1. OOPS: Inheritance (IS-A vs. HAS-A)

Inheritance (IS-A)

Inheritance allows a class (Child) to acquire the properties and methods of another class (Parent). It represents a specialized version of the parent.

- **Example:** A `Car` **is a** `Vehicle`. A `Manager` **is an** `Employee`.
- **Strength:** Excellent for code reuse when there is a clear hierarchy.

Composition (HAS-A)

Composition involves building complex classes by combining simpler objects. Instead of inheriting behavior, a class "has" an instance of another class as a member.

- **Example:** A `Car` **has an** `Engine`. An `Employee` **has a** `Department`.
- **Strength:** Much more flexible. You can swap the "Engine" of a `Car` at runtime, but you cannot easily change its "Parent" class in Inheritance.

2. OOPS Scenario: Avoiding Tight Coupling

Problem: You are building a `Robot` class. Should it inherit from `Camera`, `Wheels`, and `Speaker`?

Analysis:

1. **The Inheritance Trap:** If `Robot` inherits from `Camera`, it becomes a specialized camera. This leads to "Tight Coupling." If the `Camera` class changes, the `Robot` might break.
2. **The Composition Solution:** A `Robot` should **have a** `Camera` object as an attribute.
3. **Why HAS-A wins here:** - A robot might have two cameras or none. Inheritance makes this difficult to manage.
    - It follows the **Principle of Least Knowledge**: The Robot doesn't need to know *how* the camera works, just how to call its `take_picture()` method.

3. Programming: Recursion & The Call Stack

Recursion is a process where a function calls itself to solve smaller instances of the same problem.

Problem: Calculate Factorial of n ($n!$)

- **Base Case:** The condition where recursion stops. For factorial, $0! = 1$.
- **Recursive Step:** The logic that moves toward the base case. $n! = n \times (n-1)!$.

How the Call Stack works:

1. When `factorial(3)` is called, it pauses and calls `factorial(2)`.

2. `factorial(2)` pauses and calls `factorial(1)`.
3. Each call is "pushed" onto the **Stack Memory**.
4. Once the Base Case is reached, the stack "pops" and returns values back up the chain: $1 \to (1 \times 1) \to (2 \times 1) \to (3 \times 2) = 6$.

> **Danger:** Without a proper base case, you will get a **Stack Overflow** error because the memory fills up with infinite paused calls.

4. SQL: JOIN with GROUP BY

Combining these allows you to perform calculations across related tables.

The Logic:

1. **JOIN:** Bring the related data together (e.g., Customers and their Orders).
2. **GROUP BY:** Categorize the rows by a specific column (e.g., by Customer Name).
3. **Aggregate Function:** Perform math on the groups (e.g., `SUM` of order amounts).

Syntax Example:

Find the total spending of each customer.

```
SELECT customers.name, SUM(orders.amount) AS total_spent
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
GROUP BY customers.name;
```

Summary Table

Topic	Focus	Key Takeaway
OOPS	IS-A vs HAS-A	Use Inheritance for "type" hierarchy; use Composition for "component" flexibility.
DSA	Recursion	Every recursive function must have a Base Case to avoid Stack Overflow.
SQL	Aggregated Joins	Crucial for reporting; allows summarizing data from multiple tables at once.

One week complete! You've moved from basic objects to advanced system architecture and recursive logic.