INTERVIEW PREPARATION EBOOK

ULTIMATE CRASH PREPARATION

Abstract

This interview preparation book provides a focused guide covering programming fundamentals, DSA theory, and OOPs concepts like encapsulation and inheritance. It includes detailed SQL topics such as queries, joins, and normalization. The book also features project-based questions to help explain your work confidently. HR questions and behavioral tips prepare you for the non-technical round. Key data structures and algorithms like arrays, trees, linkedlist, stack, and queues are covered with practical examples. Ideal for quick revision, this book strengthens your understanding of OOPs, SQL, DSA theory, project explanations, and HR readiness for technical interviews.

Praful Sharma

jobsknacademy@gmail.com

Contents

Object Oriented Programming

What is OOPS? Explain the advantages.

OOPs (Object-Oriented Programming System) is a programming paradigm based on the concept of "objects", which can contain **data** (fields or attributes) and **methods** (functions or procedures) that operate on the data.

---

4 Pillars of OOPs

1. **Encapsulation**
    - Wrapping data and code (methods) into a single unit (class).
    - Access to data is controlled using access modifiers (private, public, protected).
    - Prevents unauthorized access and increases security.
2. **Abstraction**
    - Hides complex implementation details and shows only the essential features.
    - Helps in reducing programming complexity and effort.
3. **Inheritance**
    - Mechanism where a new class (child) inherits properties and behavior from another class (parent).
    - Promotes code reusability.
4. **Polymorphism**

- Ability of a single function or method to behave differently based on context (many forms).
- Two types:
  - **Compile-time (Method Overloading)**
  - **Run-time (Method Overriding)**

Advantages of OOPs

1. **Modularity:** Code is organized into classes and objects, making it easier to manage.
2. **Reusability:** Inheritance allows code reuse.
3. **Security:** Encapsulation hides internal data, protecting it from outside interference.
4. **Flexibility:** Polymorphism enables flexibility and scalability.
5. **Maintainability:** Code is easier to update and maintain.
6. **Real-world modeling:** OOP reflects real-world entities, improving understanding and design.

Explain Class and Objects

Class

A **class** is a blueprint or template for creating **objects**. It defines a set of properties (also called attributes or fields) and behaviors (methods or functions) that the created objects will have.

- It does not occupy memory until an object is created from it.
- Think of it as a **design or prototype**.

Example (Conceptual):

Class: Car

Attributes: color, model, speed

Methods: start(), accelerate(), brake()

This describes what a **Car** is and what it can do, but it's not an actual car yet.

Object

An **object** is an instance of a class. It is a **real-world entity** that has a state (attributes) and behavior (methods).

- Each object has its own values for the properties defined in the class.
- Multiple objects can be created from the same class.

Example (Conceptual):

Object: myCar

Class: Car

myCar.color = "Red"

myCar.model = "Toyota"

myCar.speed = 120

Calling: myCar.start()

Here, myCar is an actual object based on the Car class. It has real values and can perform actions.

What is Constructor?

What is a Constructor?

A **constructor** is a special method that is **automatically called** when an object of a class is created. It is used to **initialize the object's attributes**.

Key Features (Generic)

Feature	Description
Name	Same as class name (C++, Java)
Called Automatically	Yes, when an object is created
Return Type	None (not even void)
Purpose	Initialize variables and set up the object
Overloading Allowed	Yes (in C++ and Java), No (Python)

C++ Constructor

- Same name as the class
- Can be overloaded
- No return type

```
#include <iostream>
using namespace std;

class Person {
public:
   string name;
   int age;

   // Constructor
   Person(string n, int a) {
      name = n;
      age = a;
   }
};

int main() {
   Person p("Alice", 30);
   cout << p.name << " is " << p.age << " years old.";
}
```

Notes:

- Person is the constructor.
- Automatically called when p is created.

Java Constructor

- Same name as the class
- Can be overloaded
- No return type

```
public class Person {
   String name;
```

```
    int age;

    // Constructor
    Person(String n, int a) {
        name = n;
        age = a;
    }

    public static void main(String[] args) {
        Person p = new Person("Alice", 30);
        System.out.println(p.name + " is " + p.age + " years old.");
    }
}
```

Notes:

- Java provides a **default constructor** if you don't define one.
- Constructor can be overloaded.

Python Constructor

- Defined using __init__() method
- Automatically called when an object is created
- No need to match class name

```
class Person:
    def __init__(self, name, age):  # Constructor
        self.name = name
        self.age = age

p = Person("Alice", 30)
print(f"{p.name} is {p.age} years old.")
```

Notes:

- __init__ is Python's constructor method.

- self is used to refer to the instance.

Types of Constructors

1. Default Constructor

- **Definition:** A constructor that takes no parameters.
- **Purpose:** Initializes objects with default values.
- **Note:** If no constructor is defined, many languages provide a default constructor automatically.

Example:

- **C++**

```
class MyClass {
public:
    MyClass() {
        // Default constructor
    }
};
```

- **Java**

```
class MyClass {
    MyClass() {
        // Default constructor
    }
}
```

- **Python**

Python does not have a default constructor in the same way, but if __init__ is not defined, a default one exists implicitly.

```
class MyClass:
    def __init__(self):
        # Default constructor
        pass
```

---

2. Parameterized Constructor

- **Definition:** A constructor that accepts arguments to initialize an object with custom values.
- **Purpose:** Allows initializing objects with specific data at creation time.

Example:

- **C++**

```
class MyClass {
    int x;
public:
    MyClass(int val) {
        x = val;
    }
};
```

- **Java**

```
class MyClass {
    int x;
    MyClass(int val) {
        x = val;
    }
}
```

- **Python**

```
class MyClass:
    def __init__(self, val):
        self.x = val
```

3. Copy Constructor (Mostly in C++)

- **Definition:** Constructor that creates a new object by copying an existing object.
- **Purpose:** Used for copying object data safely.
- **Note:** Not explicitly available in Java or Python.

Example (C++):

```
class MyClass {
   int x;
public:
   MyClass(int val) { x = val; }
   MyClass(const MyClass &obj) {  // Copy constructor
      x = obj.x;
   }
};
```

---

4. Static Constructor (Specific to some languages like C#)

- **Definition:** A constructor to initialize static data members.
- **Purpose:** Called once when the class is first loaded.
- **Note:** Not common in C++ or Java; Python uses class methods or module-level initialization instead.

---

5. Default Parameterized Constructor

- A constructor with parameters that have default values, allowing it to act as both default and parameterized.

Example (C++):

```
class MyClass {
   int x;
public:
   MyClass(int val = 0) {  // Acts as default and parameterized
      x = val;
   }
};
```

---

6. Private Constructor

- **Definition:** A constructor declared private.
- **Purpose:** Restricts object creation, often used in singleton design pattern.

Example (Java):

```
class Singleton {
   private static Singleton instance;
   private Singleton() {}  // Private constructor

   public static Singleton getInstance() {
      if(instance == null)
         instance = new Singleton();
      return instance;
   }
}
```

Summary Table

Constructor Type	Purpose	Language Support
Default Constructor	Initialize with default values	All (implicit if none defined)
Parameterized Constructor	Initialize with given parameters	All
Copy Constructor	Create object by copying another	Mainly C++
Static Constructor	Initialize static members	C#, some others
Private Constructor	Control object creation (e.g., Singleton)	All (via access specifiers)
Default Parameterized	Acts as both default & parameterized	C++

What is encapsulation

What is Encapsulation?

Encapsulation is one of the core principles of object-oriented programming (OOP). It refers to the **bundling of data (attributes)** and **methods (functions)** that operate on that data into a single unit — typically a class — and **restricting direct access** to some of the object's components.

Encapsulation helps in:

- Protecting the internal state of an object
- Controlling how the data is accessed or modified

- Improving code modularity and maintainability

Key Points of Encapsulation

- Data members are marked **private** (not directly accessible from outside the class).
- Access to data is provided via **public methods** like **getters** and **setters**.
- This protects the data and enforces rules on how it is used.

C++ Example:

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    void setName(string n) {
        name = n;
    }

    string getName() {
        return name;
    }

    void setAge(int a) {
        if (a >= 0) age = a;
    }

    int getAge() {
```

```
    return age;
  }
};

int main() {
  Person p;
  p.setName("Alice");
  p.setAge(30);
  cout << p.getName() << " is " << p.getAge() << " years old.";
}
```

Java Example:

```
public class Person {
  private String name;
  private int age;

  public void setName(String n) {
    name = n;
  }

  public String getName() {
    return name;
  }

  public void setAge(int a) {
    if (a >= 0) age = a;
  }

  public int getAge() {
    return age;
```

```
    }

    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");
        p.setAge(30);
        System.out.println(p.getName() + " is " + p.getAge() + " years old.");
    }
}
```

Python Example:

```
class Person:
    def __init__(self):
        self.__name = ""
        self.__age = 0

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_age(self, age):
        if age >= 0:
            self.__age = age

    def get_age(self):
        return self.__age

p = Person()
p.set_name("Alice")
```

```
p.set_age(30)

print(f"{p.get_name()} is {p.get_age()} years old.")
```

What is inheritance?

Inheritance is a core concept of Object-Oriented Programming (OOP) that allows a **class (child or subclass)** to **acquire properties and behaviors (methods)** from another class (parent or superclass). It enables **code reusability**, **hierarchical classification**, and allows one class to **extend** the functionality of another.

Key Points of Inheritance

- The **child class** inherits all **non-private** members of the parent class.
- The child can also have **its own additional methods** or **override** the parent's methods.
- Promotes **code reuse** and **logical hierarchy**.

Types of Inheritance (Common Forms)

- **Single Inheritance** – One child class inherits one parent.
- **Multilevel Inheritance** – A class inherits a child class which itself inherits another class.
- **Multiple Inheritance** – A class inherits from more than one class (not supported directly in Java).

C++ Example:

```
#include <iostream>
using namespace std;

class Animal {
public:
   void speak() {
      cout << "Animal speaks" << endl;
   }
};

class Dog : public Animal {
public:
   void bark() {
      cout << "Dog barks" << endl;
   }
};

int main() {
   Dog d;
   d.speak();  // Inherited from Animal
   d.bark();   // Own method
}
```

Java Example:

```
class Animal {
   void speak() {
```

```
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.speak(); // Inherited from Animal
        d.bark();  // Own method
    }
}
```

Python Example:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()  # Inherited from Animal
d.bark()   # Own method
```

Advantages of Inheritance

Benefit	Description
Code Reusability	Reuse existing logic instead of rewriting
Modularity	Improves organization of code
Extensibility	Easily extend or override base functionality

Types of inheritance

Inheritance allows classes to inherit properties and methods from other classes. Depending on how classes are related, inheritance can be classified into several types:

---

1. Single Inheritance

- One child class inherits from one parent class.

