1. **Lambda Expressions – Sorting and Filtering Employees**

**Scenario: You are building a human resource management module. You need to: • Sort employees by name or salary. • Filter employees with a salary above a certain threshold. Use Case: Instead of creating multiple comparator classes or anonymous classes, you use Lambda expressions to sort and filter employee records in a concise and readable manner**.

import java.util.\*;

class Employee {

String name;

double salary;

Employee(String name, double salary) {

this.name = name;

this.salary = salary;

}

public String toString() {

return name + " - ₹" + salary;

}

}

public class LambdaExample {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("John", 40000),

new Employee("Alice", 55000),

new Employee("Bob", 30000)

);

// Sort by name using lambda

employees.sort((e1, e2) -> e1.name.compareTo(e2.name));

System.out.println("Sorted by name:");

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

// Sort by salary

employees.sort((e1, e2) -> Double.compare(e1.salary, e2.salary));

System.out.println("\nSorted by salary:");

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

// Filter salary > 40000

System.out.println("\nEmployees with salary > 40000:");

employees.stream()

.filter(e -> e.salary > 40000)

.forEach(System.out::println);

}

}

**2. Stream API – Order Processing System**

**Scenario: In an e-commerce application, you must: • Filter orders above a certain value. • Count total orders per customer. • Sort and group orders by product category. Use Case: Streams help to process collections like orders using operators like filter, map, collect, sorted, and groupingBy to build readable pipelines for data processing**

import java.util.\*;

import java.util.stream.\*;

import static java.util.stream.Collectors.\*;

class Order {

String customer;

String category;

double amount;

Order(String customer, String category, double amount) {

this.customer = customer;

this.category = category;

this.amount = amount;

}

public String toString() {

return customer + " - " + category + " - ₹" + amount;

}

}

public class StreamExample {

public static void main(String[] args) {

List<Order> orders = Arrays.asList(

new Order("Alice", "Electronics", 1500),

new Order("Bob", "Clothing", 700),

new Order("Alice", "Clothing", 300),

new Order("Bob", "Electronics", 1200)

);

// Filter orders > 1000

System.out.println("Orders above ₹1000:");

orders.stream()

.filter(o -> o.amount > 1000)

.forEach(System.out::println);

// Count orders per customer

System.out.println("\nOrder count per customer:");

Map<String, Long> count = orders.stream()

.collect(groupingBy(o -> o.customer, counting()));

count.forEach((k, v) -> System.out.println(k + ": " + v));

// Group orders by category

System.out.println("\nOrders grouped by category:");

Map<String, List<Order>> grouped = orders.stream()

.collect(groupingBy(o -> o.category));

grouped.forEach((k, v) -> {

System.out.println(k + ": " + v);

});

}

}

**3. Functional Interface – Custom Logger**

**Scenario: You want to create a logging utility that allows: • Logging messages conditionally. • Reusing common log filtering logic. Use Case: You define a custom LogFilter functional interface and allow users to pass behavior using lambdas. You also utilize built-in interfaces like Predicate and Consumer**.

@FunctionalInterface

interface LogFilter {

boolean shouldLog(String message);

}

public class LoggerExample {

public static void log(String message, LogFilter filter) {

if (filter.shouldLog(message)) {

System.out.println("LOG: " + message);

}

}

public static void main(String[] args) {

log("Info: App started", msg -> msg.contains("Info"));

log("Error: Something failed", msg -> msg.contains("Error"));

}

}

**4. Default Methods in Interfaces – Payment Gateway**

**Scenario: You're integrating multiple payment methods (PayPal, UPI, Cards) using interfaces. Use Case: You use default methods in interfaces to provide shared logic (like transaction logging or currency conversion) without forcing each implementation to re-define them.**

interface Payment {

void pay(double amount);

default void logTransaction(double amount) {

System.out.println("Transaction of ₹" + amount + " logged.");

}

}

class PayPal implements Payment {

public void pay(double amount) {

System.out.println("Paid using PayPal");

logTransaction(amount);

