**What are collections in java?**

In Java, a collection is a **framework** or a data structure that is used to store and manipulate a group of objects.

**Collections** provide a more flexible and efficient way to work with groups of elements compared to **arrays**, as they can dynamically grow or shrink in size and offer various operations and algorithms for common data manipulation tasks.

You can store any type in a collection (int, double, String, Product…).

The Java Collections Framework is a set of **classes and interfaces** provided by the Java Standard Library to work with collections.

It includes various **interfaces**, **classes**, and **algorithms** that help you manage and manipulate data in different ways.

Some of the key interfaces and classes in the Java Collections Framework include:

1. **Collection Interface:** This is the root interface of the Java Collections Framework. It defines the basic methods for working with collections, such as adding, removing, and iterating over elements.
2. **List Interface:** Extends the Collection interface and represents an ordered collection of elements. Implementations of this interface include ArrayList, LinkedList, and Vector.
3. **Set Interface:** Represents an **unordered collection of unique elements**. Implementations of this interface include HashSet, LinkedHashSet, and TreeSet.
4. **Map Interface:** Represents a collection of key-value pairs, where each key is associated with a value. Implementations of this interface include HashMap, LinkedHashMap, and TreeMap.
5. **Queue Interface:** Represents a collection designed for holding elements prior to processing. Implementations include LinkedList and PriorityQueue.
6. **Deque Interface:** Represents a double-ended queue, which allows you to add and remove elements from both ends. LinkedList is a common implementation of this interface.
7. **ArrayList:** A commonly used implementation of the List interface that uses an array to store elements.
8. **HashSet:** A commonly used implementation of the Set interface that uses a hash table to store elements.
9. **HashMap:** A commonly used implementation of the Map interface that uses a hash table to store key-value pairs.
10. **TreeSet:** An implementation of the Set interface that stores elements in a sorted order using a Red-Black tree.
11. **TreeMap:** An implementation of the Map interface that stores key-value pairs in a sorted order using a Red-Black tree.

The Java Collections Framework provides a rich set of methods for performing operations like searching, sorting, adding, removing, and iterating over collections. It also includes utilities for handling concurrent access, synchronizing collections, and creating unmodifiable or read-only collections.

Collections are an integral part of Java programming, and they play a crucial role in various applications, from simple data storage to complex data processing and manipulation. They make it easier to work with data structures and algorithms, simplifying many common programming tasks.

**Where collections will be used?**

Collections in Java are used in a wide range of scenarios and applications where you need to manage, manipulate, and store data efficiently. They provide a flexible and convenient way to work with groups of objects. Here are some common use cases for collections in Java:

1. **Data Storage and Retrieval:** Collections are used to store and organize data, making it easy to add, retrieve, update, and remove elements. For example, you can use lists, sets, or maps to store user data, product information, or other structured data.
2. **Iterating Over Data:** Collections offer methods to iterate over their elements, making it easy to process and analyze data. You can use loops or iterators to traverse and operate on the elements.
3. **Searching and Retrieval:** Collections provide methods to search for specific elements efficiently. For example, you can use a HashSet or a HashMap to quickly find items by their keys or values.
4. **Sorting:** Collections can be sorted in a variety of ways, which is useful for tasks like displaying data in a specific order or finding the maximum or minimum value in a dataset.
5. **Stacks and Queues:** Collections like LinkedList and PriorityQueue can be used to implement data structures such as stacks and queues, which are useful for managing data in a last-in, first-out (LIFO) or first-in, first-out (FIFO) manner.
6. **Concurrency:** Collections provide concurrent versions that support thread-safe operations, which are crucial in multi-threaded applications to avoid data corruption and race conditions.
7. **Utility Operations:** Java collections offer various utility operations for performing tasks like reversing a list, joining collections, finding the intersection or union of sets, and more.
8. **Managing Key-Value Pairs:** Collections like HashMap and TreeMap are essential for storing key-value pairs, where you can associate a value with a unique key and efficiently look up values based on their keys.
9. **Custom Data Structures:** You can create custom data structures by implementing the interfaces provided by the Java Collections Framework, enabling you to design collections tailored to specific requirements.
10. **Algorithms and Data Processing:** Collections are often used in algorithms and data processing tasks, like sorting, searching, filtering, and transforming data.
11. **Caching and Memorization:** Collections can be used for caching previously computed results, which can improve the performance of applications by avoiding redundant calculations.
12. **Graphs and Trees:** Collections can be used to represent and manipulate graph structures, such as adjacency lists, and tree structures, like binary trees.
13. **Event Handling:** Collections can be used to manage event listeners and subscribers in event-driven programming.
14. **Serialization and Deserialization:** Collections are often used to store and transfer data in a serialized form, which can be later deserialized to reconstruct objects.
15. **User Interface Development:** Collections can be used to manage data in user interfaces, such as lists of items in a GUI component.

In summary, collections in Java are a fundamental part of the language, and they find application in a wide range of domains and scenarios to manage and manipulate data effectively and efficiently. The Java Collections Framework provides a rich set of tools and data structures to support these use cases.

**Collection classes and interfaces**

**Interfaces**

**Some of the key interfaces in the Collections Framework include:**

1. **Collection Interface:**
   * **java.util.Collection**: It is the root interface for the collection hierarchy. It represents a group of objects known as elements.
2. **Set Interface:**
   * **java.util.Set**: It extends the **Collection** interface and represents a collection that does not allow duplicate elements.
3. **List Interface:**
   * **java.util.List**: It extends the **Collection** interface and represents an ordered collection that allows duplicate elements.
4. **Queue Interface:**
   * **java.util.Queue**: It extends the **Collection** interface and represents a collection used for holding elements prior to processing.
5. **Map Interface:**
   * **java.util.Map**: It represents a collection of **key-value pairs**, where each key is associated with exactly one value.
6. **SortedSet Interface:**
   * **java.util.SortedSet**: It extends the **Set** interface and represents a set that is sorted in ascending order.
7. **SortedMap Interface:**
   * **java.util.SortedMap**: It extends the **Map** interface and represents a map that is sorted in ascending order of its **keys**.
8. **Deque Interface:**
   * **java.util.Deque**: It stands for double-ended queue and extends the **Queue** interface. It represents a queue where elements can be inserted and removed from both ends.

**Classes**

**Some of the commonly used classes include:**

1. **Collection Classes:**
   * **java.util.ArrayList**: Implements the **List** interface and provides a dynamic array that can grow or shrink as needed.
   * **java.util.LinkedList**: Implements the **List** interface and provides a doubly-linked list.
   * **java.util.HashSet**: Implements the **Set** interface and stores elements in a hash table, allowing for constant-time performance for basic operations.
   * **java.util.TreeSet**: Implements the **SortedSet** interface and stores elements in a sorted tree structure.
   * **java.util.PriorityQueue**: Implements the **Queue** interface and provides a priority queue based on a priority heap.
2. **Map Classes:**
   * **java.util.HashMap**: Implements the **Map** interface and uses a hash table to store key-value pairs.
   * **java.util.TreeMap**: Implements the **SortedMap** interface and stores key-value pairs in a sorted tree structure.
   * **java.util.LinkedHashMap**: Extends **HashMap** and maintains the order of the keys based on their insertion order.
   * **java.util.IdentityHashMap**: Implements the **Map** interface and compares keys by reference equality rather than content equality.
3. **Deque Classes:**
   * **java.util.ArrayDeque**: Implements the **Deque** interface and provides a resizable-array implementation of a double-ended queue.
4. **Specialized Collections:**
   * **java.util.BitSet**: Implements a set of bits or flags.
   * **java.util.Stack**: Represents a last-in, first-out (LIFO) stack of objects.
   * **java.util.Vector**: An older implementation of a dynamic array that is synchronized (thread-safe).
   * **java.util.Hashtable**: An older implementation of a hash table that is synchronized (thread-safe).

