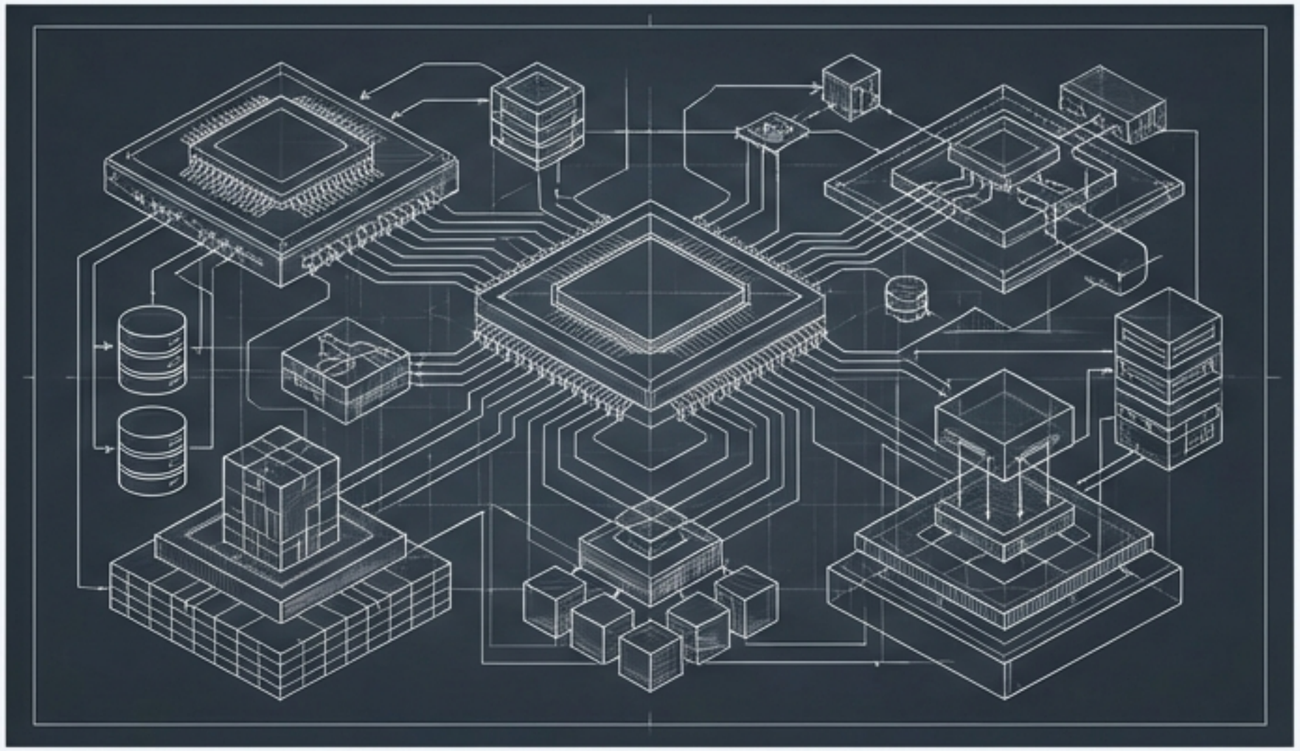
Core CS Fundamentals: Your Interview Prep Crash Course

Mastering the essential concepts of DBMS, OOPs, and Computer Networks. A definitive guide for aspiring software engineers.



What Exactly is a Database Management System (DBMS)?

A DBMS is not the database itself; it is the **software** used to create, retrieve, and manage data efficiently in a database.

Think of it as the librarian for a vast library (the database). The librarian finds, stores, and organizes the books (data) for you.

SQL (RDBMS)

Organizes data in a structured way using tables (rows and columns).

Like a perfectly organized Excel spreadsheet where tables are linked.

Examples: MySQL, PostgreSQL

NoSQL (Non-Relational)

Manages unstructured or semi-structured data (documents, key-value pairs, graphs).

