1. OOPS: Open/Closed Principle (OCP)

The Open/Closed Principle is the second of the SOLID principles. It states that software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.

- **Open for Extension:** You should be able to add new functionality to the class.
- **Closed for Modification:** Once a class has been developed and tested, its source code should not be changed to add new features.

Why is this important?

Modifying existing code is risky. It can introduce bugs into features that were previously working perfectly. By following OCP, you ensure that new features are added via new code (subclasses or implementations), keeping the core system stable.

2. OOPS Scenario: Scalable Payment Systems

Problem: You have a `PaymentProcessor` class with a method `process(payment_type)`. Initially, it only handles "CreditCard" and "PayPal". Every time a new method like "GooglePay" or "Bitcoin" is added, you have to edit the `process` method with new `if-else` or `switch` statements.

The Violation (Modification Required): This violates OCP because every new feature requires modifying the existing, tested logic of the `PaymentProcessor`.

The OCP Solution (Extension):

1. **Abstract Interface:** Create a `PaymentMethod` interface with a `pay(amount)` method.
2. **Concrete Classes:** Create `CreditCardPayment`, `PayPalPayment`, and `CryptoPayment` classes that implement the interface.
3. **The Result:** The `PaymentProcessor` now simply calls `payment_method.pay(amount)`.
4. **Adding Features:** To add "ApplePay," you simply create a new class `ApplePayPayment`. You **do not touch** the existing `PaymentProcessor` code. This is a modular, risk-free extension.

3. Programming: Two Sum (Optimized Approach)

This is perhaps the most famous hashing problem in coding interviews.

Problem: Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

Approach 1: Brute Force

- Use nested loops to check every possible pair.
- **Complexity:** $O(n^2)$ Time.

Approach 2: One-Pass Hashing (Optimized)

1. Initialize an empty Hash Map (Dictionary).
2. Traverse the array once:
   - Calculate the `complement = target - current_number`.
   - **Check:** Is the `complement` already in our Hash Map?
     - **Yes:** We found the pair! Return the index of the `complement` (from the map) and the current index.
     - **No:** Add the current number and its index to the map `{number: index}`.
3. **Complexity:** $O(n)$ Time and $O(n)$ Space.

   The Trade-off: We use extra space ($O(n)$ for the map) to achieve a massive speed boost from $O(n^2)$ to $O(n)$.

4. SQL: Indexes & Query Optimization

An index is a data structure (usually a B-Tree or Hash) that the database uses to find rows quickly without scanning the entire table.

How it works:

Think of a book. Without an index, if you want to find a specific topic, you must read every page (Full Table Scan). With an index, you look up the topic in the back and jump straight to the correct page.

Pros and Cons of Indexing:

- **Pros:**
  - Dramatically speeds up `SELECT` queries, especially those with `WHERE`, `JOIN`, and `ORDER BY` clauses.
  - Reduces the $O(n)$ search to $O(\log n)$ (for B-Trees).
- **Cons:**
  - **Storage:** Indexes take up extra disk space.
  - **Write Penalty:** Every time you `INSERT`, `UPDATE`, or `DELETE`, the database must also update the index. Over-indexing makes data modification slow.

Optimization Rules:

1. **Index the "Searchable" Columns:** Focus on columns used in `WHERE`, `JOIN` conditions, and `ORDER BY`.
2. **Avoid Indexing Low-Cardinality Columns:** Indexing a "Gender" column (where values are only M/F) usually doesn't help because the database still has to scan half the table.
3. **Composite Indexes:** If you frequently query by two columns (e.g., `first_name` and `last_name`), a multi-column index can be more efficient than two separate ones.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Open/Closed (OCP)	Add new features by adding new classes, not by editing old ones.
DSA	Two Sum	Hashing turns a nested-loop search ($O(n^2)$) into a single pass ($O(n)$).
SQL	Indexing	Speeds up reads but slows down writes; choose columns based on query patterns.

Day 19 complete! You've mastered the art of "extension without modification" and "speed via storage."