**Example 1: Using ArrayList to Store and Iterate Over Strings**

import java.util.ArrayList;

import java.util.List;

public class ArrayListExample {

public static void main(String[] args) {

// Create an ArrayList to store strings

List<String> stringList = new ArrayList<>();

// Add elements to the list

stringList.add("Apple");

stringList.add("Banana");

stringList.add("Cherry");

// Iterate over the elements using a for-each loop

for (String fruit : stringList) {

System.out.println(fruit);

}

}

}

**Example 2: Using HashMap to Store Key-Value Pairs**

import java.util.HashMap;

import java.util.Map;

public class HashMapExample {

public static void main(String[] args) {

// Create a HashMap to store key-value pairs

Map<String, Integer> ageMap = new HashMap<>();

// Add key-value pairs to the map

ageMap.put("Alice", 25);

ageMap.put("Bob", 30);

ageMap.put("Charlie", 28);

// Retrieve values by key

int bobAge = ageMap.get("Bob");

System.out.println("Bob's age is " + bobAge);

// Iterate over the key-value pairs

for (Map.Entry<String, Integer> entry : ageMap.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

**Example 3: Using HashSet for Unique Elements**

import java.util.HashSet;

import java.util.Set;

public class HashSetExample {

public static void main(String[] args) {

// Create a HashSet to store unique elements

Set<String> uniqueSet = new HashSet<>();

// Add elements to the set

uniqueSet.add("Apple");

uniqueSet.add("Banana");

uniqueSet.add("Apple"); // Duplicate element will be ignored

// Iterate over the unique elements

for (String fruit : uniqueSet) {

System.out.println(fruit);

}

}

}

**Example 4: Using LinkedList as a Queue**

import java.util.LinkedList;

import java.util.Queue;

public class LinkedListQueueExample {

public static void main(String[] args) {

// Create a LinkedList to use as a queue

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

// Enqueue (add) elements to the queue

queue.offer("First");

queue.offer("Second");

queue.offer("Third");

// Dequeue (remove) elements from the queue

while (!queue.isEmpty()) {

System.out.println("Dequeued: " + queue.poll());

}

}

}

**Example 5: Using TreeMap for Sorting**

import java.util.Map;

import java.util.TreeMap;

public class TreeMapExample {

public static void main(String[] args) {

// Create a TreeMap to store and sort key-value pairs

Map<String, Integer> scoreMap = new TreeMap<>();

// Add key-value pairs to the map

scoreMap.put("Alice", 85);

scoreMap.put("Bob", 92);

scoreMap.put("Charlie", 78);

// Iterate over the sorted entries

for (Map.Entry<String, Integer> entry : scoreMap.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

**Example 6: Using Collections.sort(list) for Sorting a List (Collections.sort(list, Collections.reverseOrder()).**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class CollectionsSortExample {

public static void main(String[] args) {

// Create a list of integers

List<Integer> numbers = new ArrayList<>();

numbers.add(5);

numbers.add(2);

numbers.add(8);

numbers.add(1);

// Sort the list in ascending order

Collections.sort(numbers);

// Iterate over the sorted list

for (int number : numbers) {

System.out.println(number);

}

}

}

**Sorting a class**

**public** **class** Student **implements** Comparable<Student> {

**private** **long** id;

**private** String name;

**private** String address;

**public** Student() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** Student(**long** id, String name, String address) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.address = address;

}

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", name=" + name + ", address=" + address + "]";

}

@Override

**public** **int** compareTo(Student s) {

// **TODO** Auto-generated method stub

**return** **this**.name.compareTo(s.name);

}

}

**public** **class** Collections011 {

**public** **static** **void** main(String[] args) {

List<Student> students = **new** ArrayList<>();

students.add(**new** Student(101, "Ramesh", "123Street"));

students.add(**new** Student(102, "Tarun", "123Street"));

students.add(**new** Student(103, "Rajesh", "123Street"));

students.add(**new** Student(104, "Anish", "123Street"));

students.add(**new** Student(105, "Divya", "123Street"));

**for**(Student student: students) {

System.***out***.println(student);

}

System.***out***.println("\nSort the students by names");

Collections.*sort*(students);

**for**(Student student: students) {

System.***out***.println(student);

}

}

}

**Example 7: Using Collections.unmodifiableList() for Creating an Unmodifiable List**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class UnmodifiableListExample {

public static void main(String[] args) {

// Create a mutable list

List<String> mutableList = new ArrayList<>();

mutableList.add("Apple");

mutableList.add("Banana");

// Make the list unmodifiable

List<String> unmodifiableList = Collections.unmodifiableList(mutableList);

// Attempt to modify the unmodifiable list will result in an exception

// unmodifiableList.add("Cherry"); // This line will throw an UnsupportedOperationException

// You can still read from the unmodifiable list

for (String fruit : unmodifiableList) {

System.out.println(fruit);

}

}

}

**Example 8: Using a Stack**

import java.util.Stack;

public class StackExample {

public static void main(String[] args) {

// Create a stack to implement a LIFO structure

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

// Push elements onto the stack

stack.push("First");

stack.push("Second");

stack.push("Third");

// Pop elements from the stack

while (!stack.isEmpty()) {

System.out.println("Popped: " + stack.pop());

}

}

}

**Example 9: Using ArrayList and Stream for Filtering Data using lambda operator(->).**

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

public class ArrayListFilterExample {

public static void main(String[] args) {

// Create an ArrayList with integers

List<Integer> numbers = new ArrayList<>();

numbers.add(5);

numbers.add(2);

numbers.add(8);

numbers.add(1);

numbers.add(8);

// Use a stream to filter the list

List<Integer> filteredNumbers = numbers.stream()

.filter(n -> n > 3)

.collect(Collectors.toList());

// Print the filtered numbers

System.out.println("Filtered Numbers: " + filteredNumbers);

}

}

**Example 10: Using TreeSet for Maintaining a Sorted Set**

import java.util.Set;

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

// Create a TreeSet to store elements in sorted order

Set<Integer> numberSet = new TreeSet<>();

// Add elements to the set

numberSet.add(5);

numberSet.add(2);

numberSet.add(8);

numberSet.add(1);

// Iterate over the sorted set

for (int number : numberSet) {

System.out.println(number);

}

}

}

**Example 11: Using ConcurrentHashMap for Thread-Safe Map**

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {

public static void main(String[] args) {

// Create a thread-safe map

Map<String, Integer> concurrentMap = new ConcurrentHashMap<>();

// Add key-value pairs to the map

concurrentMap.put("Alice", 25);

concurrentMap.put("Bob", 30);

concurrentMap.put("Charlie", 28);

// Perform thread-safe updates

concurrentMap.compute("Alice", (key, value) -> value + 1);

// Iterate over the map

concurrentMap.forEach((key, value) -> {

System.out.println(key + ": " + value);

});

}

}

**Example 12: Employee Management System Using ArrayList and HashMap**

This example demonstrates a simple employee management system where employees can be added, retrieved, and listed using ArrayList and HashMap.

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class Employee {

private String id;

private String name;

private String department;

public Employee(String id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

public String getId() {

return id;

}

public String getName() {

return name;

}

public String getDepartment() {

return department;

}

@Override

public String toString() {

return "Employee{id='" + id + "', name='" + name + "', department='" + department + "'}";

}

}

public class EmployeeManagementSystem {

private List<Employee> employeeList;

private Map<String, Employee> employeeMap;

public EmployeeManagementSystem() {

employeeList = new ArrayList<>();

employeeMap = new HashMap<>();

}

public void addEmployee(Employee employee) {

employeeList.add(employee);

employeeMap.put(employee.getId(), employee);

}

public Employee getEmployeeById(String id) {

return employeeMap.get(id);

}

public List<Employee> getAllEmployees() {

return new ArrayList<>(employeeList);

}

public static void main(String[] args) {

EmployeeManagementSystem ems = new EmployeeManagementSystem();

Employee e1 = new Employee("1", "Alice", "HR");

Employee e2 = new Employee("2", "Bob", "IT");

Employee e3 = new Employee("3", "Charlie", "Finance");

ems.addEmployee(e1);

ems.addEmployee(e2);

ems.addEmployee(e3);

System.out.println("Employee with ID 2: " + ems.getEmployeeById("2"));

System.out.println("All Employees: " + ems.getAllEmployees());

}

}

**Example 13: E-commerce Order Processing Using LinkedList and TreeMap**

This example simulates an order processing system for an e-commerce application using LinkedList to handle order queues and TreeMap to store products with their prices.

import java.util.LinkedList;

import java.util.Map;

import java.util.Queue;

import java.util.TreeMap;

public class Product {

private String name;

private double price;

public Product(String name, double price) {

this.name = name;

this.price = price;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + "}";

