranteed in a `HashMap`. The `containsKey()` method checks whether "Banana" is present in the map and the `remove()` method deletes "Cherry" from the `HashMap`.	`public class HashMapExample {`
	`  public static void main(String[] args) {`
	`  // Creating a HashMap`
	`  HashMap<String, Integer> map = new HashMap<>();`
	`  // Adding key-value pairs to the HashMap`
	`  map.put("Apple", 1);`
	`  map.put("Banana", 2);`
	`  map.put("Cherry", 3);`
	`  // Accessing values`
	`  System.out.println("Value for key 'Apple': " + map.get("Apple")); // Output: 1`
	`  // Iterating through the HashMap`
	`  for (String key : map.keySet()) {`
	`  System.out.println(key + ": " + map.get(key));`
	`  }`
	`  // Checking if a key exists`
	`  if (map.containsKey("Banana")) {`
	`  System.out.println("Banana exists in the map.");`
	`  }`
	`  // Removing a key-value pair`
	`  map.remove("Cherry");`

Close curly braces to end the `TreeMapExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates the usage of a `HashMap`, a data structure that stores key-value pairs and allows fast access to values using keys. `put(K key, V value)` adds or updates a key-value pair, `get(K key)` retrieves the value for a key, `keySet()` returns all keys, `containsKey(K key)` checks if a key exists, and `remove(K key)` deletes a key-value pair.

Using a HashMap

Description	Example

Initialize a `HashMap<String, Integer>` named `wordCount`, where the keys are `words` (Strings) and the values are the `count of occurrences of each word` (Integers). Define the input text containing a string with multiple repeated words. The `split()` method splits the text string into a `words` array based on spaces. A `for` loop iterates over each word in the `words` array. The `wordCount.getOrDefault(word, 0)` method retrieves the current count of the word if it exists. If the word is not yet in the map, it defaults to 0. The +1 increments the count for each occurrence and `put(word, newCount)` updates the count in the `HashMap`.	`HashMap<String, Integer> wordCount = new HashMap<>();`
	`String text = "apple banana apple orange banana apple";`
	`String[] words = text.split(" ");`
	
	`for (String word : words) {`
	`  wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);`
	`}`

Explanation: This Java code snippet demonstrates how to use a HashMap to count the occurrences of words in a given text string. This approach is useful for word frequency analysis in text processing. The `split(" ")` function splits text into words. `HashMap` efficiently tracks word occurrences. `getOrDefault(key, defaultValue)` avoids null values.

Creating a TreeMap

Description	Example
Import the `TreeMap` class from the java.util package to store key-value pairs in sorted order.	`import java.util.TreeMap;`

Description	Example

Initialize a `TreeMap<String, Integer>` named `treeMap` to store fruit names (keys) and their corresponding values (integers). The `TreeMap` automatically sorts the keys in ascending order (Apple → Banana → Cherry). The `treeMap.get("Apple"))` call fetches and prints the value associated with "Apple". The `for` loop calls the `keySet()` method to iterate over all keys (which are sorted) and print their associated values. The `containsKey()` method checks if "Cherry" is present and prints a message. The `treemap.remove()` method removes the "Banana" entry from the TreeMap.	`public class TreeMapExample {`
	`  public static void main(String[] args) {`
	`  // Creating a TreeMap`
	`  TreeMap<String, Integer> treeMap = new TreeMap<>();`
	`  // Adding key-value pairs to the TreeMap`
	`  treeMap.put("Banana", 2);`
	`  treeMap.put("Apple", 1);`
	`  treeMap.put("Cherry", 3);`
	`  // Accessing values`
	`  System.out.println("Value for key 'Apple': " + treeMap.get("Apple")); // Output: 1`
	
	`  // Iterating through the TreeMap`
	`  for (String key : treeMap.keySet()) {`
	`  System.out.println(key + ": " + treeMap.get(key));`
	`  }`
	`  // Checking if a key exists`
	`  if (treeMap.containsKey("Cherry")) {`
	`  System.out.println("Cherry exists in the TreeMap.");`
	`  }`
	`  // Removing a key-value pair`
	`  treeMap.remove("Banana");`

Close curly braces to end the `TreeMapExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates the use of a TreeMap, a data structure that stores key-value pairs in sorted order based on keys. `TreeMap` maintains sorted order (ascending by default).

Using a TreeMap

Description	Example

Initialize a `TreeMap<String, Integer>` named `leaderboard` where Keys (String) represent `player names` and Values (Integer) represent `player scores`. `TreeMap` automatically sorts keys in ascending order. Add three players and their scores to the `TreeMap`. Since `TreeMap` maintains sorted order by `key (name)`, the stored order will be: Alice → Bob → Charlie. Display the sorted leaderboard using the `keySet()` method.	`TreeMap<String, Integer> leaderboard = new TreeMap<>();`
	`leaderboard.put("Alice", 150);`
	`leaderboard.put("Bob", 200);`
	`leaderboard.put("Charlie", 100);`
	`// Displaying sorted leaderboard`
	`for (String player : leaderboard.keySet()) {`
	`  System.out.println(player + ": " + leaderboard.get(player));`
	`}`

Explanation: This Java code snippet demonstrates the use of a `TreeMap` to store and display a sorted leaderboard of players and their scores. TreeMap stores entries in key-sorted order (ascending). `put(K key, V value)` adds key-value pairs, `get(K key)` retrieves the value for a given key, and keySet() returns keys in sorted order.