Example:

Animal → Dog

---

2. Multilevel Inheritance

- A class inherits from a child class, which itself inherits from another class.

Example:

Animal → Dog → Puppy

---

3. Multiple Inheritance

- A class inherits from more than one parent class.

Note:

- Supported in C++ and Python.
- Java does **not** support multiple inheritance with classes (but supports it with interfaces).

Example:

Class A, Class B → Class C inherits from both A and B

---

4. Hierarchical Inheritance

- Multiple child classes inherit from a single parent class.

Example:

Animal → Dog, Cat, Bird

---

5. Hybrid Inheritance

- Combination of two or more types of inheritance.

Example:

- Combination of multiple and multilevel inheritance.

Examples in C++, Java, and Python

---

1. Single Inheritance

C++

```
class Animal {
public:
    void speak() { cout << "Animal speaks\n"; }
```

```
};


class Dog : public Animal {
public:
    void bark() { cout << "Dog barks\n"; }
};
```

Java

```
class Animal {
    void speak() { System.out.println("Animal speaks"); }
}


class Dog extends Animal {
    void bark() { System.out.println("Dog barks"); }
}
```

Python


```
class Animal:
    def speak(self):
        print("Animal speaks")


class Dog(Animal):
    def bark(self):
        print("Dog barks")
```

---

2. Multilevel Inheritance

C++

```
class Animal {
public:
    void speak() { cout << "Animal speaks\n"; }
};
```

```
class Dog : public Animal {
public:
   void bark() { cout << "Dog barks\n"; }
};

class Puppy : public Dog {
public:
   void weep() { cout << "Puppy weeps\n"; }
};
```

Java

```
class Animal {
   void speak() { System.out.println("Animal speaks"); }
}

class Dog extends Animal {
   void bark() { System.out.println("Dog barks"); }
}

class Puppy extends Dog {
   void weep() { System.out.println("Puppy weeps"); }
}
```

Python

```
class Animal:
   def speak(self):
      print("Animal speaks")

class Dog(Animal):
   def bark(self):
      print("Dog barks")
```

```
class Puppy(Dog):
    def weep(self):
        print("Puppy weeps")
```

3. Multiple Inheritance

C++

```
class Printer {
public:
    void print() { cout << "Print\n"; }
};

class Scanner {
public:
    void scan() { cout << "Scan\n"; }
};

class AllInOne : public Printer, public Scanner {
};
```

Python

```
class Printer:
    def print(self):
        print("Print")

class Scanner:
    def scan(self):
        print("Scan")

class AllInOne(Printer, Scanner):
    pass
```

Java

Java does not support multiple inheritance with classes.

---

4. Hierarchical Inheritance

C++

```
class Animal {
public:
    void speak() { cout << "Animal speaks\n"; }
};

class Dog : public Animal {};
class Cat : public Animal {};
```

Java

```
class Animal {
    void speak() { System.out.println("Animal speaks"); }
}

class Dog extends Animal {}
class Cat extends Animal {}
```

Python

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

Access Specifiers

Access specifiers (also called access modifiers) define the **visibility** or **access level** of class members (variables and methods). They control **who can access or modify** the data or functions in a class.

Common Access Specifiers

Specifier	Description
Public	Members are accessible from anywhere.
Private	Members are accessible only within the class itself. Not accessible outside the class.
Protected	Members are accessible within the class and by derived (child) classes. Not accessible outside these.

Purpose

- Protect data from unauthorized access or modification.
- Encapsulate the internal details of a class.
- Control how objects interact with each other.

Examples in C++, Java, and Python

C++

```
class Example {
public:
   int x;       // accessible everywhere

private:
   int y;       // accessible only inside Example

protected:
   int z;       // accessible in Example and derived classes

public:
```

```
    void setY(int value) { y = value; }
    int getY() { return y; }
};
```

Java

```
public class Example {
    public int x;      // accessible everywhere
    private int y;     // accessible only inside Example
    protected int z;   // accessible inside Example and subclasses

    public void setY(int value) { y = value; }
    public int getY() { return y; }
}
```

Python

Python does not enforce strict access control but uses naming conventions:

- **Public members:** Normal names, accessible everywhere.
- **Protected members:** Single underscore prefix _name, meant for internal use and subclasses (by convention).
- **Private members:** Double underscore prefix __name, name-mangled to discourage access from outside.

```
class Example:
    def __init__(self):
        self.x = 10         # public
        self._y = 20        # protected (convention)
        self.__z = 30       # private (name mangling)

    def get_z(self):
        return self.__z
```

Summary Table

Language	Public	Private	Protected
C++	public:	private:	protected:
Java	public	private	protected
Python	normal name (x)	__name (name mangling)	_name (convention)

Polymorphism

What is Polymorphism?

Polymorphism is an important concept in Object-Oriented Programming that means **"many forms."**

It allows the same operation or method to behave differently on different classes or objects. This helps in writing flexible and reusable code.

Types of Polymorphism

1. **Compile-time Polymorphism (Static Binding)**
    - Achieved by **method overloading** or **operator overloading**.
    - The method to invoke is decided at compile time.
2. **Run-time Polymorphism (Dynamic Binding)**
    - Achieved by **method overriding** in inheritance.
    - The method to invoke is decided at runtime based on the object type.

Examples with Explanation

---

1. Compile-time Polymorphism (Method Overloading)

C++ Example:

```
#include <iostream>

using namespace std;


class Calculator {
```

```
public:
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {  // Overloaded method
        return a + b + c;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(2, 3) << endl;      // Outputs 5
    cout << calc.add(2, 3, 4) << endl;   // Outputs 9
}
```


Java Example:

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {  // Overloaded method
        return a + b + c;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));      // Outputs 5
        System.out.println(calc.add(2, 3, 4));   // Outputs 9
    }
```

```
}
```

Python Example:

```
class Calculator:
    def add(self, a, b, c=None):
        if c:
            return a + b + c
        else:
            return a + b

calc = Calculator()
print(calc.add(2, 3))       # Outputs 5
print(calc.add(2, 3, 4))    # Outputs 9
```

2. Run-time Polymorphism (Method Overriding)

C++ Example:

```
#include <iostream>
using namespace std;

class Animal {
public:
    virtual void sound() {  // Virtual function for runtime polymorphism
        cout << "Animal makes a sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
```

```
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* animal = new Dog();
    animal->sound();  // Calls Dog's sound() at runtime
}
```

Java Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound();  // Calls Dog's sound() at runtime
    }
}
```

Python Example:

```
class Animal:
```

```
    def sound(self):
        print("Animal makes a sound")


class Dog(Animal):
    def sound(self):
        print("Dog barks")


animal = Dog()
animal.sound()  # Calls Dog's sound()
```

Summary

Polymorphism Type	How it Works	Example
Compile-time (Static)	Method Overloading	Multiple methods with same name but different parameters
Run-time (Dynamic)	Method Overriding	Child class changes parent's method behavior

Summary of OOPS

1. What is OOP?

Answer: Programming paradigm based on objects that contain data and methods.

2. Name the four main pillars of OOP.

Answer: Encapsulation, Inheritance, Polymorphism, Abstraction.

3. What is a class?

Answer: Blueprint for creating objects.

4. What is an object?

Answer: Instance of a class.

5. What is encapsulation?

Answer: Wrapping data and methods into a single unit and restricting access.

6. What is inheritance?

Answer: Mechanism where one class acquires properties of another.

7. What is polymorphism?

Answer: Ability of an object to take many forms.

8. What is abstraction?

Answer: Hiding complex details and showing only essential features.

9. What is a constructor?

Answer: Special method to initialize an object when created.

10. What is method overloading?

Answer: Same method name with different parameters in the same class.

11. What is method overriding?

Answer: Child class provides its own implementation of a parent's method.

12. What is the difference between class and object?

Answer: Class is blueprint; object is instance of class.

13. What is access specifier?

Answer: Keywords to define visibility of class members (public, private, protected).

14. What is a destructor?

Answer: Special method called when an object is destroyed (mainly in C++).

15. What is the difference between private and protected access?

Answer: Private is accessible only within class; protected is accessible within class and derived classes.

16. What is a static member?

Answer: Member shared by all objects of a class.

17. What is an interface?

Answer: Abstract class with only abstract methods (mainly in Java).

18. What is a pure virtual function?

Answer: Function with no definition forcing derived classes to implement it (C++).

19. What is multiple inheritance?

Answer: A class inherits from more than one class.

20. What is hierarchical inheritance?

Answer: Multiple classes inherit from one base class.

21. What is multilevel inheritance?

Answer: A class inherits from a derived class.

22. What is dynamic binding?

Answer: Linking function call to function code at runtime.