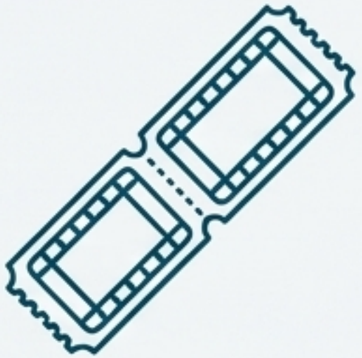
Like a folder of flexible Word documents; each can have a different structure.

Examples: MongoDB, Cassandra

When asked about DBMS, always start by clarifying that it's the *software layer* that interacts with the database, not the storage itself. This shows precision.

How Do Databases Guarantee Transaction Reliability? The ACID Properties.

ACID is an acronym that describes four properties ensuring that database transactions are processed reliably.



A: Atomicity (All or Nothing)

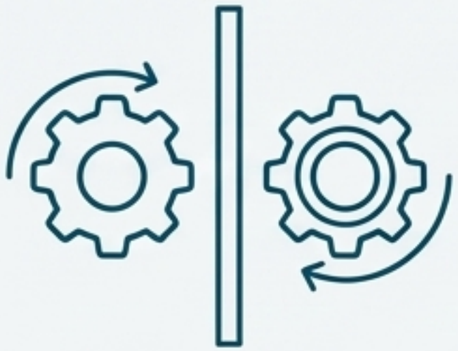
A transaction is an indivisible unit. It must be executed in its entirety or not at all.

Booking movie tickets. You either get all 10 tickets you requested, or the transaction fails completely. You can't end up with only two tickets and your money partially spent.

C: Consistency

A transaction brings the database from one valid state to another, preserving all predefined rules (constraints).

If you delete your Amazon account (parent table), all of your associated orders (child table) must also be deleted to maintain a consistent state. The relationship rules cannot be broken.



I: Isolation

Concurrent transactions are executed independently without interfering with each other. One transaction doesn't see the intermediate state of another.

You try to book a ticket for ₹250. At the exact same moment, the owner tries to update the price to ₹10,000. Isolation ensures one transaction completes fully before the other begins, so you either pay ₹250 or see the new price of ₹10,000, but never a mix-up.

D: Durability

Once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

After your 10 movie tickets are successfully booked, the change is permanent. Refreshing the page or a system crash won't make your confirmed booking disappear.

Understanding the Keys that Unlock Database Relationships

Primary Key

A column (or set of columns) that **uniquely identifies** each row in a table. It cannot contain NULL values.

Purpose

To ensure data integrity and provide a unique identifier for indexing and fast data retrieval.

Users

UserID	Name
123	'Harish'
124	'Harish'

Even if two users are named 'Harish', their `UserID` (e.g., 123) will be unique.

Foreign Key

A key used to link two tables together. It's a field in one table that refers to the PRIMARY KEY in another table.

Purpose

To create and enforce a relationship between data in two tables. This is the foundation of a ***relational*** database.

Users

UserID	Name
123	'Harish'
124	'Harish'

Orders

OrderID	OrderDate	UserID

An `Orders` table has a `UserID` column that points to the `UserID` in the `Users` table, linking an order to a specific user.

Candidate Key

A column or a set of columns that could potentially be a primary key. A table can have multiple candidate keys.

Users

UserID	Name	Email	PhoneNumber

In a `Users` table, both `UserID` and the combination of ('Email', `PhoneNumber`) could uniquely identify a user. Both are candidate keys, but we choose `UserID` as the primary key.

How Do We Organize Data to Eliminate Redundancy? Normalization.

Normalization is the process of structuring a database to reduce data redundancy and improve data integrity. We do this by dividing larger tables into smaller, well-structured tables and defining relationships between them.

1NF (First Normal Form)

Ensures each column holds atomic (indivisible) values and each entry is unique.