}

}

public class Order {

private int orderId;

private String customerName;

private Map<Product, Integer> productQuantities;

public Order(int orderId, String customerName) {

this.orderId = orderId;

this.customerName = customerName;

this.productQuantities = new TreeMap<>((p1, p2) -> p1.getName().compareTo(p2.getName()));

}

public void addProduct(Product product, int quantity) {

productQuantities.put(product, productQuantities.getOrDefault(product, 0) + quantity);

}

public Map<Product, Integer> getProductQuantities() {

return productQuantities;

}

@Override

public String toString() {

return "Order{orderId=" + orderId + ", customerName='" + customerName + "', products=" + productQuantities + "}";

}

}

public class EcommerceOrderProcessing {

private Queue<Order> orderQueue;

private Map<String, Product> productCatalog;

public EcommerceOrderProcessing() {

orderQueue = new LinkedList<>();

productCatalog = new TreeMap<>();

}

public void addProductToCatalog(Product product) {

productCatalog.put(product.getName(), product);

}

public void placeOrder(Order order) {

orderQueue.offer(order);

}

public void processOrders() {

while (!orderQueue.isEmpty()) {

Order order = orderQueue.poll();

System.out.println("Processing order: " + order);

double totalAmount = 0;

for (Map.Entry<Product, Integer> entry : order.getProductQuantities().entrySet()) {

totalAmount += entry.getKey().getPrice() \* entry.getValue();

}

System.out.println("Total amount for order " + order.getOrderId() + ": $" + totalAmount);

}

}

public static void main(String[] args) {

EcommerceOrderProcessing eop = new EcommerceOrderProcessing();

Product p1 = new Product("Laptop", 999.99);

Product p2 = new Product("Phone", 499.99);

Product p3 = new Product("Tablet", 299.99);

eop.addProductToCatalog(p1);

eop.addProductToCatalog(p2);

eop.addProductToCatalog(p3);

Order o1 = new Order(1, "Alice");

o1.addProduct(p1, 1);

o1.addProduct(p3, 2);

Order o2 = new Order(2, "Bob");

o2.addProduct(p2, 3);

eop.placeOrder(o1);

eop.placeOrder(o2);

eop.processOrders();

}

}

**Example 14: Library Management System Using HashMap and TreeSet**

This example demonstrates a simple library management system using HashMap to manage books and TreeSet to keep track of available book titles in a sorted order.

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

import java.util.TreeSet;

public class Book {

private String title;

private String author;

private String isbn;

public Book(String title, String author, String isbn) {

this.title = title;

this.author = author;

this.isbn = isbn;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public String getIsbn() {

return isbn;

}

@Override

public String toString() {

return "Book{title='" + title + "', author='" + author + "', isbn='" + isbn + "'}";

}

}

public class LibraryManagementSystem {

private Map<String, Book> booksByIsbn;

private Set<String> availableBookTitles;

public LibraryManagementSystem() {

booksByIsbn = new HashMap<>();

availableBookTitles = new TreeSet<>();

}

public void addBook(Book book) {

booksByIsbn.put(book.getIsbn(), book);

availableBookTitles.add(book.getTitle());

}

public Book getBookByIsbn(String isbn) {

return booksByIsbn.get(isbn);

}

public Set<String> getAvailableBookTitles() {

return availableBookTitles;

}

public void removeBook(String isbn) {

Book book = booksByIsbn.remove(isbn);

if (book != null) {

availableBookTitles.remove(book.getTitle());

}

}

public static void main(String[] args) {

LibraryManagementSystem lms = new LibraryManagementSystem();

Book b1 = new Book("1984", "George Orwell", "123456789");

Book b2 = new Book("To Kill a Mockingbird", "Harper Lee", "987654321");

Book b3 = new Book("The Great Gatsby", "F. Scott Fitzgerald", "111222333");

lms.addBook(b1);

lms.addBook(b2);

lms.addBook(b3);

System.out.println("Available Book Titles: " + lms.getAvailableBookTitles());

System.out.println("Details of book with ISBN 123456789: " + lms.getBookByIsbn("123456789"));

lms.removeBook("987654321");

System.out.println("Available Book Titles after removal: " + lms.getAvailableBookTitles());

}

}

**Example 15: Chat Application Using ConcurrentHashMap and CopyOnWriteArrayList**

This example simulates a chat application where multiple users can send and receive messages in a thread-safe manner using ConcurrentHashMap and CopyOnWriteArrayList.

import java.util.List;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.CopyOnWriteArrayList;

class User {

private String username;

private List<String> messages;

public User(String username) {

this.username = username;

this.messages = new CopyOnWriteArrayList<>();

}

public String getUsername() {

return username;

}

public List<String> getMessages() {

return messages;

}

public void sendMessage(String message) {

messages.add(message);

}

@Override

public String toString() {

return "User{username='" + username + "', messages=" + messages + "}";

}

}

public class ChatApplication {

private Map<String, User> users;

public ChatApplication() {

users = new ConcurrentHashMap<>();

}

public void addUser(String username) {

users.put(username, new User(username));

}

public User getUser(String username) {

return users.get(username);

}

public void sendMessage(String fromUser, String toUser, String message) {

User sender = users.get(fromUser);

User receiver = users.get(toUser);

if (sender != null && receiver != null) {

String formattedMessage = fromUser + ": " + message;

sender.sendMessage("You: " + message);

receiver.sendMessage(formattedMessage);

}

}

public static void main(String[] args) {

ChatApplication chatApp = new ChatApplication();

chatApp.addUser("Alice");

chatApp.addUser("Bob");

chatApp.sendMessage("Alice", "Bob", "Hello, Bob!");

chatApp.sendMessage("Bob", "Alice", "Hi, Alice!");

System.out.println("Alice's messages: " + chatApp.getUser("Alice").getMessages());

System.out.println("Bob's messages: " + chatApp.getUser("Bob").getMessages());

}

}

**Example 16: Social Media Feed Using PriorityQueue and HashSet**

This example simulates a social media feed where posts are displayed based on their timestamp using PriorityQueue and unique posts are ensured using HashSet.

import java.util.HashSet;

import java.util.PriorityQueue;

import java.util.Set;

class Post implements Comparable<Post> {

private String user;

private String content;

private long timestamp;

public Post(String user, String content, long timestamp) {

this.user = user;

this.content = content;

this.timestamp = timestamp;

}

public String getUser() {

return user;

}

public String getContent() {

return content;

}

public long getTimestamp() {

return timestamp;

}

@Override

public int compareTo(Post other) {

return Long.compare(other.timestamp, this.timestamp); // Newer posts come first

}

@Override

public String toString() {

return "Post{user='" + user + "', content='" + content + "', timestamp=" + timestamp + "}";

}

}

public class SocialMediaFeed {

private PriorityQueue<Post> feed;

private Set<String> postsSet;

public SocialMediaFeed() {

feed = new PriorityQueue<>();

postsSet = new HashSet<>();

}

public void addPost(Post post) {

if (postsSet.add(post.getContent())) { // Ensures unique content

feed.offer(post);

}

}

public void displayFeed() {

while (!feed.isEmpty()) {

System.out.println(feed.poll());

}

}

public static void main(String[] args) {

SocialMediaFeed smf = new SocialMediaFeed();

Post p1 = new Post("Alice", "Hello, world!", System.currentTimeMillis());

Post p2 = new Post("Bob", "Just had lunch!", System.currentTimeMillis() - 1000);

Post p3 = new Post("Alice", "Hello, world!", System.currentTimeMillis() - 500); // Duplicate content

smf.addPost(p1);

smf.addPost(p2);

smf.addPost(p3);

smf.displayFeed();

}

}

### Example 17: Inventory Management Using HashMap and LinkedList

This example simulates an inventory management system where items are added, sold, and tracked using HashMap for inventory and LinkedList for sale history.

import java.util.HashMap;

import java.util.LinkedList;

import java.util.Map;

import java.util.Queue;

class Item {

private String name;

private double price;

public Item(String name, double price) {

this.name = name;

this.price = price;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

@Override

public String toString() {

return "Item{name='" + name + "', price=" + price + "}";

}

}

public class InventoryManagement {

private Map<String, Integer> inventory;

private Queue<Item> salesHistory;

public InventoryManagement() {

inventory = new HashMap<>();

salesHistory = new LinkedList<>();

