1. OOPS: Abstraction (Focusing on 'What')

What is Abstraction?

Abstraction is the process of hiding the complex internal implementation details and showing only the essential features of an object. While **Encapsulation** hides *data*, **Abstraction** hides *complexity*.

- **The Remote Control Analogy:** You know that pressing the "Power" button turns on the TV (What it does). You do not need to know the electronic circuitry or the signal processing happening inside (How it does it).
- **Implementation:** Usually achieved through **Abstract Classes** and **Interfaces**.

Key Benefits:

- **Reduces Complexity:** Users interact with a simple interface.
- **Flexibility:** You can change the internal implementation without affecting the users of the class.

2. OOPS Scenario: Payment Gateway Integration

Problem: A website needs to support PayPal, Stripe, and Bank Transfers. How do we ensure the checkout code doesn't break every time we add a new payment method?

Analysis:

1. **The Abstract Layer:** Create an abstract class or interface called `PaymentGateway` with a method `process_payment(amount)` .
2. **Multiple Implementations:** - `PayPal(PaymentGateway)` implements the method using PayPal's API.
   - `Stripe(PaymentGateway)` implements it using Stripe's API.
3. **The Result:** The checkout system only knows it is calling `process_payment()` . It doesn't care *which* gateway is being used. This is the power of Abstraction—a common interface for diverse logic.

3. Programming: Binary Search Variant (First Occurrence)

Problem: In a sorted array with duplicate elements, find the index of the **first** occurrence of a target value.

The Logic:

A standard Binary Search might return *any* index where the target exists. To find the *first* one, we modify the condition when we find a match.

Algorithm:

1. Initialize `low = 0`, `high = n - 1`, and `result = -1`.
2. While `low <= mid`:
   - Calculate `mid = low + (high - low) // 2`.
   - If `arr[mid] == target`:
     - **Update** `result = mid` (Potential first occurrence).
     - **Move** `high = mid - 1` (Keep searching the left half to see if there's an earlier occurrence).
   - Else if `target < arr[mid]`: `high = mid - 1`.
   - Else: `low = mid + 1`.
3. Return `result`.

Complexity: $O(\log n)$ Time and $O(1)$ Space.

4. SQL: JOIN with WHERE Clause

While `JOIN` combines tables, the `WHERE` clause filters the combined result set.

The Order of Execution:

1. **FROM & JOIN:** Tables are combined into a large temporary table.
2. **WHERE:** Rows that don't meet the criteria are filtered out.
3. **SELECT:** The final columns are retrieved.

Syntax Example:

Find all customers from 'New York' who have placed orders over $\$500$.

```
SELECT customers.name, orders.amount
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.city = 'New York' AND orders.amount > 500;
```

Pro-Tip:

Always filter as much as possible in the `WHERE` clause to keep your result sets clean and your reports fast.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Abstraction	Defines a contract (What) while hiding the mess (How).

DSA	First Occurrence	Modify Binary Search to "keep looking left" even after finding a match.
SQL	Filtered Joins	Use `WHERE` after `JOIN` to narrow down multi-table results to specific business needs.