23. What is static binding?

Answer: Linking function call at compile time.

24. What is message passing in OOP?

Answer: Objects communicate by sending messages (calling methods).

25. What is the use of this pointer?

Answer: Refers to the current object.

26. What is composition?

Answer: A class contains objects of other classes as members.

27. What is aggregation?

Answer: Weak form of composition; objects can exist independently.

28. What is a virtual function?

Answer: Allows method overriding and dynamic dispatch.

29. What is operator overloading?

Answer: Giving operators new meaning for user-defined types.

30. What is the difference between overloading and overriding?

Answer: Overloading is compile-time, overriding is run-time polymorphism.

31. What is a friend function?

Answer: A function that can access private members of a class (C++).

32. What is an abstract class?

Answer: Class with at least one pure virtual method; cannot instantiate.

33. What is the difference between an abstract class and an interface?

Answer: Abstract class can have defined methods; interface only abstract methods.

34. What is the default access modifier in classes?

Answer: Private in C++, public in Java.

35. What is method hiding?

Answer: Redefining a method in a child class with the same name but not overriding.

36. Can constructors be inherited?

Answer: No, constructors are not inherited.

37. What is a copy constructor?

Answer: Constructor that creates a new object as a copy of an existing object (C++).

38. What is an inline function?

Answer: Function expanded at compile time to reduce call overhead.

39. What is runtime polymorphism?

Answer: Overriding methods resolved during program execution.

40. What is the difference between class variable and instance variable?

Answer: Class variable is shared by all objects; instance variable is unique to each object

All Differences in oops

1. Class vs Object

Class	Object
Blueprint or template	Instance of a class
Defines properties & behavior	Holds actual data & behavior
No memory allocated	Memory allocated when created

---

2. Encapsulation vs Abstraction

Encapsulation	Abstraction
Hides internal data using access specifiers	Hides complex implementation details and shows only functionality
Focus on data protection	Focus on hiding complexity
Uses private/protected members	Uses abstract classes/interfaces

---

3. Inheritance vs Polymorphism

Inheritance	Polymorphism
Mechanism to acquire properties/methods from parent	Ability to take many forms, same interface different behavior
Represents "is-a" relationship	Represents "many forms" of behavior
Enables code reuse	Enables method overriding/overloading

4. Method Overloading vs Method Overriding

Method Overloading	Method Overriding
Same method name, different parameters	Same method name, same parameters but different class
Compile-time polymorphism	Run-time polymorphism
Occurs in the same class	Occurs in inheritance (child class)

---

5. Abstract Class vs Interface (Java)

Abstract Class	Interface
Can have both abstract and concrete methods	Only abstract methods (Java 7 and earlier)
Can have constructors	Cannot have constructors
Supports single inheritance	Supports multiple inheritance
Can have member variables	Variables are final static by default

---

6. Private vs Protected Access Specifier

Private	Protected
Accessible only within the class	Accessible within class and subclasses
More restrictive	Less restrictive

---

7. Aggregation vs Composition

Aggregation	Composition
"Has-a" relationship but loosely coupled	"Has-a" relationship tightly coupled
Child can exist without parent	Child cannot exist without parent
Example: Department and Professor	Example: Car and Engine

---

8. Constructor vs Method

Constructor	Method
Initializes object at creation	Performs operations
No return type	Has a return type or void
Same name as class	Can have any name

---

9. Static Binding vs Dynamic Binding

Static Binding	Dynamic Binding
Resolved at compile time	Resolved at runtime
Uses method overloading	Uses method overriding
Faster	Slower (due to runtime decision)

---

10. Virtual Function vs Pure Virtual Function (C++)

Virtual Function	Pure Virtual Function
Has implementation	No implementation (abstract)
Can be overridden	Must be overridden
Class can be instantiated	Makes class abstract

Tricky OOPS Questions

1. Can a constructor be inherited?

Answer: No, constructors are not inherited but a derived class can call the base class constructor explicitly.

---

2. What happens if a base class pointer points to a derived class object and calls a non-virtual function?

Answer: The base class version is called (static binding), not the derived class's method.

---

3. Can you override a private method?

Answer: No, private methods are not visible to derived classes, so they can't be overridden.

4. What is the difference between overloading and overriding a static method?

Answer: Static methods can be overloaded but not overridden because they are resolved at compile-time.

---

5. What is the difference between "has-a" and "is-a" relationships?

Answer: "Is-a" refers to inheritance; "has-a" refers to composition or aggregation.

---

6. Can a class be both abstract and concrete?

Answer: No, an abstract class contains at least one abstract method and cannot be instantiated.

---

7. What happens if a derived class does not override a pure virtual function?

Answer: The derived class becomes abstract and cannot be instantiated.

---

8. Is it possible to have a virtual constructor?

Answer: No, constructors cannot be virtual in most languages.

---

9. What's the difference between an interface and an abstract class?

Answer: Interfaces only declare methods (no implementation), while abstract classes can provide method implementations.

---

10. What is object slicing?

Answer: Assigning a derived class object to a base class object causes the derived part to be sliced off, losing data.

---

11. Why can't we override static methods?

Answer: Static methods belong to the class, not objects, so they are resolved at compile time.

---

12. Can you overload a constructor?

Answer: Yes, constructors can be overloaded to allow different ways of initializing objects.

---

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

Answer: Shallow copy copies object references; deep copy copies actual data to new memory locations.

---

14. Can multiple inheritance cause ambiguity? How is it resolved?

Answer: Yes, ambiguity occurs if two base classes have methods with the same name; resolved via scope resolution or interfaces.

---

15. Why is polymorphism called "runtime polymorphism"?

Answer: Because method overriding decisions are made during program execution.

---

16. Can you declare a class inside another class?

Answer: Yes, these are called nested or inner classes.

---

17. What's the use of the final keyword in Java OOP?

Answer: Prevents method overriding or inheritance of a class.

---

18. Can abstract classes have constructors?

Answer: Yes, abstract classes can have constructors which are called during subclass instantiation.

---

19. What is method hiding?

Answer: When a subclass declares a static method with the same name as a static method in the superclass, hiding it.

---

20. What is the difference between composition and aggregation?

Answer: Composition implies ownership and lifecycle dependency; aggregation implies a weaker association without lifecycle control.

21. Can an interface have constructors?

Answer: No, interfaces cannot have constructors.

---

22. What's the difference between an interface and a class with all abstract methods?

Answer: Interfaces cannot have instance variables or method implementations (Java 7 and earlier); abstract classes can.

23. Can a final method be overridden?

Answer: No, final methods cannot be overridden.

---

24. Is multiple inheritance supported in Java?

Answer: No, Java doesn't support multiple class inheritance but supports multiple interfaces.

---

25. What is a diamond problem in inheritance?

Answer: Ambiguity caused by multiple inheritance paths leading to the same base class.

---

26. How does Java solve the diamond problem?

Answer: By supporting multiple interfaces, not multiple classes.

---

27. Can you override a private method using the same name in a subclass?

Answer: No, private methods are not visible outside the class, so the subclass defines a new method, not overriding.

---

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

Answer: == compares references; .equals() compares object content.

---

29. Can static methods be abstract?

Answer: No, static methods cannot be abstract.

---

30. What is a singleton class?

Answer: A class that allows only one instance throughout the program.

---

31. What is method signature?

Answer: Method name and parameter list (types and order), excluding return type.

---

32. Can a subclass access private members of a superclass?

Answer: No, private members are not accessible directly by subclasses.

33. What is late binding?

Answer: Another term for runtime polymorphism where method calls are resolved at runtime.

---

34. What is object-oriented design principle "Favor composition over inheritance"?

Answer: Prefer building classes by composing objects rather than inheriting to reduce tight coupling.

---

35. Can a class implement multiple interfaces?

Answer: Yes, multiple interface implementation is allowed.

---

36. What happens if you declare a variable inside a method with the same name as a class variable?

Answer: The method variable shadows the class variable within its scope.

---

37. What is covariance in method overriding?

Answer: Overridden method returns a subtype of the original method's return type.

---

38. Can abstract methods have a body?

Answer: No, abstract methods do not have a body.

---

39. What is the difference between composition and inheritance?

Answer: Inheritance models "is-a" relationship; composition models "has-a" relationship.

---

40. Can constructors be overloaded?

Answer: Yes, constructors can have multiple forms with different parameters.

---

41. What is a marker interface?

Answer: An interface with no methods used to signal a property (e.g., Serializable in Java).

---

42. What is a default method in an interface (Java 8+)?

Answer: A method with implementation inside an interface using default keyword.

43. What is method shadowing?

Answer: When a subclass defines a static method with the same signature as a static method in the superclass.

---

44. Can a class implement an interface partially?

Answer: No, all interface methods must be implemented unless the class is abstract.

---

45. What happens if you don't provide a constructor?

Answer: A default no-argument constructor is provided automatically (if no other constructor is defined).

---

46. Can you override a method with a different return type?

Answer: Only if the return type is covariant (a subtype of the original return type).

---

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

Answer: Shallow copy copies references; deep copy copies actual objects recursively.

---

48. What is the use of super keyword?

Answer: Refers to the parent class and is used to call parent class methods or constructors.

---

49. What is the effect of making a class final?

Answer: It cannot be subclassed.

---

50. Can interface extend another interface?

Answer: Yes, interfaces can extend multiple interfaces.

DBMS

What is DBMS?

DBMS (Database Management System) is software that allows you to create, manage, and interact with databases. It provides tools to store, retrieve, update, and manage data efficiently and securely.

Key points about DBMS:

- It helps organize data in structured formats (tables, records).
- Supports multiple users simultaneously.
- Ensures data integrity and security.
- Provides query languages (like SQL) to access and manipulate data.
- Examples include MySQL, Oracle, SQL Server, and PostgreSQL.

