The Ultimate OOPs Interview Master Guide for Freshers. By -Syed

Alright, future tech leader. Let's talk about building software that lasts. Object-Oriented Programming (OOPs) is more than just a set of rules; it's a way of thinking that turns chaotic code into elegant, manageable, and powerful systems. Interviewers at Google, Meta, and everywhere else don't want textbook definitions, they want to see if you can *feel* the architecture.

This guide is your personal mentorship session. We're going to transform abstract concepts into tangible ideas you can grasp, relate to, and never forget. By the end of this, you won't just be ready for your interview; you'll be a better software architect. Let's begin.

Secret_Syed

Part 1: The Core Idea - What is OOPs?

1. What is Object-Oriented Programming (OOPs)?

OOPs is a programming paradigm based on the concept of "objects." Instead of writing long lists of procedures, we model the real world by creating objects that contain both data and the functions that operate on that data.

- **Analogy:** Think of building a car. Instead of a single, massive instruction sheet (procedural), you build separate, self-contained objects: an Engine object, a Wheel object, a Steering object. Each object knows its own properties (data) and how it works (functions).

2. Why do we use OOPs?

We use OOPs to build complex software that is easier to manage, scale, and debug. It promotes code reusability and helps organize a large system into logical, understandable parts.

- **Real-World Example:** For a large application like Google Maps, OOPs is essential. The Map is an object, each Pin is an object, and the Route is an object. This separation allows thousands of engineers to work on different parts of the application without breaking the whole system.

3. What is the difference between Procedural and Object-Oriented Programming?

Procedural programming is about writing a sequence of steps or procedures to complete a task. OOPs is about creating objects that interact with each other to complete a task.

- **Analogy:** To make a sandwich, a procedural approach is a single recipe: 1. Get bread. 2. Get cheese. 3. Put cheese on bread. An OOP approach creates objects: a Bread object, a Cheese object, and a SandwichMaker object that knows how to combine them. The OOP way is more organized and reusable for making different kinds of sandwiches.

Part 2: The Building Blocks - Classes & Objects

4. What is a Class?

A class is a blueprint or a template for creating objects. It defines a set of common properties (attributes) and behaviors (methods) that all objects of that class will have.

- **Analogy:** The architectural blueprint for a house is a class. It defines that every house built from it will have bedrooms, bathrooms, and a kitchen, but it isn't a house itself.

Secret_Syed

5. What is an Object?

An object is a specific instance of a class. It's the actual "thing" created from the blueprint, with its own unique data, and it resides in the computer's memory.

- **Analogy:** If the blueprint is the class, the actual house you build at "123 Main Street" is an object. You can build many unique houses (objects) from the same blueprint (class).

6. What is the difference between a Class and an Object?

A class is the logical template; an object is the physical reality. The class Car defines that cars have a color and can drive(). The redFerrari object is a specific instance with color = "red". The class doesn't exist in memory; objects do.

7. Can we create an object without a class?

No. A class is the fundamental blueprint required to define the structure and behavior of an object. Without the blueprint, the system wouldn't know what kind of object to create.

Part 3: The Four Pillars of OOPs

Pillar 1: Encapsulation

8. What is Encapsulation?

Encapsulation is the practice of bundling an object's data (attributes) and the methods that operate on that data into a single, self-contained unit (the class). It also involves hiding the object's internal state from the outside world.

- **Analogy:** Think of a car's dashboard. It encapsulates the complex engine, wiring, and sensors. You are given a simple interface (steering wheel, pedals, speedometer) to interact with the car, without needing to know or worry about the complex mechanics hidden under the hood.

9. How is Encapsulation achieved?

It's achieved by making the data members of a class private and providing public methods (getters and setters) to access and modify that data in a controlled way.

- **Real-World Example:** A User class has a private password field. You can't access it directly. Instead, you must call a public method like changePassword(oldPassword, newPassword), which can contain logic to validate the change, ensuring the object's integrity.

10. What are Access Specifiers/Modifiers?

These are keywords that set the visibility and accessibility of classes, methods, and attributes.