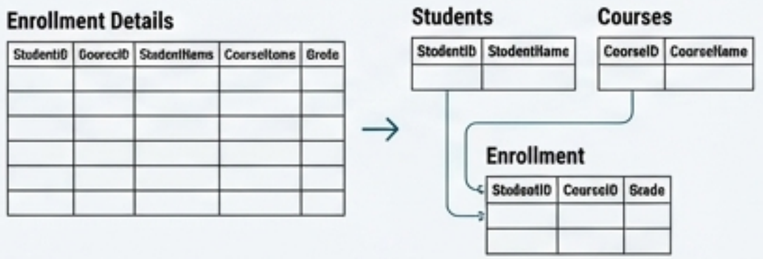
	Phone Numbers
	123-456, 789-012

→

	Phone Numbers
	123-456
	789-012

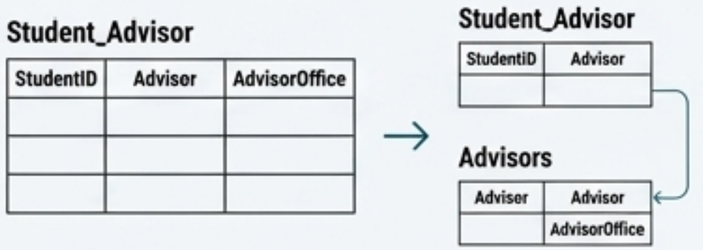
2NF (Second Normal Form)

Must be in 1NF. Removes partial dependencies by ensuring all non-key attributes are fully dependent on the entire primary key.



3NF (Third Normal Form)

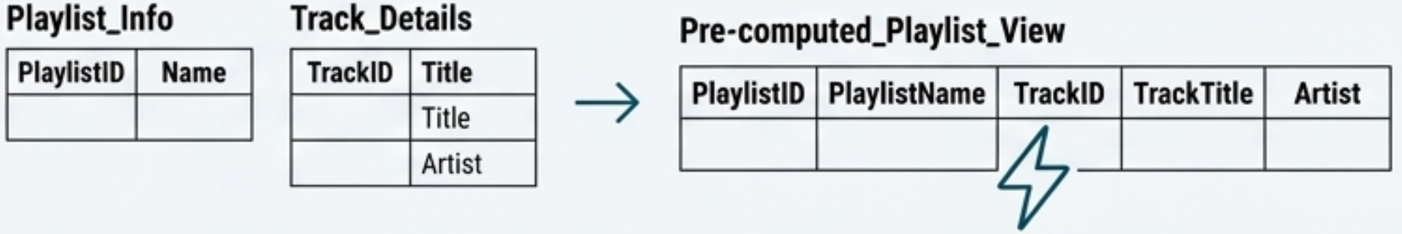
Must be in 2NF. Removes transitive dependencies, meaning non-key attributes can't depend on other non-key attributes.



The Other Side of the Coin: Denormalization

The process of intentionally adding redundant data to one or more tables.

Why? To improve query performance. By combining tables, we can avoid complex and slow JOIN operations for frequently accessed data.

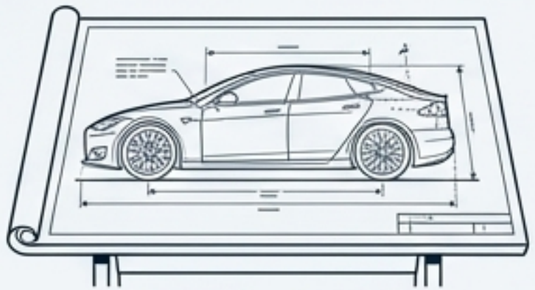


Thinking in Objects: The Core Idea Behind OOP

OOP is a programming paradigm based on the concept of 'objects'. Instead of structuring programs around functions and logic, we structure them around objects that contain both data and the functions that operate on that data.

Attributes (Properties): Things an object *has*.

Methods (Behaviors): Things an object can *do*.



Class: The blueprint.