}

public void addItem(String itemName, int quantity) {

inventory.put(itemName, inventory.getOrDefault(itemName, 0) + quantity);

}

public void sellItem(Item item) {

String itemName = item.getName();

if (inventory.getOrDefault(itemName, 0) > 0) {

inventory.put(itemName, inventory.get(itemName) - 1);

salesHistory.offer(item);

System.out.println("Sold: " + item);

} else {

System.out.println("Item out of stock: " + itemName);

}

}

public void displayInventory() {

System.out.println("Inventory: " + inventory);

}

public void displaySalesHistory() {

System.out.println("Sales History:");

for (Item item : salesHistory) {

System.out.println(item);

}

}

public static void main(String[] args) {

InventoryManagement im = new InventoryManagement();

Item i1 = new Item("Laptop", 999.99);

Item i2 = new Item("Phone", 499.99);

Item i3 = new Item("Tablet", 299.99);

im.addItem("Laptop", 10);

im.addItem("Phone", 20);

im.addItem("Tablet", 15);

im.sellItem(i1);

im.sellItem(i2);

im.sellItem(i3);

im.sellItem(i1);

im.displayInventory();

im.displaySalesHistory();

}

}

### Example 18: School Management System Using ArrayList, HashMap, and TreeSet

This example simulates a school management system where students, teachers, and courses are managed. It uses ArrayList to manage lists of students and teachers, HashMap to map students to courses, and TreeSet to maintain sorted lists of courses.

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.TreeSet;

class Person {

private String id;

private String name;

public Person(String id, String name) {

this.id = id;

this.name = name;

}

public String getId() {

return id;

}

public String getName() {

return name;

}

@Override

public String toString() {

return "Person{id='" + id + "', name='" + name + "'}";

}

}

class Student extends Person {

public Student(String id, String name) {

super(id, name);

}

}

class Teacher extends Person {

public Teacher(String id, String name) {

super(id, name);

}

}

class Course implements Comparable<Course> {

private String code;

private String title;

public Course(String code, String title) {

this.code = code;

this.title = title;

}

public String getCode() {

return code;

}

public String getTitle() {

return title;

}

@Override

public int compareTo(Course other) {

return this.code.compareTo(other.code);

}

@Override

public String toString() {

return "Course{code='" + code + "', title='" + title + "'}";

}

}

public class SchoolManagementSystem {

private List<Student> students;

private List<Teacher> teachers;

private Map<Student, Set<Course>> studentCourses;

private Set<Course> courses;

public SchoolManagementSystem() {

students = new ArrayList<>();

teachers = new ArrayList<>();

studentCourses = new HashMap<>();

courses = new TreeSet<>();

}

public void addStudent(Student student) {

students.add(student);

}

public void addTeacher(Teacher teacher) {

teachers.add(teacher);

}

public void addCourse(Course course) {

courses.add(course);

}

public void enrollStudentInCourse(Student student, Course course) {

studentCourses.computeIfAbsent(student, k -> new TreeSet<>()).add(course);

}

public List<Student> getAllStudents() {

return new ArrayList<>(students);

}

public List<Teacher> getAllTeachers() {

return new ArrayList<>(teachers);

}

public Set<Course> getAllCourses() {

return new TreeSet<>(courses);

}

public Set<Course> getCoursesForStudent(Student student) {

return studentCourses.getOrDefault(student, new TreeSet<>());

}

public static void main(String[] args) {

SchoolManagementSystem sms = new SchoolManagementSystem();

Student s1 = new Student("S001", "Alice");

Student s2 = new Student("S002", "Bob");

Teacher t1 = new Teacher("T001", "Dr. Smith");

Teacher t2 = new Teacher("T002", "Prof. Johnson");

Course c1 = new Course("C001", "Mathematics");

Course c2 = new Course("C002", "Physics");

Course c3 = new Course("C003", "Chemistry");

sms.addStudent(s1);

sms.addStudent(s2);

sms.addTeacher(t1);

sms.addTeacher(t2);

sms.addCourse(c1);

sms.addCourse(c2);

sms.addCourse(c3);

sms.enrollStudentInCourse(s1, c1);

sms.enrollStudentInCourse(s1, c2);

sms.enrollStudentInCourse(s2, c2);

sms.enrollStudentInCourse(s2, c3);

System.out.println("All Students: " + sms.getAllStudents());

System.out.println("All Teachers: " + sms.getAllTeachers());

System.out.println("All Courses: " + sms.getAllCourses());

System.out.println("Courses for Alice: " + sms.getCoursesForStudent(s1));

System.out.println("Courses for Bob: " + sms.getCoursesForStudent(s2));

}

}

### Example 19: Bank System Using ConcurrentHashMap and LinkedBlockingQueue

This example simulates a bank system where customers can create accounts, deposit, withdraw, and transfer money using ConcurrentHashMap for account management and LinkedBlockingQueue for transaction processing.

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.LinkedBlockingQueue;

class Account {

private String accountId;

private String owner;

private double balance;

public Account(String accountId, String owner, double initialBalance) {

this.accountId = accountId;

this.owner = owner;

this.balance = initialBalance;

}

public String getAccountId() {

return accountId;

}

public String getOwner() {

return owner;

}

public double getBalance() {

return balance;

}

public synchronized void deposit(double amount) {

balance += amount;

}

public synchronized boolean withdraw(double amount) {

if (amount <= balance) {

balance -= amount;

return true;

}

return false;

}

@Override

public String toString() {

return "Account{accountId='" + accountId + "', owner='" + owner + "', balance=" + balance + "}";

}

}

class Transaction {

private String type;

private String accountId;

private double amount;

public Transaction(String type, String accountId, double amount) {

this.type = type;

this.accountId = accountId;

this.amount = amount;

}

public String getType() {

return type;

}

public String getAccountId() {

return accountId;

}

public double getAmount() {

return amount;

}

@Override

public String toString() {

return "Transaction{type='" + type + "', accountId='" + accountId + "', amount=" + amount + "}";

}

}

public class BankSystem {

private Map<String, Account> accounts;

private LinkedBlockingQueue<Transaction> transactionQueue;

public BankSystem() {

accounts = new ConcurrentHashMap<>();

transactionQueue = new LinkedBlockingQueue<>();

}

public void createAccount(String accountId, String owner, double initialBalance) {

accounts.put(accountId, new Account(accountId, owner, initialBalance));

}

public void processTransactions() {

while (!transactionQueue.isEmpty()) {

Transaction transaction = transactionQueue.poll();

if (transaction != null) {

Account account = accounts.get(transaction.getAccountId());

if (account != null) {

switch (transaction.getType()) {

case "DEPOSIT":

account.deposit(transaction.getAmount());

break;

case "WITHDRAW":

account.withdraw(transaction.getAmount());

break;

}

System.out.println("Processed: " + transaction);

System.out.println("Updated Account: " + account);

}

}

}

}

public void addTransaction(Transaction transaction) {

transactionQueue.offer(transaction);

}

public static void main(String[] args) {

BankSystem bank = new BankSystem();

bank.createAccount("A001", "Alice", 1000.0);

bank.createAccount("A002", "Bob", 500.0);

bank.addTransaction(new Transaction("DEPOSIT", "A001", 200.0));

bank.addTransaction(new Transaction("WITHDRAW", "A002", 150.0));

bank.addTransaction(new Transaction("DEPOSIT", "A002", 300.0));

bank.processTransactions();

}

}

**Example 20: Online Store Using TreeMap, HashSet, and ArrayList**

This example simulates an online store where products, customers, and orders are managed using TreeMap for sorted product listings, HashSet for unique customer IDs, and ArrayList for order management.

import java.util.ArrayList;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.TreeMap;

class Product {

private String productId;

private String name;

private double price;

public Product(String productId, String name, double price) {

this.productId = productId;

this.name = name;

this.price = price;

}

public String getProductId() {

return productId;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

@Override

public String toString() {

return "Product{productId='" + productId + "', name='" + name + "', price=" + price + "}";

}

}

class Customer {

private String customerId;

private String name;

public Customer(String customerId, String name) {

this.customerId = customerId;

this.name = name;

}

public String getCustomerId() {

return customerId;

}

public String getName() {

return name;