What is RDBMS

RDBMS is a type of DBMS that stores data in **tables** (also called relations), where data is organized into rows and columns. It uses the relational model to manage data and supports operations using SQL (Structured Query Language).

Structure of RDBMS

- **Table (Relation):** The main structure, consisting of rows and columns. Each table stores data about a specific entity (e.g., Customers, Orders).
- **Row (Tuple):** Represents a single record in a table.
- **Column (Attribute):** Represents a field or property of the entity.
- **Primary Key:** A column or set of columns uniquely identifying each row.
- **Foreign Key:** A column that creates a link between two tables by referencing the primary key of another table.

Explain SQL Languages

SQL (Structured Query Language) is divided into several sublanguages based on the type of operations you perform on the database. The main categories are:

- **DDL** — Data Definition Language
- **DML** — Data Manipulation Language
- **DCL** — Data Control Language
- **TCL** — Transaction Control Language

---

1. DDL (Data Definition Language)

DDL commands define or modify the structure of the database objects like tables, indexes, and schemas.

Common DDL Commands:

Command	Purpose
CREATE	Create a new database object (table, index, view)
ALTER	Modify an existing database object (e.g., add a column)
DROP	Delete an existing database object
TRUNCATE	Remove all rows from a table quickly (structure remains)
RENAME	Rename a database object

Example:

```
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   Name VARCHAR(50),
   Department VARCHAR(50)
);
```

```
ALTER TABLE Employees ADD Salary INT;
```

```
DROP TABLE Employees;
```

---

2. DML (Data Manipulation Language)

DML commands deal with the manipulation of data stored in database tables.

Common DML Commands:

Command	Purpose
SELECT	Retrieve data from tables
INSERT	Insert new data into tables
UPDATE	Modify existing data in tables

Command	Purpose
DELETE	Remove data from tables

Example:

```
INSERT INTO Employees (EmployeeID, Name, Department) VALUES (1, 'John Doe', 'HR');
```

```
UPDATE Employees SET Department = 'Finance' WHERE EmployeeID = 1;
```

```
SELECT * FROM Employees;
```

```
DELETE FROM Employees WHERE EmployeeID = 1;
```

---

3. DCL (Data Control Language)

DCL commands control access to the data in the database.

Common DCL Commands:

Command	Purpose
GRANT	Give user permissions
REVOKE	Remove user permissions

Example:

```
GRANT SELECT, INSERT ON Employees TO User1;
```

```
REVOKE INSERT ON Employees FROM User1;
```

---

4. TCL (Transaction Control Language)

TCL commands manage transactions, which are sequences of SQL statements executed as a single unit.

Common TCL Commands:

Command	Purpose
COMMIT	Save all changes made in the transaction permanently
ROLLBACK	Undo changes made in the current transaction

Command	Purpose
SAVEPOINT	Set a point within a transaction to roll back to
SET TRANSACTION	Set transaction properties (e.g., isolation level)

Example:

```
BEGIN TRANSACTION;

UPDATE Employees SET Salary = Salary + 1000 WHERE Department = 'Sales';

SAVEPOINT IncreaseSalary;

-- If something goes wrong:
ROLLBACK TO IncreaseSalary;

COMMIT;
```

Summary Table:

Language	Purpose	Example Commands
DDL	Define/modify database structure	CREATE, ALTER, DROP, TRUNCATE
DML	Manipulate data	SELECT, INSERT, UPDATE, DELETE
DCL	Control access & permissions	GRANT, REVOKE
TCL	Manage transactions	COMMIT, ROLLBACK, SAVEPOINT

Explain all Keys

1. Primary Key

- **Definition:** A column or group of columns that uniquely identifies each row in a table.
- **Rules:**

- Cannot contain NULL.
- Must be unique for every row.
- Only one primary key is allowed per table.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100)
);
```

---

2. Candidate Key

- **Definition:** A column or set of columns that can uniquely identify any record in the table. A table can have multiple candidate keys.
- **Note:** One candidate key becomes the **primary key**, others are known as **alternate keys**.

Example:

In a Students table, both StudentID and Email might be unique — both are candidate keys.

---

3. Alternate Key

- **Definition:** A candidate key that was **not chosen** as the primary key.
- Still unique and not null, but not the primary key.

Example:

If StudentID is the primary key, then Email (also unique) is the alternate key.

---

4. Composite Key (or Compound Key)

- **Definition:** A primary key made of **two or more columns** to uniquely identify a record.
- Used when a single column is not enough to guarantee uniqueness.

Example:

```
CREATE TABLE Orders (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
```

);

---

5. Foreign Key

- **Definition:** A key used to link two tables. It refers to the **primary key** in another table.
- Ensures **referential integrity**.

Example:

```
CREATE TABLE Departments (
   DepartmentID INT PRIMARY KEY,
   DepartmentName VARCHAR(50)
);

CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   DepartmentID INT,
   FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```


---

6. Unique Key

- **Definition:** Ensures all values in a column are unique.
- Unlike primary key, it **can contain one NULL** (depends on the DBMS).
- You can have **multiple unique keys** in one table.

Example:

```
CREATE TABLE Users (
   UserID INT PRIMARY KEY,
   Email VARCHAR(100) UNIQUE
);
```

---

7. Super Key

- **Definition:** A set of one or more columns that **can uniquely identify** a record.
- **Super key ⊇ Candidate key** (includes candidate keys and any extra attributes).

Example:

In Employees(EmployeeID, Email, Name), both:

- EmployeeID and
- (EmployeeID, Email)
  are **super keys**, but only EmployeeID is a **candidate key**.

Quick Summary Table

Key Type	Uniqueness	Allows NULL	Purpose
Primary Key	Yes	No	Unique row identification
Candidate Key	Yes	No	Potential primary keys
Alternate Key	Yes	No	Candidate keys not chosen as primary
Composite Key	Yes	No	Combines multiple columns as key
Foreign Key	No	Yes	References another table's primary key
Unique Key	Yes	Yes (1 or more)	Enforces unique constraint
Super Key	Yes	Possibly	Superset of candidate keys

Explain Normalization

What is Normalization in DBMS?

Normalization is the process of organizing data in a database to minimize **redundancy** and improve **data integrity**. It involves dividing large, complex tables into smaller, related tables and defining relationships between them using keys.

Objectives of Normalization

- Eliminate duplicate data.
- Ensure logical data storage.
- Simplify queries and updates.
- Avoid data anomalies (insertion, update, deletion).
- Improve efficiency and consistency.

Normal Forms (NF)

Normalization is done through various **normal forms**, each with specific rules:

1NF (First Normal Form)

- Ensures each column contains atomic (indivisible) values.
- No repeating groups or arrays allowed.

Example:
Incorrect (Not in 1NF):

Student(ID, Name, Courses)

(1, 'Alice', 'Math, Physics')

Correct (In 1NF):

Student(ID, Name, Course)

(1, 'Alice', 'Math')

(1, 'Alice', 'Physics')

2NF (Second Normal Form)

- Must be in 1NF.
- No **partial dependency**: non-key attributes should depend on the whole primary key, not just part of it (applies to composite keys).

3NF (Third Normal Form)

- Must be in 2NF.
- No **transitive dependency**: non-key attributes should depend only on the primary key, not on other non-key attributes.

BCNF (Boyce-Codd Normal Form)

- A stricter version of 3NF.
- Handles anomalies not covered by 3NF when candidate keys exist.

Higher Normal Forms (optional in most systems)

- **4NF:** Removes multi-valued dependencies.
- **5NF:** Removes join dependencies.
- **6NF:** Deals with temporal databases and is rarely used.

Benefits of Normalization

- Reduces data redundancy.
- Improves data consistency.
- Easier maintenance and updates.
- Better data integrity.

Transactions in SQL

A **transaction** is a sequence of one or more SQL operations (like INSERT, UPDATE, DELETE) executed as a **single logical unit of work**. A transaction must be **completed fully or not at all** to ensure data integrity.

Transactions follow the **ACID** properties:

1. Atomicity

- All operations in a transaction are treated as a single unit.
- If any operation fails, the entire transaction is **rolled back**.

2. Consistency

- A transaction moves the database from one **valid state** to another, maintaining data integrity.

3. Isolation

- Transactions execute independently, ensuring that one transaction's operations are not visible to others until completion.

4. Durability

- Once a transaction is **committed**, the changes are permanent—even if there's a system failure.

Transaction States

1. **Active:** Transaction is currently executing.
2. **Partially Committed:** Final operation executed, waiting to be committed.
3. **Committed:** All operations successful, changes saved permanently.
4. **Failed:** One or more operations failed.
5. **Aborted:** Transaction rolled back, all changes undone.

SQL Commands Used in Transactions

Command	Description
BEGIN	Starts a new transaction
COMMIT	Saves all changes made in the transaction
ROLLBACK	Undoes all changes made in the transaction
SAVEPOINT	Sets a point within a transaction to roll back to
SET TRANSACTION	Sets transaction properties like isolation level

---

Example in SQL

```
BEGIN;

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 2;

COMMIT;
```

If any statement fails, you can do:

```
ROLLBACK;
```

ACID Properties

What is ACID?

ACID stands for:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

These four properties guarantee that database transactions are **processed reliably**, even in the event of system failures, concurrent access, or errors.

---

1. Atomicity – *"All or Nothing"*

- A transaction is **atomic**: either all its operations succeed, or none do.

- If any part of the transaction fails, the entire transaction is **rolled back**, and the database remains unchanged.

