1. OOPS: Refactoring Bad Design

Refactoring is the process of restructuring existing computer code—changing the factoring—without changing its external behavior.

Identifying "Code Smells":

1. **Duplicate Code:** The same logic appears in multiple places. (Solution: Extract to a single method).
2. **Long Methods:** A method that does too much is hard to test and understand. (Solution: Break into smaller, focused methods).
3. **Large Class (God Object):** A class that tries to handle every responsibility. (Solution: Apply SRP and split the class).
4. **Shotgun Surgery:** Every time you make a small change, you have to edit ten different classes. (Solution: Move related logic into one place).

2. OOPS Scenario: Decoupling for Flexibility

Problem: You have a `WeatherApp` class that directly creates an instance of `AccuWeatherAPI` inside its constructor to fetch data.

The Issue (Tight Coupling): If you want to switch to `OpenWeatherAPI` , you must modify the `WeatherApp` class code. If the internet is down during testing, you can't easily "mock" the weather data because the API call is hardcoded.

The Refactored Solution (Abstraction + Composition):

1. **Define an Interface:** Create a `WeatherProvider` interface with a `fetch_weather()` method.
2. **Implement Providers:** Create classes `AccuWeather` and `OpenWeather` that implement the interface.
3. **Inject the Dependency:** The `WeatherApp` now "has a" `WeatherProvider` . You pass the provider into the constructor.

The Result: The `WeatherApp` no longer cares which API is being used. It is now **Loosely Coupled** and easy to test or extend.

3. Programming: Hashing Basics (Frequency Counting)

Hashing is a technique that maps data of arbitrary size to fixed-size values (hash codes) using a hash function. In DSA, we use **Hash Maps** (Dictionaries) to achieve $O(1)$ average-time complexity for lookups.

Problem: Find the frequency of each element in an array.

Approach 1: Brute Force

- For each element, count its occurrences by looping through the rest of the array.
- **Complexity:** $O(n^2)$.

Approach 2: Hashing (Optimized)

1. Initialize an empty Hash Map (Dictionary).
2. Traverse the array once:
   - If the element exists in the map, increment its value (count).
   - Else, add the element to the map with a value of 1.
3. **Complexity:** $O(n)$ Time and $O(n)$ Space.

> **Key takeaway:** Hashing trades a little bit of memory (Space) for a massive gain in speed (Time).

4. SQL: Performance Considerations in JOINs

A JOIN can be extremely fast or painfully slow depending on how the database engine processes it.

Critical Performance Factors:

1. **Indexing (The #1 Factor):** Always ensure that the columns used in the `ON` clause (usually Primary and Foreign Keys) are indexed. Without an index, the database must perform a "Full Table Scan," comparing every row of Table A with every row of Table B ($O(n \times m)$).
2. **Join Type Selection:**
   - **Nested Loop Join:** Best for small tables.
   - **Hash Join:** Used for large, non-indexed joins (heavy on memory).
   - **Merge Join:** Very fast if both tables are already sorted by the join key.
3. **Selectivity:** Only `SELECT` the columns you need. `SELECT *` across three joined tables pulls massive amounts of unnecessary data into memory, slowing down the network and the engine.

Pro-Tip:

Use the `EXPLAIN` or `EXPLAIN ANALYZE` command before your query to see how the database plans to execute the join and whether it is utilizing indexes correctly.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Refactoring	Cleaning "Code Smells" prevents technical debt and improves maintainability.

DSA	Hashing	Reduces search/count operations from $O(n)$ to $O(1)$ on average.
SQL	Join Performance	Indexes on Join columns are mandatory for production-scale databases.

Day 16 complete! You are now moving from just "making it work" to "making it professional and fast."