}

@Override

public String toString() {

return "Customer{customerId='" + customerId + "', name='" + name + "'}";

}

}

class Order {

private String orderId;

private Customer customer;

private List<Product> products;

private double totalAmount;

public Order(String orderId, Customer customer) {

this.orderId = orderId;

this.customer = customer;

this.products = new ArrayList<>();

this.totalAmount = 0.0;

}

public String getOrderId() {

return orderId;

}

public Customer getCustomer() {

return customer;

}

public List<Product> getProducts() {

return products;

}

public double getTotalAmount() {

return totalAmount;

}

public void addProduct(Product product) {

products.add(product);

totalAmount += product.getPrice();

}

@Override

public String toString() {

return "Order{orderId='" + orderId + "', customer=" + customer + ", products=" + products + ", totalAmount=" + totalAmount + "}";

}

}

public class OnlineStore {

private Map<String, Product> products;

private Set<String> customers;

private List<Order> orders;

public OnlineStore() {

products = new TreeMap<>();

customers = new HashSet<>();

orders = new ArrayList<>();

}

public void addProduct(Product product) {

products.put(product.getProductId(), product);

}

public void addCustomer(Customer customer) {

customers.add(customer.getCustomerId());

}

public void placeOrder(Order order) {

if (customers.contains(order.getCustomer().getCustomerId())) {

orders.add(order);

System.out.println("Order placed: " + order);

} else {

System.out.println("Customer not found: " + order.getCustomer().getCustomerId());

}

}

public List<Product> listAllProducts() {

return new ArrayList<>(products.values());

}

public List<Order> listAllOrders() {

return new ArrayList<>(orders);

}

public static void main(String[] args) {

OnlineStore store = new OnlineStore();

Product p1 = new Product("P001", "Laptop", 999.99);

Product p2 = new Product("P002", "Phone", 599.99);

Product p3 = new Product("P003", "Tablet", 299.99);

store.addProduct(p1);

store.addProduct(p2);

store.addProduct(p3);

Customer c1 = new Customer("C001", "Alice");

Customer c2 = new Customer("C002", "Bob");

store.addCustomer(c1);

store.addCustomer(c2);

Order o1 = new Order("O001", c1);

o1.addProduct(p1);

o1.addProduct(p2);

Order o2 = new Order("O002", c2);

o2.addProduct(p3);

store.placeOrder(o1);

store.placeOrder(o2);

System.out.println("All Products: " + store.listAllProducts());

System.out.println("All Orders: " + store.listAllOrders());

}

}

### Example 21: Library Management System

Here is more comprehensive example of a Library Management System.

This system will include features for managing books, members, and transactions such as borrowing and returning books.

It will use a combination of Java Collections and Object-Oriented Programming principles.

#### Book Class

public class Book {

private String isbn;

private String title;

private String author;

private int copiesAvailable;

public Book(String isbn, String title, String author, int copiesAvailable) {

this.isbn = isbn;

this.title = title;

this.author = author;

this.copiesAvailable = copiesAvailable;

}

public String getIsbn() {

return isbn;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public int getCopiesAvailable() {

return copiesAvailable;

}

public void borrowBook() {

if (copiesAvailable > 0) {

copiesAvailable--;

} else {

throw new IllegalStateException("No copies available");

}

}

public void returnBook() {

copiesAvailable++;

}

@Override

public String toString() {

return "Book{" +

"isbn='" + isbn + '\'' +

", title='" + title + '\'' +

", author='" + author + '\'' +

", copiesAvailable=" + copiesAvailable +

'}';

}

}

**Member Class**

import java.util.HashSet;

import java.util.Set;

public class Member {

private String memberId;

private String name;

private Set<Book> borrowedBooks;

public Member(String memberId, String name) {

this.memberId = memberId;

this.name = name;

this.borrowedBooks = new HashSet<>();

}

public String getMemberId() {

return memberId;

}

public String getName() {

return name;

}

public Set<Book> getBorrowedBooks() {

return borrowedBooks;

}

public void borrowBook(Book book) {

book.borrowBook();

borrowedBooks.add(book);

}

public void returnBook(Book book) {

if (borrowedBooks.remove(book)) {

book.returnBook();

} else {

throw new IllegalStateException("Book not borrowed by this member");

}

}

@Override

public String toString() {

return "Member{" +

"memberId='" + memberId + '\'' +

", name='" + name + '\'' +

", borrowedBooks=" + borrowedBooks +

'}';

}

}

**Library Class**

import java.util.HashMap;

import java.util.HashSet;

import java.util.Map;

import java.util.Set;

public class Library {

private Map<String, Book> books;

private Map<String, Member> members;

public Library() {

books = new HashMap<>();

members = new HashMap<>();

}

public void addBook(Book book) {

books.put(book.getIsbn(), book);

}

public void addMember(Member member) {

members.put(member.getMemberId(), member);

}

public Book getBook(String isbn) {

return books.get(isbn);

}

public Member getMember(String memberId) {

return members.get(memberId);

}

public void borrowBook(String memberId, String isbn) {

Member member = getMember(memberId);

Book book = getBook(isbn);

if (member != null && book != null) {

member.borrowBook(book);

} else {

throw new IllegalStateException("Member or Book not found");

}

}

public void returnBook(String memberId, String isbn) {

Member member = getMember(memberId);

Book book = getBook(isbn);

if (member != null && book != null) {

member.returnBook(book);

} else {

throw new IllegalStateException("Member or Book not found");

}

}

public Set<Book> listAvailableBooks() {

Set<Book> availableBooks = new HashSet<>();

for (Book book : books.values()) {

if (book.getCopiesAvailable() > 0) {

availableBooks.add(book);

}

}

return availableBooks;

}

public Set<Member> listMembers() {

return new HashSet<>(members.values());

}

public static void main(String[] args) {

Library library = new Library();

Book book1 = new Book("123456", "Java Programming", "Author A", 5);

Book book2 = new Book("789101", "Data Structures", "Author B", 3);

Book book3 = new Book("112131", "Algorithms", "Author C", 2);

library.addBook(book1);

library.addBook(book2);

library.addBook(book3);

Member member1 = new Member("M001", "Alice");

Member member2 = new Member("M002", "Bob");

library.addMember(member1);

library.addMember(member2);

library.borrowBook("M001", "123456");

library.borrowBook("M001", "789101");

library.borrowBook("M002", "112131");

System.out.println("Available Books: " + library.listAvailableBooks());

System.out.println("Members: " + library.listMembers());

library.returnBook("M001", "123456");

System.out.println("Available Books after return: " + library.listAvailableBooks());

}

}

### Breakdown of Features

* **Book Class**:
  + Holds details of the book, including ISBN, title, author, and the number of copies available.
  + Methods for borrowing and returning books, ensuring the number of available copies is correctly updated.
* **Member Class**:
  + Holds details of the member, including ID, name, and a set of borrowed books.
  + Methods for borrowing and returning books, updating both the member's borrowed books and the book's available copies.
* **Library Class**:
  + Manages collections of books and members.
  + Methods for adding books and members, borrowing and returning books, and listing available books and members.
  + Ensures transactions are valid and updates the state of books and members appropriately.

### Execution

Running the main method in the Library class will:

1. Create a library instance.
2. Add books and members to the library.
3. Simulate borrowing and returning books by members.
4. Display available books and members before and after transactions.

This system showcases an extensive example of object-oriented design, encapsulation, and usage of various Java collections for efficient data management and operations.

### Example 22: Bank Management System

Here we'll implement a simple **Bank Management System**.

The system will include features for managing accounts, transactions (deposit and withdrawal), and viewing account details.

It will make use of Java Collections and OOP principles.