Object: The actual instance created from the blueprint.

```
Car myCar = new Car(); // The 'new' keyword creates the object from the class.
```

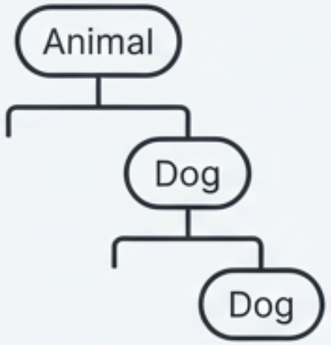
The Four Pillars of Object-Oriented Programming

1. Encapsulation (Bundling & Protection)

Bundling data (attributes) and methods into a single unit (an object). It restricts direct access to some of an object's components.

A protective case on your phone. It bundles the phone and protects its internal components from accidental damage. You interact through defined buttons (public methods).

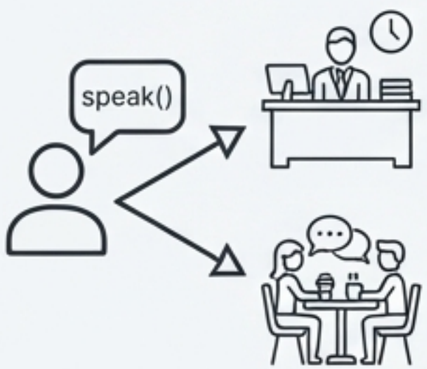
2. Inheritance (Reusability)



A mechanism where a new class (child) derives attributes and methods from an existing class (parent). This promotes code reuse.

Inherited property. Just as a child inherits traits from their parents, a `Dog` class can inherit the `eat()` method from a general `Animal` class.

3. Polymorphism (Many Forms)



The ability of a method or object to take on many forms. A single action can be performed in different ways depending on the context.

Your communication style. You `speak()' differently to your manager (formally) than you do to your friends (informally). The action (`speak`) is the same, but the implementation changes.

4. Abstraction (Hiding Complexity)

Hiding complex implementation details and showing only the essential features of the object.
A TV remote. You know pressing 'power' turns the TV on, but you don't need to know the complex circuitry and signals involved. The internal details are abstracted away.

Encapsulation in Action: Protecting Your Data's Integrity

Don't let other parts of your code reach in and change an object's data directly. Instead, provide public methods to control how the data is accessed and modified.

How It's Done: Access Modifiers

`public`: Accessible from anywhere.

`private`: Accessible only within the same class.

`protected`: Accessible within the same package and by subclasses.

Code Example: A Bank Account

❌ The Wrong Way (No Encapsulation)

```
class BankAccount {
    public double balance; // Public - anyone can change it!
}

// another class...
myAccount.balance = -1000.00; // Uncontrolled, invalid state
```

✅ The Right Way (With Encapsulation)

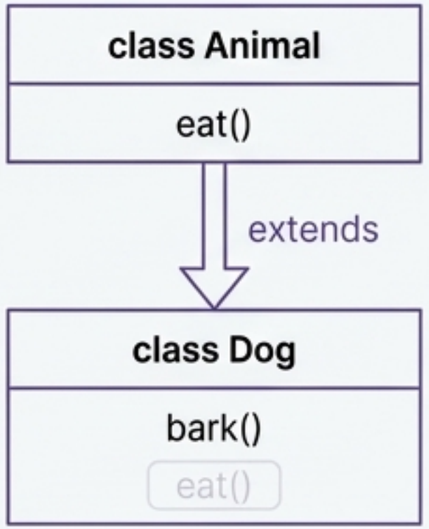
```
class BankAccount {
    private double balance; // Private - protected!

    public double getBalance() { // Public "getter"
        return this.balance;
    }

    public void deposit(double amount) { // Public "setter"
        if (amount > 0) {
            this.balance += amount;
        }
    }
}
```

Why It Matters

Encapsulation prevents unauthorized or invalid modifications to an object's state, ensuring data remains valid and consistent.