- **public:** Accessible from anywhere. (The car's steering wheel).
- **private:** Accessible only within the same class. (The engine's internal workings).
- **protected:** Accessible within the same class and by its subclasses. (A mechanic's diagnostic port, accessible to authorized service centers).

Secret_Syed

Pillar 2: Abstraction

11. What is Abstraction?

Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object. It focuses on what an object does, not how it does it.

- **Analogy:** A TV remote is a perfect example of abstraction. You just see buttons like "Volume Up" and "Power." You don't need to know the complex circuitry, infrared signals, or electronics that make it work.

12. How is Abstraction different from Encapsulation?

Abstraction is about hiding complexity to provide a simple interface. Encapsulation is about bundling data and methods together to protect an object's state. Encapsulation is a technique used to help achieve abstraction.

- **Analogy:** Abstraction is the design of the car's simple dashboard. Encapsulation is the actual manufacturing process of building the engine and sealing it under the hood.

Secret_Syed

13. What is an abstract class?

An abstract class is a special type of class that cannot be used to create objects. It acts as a template for other classes to inherit from, and it can contain both regular methods and "abstract methods" (methods without an implementation).

- **Real-World Example:** You could have an abstract class Vehicle with an abstract method startEngine(). You can't create a generic Vehicle object, but you can create Car and Motorcycle classes that inherit from Vehicle and provide their own specific way to startEngine().

14. What is an interface?

An interface is a completely abstract type that contains only the signatures of methods, without any implementation. It's a contract that guarantees any class implementing it will provide functionality for its methods.

- **Analogy:** An interface is like a standard electrical wall socket. It doesn't provide power itself, but it defines a contract: anything that

wants to get power from the wall (a Toaster class, a Lamp class) *must* implement the standard plug shape.

15. How is an abstract class different from an interface?

An abstract class can have both abstract and non-abstract (implemented) methods, and a class can only inherit from one. An interface can only have abstract methods, but a class can implement multiple interfaces. Think of an abstract class as defining what an object is, and an interface as defining what an object can do.

Pillar 3: Inheritance

16. What is Inheritance?

Inheritance is a mechanism where a new class (the child or subclass) derives properties and behaviors from an existing class (the parent or superclass). It promotes code reusability and creates a logical hierarchy.

- **Analogy:** Inheritance is like genetics. You (the child class) inherit traits like eye color and height (attributes and methods) from your parents (the parent class), but you can also have your own unique traits.

Secret_Syed

17. What is a superclass and a subclass?

- **Superclass (Parent):** The class whose features are inherited. (e.g., Vehicle).
- **Subclass (Child):** The class that inherits from another class. (e.g., Car).

18. What are the different types of inheritance?

- **Single Inheritance:** One child class inherits from one parent class. (Car -> Vehicle).
- **Multilevel Inheritance:** A child class inherits from a parent, which in turn inherits from a grandparent. (SportsCar -> Car -> Vehicle).
- **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class. (Car, Bus, Motorcycle all inherit from Vehicle).
- **Multiple Inheritance (not in all languages):** A child class inherits

from more than one parent class. (A FlyingCar could inherit from both Car and Airplane).
- **Hybrid Inheritance:** A combination of two or more types of inheritance.

19. What is the super (or base) keyword?

The super keyword is used in a subclass to call methods or constructors of its immediate parent class. It's how a child can access its parent's functionality.

- **Real-World Example:** When creating a Car object, its constructor might first call super() to run the Vehicle constructor to initialize common properties like numberOfWheels, before setting its own car-specific properties like hasAirConditioning.

20. What is the diamond problem in multiple inheritance?

This is a deadly ambiguity that occurs when a class inherits from two parent classes that both inherit from the same grandparent class. If the child class calls a method from the grandparent, the compiler doesn't know which parent's path to take. Languages like Java prevent this by not allowing multiple inheritance of classes.

Pillar 4: Polymorphism

21. What is Polymorphism?

Polymorphism, meaning "many forms," is the ability of an object, method, or variable to take on multiple forms. It allows a single action to be performed in different ways depending on the context.