**Account Class**

import java.util.ArrayList;

import java.util.List;

public class Account {

private String accountId;

private String accountHolderName;

private double balance;

private List<String> transactions;

public Account(String accountId, String accountHolderName, double initialDeposit) {

this.accountId = accountId;

this.accountHolderName = accountHolderName;

this.balance = initialDeposit;

this.transactions = new ArrayList<>();

this.transactions.add("Initial deposit: $" + initialDeposit);

}

public String getAccountId() {

return accountId;

}

public String getAccountHolderName() {

return accountHolderName;

}

public double getBalance() {

return balance;

}

public List<String> getTransactions() {

return transactions;

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

transactions.add("Deposited: $" + amount);

} else {

throw new IllegalArgumentException("Deposit amount must be positive");

}

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

transactions.add("Withdrew: $" + amount);

} else {

throw new IllegalArgumentException("Invalid withdrawal amount");

}

}

@Override

public String toString() {

return "Account{" +

"accountId='" + accountId + '\'' +

", accountHolderName='" + accountHolderName + '\'' +

", balance=" + balance +

'}';

}

}

**Bank Class**

import java.util.HashMap;

import java.util.Map;

public class Bank {

private Map<String, Account> accounts;

public Bank() {

accounts = new HashMap<>();

}

public void createAccount(String accountId, String accountHolderName, double initialDeposit) {

if (!accounts.containsKey(accountId)) {

Account account = new Account(accountId, accountHolderName, initialDeposit);

accounts.put(accountId, account);

System.out.println("Account created: " + account);

} else {

throw new IllegalArgumentException("Account ID already exists");

}

}

public Account getAccount(String accountId) {

return accounts.get(accountId);

}

public void deposit(String accountId, double amount) {

Account account = getAccount(accountId);

if (account != null) {

account.deposit(amount);

System.out.println("Deposited $" + amount + " into account: " + accountId);

} else {

throw new IllegalArgumentException("Account not found");

}

}

public void withdraw(String accountId, double amount) {

Account account = getAccount(accountId);

if (account != null) {

account.withdraw(amount);

System.out.println("Withdrew $" + amount + " from account: " + accountId);

} else {

throw new IllegalArgumentException("Account not found");

}

}

public void viewAccountDetails(String accountId) {

Account account = getAccount(accountId);

if (account != null) {

System.out.println(account);

System.out.println("Transactions:");

for (String transaction : account.getTransactions()) {

System.out.println(transaction);

}

} else {

throw new IllegalArgumentException("Account not found");

}

}

public static void main(String[] args) {

Bank bank = new Bank();

// Creating accounts

bank.createAccount("A001", "John Doe", 1000.0);

bank.createAccount("A002", "Jane Smith", 1500.0);

// Depositing money

bank.deposit("A001", 500.0);

bank.deposit("A002", 200.0);

// Withdrawing money

bank.withdraw("A001", 300.0);

bank.withdraw("A002", 100.0);

// Viewing account details

bank.viewAccountDetails("A001");

bank.viewAccountDetails("A002");

}

}

**Breakdown of Features**

* **Account Class**:
  + Holds details of the account, including ID, holder's name, balance, and a list of transactions.
  + Methods for depositing and withdrawing money, updating the balance and recording the transactions.
  + Provides account details including transaction history.
* **Bank Class**:
  + Manages a collection of accounts using a HashMap.
  + Methods for creating accounts, depositing money, withdrawing money, and viewing account details.
  + Ensures transactions are valid and updates the state of accounts appropriately.

**Execution**

Running the main method in the Bank class will:

1. Create a bank instance.
2. Create two accounts with initial deposits.
3. Perform deposits and withdrawals on the accounts.
4. Display account details and transaction histories.

This system showcases an extensive example of object-oriented design, encapsulation, and usage of various Java collections for efficient data management and operations in a banking context.

**Some more examples of code on collections**

**ArrayList**

**Example 1: Create and Print an ArrayList of Strings**

import java.util.ArrayList;

public class Example1 {

public static void main(String[] args) {

ArrayList<String> fruits = new ArrayList<>();

fruits.add("Apple");

fruits.add("Mango");

fruits.add("Banana");

System.out.println("Fruits: " + fruits);

}

}

**Example 2: Access, Update, and Remove Elements**

import java.util.ArrayList;

public class Example2 {

public static void main(String[] args) {

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

numbers.add(10);

numbers.add(20);

numbers.add(30);

// Access element

System.out.println("First element: " + numbers.get(0));

// Update element

numbers.set(1, 25);

System.out.println("After update: " + numbers);

// Remove element

numbers.remove(2);

System.out.println("After removal: " + numbers);

}

}

**Example 3: Iterate Through an ArrayList**

import java.util.ArrayList;

public class Example3 {

public static void main(String[] args) {

ArrayList<String> animals = new ArrayList<>();

animals.add("Dog");

animals.add("Cat");

animals.add("Elephant");

for (String animal : animals) {

System.out.println(animal);

}

}

}

**Example 4: Check if an Element Exists**

import java.util.ArrayList;

public class Example4 {

public static void main(String[] args) {

ArrayList<String> cities = new ArrayList<>();

cities.add("New York");

cities.add("London");

cities.add("Tokyo");

String search = "London";

if (cities.contains(search)) {

System.out.println(search + " is in the list.");

} else {

System.out.println(search + " is not in the list.");

}

}

}

**Example 5: Sort an ArrayList**

import java.util.ArrayList;

import java.util.Collections;

public class Example5 {

public static void main(String[] args) {

ArrayList<String> names = new ArrayList<>();

names.add("Zara");

names.add("Amit");

names.add("John");

Collections.sort(names);

System.out.println("Sorted Names: " + names);

}

}

**LinkedList**

**Example 1: Create and Display a LinkedList of Strings**

import java.util.LinkedList;

public class LinkedListExample1 {

public static void main(String[] args) {

LinkedList<String> cars = new LinkedList<>();

cars.add("BMW");

cars.add("Tesla");

cars.add("Toyota");

System.out.println("Cars: " + cars);

}

}

**Example 2: Add Elements at First and Last Positions**

import java.util.LinkedList;

public class LinkedListExample2 {

public static void main(String[] args) {

LinkedList<String> colors = new LinkedList<>();

colors.add("Red");

colors.add("Blue");

colors.addFirst("Green");

colors.addLast("Yellow");

System.out.println("Colors: " + colors);

}

}

**Example 3: Remove Elements from LinkedList**

import java.util.LinkedList;

public class LinkedListExample3 {

public static void main(String[] args) {

LinkedList<Integer> numbers = new LinkedList<>();

numbers.add(5);

numbers.add(10);

numbers.add(15);

numbers.add(20);

numbers.remove(1); // removes element at index 1

numbers.removeFirst(); // removes the first element

numbers.removeLast(); // removes the last element (if any)

System.out.println("Numbers: " + numbers);

}

}

**Example 4: Iterate Through LinkedList**

import java.util.LinkedList;

public class LinkedListExample4 {

public static void main(String[] args) {

LinkedList<String> languages = new LinkedList<>();

languages.add("Java");

languages.add("Python");

languages.add("C++");

for (String lang : languages) {

System.out.println(lang);

}

}

}

**Example 5: Use LinkedList as a Queue**

import java.util.LinkedList;

import java.util.Queue;

public class LinkedListExample5 {

public static void main(String[] args) {

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

queue.offer("Customer1");

queue.offer("Customer2");

queue.offer("Customer3");

System.out.println("Serving: " + queue.poll()); // removes and returns head

System.out.println("Next in queue: " + queue.peek()); // views head

System.out.println("Queue: " + queue);

}

}

**HashSet**

**Example 1: Create and Display a HashSet**

import java.util.HashSet;

public class HashSetExample1 {

public static void main(String[] args) {

HashSet<String> fruits = new HashSet<>();

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Mango");

System.out.println("Fruits: " + fruits);

}

}

**Note: HashSet does not maintain order of elements.**

**Example 2: Check for Existence and Remove Elements**

import java.util.HashSet;

public class HashSetExample2 {

public static void main(String[] args) {

HashSet<Integer> numbers = new HashSet<>();

numbers.add(10);

numbers.add(20);

numbers.add(30);

System.out.println("Contains 20? " + numbers.contains(20));

numbers.remove(20);

System.out.println("After removal: " + numbers);

}

}

**Example 3: Iterate Through a HashSet**

import java.util.HashSet;

public class HashSetExample3 {

public static void main(String[] args) {

HashSet<String> countries = new HashSet<>();

countries.add("India");

countries.add("USA");

countries.add("UK");

for (String country : countries) {

System.out.println(country);

}

}

}

**Example 4: Union of Two Sets**

import java.util.HashSet;

public class HashSetExample4 {

public static void main(String[] args) {

HashSet<Integer> set1 = new HashSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

HashSet<Integer> set2 = new HashSet<>();

set2.add(3);

set2.add(4);

set2.add(5);

set1.addAll(set2); // Union

System.out.println("Union: " + set1);

}

}

**Example 5: Intersection of Two Sets**

import java.util.HashSet;

public class HashSetExample5 {

public static void main(String[] args) {

HashSet<String> setA = new HashSet<>();

setA.add("Java");

setA.add("Python");

setA.add("C++");

HashSet<String> setB = new HashSet<>();

setB.add("Python");

setB.add("Ruby");

setA.retainAll(setB); // Intersection

System.out.println("Common Elements: " + setA);

}

}

**TreeSet**

**Example 1: Create and Display a TreeSet**

import java.util.TreeSet;

public class TreeSetExample1 {

public static void main(String[] args) {

TreeSet<String> names = new TreeSet<>();

names.add("Charlie");

names.add("Alice");

names.add("Bob");

System.out.println("Sorted Names: " + names);

}

}

TreeSet stores elements in ascending order automatically.