Example:

```
BEGIN;

UPDATE Accounts SET Balance = Balance - 1000 WHERE AccountID = 1;

UPDATE Accounts SET Balance = Balance + 1000 WHERE AccountID = 2;

-- If the second update fails

ROLLBACK;
```

---

2. Consistency – *"Valid State → Valid State"*

- Ensures the database remains in a **valid state** before and after the transaction.
- A transaction must follow **all integrity constraints, rules, and relationships**.

Example:

- You cannot insert a record with a foreign key that doesn't exist in the parent table.
- After a transaction, the total balance of all accounts must still match expected values.

---

3. Isolation – *"No Interference"*

- Ensures that **concurrent transactions** do not interfere with each other.
- One transaction's changes are not visible to others until it's committed.

Isolation Levels:

Level	Description
Read Uncommitted	Can read uncommitted data (dirty read)
Read Committed	Can only read committed data
Repeatable Read	Same query returns same result every time
Serializable	Transactions are completely isolated

Example Problem Without Isolation:

- Two users withdrawing from the same account at the same time may overdraw if isolation is not enforced.

---

4. Durability – *"Permanent Changes"*

- Once a transaction is **committed**, its changes are permanent—even if the system crashes.

- Data is written to **non-volatile storage**, ensuring persistence.

Example:

- After a COMMIT, even if the power goes out, changes remain in the database.

Summary Table

Property	Purpose	Ensures that...
Atomicity	All operations complete or none do	Partial updates don't corrupt the database
Consistency	Transactions preserve DB rules	DB goes from one valid state to another
Isolation	Transactions don't affect each other	Concurrency doesn't cause data conflicts
Durability	Committed changes are permanent	Data remains even after crashes

What are Joins in DBMS?

Joins are used in SQL to combine rows from two or more tables based on a related column, typically a primary key in one table and a foreign key in another.

Why Use Joins?

- To retrieve meaningful data spread across multiple tables.
- To utilize the relationships defined between tables using keys.

Types of Joins

1. INNER JOIN

- Returns only rows that have matching values in both tables.
- Rows without matches in either table are excluded.

Syntax:

```
SELECT A.name, B.salary
FROM Employee A
INNER JOIN Salary B ON A.emp_id = B.emp_id;
```

---

2. LEFT JOIN (or LEFT OUTER JOIN)

- Returns all rows from the **left** table, and matched rows from the right table.

- If there is no match in the right table, the result is NULL for right table columns.

Syntax:

```
SELECT A.name, B.salary
FROM Employee A
LEFT JOIN Salary B ON A.emp_id = B.emp_id;
```

---

3. RIGHT JOIN (or RIGHT OUTER JOIN)

- Returns all rows from the **right** table, and matched rows from the left table.
- If there is no match in the left table, the result is NULL for left table columns.

Syntax:

```
SELECT A.name, B.salary
FROM Employee A
RIGHT JOIN Salary B ON A.emp_id = B.emp_id;
```

---

4. FULL JOIN (or FULL OUTER JOIN)

- Returns all rows from both tables.
- Where there is a match, rows are combined.
- Where there is no match, NULLs are placed for missing values.

Syntax:

```
SELECT A.name, B.salary
FROM Employee A
FULL OUTER JOIN Salary B ON A.emp_id = B.emp_id;
```

---

5. CROSS JOIN

- Returns the **Cartesian product** of both tables.
- Every row of the first table is combined with every row of the second table.

Syntax:

```
SELECT A.name, B.salary
FROM Employee A
CROSS JOIN Salary B;
```

---

6. SELF JOIN

- A table is joined with itself.
- Useful when rows in the same table are related to each other.

Syntax:

```
SELECT A.name AS Employee, B.name AS Manager
FROM Employee A, Employee B
WHERE A.manager_id = B.emp_id;
```

Summary Table

Join Type	Description
INNER JOIN	Rows with matching values in both tables
LEFT JOIN	All rows from left table + matched rows from right table
RIGHT JOIN	All rows from right table + matched rows from left table
FULL JOIN	All rows from both tables, matched or not
CROSS JOIN	All combinations (Cartesian product)
SELF JOIN	Join a table with itself

DBMS Short Summary

1. **What is DBMS?**
   A software system to create, manage, and access databases efficiently.
2. **What is RDBMS?**
   A type of DBMS based on the relational model, storing data in tables.
3. **Difference between DBMS and RDBMS?**
   RDBMS supports relational constraints and keys; DBMS doesn't enforce relationships strictly.
4. **What is a table?**
   A collection of rows and columns representing structured data.
5. **What is a tuple?**
   A single row in a table.
6. **What is an attribute?**
   A column in a table.

7. **What is a primary key?**
   A unique, non-null identifier for table rows.
8. **What is a foreign key?**
   A key in one table that refers to the primary key of another table.
9. **What is a candidate key?**
   A column or set of columns that can uniquely identify rows.
10. **What is a super key?**
    A set of attributes that uniquely identify rows (may include extra attributes).

---

11. **What is SQL?**
    Structured Query Language used to interact with databases.
12. **What is DDL?**
    Data Definition Language (e.g., CREATE, ALTER, DROP).
13. **What is DML?**
    Data Manipulation Language (e.g., SELECT, INSERT, UPDATE, DELETE).
14. **What is DCL?**
    Data Control Language (e.g., GRANT, REVOKE).
15. **What is TCL?**
    Transaction Control Language (e.g., COMMIT, ROLLBACK, SAVEPOINT).
16. **What is a view?**
    A virtual table based on a query result.
17. **What is a stored procedure?**
    A precompiled set of SQL statements stored in the database.
18. **What is a trigger?**
    A procedure that runs automatically on specific events (e.g., insert).
19. **What is a join?**
    Combines rows from two or more tables based on related columns.
20. **Types of joins?**
    INNER, LEFT, RIGHT, FULL, CROSS, SELF.

---

21. **What is normalization?**
    Process of organizing data to reduce redundancy.
22. **1NF?**
    Eliminates repeating groups; ensures atomic values.
23. **2NF?**
    Removes partial dependency (must be in 1NF).

24. **3NF?**
    Removes transitive dependency (must be in 2NF).
25. **BCNF?**
    Stricter version of 3NF; every determinant must be a candidate key.
26. **What is denormalization?**
    Introducing redundancy for performance optimization.
27. **What is schema?**
    Structure or blueprint of a database (tables, columns, etc.).
28. **What is ER model?**
    Entity-Relationship model representing data as entities and relationships.
29. **What is an entity?**
    A real-world object or concept (e.g., Student, Course).
30. **What is a relationship?**
    An association between two or more entities.

---

31. **What is a transaction?**
    A sequence of operations performed as a single logical unit.
32. **What are ACID properties?**
    Atomicity, Consistency, Isolation, Durability.
33. **What is concurrency control?**
    Managing simultaneous transactions without conflict.
34. **What is a deadlock?**
    Two or more transactions waiting for each other to release resources.
35. **What is indexing?**
    Improves search performance by creating pointers to data.
36. **What is a cursor?**
    A database object used to retrieve row-by-row results.
37. **What is data integrity?**
    Accuracy and consistency of stored data.
38. **What is a constraint?**
    Rules applied to columns (e.g., NOT NULL, UNIQUE, CHECK).
39. **What is a subquery?**
    A query inside another SQL query.
40. **What is a correlated subquery?**
    A subquery that uses values from the outer query.

All Differences

1. DBMS vs RDBMS

Feature	DBMS	RDBMS
Data Structure	File-based or hierarchical	Table-based (relational model)
Support for Keys	No strict support	Supports primary key, foreign key
Relationships	Not enforced	Enforces relationships with keys
Normalization	Rarely supports	Supports normalization
Scalability	Smaller databases	Handles large-scale databases
Examples	XML, JSON files	MySQL, Oracle, SQL Server

---

2. Primary Key vs Unique Key

Feature	Primary Key	Unique Key
Uniqueness	Must be unique	Must be unique
Null values allowed	No	Yes, one NULL allowed
Purpose	Uniquely identifies a record	Uniquely identifies but allows NULL

---

3. DELETE vs TRUNCATE

Feature	DELETE	TRUNCATE
Operation type	DML	DDL
Can be rolled back	Yes	No
Triggers fired	Yes	No
Speed	Slower (row by row)	Faster (deallocates pages)

---

4. WHERE vs HAVING

Feature	WHERE	HAVING
Applied on	Rows before grouping	Groups after aggregation
Used with aggregate functions?	No	Yes

Feature	WHERE	HAVING
Filtering type	Row-level filter	Group-level filter

---

5. Clustered Index vs Non-Clustered Index

Feature	Clustered Index	Non-Clustered Index
Data storage order	Data stored in index order	Separate from data storage
Number per table	Only one	Multiple allowed
Speed for range queries	Faster	Slightly slower

---

6. Static SQL vs Dynamic SQL

Feature	Static SQL	Dynamic SQL
Compilation	Precompiled	Compiled at runtime
Flexibility	Less flexible	More flexible
Usage	Fixed queries	Queries constructed at runtime

---

7. 1NF vs 2NF vs 3NF

Normal Form	Description	Requirement
1NF	Atomic values, no repeating groups	No multi-valued attributes
2NF	1NF + no partial dependency	Non-key attributes depend on full primary key
3NF	2NF + no transitive dependency	Non-key attributes depend only on primary key

---

8. Physical Data Independence vs Logical Data Independence

Feature	Physical Data Independence	Logical Data Independence
Definition	Change in physical storage without affecting logical schema	Change in logical schema without affecting external views
Example	Change file organization	Add/remove fields in table

---

9. DELETE vs DROP

Feature	DELETE	DROP
Operation	Removes rows	Removes entire table
Can be rolled back	Yes	No
Structure	Table remains	Table structure removed

---

10. HAVING vs GROUP BY

Feature	HAVING	GROUP BY
Purpose	Filters groups after aggregation	Groups rows before aggregation
Usage	Used with aggregate functions	Defines groupings