- **Analogy:** Think of a USB port. It's a single interface (polymorphism), but it can accept many different forms of devices: a mouse, a keyboard, a flash drive, a smartphone. Each device behaves differently when connected to the same port.

22. What is Compile-Time (Static) Polymorphism?

This is when the decision of which method to call is made at compile time. The most common way to achieve this is through method overloading.

- **Real-World Example:** You have a Calculator class with two add methods: one that takes two integers (add(int, int)) and one that

Secret_Syed

takes two floating-point numbers (add(double, double)). The compiler knows exactly which one to use based on the arguments you provide.

23. What is Run-Time (Dynamic) Polymorphism?

This is when the decision of which method to call is delayed until run time. It's achieved through method overriding.

- **Real-World Example:** You have a Vehicle class with a move() method. A Car subclass and a Boat subclass both override this method. If you have a variable of type Vehicle that holds a Boat object, calling move() will execute the boat's version of the method. The system figures this out "on the fly" as the program runs.

24. What is Method Overloading?

Method overloading is defining multiple methods in the same class with the same name but with different parameters (either different types or a different number of parameters). This is an example of static polymorphism.

25. What is Method Overriding?

Method overriding is when a subclass provides its own specific implementation of a method that is already defined in its parent class. The method signature (name and parameters) must be the same. This is the key to run-time polymorphism.

26. What is the difference between Overloading and Overriding?

Overloading is about having multiple methods with the same name but different signatures in the same class. Overriding is about a subclass providing a new implementation for a method from its parent class with the same signature. Overloading is resolved at compile time; overriding is resolved at run time.

27. What is a virtual function?

A virtual function is a member function in a base class that you expect to be overridden in derived classes. In languages like C++, you use the virtual keyword to declare that a function can be overridden, enabling run-time polymorphism.

Part 4: Constructors & Destructors

Secret Syed

28. What is a Constructor?

A constructor is a special method that is automatically called when an object of a class is created. Its primary job is to initialize the object's state (its data members).

- **Analogy:** A constructor is like the setup and assembly process on a factory line. When a new car (object) is created, the constructor is the process that installs the engine, attaches the wheels, and paints it, ensuring it's in a valid state before it leaves the factory.

29. What are the rules for a constructor?

A constructor must have the exact same name as the class, and it does not have a return type, not even void.

30. What are the types of constructors?

- **Default Constructor:** A constructor with no parameters.
- **Parameterized Constructor:** A constructor that accepts parameters to initialize the object with specific values.
- **Copy Constructor:** A constructor that creates a new object as a copy of an existing object.

Secret_Syed

31. Can a constructor be overloaded?

Yes, absolutely. This is very common. You can have multiple constructors in the same class as long as they have different parameter lists, allowing you to create objects in different ways.

32. What is a Destructor?

A destructor is a special method that is automatically called when an object is destroyed or goes out of scope. Its job is to clean up any resources the object might have acquired during its lifetime (like closing files or network connections).

- **Analogy:** A destructor is like the teardown crew after a concert. Once the show (the object's life) is over, the destructor comes in to turn off the lights, unplug the equipment, and clean up the venue, releasing the resources for others to use.

Part 5: Advanced Concepts & Keywords

33. What is a static keyword?

The static keyword means something belongs to the class itself, not to any individual instance (object) of the class. There is only one copy of a static member, shared by all objects of that class.

- **Real-World Example:** In a Car class, the numberOfWheels could be a static variable (all cars have 4), while color would be an instance variable (each car has its own color). You can access a static member using the class name directly, without creating an object.

34. What is the this keyword?

The this keyword is a reference to the current object—the object whose method is being called. It's used to distinguish between instance variables and local parameters when they have the same name.

- **Analogy:** this is like saying "me" or "myself" in a conversation. If you are setting a person's name with setName(String name), you would use this.name = name; to clarify that you are setting "my own" name to the value that was passed in.

35. What is the final (or sealed/const) keyword?

This keyword is used to declare something as unchangeable.