**Example 2: Add, Remove, and Check Elements**

import java.util.TreeSet;

public class TreeSetExample2 {

public static void main(String[] args) {

TreeSet<Integer> numbers = new TreeSet<>();

numbers.add(40);

numbers.add(10);

numbers.add(30);

numbers.add(20);

numbers.remove(30);

System.out.println("Numbers: " + numbers);

System.out.println("Contains 20? " + numbers.contains(20));

}

}

**Example 3: Iterate Through TreeSet**

import java.util.TreeSet;

public class TreeSetExample3 {

public static void main(String[] args) {

TreeSet<String> countries = new TreeSet<>();

countries.add("India");

countries.add("USA");

countries.add("Brazil");

for (String country : countries) {

System.out.println(country);

}

}

}

**Example 4: Get First, Last, Higher, and Lower Elements**

import java.util.TreeSet;

public class TreeSetExample4 {

public static void main(String[] args) {

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

set.add(10);

set.add(20);

set.add(30);

set.add(40);

System.out.println("First: " + set.first());

System.out.println("Last: " + set.last());

System.out.println("Higher than 20: " + set.higher(20));

System.out.println("Lower than 30: " + set.lower(30));

}

}

**Example 5: Subset, HeadSet, and TailSet**

import java.util.TreeSet;

public class TreeSetExample5 {

public static void main(String[] args) {

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

for (int i = 10; i <= 100; i += 10) {

set.add(i);

}

System.out.println("Subset (20 to 60): " + set.subSet(20, 70));

System.out.println("HeadSet (<50): " + set.headSet(50));

System.out.println("TailSet (>=60): " + set.tailSet(60));

}

}

**PriorityQueue**

A **queue** is a linear data structure that follows the **First In First Out (FIFO)** principle. This means that elements are added to the end of the queue and removed from the front. It is commonly used in scenarios where order needs to be preserved, such as in breadth-first search algorithms.

A **priority queue**, on the other hand, is an extension of the queue where each element is associated with a priority. Elements with higher priority are served before elements with lower priority, regardless of their order in the queue. This makes priority queues suitable for scenarios where certain tasks need to be prioritized over others, such as in Dijkstra’s algorithm for finding the shortest path.

**Key Differences**

1. **Order of Processing**: Queue: Processes elements in the order they were added (FIFO). Priority Queue: Processes elements based on their priority. Higher priority elements are processed first^1^.
2. **Complexity**: Queue: Enqueue and dequeue operations are typically O(1). Priority Queue: Enqueue and dequeue operations are O(log n) when implemented using binary heaps^1^.
3. **Use Cases**: Queue: Used in breadth-first search, scheduling tasks in a round-robin fashion, etc. Priority Queue: Used in Dijkstra’s algorithm, Prim’s algorithm, CPU scheduling, etc^2^.

**Example 1: Create and Display a PriorityQueue (Natural Order)**

import java.util.PriorityQueue;

public class PriorityQueueExample1 {

public static void main(String[] args) {

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(30);

pq.add(10);

pq.add(20);

System.out.println("PriorityQueue: " + pq);

}

}

Output order might appear unordered, but the smallest element is always at the **head**.

**Example 2: Polling Elements (Min-Heap Behavior)**

import java.util.PriorityQueue;

public class PriorityQueueExample2 {

public static void main(String[] args) {

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(5);

pq.add(1);

pq.add(10);

while (!pq.isEmpty()) {

System.out.println("Polled: " + pq.poll());

}

}

}

Elements come out in ascending order: 1, 5, 10.

**Example 3: PriorityQueue with Strings**

import java.util.PriorityQueue;

public class PriorityQueueExample3 {

public static void main(String[] args) {

PriorityQueue<String> pq = new PriorityQueue<>();

pq.add("Banana");

pq.add("Apple");

pq.add("Cherry");

while (!pq.isEmpty()) {

System.out.println(pq.poll());

}

}

}

Strings are ordered alphabetically.

**Example 4: PriorityQueue with Custom Comparator (Max-Heap)**

import java.util.PriorityQueue;

import java.util.Collections;

public class PriorityQueueExample4 {

public static void main(String[] args) {

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

maxHeap.add(3);

maxHeap.add(1);

maxHeap.add(4);

while (!maxHeap.isEmpty()) {

System.out.println(maxHeap.poll()); // Output: 4, 3, 1

}

}

}

**Example 5: PriorityQueue with Custom Objects and Comparator**

import java.util.PriorityQueue;

import java.util.Comparator;

class Task {

String name;

int priority;

public Task(String name, int priority) {

this.name = name;

this.priority = priority;

}

public String toString() {

return name + " (priority " + priority + ")";

}

}

public class PriorityQueueExample5 {

public static void main(String[] args) {

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

taskQueue.add(new Task("Write report", 2));

taskQueue.add(new Task("Fix bugs", 1));

taskQueue.add(new Task("Deploy app", 3));

while (!taskQueue.isEmpty()) {

System.out.println("Processing: " + taskQueue.poll());

}

}

}

Tasks are processed in ascending priority order.

**HashMap**

**Example 1: Create and Display a HashMap**

import java.util.HashMap;

public class HashMapExample1 {

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("HashMap: " + map);

}

}

**Example 2: Access and Remove Elements**

import java.util.HashMap;

public class HashMapExample2 {

public static void main(String[] args) {

HashMap<String, Integer> scores = new HashMap<>();

scores.put("Alice", 90);

scores.put("Bob", 80);

scores.put("Charlie", 70);

System.out.println("Bob's score: " + scores.get("Bob"));

scores.remove("Charlie");

System.out.println("Updated map: " + scores);

}

}

**Example 3: Iterate Over Entries**

import java.util.HashMap;

import java.util.Map;

public class HashMapExample3 {

public static void main(String[] args) {

HashMap<String, String> countries = new HashMap<>();

countries.put("IN", "India");

countries.put("US", "United States");

countries.put("UK", "United Kingdom");

for (Map.Entry<String, String> entry : countries.entrySet()) {

System.out.println(entry.getKey() + " => " + entry.getValue());

}

}

}

**Example 4: Check Existence of Key or Value**

import java.util.HashMap;

public class HashMapExample4 {

public static void main(String[] args) {

HashMap<String, String> loginMap = new HashMap<>();

loginMap.put("admin", "1234");

loginMap.put("user", "abcd");

System.out.println("Contains key 'admin'? " + loginMap.containsKey("admin"));

System.out.println("Contains value 'abcd'? " + loginMap.containsValue("abcd"));

}

}

**Example 5: Using HashMap with Custom Objects**

import java.util.HashMap;

class Student {

int id;

String name;

Student(int id, String name) {

this.id = id;

this.name = name;

}

public String toString() {

return id + " - " + name;

}

}

public class HashMapExample5 {

public static void main(String[] args) {

HashMap<Integer, Student> studentMap = new HashMap<>();

studentMap.put(101, new Student(101, "Alice"));

studentMap.put(102, new Student(102, "Bob"));

for (Integer key : studentMap.keySet()) {

System.out.println("Key: " + key + ", Value: " + studentMap.get(key));

}

}

}

Here are **Java examples of frequency counting using HashMap** — a common pattern used to count how many times each element appears in a list, string, or array.