}

}

class UPI implements Payment {

public void pay(double amount) {

System.out.println("Paid using UPI");

logTransaction(amount);

}

}

public class PaymentExample {

public static void main(String[] args) {

Payment paypal = new PayPal();

paypal.pay(1000);

Payment upi = new UPI();

upi.pay(500);

}

}

**5. Method References – Notification System**

**Scenario: You’re sending different types of notifications (Email, SMS, Push). The methods for sending are already defined in separate classes. Use Case: You use method references (e.g., NotificationService::sendEmail) to refer to existing static or instance methods, making your event dispatcher concise and readable.**

import java.util.function.Consumer;

class NotificationService {

public static void sendEmail(String message) {

System.out.println("Email sent: " + message);

}

public static void sendSMS(String message) {

System.out.println("SMS sent: " + message);

}

public void sendPush(String message) {

System.out.println("Push Notification: " + message);

}

}

public class MethodReferenceExample {

public static void main(String[] args) {

Consumer<String> emailSender = NotificationService::sendEmail;

Consumer<String> smsSender = NotificationService::sendSMS;

NotificationService service = new NotificationService();

Consumer<String> pushSender = service::sendPush;

emailSender.accept("Welcome Email");

smsSender.accept("OTP SMS");

pushSender.accept("New Message Alert");

}

}

**6. Optional Class – User Profile Management**

**Scenario: User details like email or phone number may be optional during registration. Use Case: To avoid NullPointerException, you wrap potentially null fields in Optional. This forces developers to handle absence explicitly using methods like orElse, ifPresent, or map**

import java.util.Optional;

class User {

private Optional<String> email;

User(String email) {

this.email = Optional.ofNullable(email); // may be null

}

public void showEmail() {

email.ifPresentOrElse(

e -> System.out.println("Email: " + e),

() -> System.out.println("Email not provided")

);

}

}

public class OptionalExample {

public static void main(String[] args) {

User user1 = new User("abc@gmail.com");

User user2 = new User(null);

user1.showEmail();

user2.showEmail();

}

}

**7. Date and Time API – Booking System**

**Scenario: A hotel or travel booking system that: • Calculates stay duration. • Validates check-in/check-out dates. • Schedules recurring events. Use Case: You use the new LocalDate, LocalDateTime, Period, and Duration classes to perform safe and readable date/time calculations**

import java.time.\*;

import java.time.temporal.ChronoUnit;

public class BookingSystem {

public static void main(String[] args) {

LocalDate checkIn = LocalDate.of(2025, 8, 1);

LocalDate checkOut = LocalDate.of(2025, 8, 5);

if (checkOut.isAfter(checkIn)) {

long days = ChronoUnit.DAYS.between(checkIn, checkOut);

System.out.println("Stay duration: " + days + " days");

} else {

System.out.println("Check-out date must be after check-in date.");

}

// Schedule weekly events

LocalDate startDate = LocalDate.now();

for (int i = 1; i <= 4; i++) {

System.out.println("Event " + i + ": " + startDate.plusWeeks(i));

}

}

}

**8. ExecutorService – File Upload Service**

**Scenario: You allow users to upload multiple files simultaneously and want to manage the processing efficiently. Use Case: You use ExecutorService to handle concurrent uploads by creating a thread pool, managing background tasks without blocking the UI or main thread**

import java.util.concurrent.\*;

class FileUploadTask implements Runnable {

String fileName;

FileUploadTask(String fileName) {

this.fileName = fileName;

}

public void run() {

System.out.println("Uploading: " + fileName);

try {

Thread.sleep(1000); // Simulate time delay

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Uploaded: " + fileName);

}

}

public class UploadService {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(3);

executor.submit(new FileUploadTask("file1.jpg"));

executor.submit(new FileUploadTask("file2.pdf"));

executor.submit(new FileUploadTask("file3.png"));

executor.shutdown();

}

}