- **final variable:** A constant. Its value cannot be changed once assigned.
- **final method:** Cannot be overridden by a subclass.
- **final class:** Cannot be inherited from.

36. What is Garbage Collection?

Garbage collection is an automatic memory management feature in languages like Java and C#. The garbage collector automatically finds and reclaims memory that is occupied by objects that are no longer in use by the program, preventing memory leaks.

- **Analogy:** The garbage collector is like an automatic cleaning crew for your computer's memory. It periodically walks through, finds any toys (objects) that no one is playing with anymore, and puts them back in the toy box (frees the memory).

Sabrut Syed

37. What is an Exception?

An exception is an error or an unexpected event that occurs during the execution of a program, disrupting its normal flow.

38. What is Exception Handling?

Exception handling is a mechanism to gracefully handle runtime errors. Using try-catch blocks, you can "try" a piece of code that might fail, and "catch" the exception if it does, allowing you to handle the error without crashing the entire program.

- **Real-World Example:** When your app tries to download a file, you'd wrap the download code in a try block. In the catch block, you'd handle a potential NetworkException by showing the user a friendly message like "Could not connect to the internet," instead of just crashing.

39. What is the difference between an Error and an Exception?

An Exception is typically a recoverable error that your program can and should handle (like FileNotFoundException). An Error usually represents a more serious, unrecoverable problem with the system itself (like OutOfMemoryError), which your application generally cannot handle.

Part 6: Relationships Between Objects

40. What is Association?

Association is a relationship between two separate classes that establishes a link between their objects. It's the most general "uses-a" or "has-a" relationship.

- **Real-World Example:** A Doctor and a Patient. They are related, but their lifecycles are independent. A doctor can exist without a specific patient, and a patient can exist without a specific doctor.

41. What is Aggregation?

Aggregation is a specialized form of association that represents a "has-a" relationship. It implies a whole-part relationship, but the part can exist independently of the whole.

- **Analogy:** A Car "has-a" Wheel. The car is the whole, and the wheel is the part. But if you destroy the car, the wheel can still exist on its own

and could be used on another car.

42. What is Composition?

Composition is a stricter form of aggregation. It also represents a "has-a" relationship, but the part cannot exist independently of the whole. The whole is responsible for the creation and destruction of its parts.

- **Analogy:** A House is "composed of" Rooms. The house is the whole, and the rooms are the parts. If you demolish the house, the rooms are destroyed along with it. They cannot exist on their own.

43. What is the difference between Aggregation and Composition?

The key difference is the lifecycle. In Aggregation, the part has its own lifecycle. In Composition, the part's lifecycle is tied to the whole. Composition is a "death relationship."

44. What is Coupling?

Coupling is the measure of how much two classes depend on each other. Low coupling is desirable; it means a change in one class is less likely to break another class.

- **Analogy:** Low coupling is like having modular stereo components connected by standard cables. You can swap out your CD player for a new one without affecting your amplifier. High coupling is like an all-in-one stereo where if the tape deck breaks, the whole unit is useless.

Secret Syed

45. What is Cohesion?

Cohesion is the measure of how closely related and focused the responsibilities of a single class are. High cohesion is desirable; it means a class is designed to do one thing and do it well.

- **Analogy:** A highly cohesive class is like a sharp, well-made screwdriver. It has one clear purpose. A class with low cohesion is like a Swiss Army knife with too many flimsy tools; it tries to do everything but does nothing particularly well.

Part 7: The Rules of Good Design (SOLID)

46. What are the SOLID principles?

SOLID is an acronym for five fundamental principles of object-oriented design that help create software that is understandable, flexible, and maintainable.

47. S: Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning it should have only one job or responsibility.

- **Analogy:** A Swiss Army knife violates SRP. It's a knife, a screwdriver, a can opener, etc. A better design is to have a separate, high-quality screwdriver and a separate, high-quality knife. In code, don't create a God class that handles user authentication, database connections, and email notifications all at once.

48. O: Open/Closed Principle (OCP)

Software entities (classes, modules, etc.) should be open for extension but closed for modification. You should be able to add new functionality without changing existing, tested code.