Inheritance: Building on Existing Code with Parent-Child Relationships

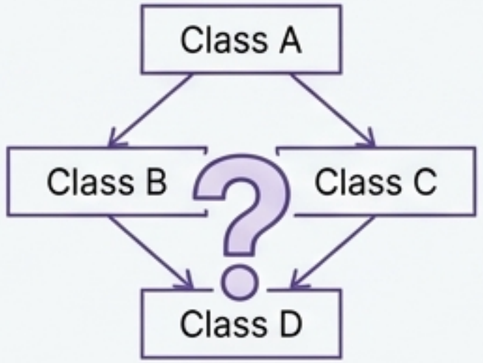
Core Principle: Inheritance allows a class (Child or Subclass) to inherit the properties and methods of another class (Parent or Superclass). This is an “is-a” relationship (a Dog is an Animal).



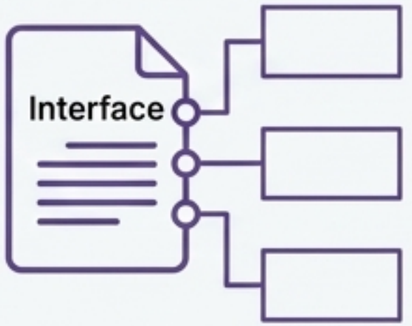

```java
class Animal { // Parent Class
  public void eat() {
     System.out.println("This animal eats food.");
  }
}

class Dog extends Animal { // Child Class
  public void bark() {
     System.out.println("Woof!");
  }
}

// Usage
Dog myDog = new Dog();
myDog.eat();    // Inherited from Animal
myDog.bark();   // Defined in Dog
```

The Diamond Problem: A Cautionary Tale

The Problem
Arises in languages that allow multiple inheritance. If Class B and Class C both inherit from Class A, and Class D inherits from both B and C, which version of a method from A does D use if both B and C have overridden it? This creates ambiguity.



The Solution in Java:
Java avoids this by not allowing multiple inheritance of classes. Instead, it allows a class to implement multiple interfaces.



Polymorphism: One Name, Many Forms

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single method name to have different implementations.

Type 1: Compile-Time Polymorphism (Method Overloading)

Defining multiple methods with the same name but different parameters (number, type, or order) within the same class. The correct method is chosen at compile-time.

```
class MathUtils {
  int add(int a, int b) { return a + b; }
  double add(double a, double b) { return a + b; }
  int add(int a, int b, int c) { return a + b + c; }
}
```

Type 2: Run-Time Polymorphism (Method Overriding)

A subclass provides a specific implementation for a method that is already defined in its superclass. The correct method is chosen at run-time based on the object type.

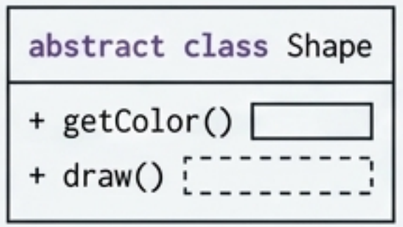
```
class Animal {
  public void makeSound() { System.out.println("Animal sound"); }
}
class Cat extends Animal {
  @Override
  public void makeSound() { System.out.println("Meow"); }
}

// Usage
Animal myPet = new Cat();
myPet.makeSound(); // Prints "Meow"
```

Abstraction: Hiding the "How," Focusing on the "What"

Abstraction simplifies complex systems by modeling classes appropriate to the problem and hiding the complex implementation details from the user. It defines a contract that implementing classes must follow.