Using Java collections in the real world

Managing books in a library management system

Description	Example

Import the `ArrayList` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `Library` class to represent a collection of books. The `books` variable is a private ArrayList<String>, meaning it stores book titles as strings and it cannot be accessed directly from outside the class. The `Library()` constructor initializes the books list	`import java.util.ArrayList;`
	`public class Library {`
	`  private ArrayList<String> books;`
	
	`  public Library() {`

Description	Example

when a `Library` object is created. The `addBook()` method adds a new book to the books list. The `displayBooks()` method prints all books stored in the `books` list using a for-each loop. The `main` method creates a `Library` object named `myLibrary`, adds two books: "The Great Gatsby" and "To Kill a Mockingbird", and calls the displayBooks() method to print the book list.	`        books = new ArrayList<>();`
	`    }`
	
	`    public void addBook(String book) {`
	`        books.add(book);`
	`    }`
	
	`    public void displayBooks() {`
	`        System.out.println("Books in the Library:");`
	`        for (String book : books) {`
	`            System.out.println(book);`
	`        }`
	`    }`
	
	`    public static void main(String[] args) {`
	`        Library myLibrary = new Library();`
	`        myLibrary.addBook("The Great Gatsby");`
	`        myLibrary.addBook("To Kill a Mockingbird");`
	`        myLibrary.displayBooks();`

Close curly braces to end the `main` and `Library` class definition.	`    }` `}`

Managing customer orders in an e-commerce application

Description	Example

Import the `HashMap` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `OrderManagement` class to manage orders. The `orders` variable is private, meaning it cannot be accessed directly from outside the class. It is encapsulated to ensure data integrity. The Java constructor `OrderManagement()` initializes the orders `HashMap` when an `OrderManagement` object is created. The `addOrder()` method adds a new order using the `.put(orderId, customerName)` method. If the same `orderId` is added again, it overwrites the previous entry. The `displayOrders()` method iterates over the `HashMap` using `keySet()` to get all order IDs, retrieves and prints the corresponding customer names. The `main` method creates an instance of `OrderManagement`, adds two orders: Order #101 for Alice and Order #102 for Bob, and calls the `displayOrders()` method to show all orders.	`import java.util.HashMap;`
	
	`public class OrderManagement {`
	`    private HashMap<Integer, String> orders;`
	
	`    public OrderManagement() {`
	`        orders = new HashMap<>();`
	`    }`
	
	`    public void addOrder(int orderId, String customerName) {`
	`        orders.put(orderId, customerName);`
	`    }`
	
	`    public void displayOrders() {`
	`        System.out.println("Customer Orders:");`
	`        for (int orderId : orders.keySet()) {`
	`            System.out.println("Order ID: " + orderId + ", Customer Name: " + orders.get(orderId));`
	`        }`
	`    }`
	
	`    public static void main(String[] args) {`
	`        OrderManagement orderManagement = new OrderManagement();`
	`        orderManagement.addOrder(101, "Alice");`
	`        orderManagement.addOrder(102, "Bob");`
	`        orderManagement.displayOrders();`

Close curly braces to end the `main` and `OrderManagement` class definition.	`    }` `}`

Explanation: This Java program implements a basic Order Management system using a HashMap to store and manage customer orders. The program uses `HashMap<Integer, String>`, which stores Keys (Integer) to represent Order IDs and Values (String) to represent Customer Names.

Managing employee information in an employee management system

Description	Example

Import the `HashSet` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `EmployeeManager` class with a private variable named `employee` that stores employee names. Encapsulation ensures the set is only modified via class methods. The constructor `EmployeeManager()` initializes the `employees` set when an `EmployeeManager` objet is created. The `addEmployee()` method adds an employee name to the `HashSet`. If the employee already exists, the `HashSet` prevents duplicate entries. The `displayEmployees()` method iterates over the `HashSet` to display all employees. The order is not guaranteed because `HashSet` does not maintain insertion order. The Java `main` method creates an instance of `EmployeeManager` and adds three employees: "John Doe", "Jane Smith", and "John Doe". Because "John Doe" is a duplicate, it is ignored by `HashSet`. Calling `displayEmployees()` showas all employees.	`import java.util.HashSet;`
	
	`public class EmployeeManager {`
	`  private HashSet<String> employees;`
	
	`  public EmployeeManager() {`
	`  employees = new HashSet<>();`
	`  }`
	
	`  public void addEmployee(String employee) {`
	`  employees.add(employee);`
	`  }`
	
	`  public void displayEmployees() {`
	`  System.out.println("Employees in the Company:");`
	`  for (String employee : employees) {`
	`  System.out.println(employee);`
	`  }`
	`  }`
	
	`  public static void main(String[] args) {`
	`  EmployeeManager manager = new EmployeeManager();`
	`  manager.addEmployee("John Doe");`
	`  manager.addEmployee("Jane Smith");`
	`  manager.addEmployee("John Doe"); // Duplicate will not be added`
	`  manager.displayEmployees();`

Close curly braces to end the `main` and `EmployeeManager` class definition.	`  }` `}`

Explanation: This Java program implements a basic Employee Management system using a HashSet to store and manage employee names. It uses `LinkedHashSet` to maintain insertion order and `TreeSet` to store employees in sorted order.

