Day 11 | #60-DayOOPSDSASQLJourney

Dhee Coding Lab: Composition, Circular Queues, and Complex Aggregations

1. OOPS: Composition vs. Inheritance

While Day 7 introduced the basic definitions, Day 12 focuses on the architectural choice of **Composition over Inheritance**.

Why Composition (HAS-A) is often preferred:

- **Flexibility:** You can change the behavior of a class at runtime by swapping its components. Inheritance is "static"—once a class inherits from a parent, it is stuck with that relationship.
- **Avoids the "Fragile Base Class" Problem:** If you modify a base class in a deep inheritance hierarchy, you might accidentally break dozens of child classes. In composition, changes in one component rarely break the containing class.
- **Loosely Coupled:** Components only interact through well-defined interfaces.

When to use which?

- **Inheritance:** Use only when there is a strict, permanent "is-a" relationship (e.g., `Square` is a `Shape`).
- **Composition:** Use for everything else (e.g., `Car` has an `Engine` , `User` has a `Role`).

2. OOPS Scenario: Scalability and Reuse

Problem: You are building a `Logger` for an application. Initially, it only logs to a `File` . Later, you need to log to `Database` , `Console` , and `Cloud` .

The Inheritance Fail: If you create `FileLogger` , `DatabaseLogger` , etc., as subclasses, what happens if you want a logger that logs to both File AND Database? You end up with a "Class Explosion" (e.g., `FileAndDatabaseLogger`).

The Composition Win: Create a `Logger` class that "has a" list of `Destination` objects.

1. The `Logger` simply iterates through its destinations and calls `.write()` .
2. To log to a new place, you just add a new `Destination` object to the list.
3. This is **Scalable**: You can combine any number of destinations without creating new classes.

3. Programming: Queue Using Array (Circular)

A **Queue** follows the **FIFO** (First-In, First-Out) principle. Implementing it with a simple array leads to "wasted space" as the `front` moves forward. A **Circular Queue** solves this.

Key Variables:

- `front` : Points to the first element.
- `rear` : Points to the last element.
- `capacity` : Maximum size of the array.

Circular Logic:

Instead of `rear = rear + 1` , we use the **Modulo Operator**:

$$rear = (rear + 1) \pmod{capacity}$$

Operations:

1. **Enqueue (Overflow Check):**
   - Condition: `(rear + 1) % capacity == front`
   - Action: If not full, increment `rear` circularly and add element.
2. **Dequeue (Underflow Check):**
   - Condition: `front == -1`
   - Action: If not empty, retrieve element at `front` . If `front == rear` , the queue is now empty (reset both to -1). Otherwise, increment `front` circularly.

Complexity: $O(1)$ for both Enqueue and Dequeue.

4. SQL: Multiple JOINs with Aggregation

This combines the "Chain Reaction" of joins with the mathematical power of `SUM` , `AVG` , or `COUNT` .

Scenario: Finding total revenue per Product Category

We need to link `Categories` $\rightarrow$ `Products` $\rightarrow$ `OrderDetails` .

Syntax Example:

```
SELECT
    c.category_name,
    COUNT(od.order_id) AS total_orders,
    SUM(od.quantity * od.unit_price) AS total_revenue
FROM categories c
INNER JOIN products p ON c.category_id = p.category_id
INNER JOIN order_details od ON p.product_id = od.product_id
GROUP BY c.category_name
ORDER BY total_revenue DESC;
```

Key takeaway for complex reports:

1. **Join first:** Create the wide "flat" view of your data.
2. **Group second:** Collapse that data into categories.
3. **Aggregate third:** Calculate the metrics for those categories.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Composition	"HAS-A" allows for runtime flexibility and prevents class hierarchy bloat.
DSA	Circular Queue	Modulo arithmetic $(n\%size)$ allows array reuse, preventing memory waste.
SQL	Multi-Aggregations	Join all necessary tables before applying `GROUP BY` for complex business metrics.

Day 12 complete. You've learned how to build systems that are modular, memory-efficient, and data-rich!