1. OOPS: Polymorphism (Many Forms)

Polymorphism allows a single interface (method name) to handle different underlying forms (data types or classes).

Compile-Time Polymorphism (Static Binding)

Achieved through **Method Overloading**. The compiler knows which method to call based on the number or type of arguments.

- **Example:** `add(int a, int b)` vs. `add(double a, double b)` .

Runtime Polymorphism (Dynamic Binding)

Achieved through **Method Overriding**. The method that gets executed is determined at runtime based on the actual object type, not the reference type.

- **Requirement:** Must have an Inheritance relationship.
- **Example:** A `Shape` class has `draw()` , and `Circle` overrides it to draw a circle.

2. OOPS Scenario: Flexible Notification Systems

Problem: You need a system that sends notifications. Today it's Email and SMS; tomorrow it might be WhatsApp or Slack. How do you design it so the main code doesn't change?

Analysis:

1. **The Polymorphic Base:** Create a base class `Notification` with a method `send(message)` .
2. **Specialized Classes:** - `EmailNotification` overrides `send()` to use an SMTP server.
   - `SMSNotification` overrides `send()` to use a Telecom API.
3. **The Power of Runtime Polymorphism:** Your main application can hold a list of `Notification` objects. It simply loops through them and calls `.send()` . The system "decides" at the last millisecond whether to trigger the Email or SMS logic based on the object's actual type.

3. Programming: Stack Application (Balanced Parentheses)

A **Stack** follows the **LIFO** (Last-In, First-Out) principle. This makes it perfect for problems where we need to "match" the most recent item.

Problem: Check if a string like `{[()]}` is balanced.

The Algorithm:

1. Initialize an empty stack.
2. Traverse the string character by character:

- If it's an **Opening Bracket** (`(` , `{` , `[`): Push it onto the stack.
- If it's a **Closing Bracket** (`)` , `}` , `]`):
    - If the stack is empty $\rightarrow$ **Unbalanced** (e.g., `())`).
    - Pop the top element. If it doesn't match the closing bracket type $\rightarrow$ **Unbalanced** (e.g., `(]`).

3. After the loop, if the stack is empty $\rightarrow$ **Balanced**. If not $\rightarrow$ **Unbalanced** (e.g., `(()`).

Complexity: $O(n)$ Time and $O(n)$ Space.

4. SQL: Multiple Table JOINs

In real-world databases, data is often spread across many tables. You can chain `JOIN` clauses to pull it all together.

The Logic: "The Chain Reaction"

You join Table A to Table B, then Table B to Table C. The "bridge" is usually a Foreign Key relationship.

Syntax Example:

Get the Customer Name, the Product they bought, and the Order Date.

```
SELECT
    customers.name,
    products.product_name,
    orders.order_date
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id
INNER JOIN products ON orders.product_id = products.product_id;
```

Pro-Tip:

When joining many tables, always use **Table Aliases** (e.g., `FROM orders AS o`) to keep your query readable and avoid "Ambiguous Column" errors.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Polymorphism	Overloading (Compile-time) vs. Overriding (Runtime).
DSA	Stack (LIFO)	Ideal for nested structures and "undo" or "matching" logic.
SQL	Multi-JOINs	Use Foreign Keys as "bridges" to link three or more tables.

Day 8 complete. You've mastered how systems adapt behavior and how complex data relates!