---

11. Full Outer Join vs Inner Join

Feature	Full Outer Join	Inner Join
Result rows	All matched + unmatched rows from both tables	Only matched rows
NULLs in result	Present for unmatched rows	No

---

12. Primary Key vs Foreign Key

Feature	Primary Key	Foreign Key
Purpose	Uniquely identifies a record	References primary key in another table
Values allowed	Unique, no NULL	Can be duplicate, can be NULL

SQL Important Syntax

SQL Command	Syntax Example	Description
CREATE DATABASE	CREATE DATABASE database_name;	Create a new database
DROP DATABASE	DROP DATABASE database_name;	Delete an existing database
CREATE TABLE	CREATE TABLE table_name (column1 datatype, column2 datatype);	Create a new table

SQL Command	Syntax Example	Description
DROP TABLE	DROP TABLE table_name;	Delete a table
INSERT INTO	INSERT INTO table_name (col1, col2) VALUES (val1, val2);	Insert data into a table
SELECT	SELECT col1, col2 FROM table_name WHERE condition;	Retrieve data from a table
UPDATE	UPDATE table_name SET col1 = val1 WHERE condition;	Modify existing data
DELETE	DELETE FROM table_name WHERE condition;	Remove rows
ALTER TABLE	ALTER TABLE table_name ADD col datatype;	Add a new column
	ALTER TABLE table_name DROP COLUMN col;	Remove a column
WHERE	SELECT * FROM table_name WHERE condition;	Filter rows
ORDER BY	`SELECT * FROM table_name ORDER BY col ASC	DESC;`
GROUP BY	SELECT col, COUNT(*) FROM table_name GROUP BY col;	Group rows
HAVING	SELECT col, COUNT(*) FROM table_name GROUP BY col HAVING COUNT(*) > n;	Filter groups after aggregation
INNER JOIN	SELECT * FROM t1 INNER JOIN t2 ON t1.key = t2.key;	Join matching rows from two tables
LEFT JOIN	SELECT * FROM t1 LEFT JOIN t2 ON t1.key = t2.key;	Left table rows + matching right table
DISTINCT	SELECT DISTINCT col FROM table_name;	Get unique values
LIMIT	SELECT * FROM table_name LIMIT n;	Limit number of rows returned
CREATE INDEX	CREATE INDEX idx_name ON table_name(col);	Create index for faster queries
DROP INDEX	DROP INDEX idx_name;	Remove an index
COMMIT	COMMIT;	Save transaction changes
ROLLBACK	ROLLBACK;	Undo transaction changes
SUBQUERY	SELECT * FROM t1 WHERE col IN (SELECT col FROM t2 WHERE cond);	Query inside another query

Important SQL Queries

Table: departments

id	department_name
1	HR
2	Finance
3	IT
4	Marketing

Table: employees

id	name	salary	hire_date	department_id	manager_id
1	Alice	90000.00	2020-01-15	3	NULL
2	Bob	60000.00	2021-03-20	3	1
3	Charlie	70000.00	2019-07-30	2	NULL
4	David	65000.00	2022-02-25	2	3
5	Eva	48000.00	2023-01-10	1	NULL
6	Frank	30000.00	2023-04-01	1	5
7	Grace	75000.00	2021-12-15	4	NULL

1. **Select all records from a table**

SELECT * FROM employees;

Fetches all columns and rows from the employees table.

---

2. **Select specific columns**

SELECT name, salary FROM employees;

Fetches only name and salary columns.

3. **Filter records with WHERE**

SELECT * FROM employees WHERE salary > 50000;

Selects employees with salary greater than 50,000.

---

4. **Order results**

SELECT * FROM employees ORDER BY salary DESC;

Sorts employees by salary in descending order.

---

5. **Count rows**

SELECT COUNT(*) FROM employees;

Counts total number of employees.

---

6. **Group by department**

SELECT department, COUNT(*) FROM employees GROUP BY department;

Counts employees in each department.

---

7. **Filter groups with HAVING**

SELECT department, AVG(salary) FROM employees

GROUP BY department

HAVING AVG(salary) > 60000;

Shows departments with average salary above 60,000.

---

8. **Inner Join two tables**

SELECT e.name, d.department_name

FROM employees e

INNER JOIN departments d ON e.department_id = d.id;

Fetches employee names with their department names.

---

9. **Left Join two tables**

SELECT e.name, d.department_name

FROM employees e

LEFT JOIN departments d ON e.department_id = d.id;

Fetches all employees, showing department name if available.

---

10. **Insert new record**

INSERT INTO employees (name, salary, department_id) VALUES ('John', 70000, 3);

Adds a new employee record.

---

11. **Update existing records**

UPDATE employees SET salary = salary * 1.1 WHERE department_id = 3;

Increases salary by 10% for department 3 employees.

---

12. **Delete records**

DELETE FROM employees WHERE salary < 30000;

Removes employees with salary less than 30,000.

---

13. **Find distinct values**

SELECT DISTINCT department_id FROM employees;

Lists unique department IDs.

---

14. **Use subquery**

SELECT name, salary FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

Finds employees earning more than average salary.

---

15. **Limit rows returned**

SELECT * FROM employees ORDER BY salary DESC LIMIT 5;

Fetches top 5 highest paid employees.

Difficult Queries

1. **Find the second highest salary**

SELECT MAX(salary) FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

---

2. **Find employees who earn more than their manager**

SELECT e.name, e.salary, m.name AS manager_name, m.salary AS manager_salary

FROM employees e

JOIN employees m ON e.manager_id = m.id

WHERE e.salary > m.salary;

---

3. **Get the running total of salaries ordered by hire date**

SELECT name, salary, hire_date,

SUM(salary) OVER (ORDER BY hire_date) AS running_total

FROM employees;

---

4. **Find departments with more than 5 employees earning above average salary**

```
SELECT department_id, COUNT(*)
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees)
GROUP BY department_id
HAVING COUNT(*) > 5;
```

---

5. **Get the employee with the highest salary in each department**

```
SELECT department_id, name, salary
FROM employees e1
WHERE salary = (
  SELECT MAX(salary)
  FROM employees e2
  WHERE e2.department_id = e1.department_id
);
```


---

6. **Find employees who have the same salary as at least one other employee**

```
SELECT name, salary
FROM employees
WHERE salary IN (
  SELECT salary FROM employees
  GROUP BY salary
  HAVING COUNT(*) > 1
);
```

---

7. **Find the Nth highest salary (e.g., 3rd highest)**

```
SELECT DISTINCT salary
FROM employees
```

```
ORDER BY salary DESC
LIMIT 1 OFFSET 2;
```

---

8. **Find departments without any employees**

```
SELECT d.department_name
FROM departments d
LEFT JOIN employees e ON d.id = e.department_id
WHERE e.id IS NULL;
```

---

9. **Find the difference in days between each employee's hire date and today**

```
SELECT name, hire_date, DATEDIFF(CURRENT_DATE, hire_date) AS days_worked
FROM employees;
```

(Note: DATEDIFF syntax varies by DBMS.)

---

10. **Find employees who joined in the last 3 months**

```
SELECT name, hire_date
FROM employees
WHERE hire_date >= DATE_SUB(CURRENT_DATE, INTERVAL 3 MONTH);
```

Project Based Questions (Modify According to your project)

Question 1: Can you explain your project?

Sample Answer(Modify According to your project):

I developed a **Full Stack Task Management Web Application** where users can register, create task lists, assign priorities, and track deadlines. The application includes user authentication, CRUD operations for tasks, and real-time status updates.

- The **frontend** is built using **React.js**, with React Router for navigation and Axios for API calls. I used Tailwind CSS for UI styling.
- The **backend** is developed using **Node.js and Express.js**, exposing a RESTful API for task and user operations.
- For **authentication**, I used **JWT tokens**, stored in HTTP-only cookies to improve security.
- The **database** is **MongoDB**, accessed via **Mongoose**. I structured tasks and users in separate collections with reference relationships.
- I deployed the **frontend** on **Vercel** and the **backend** on **Render**. The project supports environment variables and has separate dev and production configurations.

The goal was to make it responsive, secure, and easy to scale with role-based access and modular architecture.

Question 2: What was the biggest technical challenge you faced and how did you solve it?

Sample Answer(Modify according to your project):

The biggest challenge I faced was handling **authentication across client and server**, especially making sure the JWT token persisted securely and wasn't vulnerable to XSS or CSRF attacks.

Initially, I stored the JWT token in localStorage, but that raised concerns about **XSS vulnerabilities**. I researched and moved to storing the JWT in **HTTP-only cookies**, which cannot be accessed via JavaScript

However, this introduced **CORS issues** and **cookie handling problems** during cross-origin requests. To fix this, I:

- Set up proper CORS headers on the Express server using the cors package with credentials: true.
- Enabled withCredentials in Axios requests from the frontend.
- Configured the backend to send cookies securely in production (with Secure and SameSite flags).

This challenge taught me a lot about **web security**, and helped me implement a more professional and secure auth flow.

Question 3: Can you describe your project architecture?

Sample Answer:

My project follows a **typical 3-tier architecture**: **Frontend (Client)**, **Backend (Server/API)**, and **Database**.

- The **frontend** is built using **React.js**. It's a Single Page Application (SPA) using React Router for navigation and Axios for making API calls. I structured it with components and custom hooks, and used context for managing authentication state.