- **Analogy:** Think of a smartphone. It's "closed" for modification (you can't easily open it up and change its internal wiring). But it's "open" for extension—you can add new functionality by installing apps from the App Store without ever changing the phone's core OS.

49. L: Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In simple terms, a subclass must be a true substitute for its parent.

- **Analogy:** If you have a Bird class, and a Duck is a subclass of Bird, then any code that works with a Bird object must also work perfectly if you give it a Duck object. An Ostrich class might violate this if it can't fly(), breaking code that expects all birds to fly.

50. I: Interface Segregation Principle (ISP)

No client should be forced to depend on methods it does not use. It's better to have many small, specific interfaces than one large, general-purpose one.

- **Analogy:** Don't create one giant RestaurantWorker interface with methods for cookFood(), takeOrder(), and washDishes(). A chef

Sajjad Syed

shouldn't be forced to know how to wash dishes. Instead, create separate Cook, Waiter, and Dishwasher interfaces.

51. D: Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces). This decouples your code.

- **Analogy:** A high-level lamp shouldn't be directly wired to a specific brand of light bulb (a low-level detail). Instead, both the lamp and the bulb should depend on a common abstraction: the standard light socket interface. This allows you to easily swap out any brand of bulb that fits the standard socket.

Part 8: Common Blueprints (Design Patterns)

52. What is a Design Pattern?

A design pattern is a general, reusable, and well-tested solution to a commonly occurring problem within a given context in software design. It's not a finished piece of code, but a template for how to solve a problem.

53. What are the main types of Design Patterns?

- **Creational Patterns:** Provide ways to create objects, increasing flexibility. (e.g., Singleton, Factory).
- **Structural Patterns:** Deal with how classes and objects are composed to form larger structures. (e.g., Adapter, Decorator, Facade).
- **Behavioral Patterns:** Are concerned with communication between objects and how they interact. (e.g., Observer, Strategy).

54. What is the Singleton Pattern?

A creational pattern that ensures a class has only one instance and provides a single, global point of access to it.

- **Real-World Example:** A system can only have one ConfigurationManager or one Logger instance. The Singleton pattern ensures that no matter how many times you ask for the manager object, you always get back the exact same one.

Secret_Syed

55. What is the Factory Pattern?

A creational pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

- **Analogy:** Imagine a logistics company. The main Logistics class has a createTransport() method. A RoadLogistics subclass will implement this to create a Truck object, while a SeaLogistics subclass will implement it to create a Ship object. The client code doesn't need to know which type of transport it's getting.

56. What is the Strategy Pattern?

A behavioral pattern that allows you to define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

- **Real-World Example:** In a map application, you might have different strategies for calculating a route: WalkingStrategy, DrivingStrategy, PublicTransportStrategy. The main Map object can be configured with any of these strategies to calculate the path, and you can switch between them at run time.

57. What is the Observer Pattern?

A behavioral pattern where an object (the "subject") maintains a list of its dependents (the "observers") and notifies them automatically of any state changes.

- **Real-World Example:** A YouTube channel is the subject. You and other subscribers are the observers. When the creator uploads a new video (a state change), the YouTube system automatically sends a notification to all subscribers.

58. What is the Decorator Pattern?

A structural pattern that allows you to add new functionality to an object dynamically by wrapping it in a special "decorator" object.

- **Analogy:** Think of ordering a coffee. You start with a plain Coffee object. You can then "decorate" it with a MilkDecorator, and then

Secret_Syed

further decorate that with a SugarDecorator. Each decorator adds cost and changes the description without altering the original coffee object.

59. What is the Adapter Pattern?

A structural pattern that acts as a bridge between two incompatible interfaces. It allows objects with incompatible interfaces to work together.

- **Analogy:** A travel power adapter. Your laptop charger has a US plug, but the wall socket in Europe is different. The adapter sits in the middle, making the two incompatible interfaces work together seamlessly.

60. What is the Facade Pattern?

A structural pattern that provides a simplified, unified interface to a complex subsystem of classes.

