1 **)Employee Attendance Tracker (Streams, Optional, Date/Time API)**

**Program:**

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

import java.util.\*;

import java.util.stream.Collectors;

class Employee {

private final int id;

private final String name;

private LocalDateTime checkInTime;

private LocalDateTime checkOutTime;

public Employee(int id, String name) {

this.id = id;

this.name = name;

}

public void checkIn() {

this.checkInTime = LocalDateTime.now();

System.out.println(name + " checked in at " + formatDateTime(checkInTime));

}

public void checkOut() {

this.checkOutTime = LocalDateTime.now();

System.out.println(name + " checked out at " + formatDateTime(checkOutTime));

}

public Optional<LocalDateTime> getCheckInTime() {

return Optional.ofNullable(checkInTime);

}

public Optional<LocalDateTime> getCheckOutTime() {

return Optional.ofNullable(checkOutTime);

}

public String getName() {

return name;

}

private String formatDateTime(LocalDateTime dateTime) {

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

return dateTime.format(formatter);

}

}

public class AttendanceTracker {

private final Map<Integer, Employee> employees = new HashMap<>();

public void addEmployee(int id, String name) {

employees.put(id, new Employee(id, name));

}

public void checkIn(int id) {

Optional.ofNullable(employees.get(id)).ifPresentOrElse(

Employee::checkIn,

() -> System.out.println("Employee not found."));

}

public void checkOut(int id) {

Optional.ofNullable(employees.get(id)).ifPresentOrElse(

Employee::checkOut,

() -> System.out.println("Employee not found."));

}

public void displayAttendance() {

System.out.println("\nEmployee Attendance Records:");

employees.values().stream()

.forEach(employee -> {

String checkIn = employee.getCheckInTime()

.map(time -> time.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")))

.orElse("Not Checked In");

String checkOut = employee.getCheckOutTime()

.map(time -> time.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")))

.orElse("Not Checked Out");

System.out.println(employee.getName() + " - Check-In: " + checkIn + " | Check-Out: " + checkOut);

});

}

public static void main(String[] args) {

AttendanceTracker tracker = new AttendanceTracker();

tracker.addEmployee(1, "Alice");

tracker.addEmployee(2, "Bob");

tracker.checkIn(1);

tracker.checkIn(2);

try { Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace(); }

tracker.checkOut(1);

tracker.checkOut(2);

tracker.displayAttendance();

}

}

2) **Movie Streaming Recommendation System (Lambda, Streams, Comparator)**

**Program**:

import java.util.\*;

import java.util.stream.Collectors;

class Movie {

private final String title;

private final String genre;

private final double rating;

public Movie(String title, String genre, double rating) {

this.title = title;

this.genre = genre;

this.rating = rating;

}

public String getTitle() {

return title;

}

public String getGenre() {

return genre;

}

public double getRating() {

return rating;

}

@Override

public String toString() {

return title + " (" + genre + ") - Rating: " + rating;

}

}

public class MovieRecommendationSystem {

private final List<Movie> movies;

public MovieRecommendationSystem() {

movies = new ArrayList<>();

}

public void addMovie(String title, String genre, double rating) {

movies.add(new Movie(title, genre, rating));

}

public List<Movie> recommendByGenre(String genre) {

return movies.stream()

.filter(movie -> movie.getGenre().equalsIgnoreCase(genre))

.sorted(Comparator.comparingDouble(Movie::getRating).reversed())

.collect(Collectors.toList());

}

public List<Movie> recommendTopMovies(int count) {

return movies.stream()

.sorted(Comparator.comparingDouble(Movie::getRating).reversed())

.limit(count)

.collect(Collectors.toList());

}

public static void main(String[] args) {

MovieRecommendationSystem system = new MovieRecommendationSystem();

system.addMovie("Inception", "Sci-Fi", 8.8);

system.addMovie("Interstellar", "Sci-Fi", 8.6);

system.addMovie("The Dark Knight", "Action", 9.0);

system.addMovie("Avengers: Endgame", "Action", 8.4);

system.addMovie("The Matrix", "Sci-Fi", 8.7);