Managing tasks in a task management system

Description	Example

Import the `LinkedList` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `TaskManager` class with a private variable named `tasks` that stores tasks. Encapsulation ensures the set is only modified via class methods. The constructor `TaskManager()` initializes the `tasks` list when a `TaskManager` object is created. The `addTask()` method adds a task to the end of the list using `add()` and preserves the insertion order (LinkedList maintains order). The `completeTask()` method removes the first task using `removeFirst()`, prevents errors by checking `isEmpty()` before removal, and printes the completed task. The `displayTasks()` method iterates over the `LinkedList` and prints all taks. Tasks remain ordered by insertion. The Java `main` method creates an instance of `TaskManager`, adds two tasks: "Finish report" and "Email client", displays tasks, completes the first task, and displays remaining tasks.	`import java.util.LinkedList;`
	
	`public class TaskManager {`
	`  private LinkedList<String> tasks;`
	
	`  public TaskManager() {`
	`  tasks = new LinkedList<>();`
	`  }`
	
	`  public void addTask(String task) {`
	`  tasks.add(task);`
	`  }`
	
	`  public void completeTask() {`
	`  if (!tasks.isEmpty()) {`
	`  String completedTask = tasks.removeFirst();`
	`  System.out.println("Completed Task: " + completedTask);`
	`  } else {`
	`  System.out.println("No tasks to complete.");`
	`  }`
	`  }`
	
	`  public void displayTasks() {`
	`  System.out.println("Current Tasks:");`
	`  for (String task : tasks) {`
	`  System.out.println(task);`
	`  }`
	`  }`
	
	`  public static void main(String[] args) {`
	`  TaskManager manager = new TaskManager();`
	`  manager.addTask("Finish report");`
	`  manager.addTask("Email client");`
	`  manager.displayTasks();`
	
	`  manager.completeTask();`
	`  manager.displayTasks();`

Description	Example
Close curly braces to end the `main` and `TaskManager` class definition.	`    }` `}`

Explanation: This Java program implements a simple Task Manager using a LinkedList to store and manage tasks. It supports fast insertions/removals at both ends for addFirst() and removeFirst().

Managing followers in a social media application

Description	Example

Import the `HashSet` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `SocialMedia` class with a `HashMap` where Key (String) represents a user and Value (HashSet<String>) stores a set of followers (ensuring uniqueness). The constructor `SocialMedia()` initializes `userFollowers` as an empty `HashMap`. The `addFoower()` method ensures the user exists in the `HashMap` using the `putIfAbsent(user, new HashSet<>())` method and adds the follower to the user's `HashSet` (no duplicates allowed). The `displayFollowers()` method checks if the user exists, prints all followers of the user, and handles missing users by displaying "No followers found". The Java `main` method creates an instance of `SocialMedia` class, adds followers: "Bob" follows "Alice", "Charlie" follows "Alice", and displays Alice's followers.	`import java.util.HashSet;`
	
	`public class SocialMedia {`
	`    private HashMap<String, HashSet<String>> userFollowers;`
	
	`    public SocialMedia() {`
	`        userFollowers = new HashMap<>();`
	`    }`
	
	`    public void addFollower(String user, String follower) {`
	`        userFollowers.putIfAbsent(user, new HashSet<>());`
	`        userFollowers.get(user).add(follower);`
	`    }`
	
	`    public void displayFollowers(String user) {`
	`        System.out.println("Followers of " + user + ":");`
	`        HashSet<String> followers = userFollowers.get(user);`
	`        if (followers != null) {`
	`            for (String follower : followers) {`
	`                System.out.println(follower);`
	`            }`
	`        } else {`
	`            System.out.println("No followers found.");`
	`        }`
	`    }`
	
	`    public static void main(String[] args) {`
	`        SocialMedia socialMedia = new SocialMedia();`
	`        socialMedia.addFollower("Alice", "Bob");`
	`        socialMedia.addFollower("Alice", "Charlie");`
	`        socialMedia.displayFollowers("Alice");`

Close curly braces to end the `main` and `EmployeeManager` class definition.	`    }` `}`

Explanation: This Java program implements a basic social media follower system using `HashMap` and `HashSet`. `HashSet` ensures no user follows the same person twice. If a user has no followers, it prints "No followers found". HashMap provides average time complexity for lookups. Followers cannot be accessed directly, only via class methods.

Author(s)

Ramanujam Srinivasan
Lavanya Thiruvali Sunderarajan