- **Analogy:** Ordering a product from Amazon is a facade. You just click "Buy Now." Behind that single button, a complex subsystem is at work: checking inventory, processing your payment, verifying your address, and arranging for shipping. The facade hides all that complexity from you.

Secret_Syed

Bonus Round: The Final Gauntlet (61-100)

61. What is an object reference? An object reference is a variable that holds the memory address of an object, not the object itself. It's a pointer or a "handle" to the object.

62. What is object cloning? The process of creating an exact copy of an object, with the same state as the original.

63. What is the difference between shallow and deep copy?

- **Shallow Copy:** Copies the main object but only copies the *references* to any objects it contains. Both old and new objects point to the same internal objects.
- **Deep Copy:** Copies the main object *and* recursively copies all the

objects it contains, creating a completely independent duplicate.

64. What is a pure virtual function (or abstract method)? A method in a base class that has no implementation and *must* be overridden by any concrete subclass. This is how interfaces are often implemented.

65. Can you override a private method? No. Private methods are not visible to subclasses, so they cannot be overridden.

66. Can you override a static method? No. Static methods belong to the class, not the object. You can "hide" a static method by defining another one with the same signature in the subclass, but this is not run-time polymorphism.

67. What is constructor chaining? The practice of calling one constructor from another constructor within the same class or from a subclass constructor calling a parent class constructor. This avoids code duplication.

68. What is an immutable object? An object whose state cannot be changed after it is created.

- **Real-World Example:** The String class in Java is immutable. When you "change" a string, you are actually creating a brand new string object.

69. Why is immutability useful? Immutable objects are inherently thread-safe and are great for use as keys in hash maps because their hash code will never change.

70. What is the difference between == and .equals() in Java?

- **==:** Checks if two references point to the exact same object in memory.
- **.equals():** A method that checks if two objects are "meaningfully" equal. By default, it does the same as ==, but it's often overridden

Secret_Syad

(like in the String class) to compare the actual content of the objects.

71. Can you prevent a class from being inherited? Yes, by declaring the class as final in Java or C++, or sealed in C#.

72. What is the "is-a" relationship? This represents inheritance. A Car "is-a" Vehicle.

73. What is the "has-a" relationship? This represents association, aggregation, or composition. A Car "has-a" Wheel.

74. What is early (static) binding? The process of resolving method calls at compile time. This is used for overloaded methods and static methods.

75. What is late (dynamic) binding? The process of resolving method calls at run time, based on the actual type of the object. This is used for overridden methods and is the essence of polymorphism.

76. What is the difference between a structure and a class? In C++, the only difference is default access: structure members are public by default, while class members are private by default. In C#, structures are value types, while classes are reference types.

77. What is a friend function or friend class? In C++, a friend is a function or class that is granted special access to the private and protected members of another class. It's a way to selectively break encapsulation.

78. What is operator overloading? A feature in languages like C++ that allows you to redefine the way standard operators (like +, -, ==) work for your own custom classes.

79. Can you overload a destructor? No. A destructor has a fixed signature and cannot be overloaded.

Sagar Syed

80. What is a virtual destructor? In C++, if you are deleting a derived class object through a base class pointer, you need to declare the base class destructor as virtual. This ensures that the derived class's destructor is called first, preventing resource leaks.

81. What is the "new" keyword? The new keyword is an operator used to dynamically allocate memory for a new object and call its constructor.

82. What is an anonymous object? An object that is created without being assigned to a reference variable. It is used for an immediate, one-time task, after which it becomes eligible for garbage collection.

83. What is an abstract data type (ADT)? An ADT is a mathematical model for a data type, defined by its behavior (the operations you can perform on it) rather than its implementation. A List or a Stack are ADTs.

84. How does OOPs support ADTs? Classes are the perfect way to implement ADTs. A class can encapsulate the data structure and provide public methods that correspond to the ADT's operations, hiding the implementation details.

85. What is the relationship between OOPs and Design Patterns? OOPs provides the fundamental tools (like inheritance and polymorphism). Design patterns are the clever, proven recipes that show you how to combine those tools to solve common architectural problems.