System.out.println("\nRecommended Sci-Fi Movies:");

system.recommendByGenre("Sci-Fi").forEach(System.out::println);

System.out.println("\nTop 3 Movies:");

system.recommendTopMovies(3).forEach(System.out::println);

}

}

3**) Online Shopping Order Processing (Streams, Lambda, Date API, Optional)**

**Program:**

import java.util.ArrayList;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

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 OrderItem {

private Product product;

private int quantity;

public OrderItem(Product product, int quantity) {

this.product = product;

this.quantity = quantity;

}

public Product getProduct() {

return product;

}

public int getQuantity() {

return quantity;

}

public double getTotalItemPrice(){

return product.getPrice() \* quantity;

}

@Override

public String toString() {

return "OrderItem{" +

"product=" + product +

", quantity=" + quantity +

'}';

}

}

class Order {

private String orderId;

private List<OrderItem> items;

private String customerId;

public Order(String orderId, List<OrderItem> items, String customerId) {

this.orderId = orderId;

this.items = items;

this.customerId = customerId;

}

public String getOrderId() {

return orderId;

}

public List<OrderItem> getItems() {

return items;

}

public String getCustomerId() {

return customerId;

}

public double getTotalOrderPrice(){

return items.stream().mapToDouble(OrderItem::getTotalItemPrice).sum();

}

@Override

public String toString() {

return "Order{" +

"orderId='" + orderId + '\'' +

", items=" + items +

", customerId='" + customerId + '\'' +

'}';

}

}

public class OnlineShoppingOrderProcessing {

public static void main(String[] args) {

List<Product> products = new ArrayList<>();

products.add(new Product("P001", "Laptop", 1200.00));

products.add(new Product("P002", "Smartphone", 800.00));

products.add(new Product("P003", "Headphones", 100.00));

products.add(new Product("P004", "Mouse", 25.00));

List<Order> orders = new ArrayList<>();

orders.add(new Order("O001", List.of(new OrderItem(products.get(0), 1), new OrderItem(products.get(2), 2)), "C001"));

orders.add(new Order("O002", List.of(new OrderItem(products.get(1), 1), new OrderItem(products.get(3), 3)), "C002"));

orders.add(new Order("O003", List.of(new OrderItem(products.get(0), 2)), "C001"));

String customerIdToFind = "C001";

double totalCustomerOrdersPrice = orders.stream()

.filter(order -> order.getCustomerId().equals(customerIdToFind))

.mapToDouble(Order::getTotalOrderPrice)

.sum();

System.out.println("Total order price for customer " + customerIdToFind + ": $" + totalCustomerOrdersPrice);

Optional<Product> mostExpensiveProduct = products.stream()

.max((p1, p2) -> Double.compare(p1.getPrice(), p2.getPrice()));

mostExpensiveProduct.ifPresent(product -> System.out.println("Most expensive product: " + product));

String orderIdToFindProducts = "O002";

Optional<Order> orderOptional = orders.stream()

.filter(order -> order.getOrderId().equals(orderIdToFindProducts))

.findFirst();

orderOptional.ifPresent(order -> {

List<String> productNames = order.getItems().stream()

.map(item -> item.getProduct().getName())

.collect(Collectors.toList());

System.out.println("Products in order " + orderIdToFindProducts + ": " + productNames);

});

String productIdToCount = "P001";

int totalProductQuantity = orders.stream()

.flatMap(order -> order.getItems().stream())

.filter(item -> item.getProduct().getProductId().equals(productIdToCount))

.mapToInt(OrderItem::getQuantity)

.sum();

System.out.println("Total quantity of product "+ productIdToCount + " sold: " + totalProductQuantity);

double priceThreshold = 1000.0;

List<Order> highValueOrders = orders.stream()

.filter(order -> order.getTotalOrderPrice() > priceThreshold)

.collect(Collectors.toList());

System.out.println("Orders with total price over $" + priceThreshold + ": " + highValueOrders);

}

}