- The **backend** is built using **Node.js with Express.js**, where each feature (like tasks, users, auth) is modularized into routes, controllers, and services. It exposes a RESTful API that the frontend consumes.
- **Authentication** is handled using **JWT**, where tokens are stored in HTTP-only cookies for secure session persistence.
- The **database** is **MongoDB**, accessed through **Mongoose**. I defined clear schemas for users and tasks. The user schema includes hashed passwords (using bcrypt), while the task schema includes fields like title, description, priority, due date, and a reference to the user.
- For **deployment**, the frontend is hosted on **Vercel**, and the backend is on **Render**. Both environments are configured to use environment variables, and I ensured HTTPS and domain routing are properly set up.
- I used **Postman** for API testing and **GitHub** for version control. The project follows RESTful conventions and is built with scalability in mind.

Question 4: How does your frontend communicate with the backend?

Sample Answer:

The frontend and backend communicate via **RESTful APIs** over HTTP, using **Axios** for making requests in the frontend.

- Each frontend action (e.g., logging in, adding a task) triggers an Axios call to the backend.
- For example, when a user logs in, the frontend sends a POST request to /api/auth/login with email and password. The backend validates the credentials, and if successful, it sends back a **JWT token** in an HTTP-only cookie.
- For subsequent requests like creating or fetching tasks, the token in the cookie is automatically sent with the request. The backend reads and verifies it to identify the user.
- On the frontend, I set withCredentials: true in Axios so that cookies are included in each request.
- On the backend, I configured **CORS middleware** to allow cross-origin requests from the frontend domain and to accept credentials.
- The API follows a **REST structure**, for example:
    - GET /api/tasks – fetch all tasks for the logged-in user
    - POST /api/tasks – create a new task
    - PUT /api/tasks/:id – update a task
    - DELETE /api/tasks/:id – delete a task

This design ensures **clear separation of concerns**, allows **stateless communication**, and makes the system easier to maintain and scale.

Question 5 . Why did you use this tech stack for your full stack project?

Sample Answer:

I chose my tech stack based on **scalability, developer efficiency, and community support**. Here's why I selected each component:

- **Frontend – React.js**:
  React is component-based and allows for building reusable UI elements. Its virtual DOM improves performance, and with tools like React Router and Axios, I could build a responsive and dynamic single-page application efficiently.
- **Backend – Node.js with Express.js**:
  Node.js is fast and event-driven, making it great for handling API requests. Express.js is lightweight and flexible for building RESTful APIs. The combination allowed me to use JavaScript end-to-end, simplifying development and reducing context switching.
- **Database – MongoDB with Mongoose**:
  MongoDB is schema-less and fits well with agile development. It works great with JSON-like data, which is ideal for JavaScript applications. Mongoose added structure and validation through schemas while still allowing flexibility.
- **Authentication – JWT**:
  JWT is a standard method for stateless authentication. I used it with HTTP-only cookies to improve security by avoiding XSS exposure. It's scalable and integrates well with both frontend and backend.
- **Deployment – Vercel & Render**:
  Vercel is optimized for frontend apps and supports continuous deployment. Render offers a simple way to deploy Express apps with environment variable support and automatic HTTPS. Together, they allowed me to easily separate and deploy the frontend and backend.

Overall, this stack gave me a **modern, full JavaScript ecosystem**, with tools and practices aligned with industry standards. It also helped me learn important concepts like component-based UI, API design, NoSQL databases, and secure web deployment.

HR Questions

1. Tell me about yourself.

Sample Answer:

Good morning/afternoon. My name is [Your Name], and I recently completed my [B.Tech/B.E./B.Sc./MCA] in [Your Branch] from [Your College/University Name]. During my academics, I developed a strong interest in software development, particularly full stack web development.

To apply my learning practically, I worked on a full stack project—a **Task Management Web App**—where users can create, track, and update tasks with deadlines. I used **React.js** for the frontend, **Node.js with Express** for the backend, and **MongoDB** as the database. The project helped me

understand end-to-end application development, REST APIs, authentication using JWT, and deploying applications on platforms like **Vercel** and **Render**.

Along with my technical skills, I've also worked on improving my problem-solving abilities through platforms like LeetCode and HackerRank, and I'm comfortable working with Git, Postman, and basic DevOps tools.

I'm someone who learns quickly, enjoys working on real-world problems, and believes in writing clean and maintainable code. I'm now looking for an opportunity where I can contribute to a real development team, learn from experienced professionals, and grow as a software engineer.

---

2. Why should we hire you?

Sample Answer:

I have a strong foundation in full stack development, I'm a quick learner, and I enjoy solving technical problems. I've also demonstrated the ability to take a project from design to deployment. Beyond coding, I work well in teams, communicate clearly, and am always open to feedback and improvement.

---

3. What are your strengths?

Sample Answer:

My biggest strengths are my problem-solving skills and adaptability. I'm comfortable learning new technologies quickly, and I enjoy breaking down complex problems into simpler parts. I'm also consistent and reliable when it comes to meeting deadlines.

---

4. What are your weaknesses?

Sample Answer:

I used to struggle with speaking up during team discussions, especially in larger groups. But recently I've been pushing myself to participate more in meetings and take initiative in group projects, which has helped me improve.

---

5. Where do you see yourself in 5 years?

Sample Answer:

I see myself becoming a skilled and experienced full stack developer who can contribute to both architecture and product planning. Over time, I'd like to take on mentorship responsibilities and maybe even explore roles in product design or technical leadership.

---

6. Tell me about a challenge you faced and how you handled it.

Sample Answer:

In my project, I had trouble implementing secure authentication. Tokens were not being stored safely, and login sessions failed unexpectedly. I researched the issue and learned to use HTTP-only cookies with JWT, which solved the problem securely. It taught me the value of debugging calmly and learning from documentation.

---

7. Are you comfortable working in a team?

Sample Answer:

Yes, definitely. I've worked on team projects during college and also collaborated with peers while building my final year project. I believe in clear communication, respecting different ideas, and dividing tasks based on strengths.

---

8. Why do you want to work at our company?

Sample Answer:

I'm impressed by your company's focus on modern technology and innovation. I looked at some of your products and the tech stack you use aligns with what I've worked on. More importantly, I'm excited by the opportunity to learn and grow in a collaborative environment like yours.

---

9. Are you willing to relocate or work in shifts?

Sample Answer:

Yes, I'm open to relocation and flexible with work shifts. I understand that learning opportunities and career growth often require adaptability.

---

10. Do you have any questions for us?

Sample Answer:

Yes, I'd love to know more about your current development projects and what the team structure looks like for new developers. Also, what does a typical learning path look like for freshers in your company?

Cross Questions

1. You said you're a quick learner. Can you give an example?

Sample Answer:

Yes, during my final year project, I had to use JWT for authentication, but I had no prior experience with it. I watched documentation videos, read articles, and within a day, I implemented secure login and token storage using HTTP-only cookies. It was fully functional and passed all test cases, and I even explained the flow to my teammates.

2. If you're so interested in development, why didn't you do an internship?

Sample Answer:

That's a fair question. I actively looked for internships, but due to timing and availability, I didn't get one. However, I used that time productively to build my full stack project from scratch, which gave me hands-on experience with real-world tools and challenges, similar to what an internship would provide.

---

3. What if you're assigned a technology you don't know?

Sample Answer:

I'd be absolutely fine with that. In fact, I see it as an opportunity to grow. During my project, I had to learn MongoDB, which was new to me. I dedicated time to tutorials and documentation, and within a week, I was writing queries, designing schemas, and connecting it with my backend. I believe adaptability is key in tech.

---

4. You mentioned teamwork as a strength. What if your team isn't cooperative?

Sample Answer:

I believe communication is the key to handling such situations. If there's an issue, I'd first try to understand their perspective and talk it out professionally. If that doesn't help, I'd involve a mentor or lead to resolve it respectfully. In the end, the focus should always be on the project, not personal differences.

---

5. How do we know you're not just memorizing answers for the interview?

Sample Answer:

I understand why you'd ask that. But everything I'm saying comes from personal experience. My answers are based on the real challenges I faced while building my project. If you'd like, I'm happy to open the code or walk through any part of it. I believe in being honest and practical rather than trying to sound perfect.

6. What if you don't get the role you're hoping for? Will you still join us?

Sample Answer:

Yes, I will. My goal is to work in a strong learning environment where I can grow technically and professionally. Even if the role is different from what I expected, I believe I can still contribute and learn. Once inside, I'll work hard and prove myself to earn the responsibilities I'm aiming for.

---

7. You've done a full stack project alone. Can you really work in a team?

Sample Answer:

That's true—I developed the project individually to strengthen my technical skills. But I've also worked in teams during academic assignments, hackathons, and group studies. I'm comfortable collaborating, dividing work, giving and receiving feedback, and helping others when needed. Solo work improved my focus, but teamwork brings new ideas—and I value both.

---

8. What if you are asked to do repetitive or non-development work initially?

Sample Answer:

I believe every task has value, especially at the beginning of a career. If I'm trusted with something, even if it's repetitive, I'll do it sincerely. Meanwhile, I'll keep learning and improving so I'm ready when more challenging work comes my way. I'm not afraid to start small.

---

9. You say you're committed, but what if you get a better offer later?

Sample Answer:

I value growth, but I also value commitment and professionalism. If I accept an offer, it means I've done my research and see a future there. I believe in staying, learning, and building something meaningful—not just hopping for money or titles. Loyalty reflects both ethics and long-term vision.

---

10. You come from a different background. How do you know you'll like IT work long-term?

Sample Answer:

That's a fair question. But I've explored IT deeply through my project, coding challenges, and self-learning. I've enjoyed solving real problems and building things from scratch. The process excites me—debugging, deploying, and improving. So I'm confident this is not just a temporary interest, but something I want to build a career in.

Situational Questions

1. Tell me about a time you missed a deadline. What did you do?

Sample Answer:

During my final year project, I underestimated the time required for integrating user authentication. I informed my mentor early, broke the task into smaller parts, and worked extra hours to complete it with quality. I learned to buffer time and plan dependencies better in future tasks.

