1. OOPS: Introduction to Design Patterns

Design patterns are typical solutions to common problems in software design. They aren't snippets of code you can copy-paste; they are **blueprints** that you can customize to solve a specific architectural problem.

Why use them?

- **Standardization:** They provide a common vocabulary for developers (e.g., "Use a Factory here").
- **Proven Solutions:** They are tried-and-tested techniques that prevent subtle bugs that might not be obvious until the system scales.
- **Maintainability:** Patterns like Singleton and Factory reduce the complexity of object creation.

2. OOPS Scenario: Singleton and Factory Patterns

Problem: In a massive enterprise application, you have multiple modules trying to create their own `DatabaseConnection` and `ReportGenerator`.

The Singleton Solution (For Database): Instead of every module creating a new connection, the **Singleton Pattern** ensures that a class has only one instance and provides a global point of access to it.

- **Use Case:** Logging, Database connections, Configuration settings.

The Factory Solution (For Reports): Instead of the client code knowing how to instantiate `PDFReport`, `ExcelReport`, or `HTMLReport`, the **Factory Pattern** provides an interface/method for creating objects but allows the factory to decide which class to instantiate.

- **Analysis:** This simplifies object creation and ensures consistency. If you add a `CSVReport` later, you only change the Factory, not the 50 places in the code that request a report.

3. Programming: HashMap Problem (First Non-Repeating Element)

This problem builds on the frequency counting logic from Day 16, adding a second pass to find a specific pattern.

Problem: Find the first non-repeating element in an array.

The Algorithm (Two-Pass Hashing):

1. **First Pass (Frequency Count):** Iterate through the array and store the count of each element in a Hash Map.
2. **Second Pass (Discovery):** Iterate through the **original array** again. For each element, look up its count in the Hash Map.
3. The **first element** you encounter with a `count == 1` is your answer.

Why it's Efficient:

- **Approach 1 (Nested Loops):** For every element, search the rest of the array. $O(n^2)$.
- **Approach 2 (Hashing):** Two linear traversals. $O(n + n) = O(n)$ Time.
- **Space Complexity:** $O(k)$ where k is the number of unique elements.

4. SQL: JOIN with DISTINCT

When joining tables in a **One-to-Many** relationship, the "One" side will appear duplicated for every match in the "Many" side. `DISTINCT` is used to remove these duplicates.

Scenario: Identifying Active Customers

We want a list of customers who have placed at least one order. If a customer has placed 100 orders, an `INNER JOIN` will return their name 100 times.

Syntax Example:

```
SELECT DISTINCT c.customer_id, c.customer_name
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;
```

When to use vs. when to avoid:

- **Use Case:** When you only need to know the *existence* of a relationship, not the details of every individual match.
- **Avoidance:** `DISTINCT` can be expensive on very large result sets because the database has to sort the data to find duplicates. If you find yourself using `DISTINCT` constantly, your JOIN logic or schema design might need review.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Design Patterns	Patterns provide a shared language and proven blueprints for scalable architecture.
DSA	First Non-Repeating	Using a two-pass HashMap approach maintains $O(n)$ time while ensuring order.
SQL	JOIN + DISTINCT	Collapses duplicate rows generated by one-to-many relationships in a join.