How to Achieve Abstraction

Abstract Classes

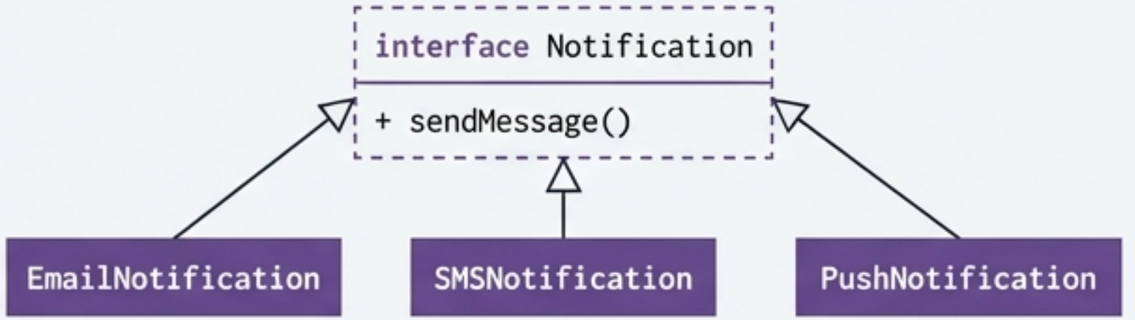


What: A class that cannot be instantiated and may contain a mix of `abstract` methods (no implementation) and concrete methods (with implementation).

When to use: When you want to provide a common base class with some default behavior that subclasses can share and/or override. A class can only **extend** one abstract class.

Example: An abstract Shape class could have an abstract `draw()` method but a concrete `getColor()` method.

Interfaces



What: A completely abstract "class" that can only contain abstract methods and static constants. It is a pure contract.

When to use: When you want to define a contract for behavior that can be implemented by completely unrelated classes. A class can **implement** multiple interfaces.

Example: A `Notification` interface with a `sendMessage()` method could be implemented by `EmailNotification`, `SMSNotification`, and `PushNotification` classes.

The OSI Model: A 7-Layer Map for Network Communication

The Open Systems Interconnection (OSI) model is a conceptual framework that standardizes the functions of a computing system into seven abstract layers. Each layer serves the layer above it and is served by the layer below it.

Encapsulation

7		**Application**	Closest to the end user. Where application protocols like HTTP and FTP operate.
6		**Presentation**	Translates, encrypts, and compresses data, ensuring it's readable by the next system.
5		**Session**	Manages sessions between applications: establishment, management, and termination.
4		**Transport**	Provides reliable data transfer between end points, managing flow control and error checking.
3		**Network**	Responsible for packet forwarding and routing. This is where IP addresses live.
2		**Data Link**	Manages node-to-node data transfer and error correction. Uses MAC addresses.
1		**Physical**	Transmits raw bits over a physical medium. It's all about the hardware.

De-encapsulation

The Internet's Core Protocols and Address Book

Transport Layer Showdown: TCP vs. UDP

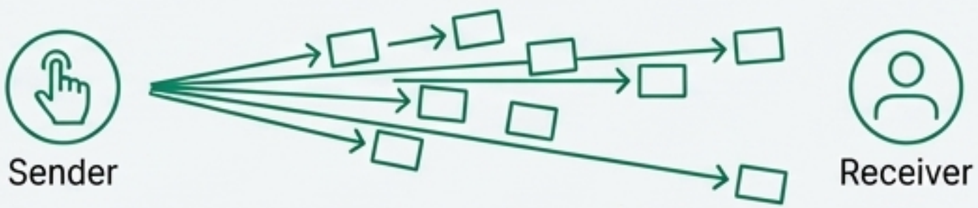
TCP (Transmission Control Protocol)

A connection-oriented protocol that guarantees reliable, ordered delivery of data packets. Uses a "three-way handshake" to establish a connection and requires acknowledgments for packets received.

Use Case: File transfers, email, web browsing.

UDP (User Datagram Protocol)

A connectionless protocol that is fast but does not guarantee delivery or order. "Fire and forget." Sends packets without establishing a connection or waiting for acknowledgments.



Use Case: Video streaming, online gaming, DNS lookups.

The Address System

IP Address (Internet Protocol Address)

A unique numerical label assigned to each device on a network.

Your home's street address

DNS (Domain Name System)

The system that translates human-readable domain names into machine-readable IP addresses.