86. What is dependency injection (DI)? A design pattern used to implement Inversion of Control. It's the practice of giving an object its dependencies from an external source, rather than having the object create them itself. This makes code more modular and testable.

- **Analogy:** Instead of a Car building its own Engine, the Engine is built elsewhere and "injected" into the Car at the factory. This allows you

Scan to CVFD

to easily swap in a different kind of engine later.

87. What is Inversion of Control (IoC)? A broad design principle where the control over the flow of a program is transferred from the application code to a framework or container. Dependency Injection is one way to achieve IoC.

88. What is the Law of Demeter (Principle of Least Knowledge)? A design guideline that says an object should only talk to its immediate friends and not to strangers. It helps to reduce coupling.

- **Analogy:** To pay a cashier, you give your credit card to the cashier. You don't reach into the cashier's wallet yourself.

89. What is the difference between an SDK and an API?

- **API (Application Programming Interface):** The set of rules and definitions for how to talk to a service. It's the menu at a restaurant.
- **SDK (Software Development Kit):** A full set of tools that helps you build applications for a specific platform. It usually includes libraries, documentation, code samples, and the API itself. It's the menu, plus the pots, pans, and ingredients.

Secret Sylo

90. What is serialization? The process of converting an object's state into a format (like a byte stream or JSON) that can be stored (e.g., in a file) or transmitted (e.g., over a network) and then recreated later.

91. What is a marker interface? An empty interface with no methods or fields. It's used to "mark" a class as having a certain capability. (e.g., Java's Serializable interface).

92. What is a pure function? A function whose return value is determined only by its input values, with no observable side effects. Pure functions are predictable and easy to test.

93. How does functional programming relate to OOPs? They are

different paradigms, but they can be used together. Modern OOP languages have incorporated many functional concepts (like lambdas and streams) to write more concise and expressive code.

94. What is a use case for the Facade pattern in a real system? A DatabaseFacade class that provides simple methods like getUser(id) or saveOrder(order), hiding the complex underlying logic of creating connections, building SQL queries, and managing transactions.

95. What is a use case for the Strategy pattern in a real system? An e-commerce site using different DiscountStrategy objects. One might be for a percentage discount, another for a "buy one, get one free" offer. The ShoppingCart can be configured with the appropriate strategy to calculate the final price.

96. What is a use case for the Observer pattern in a real system? A spreadsheet application. The data model is the "subject," and various charts and graphs are the "observers." When you change a value in a cell, the subject notifies all the charts, which then automatically update themselves.

97. What is a use case for the Adapter pattern in a real system? Your application uses a modern logging library, but you need to integrate an old, third-party component that uses a completely different logging format. You would write an Adapter that makes the old component's logging calls compatible with your modern library.

98. What is the Builder Pattern? A creational pattern used to construct a complex object step by step. It's useful when an object has many optional parameters or requires a specific creation process.

- **Analogy:** It's like ordering a custom sandwich at Subway. You start with a base object (the bread) and then call methods to add each ingredient (.addLettuce(), .addCheese(), .addToast()) before finally

calling .build() to get the finished sandwich.

99. What is the Prototype Pattern? A creational pattern that lets you create new objects by copying an existing object (a "prototype"), rather than creating them from scratch. This is useful when the creation process is expensive.

- **Real-World Example:** In a game, instead of creating a new Enemy object from scratch every time (which might involve loading files from disk), you create one prototype enemy and then just clone it to quickly create hundreds of identical enemies.

100. How would you design a simple ATM system using OOPs principles?
You would model the real world with objects:

- A Customer class to hold user data.
- A BankAccount class (encapsulating the balance) with methods like deposit() and withdraw().
- An ATM class (as a Facade) with methods like insertCard(), enterPin(), and dispenseCash().
- A Transaction class to represent each operation.
- Inheritance could be used for different account types, like SavingsAccount and CheckingAccount, which might have different withdrawal rules (Polymorphism).

Secret_Syed