**Example 1: Frequency of Characters in a String**

import java.util.HashMap;

public class FrequencyExample1 {

public static void main(String[] args) {

String str = "programming";

HashMap<Character, Integer> freq = new HashMap<>();

for (char ch : str.toCharArray()) {

freq.put(ch, freq.getOrDefault(ch, 0) + 1);

}

System.out.println("Character Frequencies: " + freq);

}

}

**Example 2: Frequency of Words in a Sentence**

import java.util.HashMap;

public class FrequencyExample2 {

public static void main(String[] args) {

String sentence = "java is easy and java is powerful";

String[] words = sentence.split(" ");

HashMap<String, Integer> freq = new HashMap<>();

for (String word : words) {

freq.put(word, freq.getOrDefault(word, 0) + 1);

}

System.out.println("Word Frequencies: " + freq);

}

}

**Example 3: Frequency of Integers in an Array**

import java.util.HashMap;

public class FrequencyExample3 {

public static void main(String[] args) {

int[] nums = {1, 2, 2, 3, 1, 4, 2, 3, 1};

HashMap<Integer, Integer> freq = new HashMap<>();

for (int num : nums) {

freq.put(num, freq.getOrDefault(num, 0) + 1);

}

System.out.println("Integer Frequencies: " + freq);

}

}

**TreeMap**

**Example 1: Create and Display a TreeMap (Sorted by Keys)**

import java.util.TreeMap;

public class TreeMapExample1 {

public static void main(String[] args) {

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

map.put(3, "Banana");

map.put(1, "Apple");

map.put(2, "Cherry");

System.out.println("TreeMap (Sorted by Key): " + map);

}

}

**Output: {1=Apple, 2=Cherry, 3=Banana}**

**Example 2: Iterate Over TreeMap Entries**

import java.util.TreeMap;

import java.util.Map;

public class TreeMapExample2 {

public static void main(String[] args) {

TreeMap<String, String> countries = new TreeMap<>();

countries.put("IN", "India");

countries.put("US", "United States");

countries.put("CN", "China");

for (Map.Entry<String, String> entry : countries.entrySet()) {

System.out.println(entry.getKey() + " => " + entry.getValue());

}

}

}

Sorted by ISO country codes: CN, IN, US

**Example 3: Access First and Last Keys**

import java.util.TreeMap;

public class TreeMapExample3 {

public static void main(String[] args) {

TreeMap<Integer, String> scores = new TreeMap<>();

scores.put(10, "Poor");

scores.put(70, "Average");

scores.put(90, "Excellent");

System.out.println("Lowest score key: " + scores.firstKey());

System.out.println("Highest score key: " + scores.lastKey());

}

}

**Example 4: TreeMap with Custom Comparator (Descending Order)**

import java.util.TreeMap;

import java.util.Comparator;

public class TreeMapExample4 {

public static void main(String[] args) {

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

reverseMap.put(100, "Top");

reverseMap.put(50, "Mid");

reverseMap.put(10, "Low");

System.out.println("TreeMap (Descending): " + reverseMap);

}

}

**Example 5: Frequency Count and Sort by Key (Alphabetically)**

import java.util.TreeMap;

public class TreeMapExample5 {

public static void main(String[] args) {

String sentence = "apple banana apple orange banana apple";

String[] words = sentence.split(" ");

TreeMap<String, Integer> freq = new TreeMap<>();

for (String word : words) {

freq.put(word, freq.getOrDefault(word, 0) + 1);

}

System.out.println("Sorted Frequency Map: " + freq);

}

}

Output: {apple=3, banana=2, orange=1} (keys sorted alphabetically)

**LinkedHashMap**

**Example 1: Create and Display a LinkedHashMap**

import java.util.LinkedHashMap;

public class LinkedHashMapExample1 {

public static void main(String[] args) {

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

map.put(10, "Ten");

map.put(5, "Five");

map.put(20, "Twenty");

System.out.println("LinkedHashMap (insertion order): " + map);

}

}

Output: {10=Ten, 5=Five, 20=Twenty}

**Example 2: Iterate Over Entries**

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapExample2 {

public static void main(String[] args) {

LinkedHashMap<String, Integer> scores = new LinkedHashMap<>();

scores.put("Alice", 85);

scores.put("Bob", 90);

scores.put("Charlie", 75);

for (Map.Entry<String, Integer> entry : scores.entrySet()) {

System.out.println(entry.getKey() + " => " + entry.getValue());

}

}

}

**Example 3: Preserve Order of Words in Frequency Count**

import java.util.LinkedHashMap;

public class LinkedHashMapExample3 {

public static void main(String[] args) {

String[] words = {"apple", "banana", "apple", "orange", "banana", "apple"};

LinkedHashMap<String, Integer> freq = new LinkedHashMap<>();

for (String word : words) {

freq.put(word, freq.getOrDefault(word, 0) + 1);

}

System.out.println("Word Frequencies (in insertion order): " + freq);

}

}

**Example 4: Remove an Entry and Maintain Order**

import java.util.LinkedHashMap;

public class LinkedHashMapExample4 {

public static void main(String[] args) {

LinkedHashMap<String, String> capitals = new LinkedHashMap<>();

capitals.put("France", "Paris");

capitals.put("Germany", "Berlin");

capitals.put("Italy", "Rome");

capitals.remove("Germany");

System.out.println("After removing Germany: " + capitals);

}

}

**Example 5: Access Order Mode (Optional)**

You can use a **constructor with accessOrder=true** to reorder based on recent access:

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapExample5 {

public static void main(String[] args) {

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

map.put(1, "A");

map.put(2, "B");

map.put(3, "C");

map.get(2); // Access element 2

map.get(1); // Access element 1

System.out.println("LinkedHashMap (access order): " + map);

}

}

Output will be reordered: {3=C, 2=B, 1=A} if iterated

**IdentityHashMap**

**Example 1: Basic Usage of IdentityHashMap**

import java.util.IdentityHashMap;

public class IdentityHashMapExample1 {

public static void main(String[] args) {

IdentityHashMap<String, String> map = new IdentityHashMap<>();

String key1 = new String("Java");

String key2 = new String("Java");

map.put(key1, "Language A");

map.put(key2, "Language B");

System.out.println("Map Size: " + map.size()); // Output: 2

System.out.println(map); // Both keys are considered different

}

}

**Example 2: Compare with HashMap**

import java.util.HashMap;

import java.util.IdentityHashMap;

public class IdentityHashMapExample2 {

public static void main(String[] args) {

String a = new String("Hello");

String b = new String("Hello");

HashMap<String, String> hashMap = new HashMap<>();

hashMap.put(a, "First");

hashMap.put(b, "Second");

IdentityHashMap<String, String> identityMap = new IdentityHashMap<>();

identityMap.put(a, "First");

identityMap.put(b, "Second");

System.out.println("HashMap size: " + hashMap.size()); // 1 (uses equals())

System.out.println("IdentityHashMap size: " + identityMap.size()); // 2 (uses ==)

}

}

**Example 3: Iterating Over IdentityHashMap**

import java.util.IdentityHashMap;

import java.util.Map;

public class IdentityHashMapExample3 {

public static void main(String[] args) {

IdentityHashMap<Integer, String> map = new IdentityHashMap<>();

Integer x = new Integer(10);

Integer y = new Integer(10);

map.put(x, "Ten A");

map.put(y, "Ten B");

for (Map.Entry<Integer, String> entry : map.entrySet()) {

System.out.println(entry.getKey() + " => " + entry.getValue());

}

}

}

**Example 4: Using Same Reference**

import java.util.IdentityHashMap;

public class IdentityHashMapExample4 {

public static void main(String[] args) {

IdentityHashMap<String, Integer> map = new IdentityHashMap<>();

String key = "Key";

map.put(key, 1);

map.put(key, 2); // Overwrites because it's the same reference

System.out.println("Map: " + map); // Output: {Key=2}

}

}

**Example 5: Storing Different References with Equal Values**

import java.util.IdentityHashMap;

public class IdentityHashMapExample5 {

public static void main(String[] args) {

IdentityHashMap<Object, String> map = new IdentityHashMap<>();

Object obj1 = new String("same");

Object obj2 = new String("same");

map.put(obj1, "Object 1");

map.put(obj2, "Object 2");

System.out.println("Map contents: " + map);

}

}

Even though obj1.equals(obj2) is true, IdentityHashMap stores both.