The internet's phonebook

The CAP Theorem: The Fundamental Trade-Off of Distributed Systems

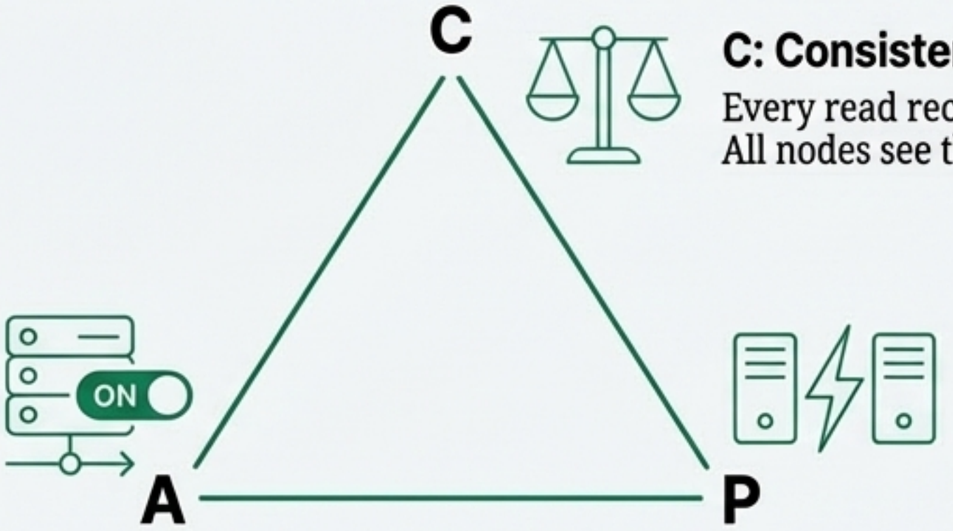
In any distributed data store, you can only provide **two out of three** of the following guarantees.

C: Consistency

Every read receives the most recent write or an error. All nodes see the same data at the same time.

A: Availability

Every request receives a (non-error) response, without guaranteeing it contains the most recent write. The system is always operational.



P: Partition Tolerance

The system continues to operate despite network messages being dropped between nodes. (In modern systems, this is a must-have).

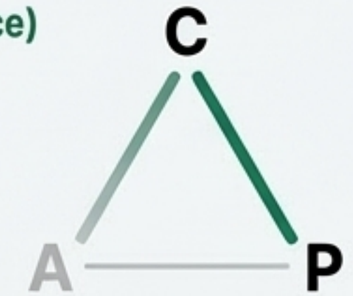
The Inevitable Choice

Since network partitions (P) are a reality, distributed systems must choose between Consistency and Availability.

CP Systems (Consistency + Partition Tolerance)

Sacrifice availability to ensure consistency. May return an error to prevent returning stale data.

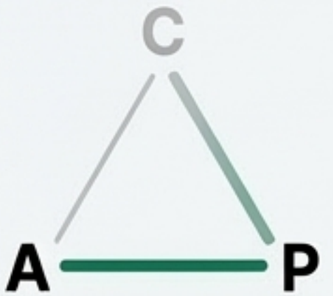
Example: Banking systems, where data correctness is non-negotiable.



AP Systems (Availability + Partition Tolerance)

Sacrifice consistency for availability. Will always respond, even if with slightly out-of-date data (eventual consistency).

Example: Social media feeds, where seeing a post a few seconds late is acceptable.



Key Strategies for Your Technical Interview

1. Go Beyond Definitions with Real-World Scenarios.

Don’t just define Normalization. Take a messy, hypothetical table and walk through how you would normalize it to 3NF. Use analogies like the movie ticket booking or Amazon user account examples.

2. Be Prepared to Write Code and Queries Live.

Practice writing SQL queries from scratch, especially different types of JOINs. Be ready to sketch out class structures for OOP problems and explain your design choices.

3. Master the "Why" Behind the Concepts.

Why use Normalization? To reduce redundancy. Why use Encapsulation? To protect data integrity. Knowing the 'why' demonstrates deeper understanding.

4. Understand the Trade-Offs.

SQL vs. NoSQL (Structure vs. Flexibility). TCP vs. UDP (Reliability vs. Speed). CAP Theorem (Consistency vs. Availability). Articulating these trade-offs is a hallmark of a senior engineering mindset.