1. OOPS: Class vs Object (Design Perspective)

The Blueprint Metaphor

In modern software design, we move beyond just "templates."

- **Class (Design Time):** Defines the **Structure** and **Behavior**. It exists in the source code. It doesn't occupy memory for data until instantiated.
- **Object (Runtime):** Defines the **Identity** and **State**. It exists in the RAM. Each object has a unique memory address, even if its attributes are identical to another object.

Design Principles:

- **Low Coupling:** Classes should depend on each other as little as possible.
- **High Cohesion:** A class should have one clear purpose (e.g., a `Student` class shouldn't handle `Database` connection logic).

2. OOPS Scenario: Multi-Object Systems

Problem: Why do we need multiple objects of the same class in a Student Management System?

Analysis:

1. **Unique State:** While every `Student` has a `roll_number` and `grade` (defined in the class), "Alice" and "Bob" have different values for those attributes.
2. **Independent Behavior:** If Alice "updates" her email, Bob's email remains unchanged.
3. **Scalability:** A single class `Student` can represent 10 or 10,000 students. We don't write new code for new students; we just instantiate new objects.

3. Programming: Pair Sum Problem

Problem: Find all pairs in an array that sum up to a specific target value.

Approach 1: Brute Force (Nested Loops)

- Compare every element with every other element.
- **Complexity:** $O(n^2)$.
- **Downside:** Very slow for large datasets.

Approach 2: Two-Pointer Technique (Sorted Array)

1. Sort the array ($O(n \log n)$).
2. Use two pointers: `left` at the start, `right` at the end.
3. While `left < right` :

- If $arr[left] + arr[right] == target$: Pair found! Move both pointers.
- If $sum < target$: Increment `left` (to increase the sum).
- If $sum > target$: Decrement `right` (to decrease the sum).

- **Complexity:** $O(n \log n)$ due to sorting.

Approach 3: Hashing (Optimized)

1. Use a Hash Set to store visited numbers.
2. For each number x, check if $(target - x)$ exists in the set.

- **Complexity:** $O(n)$ Time and $O(n)$ Space.

4. SQL: LEFT JOIN (LEFT OUTER JOIN)

The `LEFT JOIN` returns **all** records from the left table and the **matched** records from the right table.

The Logic: "The Inclusive Left"

If there is no match, the result is `NULL` on the right side.

Syntax

```
SELECT students.name, marks.score
FROM students
LEFT JOIN marks ON students.student_id = marks.student_id;
```

Why use it?

- **Finding "Missing" Data:** Useful to see which students haven't taken an exam yet (their `score` would be `NULL`).
- **Data Integrity:** Unlike `INNER JOIN`, you don't lose the "Left" entity just because it lacks a related record in the "Right" table.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Design Perspective	Classes are static blueprints; Objects are dynamic instances with identity.
DSA	Pair Sum	Hashing or Two-Pointers are significantly more efficient than nested loops.
SQL	LEFT JOIN	Preserves all records from the first table, filling gaps with `NULL`.

Consistency builds expertise. See you for Day 3!