Java 8 Features - Assessment Solutions

1. Lambda Expressions – Employee Sorting & Filtering

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

```
List<Employee> employees = Arrays.asList(
   new Employee("Meera", 50000),
   new Employee("Sahil", 70000),
   new Employee("Anu", 45000)
);

// Sort by name
employees.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));

// Filter salary > 50000
employees.stream()
       .filter(e -> e.getSalary() > 50000)
       .forEach(System.out::println);
```

2. Stream API – Order Processing

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

```
Map<String, Long> ordersCountPerCustomer = orders.stream()
   .filter(o -> o.getTotal() > 1000)
   .collect(Collectors.groupingBy(Order::getCustomer, Collectors.counting()));
```

3. Functional Interfaces – 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

Code:

```
@FunctionalInterface
interface LogFilter {
   boolean filter(String message);
}

LogFilter errorFilter = msg -> msg.contains("ERROR");
if (errorFilter.filter("ERROR: File not found")) {
   System.out.println("Log: ERROR: File not found");
}
```

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.

Code:

```
interface Payment {
   void pay(double amount);
```

```
  default void logTransaction(double amount) {
    System.out.println("Paid: Rs" + amount);
  }
}

class PayPal implements Payment {
  public void pay(double amount) {
    logTransaction(amount);
  }
}
```

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.

Code:

```
class NotificationService {
  public static void sendEmail(String msg) {
    System.out.println("Email: " + msg);
  }
}

Consumer<String> notifier = NotificationService::sendEmail;
notifier.accept("Hello, this is a reminder!");
```

6. Optional Class – User Profile

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.

```
Optional<String> email = Optional.ofNullable(user.getEmail());
email.ifPresent(e -> System.out.println("Email: " + e));
```

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

Code:

```
LocalDate checkIn = LocalDate.of(2025, 7, 26);
LocalDate checkOut = LocalDate.of(2025, 7, 30);
Period stayDuration = Period.between(checkIn, checkOut);
System.out.println("Stay Duration: " + stayDuration.getDays() + " days");
```

8. ExecutorService – File Upload

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.

Code:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> System.out.println("Uploading file..."));
executor.shutdown();
```