1. OOPS: When NOT to Use Inheritance

While inheritance is a core pillar, overusing it is a common mistake in software engineering.

Common Pitfalls:

1. **The Fragile Base Class:** Small changes in a parent class can ripple down and break dozens of child classes. This makes the system hard to maintain.
2. **Tight Coupling:** The child class becomes heavily dependent on the internal details of the parent.
3. **Deep Hierarchies:** Having many levels (A -> B -> C -> D -> E) makes the code incredibly hard to trace and debug.
4. **Violation of "Is-A":** Using inheritance just to "borrow" code (e.g., making a `Person` class inherit from `Database` just to use a save method). This is better handled by **Composition**.

2. OOPS Scenario: Refactoring for Stability

Problem: A game development team used inheritance for characters: `Actor -> Fighter -> Knight`. When they tried to add a "Magic Knight," they realized the hierarchy couldn't handle it without duplicating code from a `Mage` class.

Analysis:

1. **The Inheritance Mess:** To add magic to a `Knight`, you either inherit from `Mage` (leading to the Diamond Problem) or copy-paste code.
2. **The Refactoring (Composition):** Instead of `Knight` being a `Fighter`, the `Knight` class "has a" `CombatStyle` and "has a" `MagicAbility`.
3. **The Result:** Now, a "Magic Knight" is simply a `Knight` object where you plug in both a `SwordCombat` component and a `FireMagic` component. The system is now infinitely more scalable and less likely to break.

3. Programming: Implement Stack Using Queues

This problem tests your understanding of the difference between **LIFO** (Stack) and **FIFO** (Queue).

The Challenge

How do you make a "First-In, First-Out" structure (Queue) behave like a "Last-In, First-Out" structure (Stack)?

Approach: Making 'Push' Costly ($O(n)$)

1. Use two queues: `Q1` and `Q2`.
2. When pushing an element x:

- Enqueue x to `Q2`.
- One by one, dequeue everything from `Q1` and enqueue it to `Q2`.
- Swap the names of `Q1` and `Q2`.

3. **Pop:** Simply dequeue from `Q1` ($O(1)$).

Trade-offs:

- **Time Complexity:** Push is $O(n)$, Pop is $O(1)$.
- **Space Complexity:** $O(n)$ for storing elements.
- **Alternative:** You can also use a single queue and rotate elements $n - 1$ times during push.

4. SQL: Subqueries with JOINs

Subqueries can be used as "Derived Tables" within a JOIN to pre-filter or pre-aggregate data before combining it with other tables.

Why combine them?

- **Performance:** Pre-filtering a large table in a subquery before joining can be much faster than joining first and filtering later.
- **Complexity:** Allows you to join an aggregated result (like a total) back to individual records.

Syntax Example:

Find employees whose salary is higher than the average salary of their specific department.

```
SELECT e.name, e.salary, dept_avgs.avg_sal
FROM employees e
INNER JOIN (
    -- Subquery acting as a derived table
    SELECT dept_id, AVG(salary) AS avg_sal
    FROM employees
    GROUP BY dept_id
) AS dept_avgs ON e.dept_id = dept_avgs.dept_id
WHERE e.salary > dept_avgs.avg_sal;
```

Summary Table

Topic	Focus	Key Takeaway
OOPS	Anti-Inheritance	Avoid deep hierarchies; prefer composition for "feature" building.
DSA	Stack via Queue	Reversing order using an auxiliary queue or rotation; involves a $O(n)$ trade-off.

SQL	Subquery Joins	Use subqueries to create "virtual tables" for joining complex or pre-calculated data.

Two weeks complete! You've moved from simple coding to architectural refactoring and