---

2. Describe a situation where you had a conflict with a team member.

Sample Answer:

During a group assignment, a teammate wasn't contributing equally. Instead of accusing, I asked if they were facing any difficulty. It turned out they didn't understand the module well. I offered to

explain, and we ended up collaborating more effectively. This taught me the value of empathy in conflict resolution.

---

3. Have you ever worked under pressure? How did you handle it?

Sample Answer:

Yes, during exam week I also had a coding challenge submission. I created a strict schedule, cut distractions, and focused on one task at a time. I submitted the challenge on time and did well in exams. Pressure taught me prioritization and time management.

---

4. Tell me about a time you took initiative.

Sample Answer:

In one of our labs, many students were confused about using Git. I created a short step-by-step guide and shared it in the class group. It helped several students and even the faculty appreciated it. It felt good to take initiative and help others learn.

---

5. Describe a situation where you had to learn something quickly.

Sample Answer:

While building my full stack project, I realized I needed to implement JWT-based authentication, which I hadn't done before. I read documentation and watched tutorials, and in two days, I had it up and running securely. It proved that I can learn on the go when required.

---

6. Tell me about a time you failed. What did you learn?

Sample Answer:

I once submitted a buggy version of a project because I didn't test it enough. The result was poor feedback. I learned the importance of thorough testing and review. Since then, I always run test cases and do dry runs before submitting any work.

---

7. Describe a time when you had to make a difficult decision.

Sample Answer:

I had to choose between preparing for a campus drive and continuing with a hackathon team. After assessing priorities, I chose the placement preparation because it had long-term benefits. I informed my team well in advance and helped them transition the work. It taught me decision-making under pressure.

---

8. What would you do if you were assigned a task outside your skillset?

Sample Answer:

I would approach it as a learning opportunity. I'd start by researching and asking peers or seniors for guidance. I've done this during my project as well—like when I had to integrate email notifications. I had never done it before, but I figured it out through trial, documentation, and persistence.

---

9. Describe a time when you had multiple responsibilities. How did you manage?

Sample Answer:

During final exams, I also had a side freelance assignment. I created a daily planner, split the time blocks, and avoided multitasking. I managed to deliver both without compromising quality. Planning and focus helped me handle both efficiently.

---

10. How would you handle receiving criticism from your manager or peer?

Sample Answer:

I would listen carefully, ask clarifying questions if needed, and take the feedback positively. I see criticism as a way to improve. In fact, during my project review, a senior pointed out inefficient API structuring. I went back, studied REST principles, and refactored my code. It made me better.

CS Fundamentals (Short Answers)

Operating Systems

1. **What is an Operating System?**
   An OS is system software that manages hardware, software resources, and provides services for computer programs.
2. **Types of Operating Systems?**
   Batch, Time-sharing, Distributed, Real-time, Network, and Mobile OS.
3. **What is a Process?**
   A program in execution with its own memory space and system resources.
4. **What is a Thread?**
   A lightweight subprocess; a unit of CPU execution within a process.
5. **Difference between Process and Thread?**
   Processes have separate memory; threads share the same memory within a process.
6. **What is Deadlock?**
   A condition where processes wait indefinitely for resources held by each other.
7. **Conditions for Deadlock?**
   Mutual exclusion, Hold and wait, No preemption, Circular wait.

8. **What is Virtual Memory?**
   A memory management technique using disk as an extension of RAM.
9. **What is Paging?**
   Memory is divided into fixed-size pages to avoid fragmentation.
10. **What is Segmentation?**
    Memory is divided into variable-sized segments based on logical divisions.
11. **What is a Scheduler?**
    A component that selects the next process for execution.
12. **Types of Schedulers?**
    Long-term, Short-term, and Medium-term scheduler.
13. **What is Context Switching?**
    Saving the state of one process and loading another.
14. **What is Multitasking?**
    The ability to run multiple processes seemingly at the same time.
15. **What is Thrashing?**
    Excessive paging causing low performance.
16. **Difference: Kernel vs User Mode?**
    Kernel mode has full access to hardware; user mode is restricted.
17. **What is a System Call?**
    An interface for user programs to request services from the OS.
18. **What is IPC (Interprocess Communication)?**
    Mechanism for processes to communicate and synchronize.
19. **What is File System?**
    The method OS uses to store, organize, and manage files.
20. **What is a Real-Time OS?**
    An OS that guarantees response within a strict time constraint.

Networking

1. **What is a Computer Network?**
   A system of interconnected computers for sharing data and resources.
2. **Types of Networks?**
   LAN, MAN, WAN, PAN.
3. **What is IP Address?**
   A unique address identifying a device on a network.
4. **What is DNS?**
   Domain Name System – translates domain names to IP addresses.

5. **Difference: TCP vs UDP?**
   TCP is reliable and connection-oriented; UDP is faster and connectionless.
6. **What is HTTP/HTTPS?**
   Protocols for transferring web content. HTTPS adds encryption.
7. **What is a Protocol?**
   A set of rules for data communication.
8. **What is MAC Address?**
   A unique hardware address assigned to a network interface.
9. **What is a Subnet?**
   A subdivision of an IP network to improve routing and security.
10. **OSI Model Layers?**
    Physical, Data Link, Network, Transport, Session, Presentation, Application.
11. **Difference: OSI vs TCP/IP?**
    OSI has 7 layers; TCP/IP has 4. TCP/IP is the practical model.
12. **What is ARP?**
    Address Resolution Protocol – maps IP address to MAC address.
13. **What is NAT?**
    Network Address Translation – allows private IPs to access the internet.
14. **What is DHCP?**
    Dynamic Host Configuration Protocol – assigns IPs automatically.
15. **What is Bandwidth?**
    The maximum data transfer rate over a network path.
16. **What is Latency?**
    Time delay between request and response.
17. **What is a Firewall?**
    Security system that monitors and controls network traffic.
18. **What is a VPN?**
    Virtual Private Network – encrypts internet traffic and hides identity.
19. **What is a Router?**
    A device that connects multiple networks and routes traffic.
20. **What is Packet Switching?**
    Data is split into packets and sent independently across the network.

SDLC

1. **What is SDLC?**
   A process for planning, creating, testing, and deploying software.

2. **Phases of SDLC?**
   Requirement gathering, Design, Implementation, Testing, Deployment, Maintenance.
3. **Why is SDLC important?**
   It ensures systematic, high-quality software development.
4. **What is Requirement Gathering?**
   Understanding client needs and documenting requirements.
5. **What is Feasibility Study?**
   Evaluating technical and financial viability of the project.
6. **Difference: Functional vs Non-Functional Requirements?**
   Functional: What the system does.
   Non-functional: How the system performs (e.g., speed, security).
7. **What is Software Design?**
   Defining the architecture and components of a software solution.
8. **Difference: HLD vs LLD?**
   High-Level Design (HLD): System architecture.
   Low-Level Design (LLD): Component-level detail.
9. **What is Implementation Phase?**
   Actual coding and building of the software.
10. **What is Testing in SDLC?**
    Validating that software works as expected.
11. **Types of Testing?**
    Unit, Integration, System, Acceptance, Regression Testing.
12. **What is Waterfall Model?**
    A linear SDLC model where each phase is completed before the next.
13. **What is Agile Model?**
    Iterative and incremental approach with frequent feedback and changes.
14. **Scrum in Agile?**
    A framework in Agile where work is divided into small sprints.
15. **Difference: Agile vs Waterfall?**
    Agile is iterative and flexible; Waterfall is linear and rigid.
16. **What is Deployment?**
    Releasing software to a live environment.
17. **What is Maintenance in SDLC?**
    Updating and fixing the software post-deployment.
18. **What is Version Control?**
    Managing changes to code using tools like Git.
19. **What is DevOps?**
    Combines development and operations to improve software delivery.

20. **What is CI/CD?**
    Continuous Integration/Continuous Deployment – automates testing and deployment.

DSA Important Theory Questions

Array

1. What is an array?

An array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. The elements are stored in contiguous memory locations. Arrays store elements of the same data type, allowing fast access using an index.

2. Advantages of arrays:

- **Random Access:** You can access any element in O(1) time using its index.
- **Memory locality:** Since elements are stored contiguously, CPU caching is efficient.
- **Simple data structure:** Easy to implement and use.

3. Disadvantages of arrays:

- **Fixed size:** Once declared, size cannot be changed (in static arrays).
- **Insertion/deletion cost:** To insert or delete elements (except at the end), elements must be shifted, resulting in O(n) time.
- **Wastage of memory:** If size is overestimated, unused space remains allocated.

4. Static vs Dynamic Arrays:

- **Static arrays** have fixed size, allocated at compile time or start of program.
- **Dynamic arrays** can resize during runtime, usually by allocating a new larger array and copying data (e.g., ArrayList in Java, vector in C++).

5. Memory allocation:

- Arrays can be allocated on stack (local arrays) or heap (dynamically allocated arrays). Stack allocation is faster but limited in size.

LinkedList

1. What is a linked list?

A linked list is a linear collection of nodes where each node contains data and a pointer/reference to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations.

2. Array vs Linked List:

Aspect	Array	Linked List
Size	Fixed (static)	Dynamic
Memory	Contiguous	Non-contiguous
Access Time	O(1) random access	O(n) sequential access
Insertion/Deletion	Costly (O(n)) due to shifts	Efficient (O(1) if pointer known)

3. Types of linked lists:

- **Singly linked list:** Each node points to the next node only.
- **Doubly linked list:** Each node has pointers to both next and previous nodes.
- **Circular linked list:** Last node points back to the head, making the list circular.

4. Advantages over arrays:

- Dynamic size allows easy insertion and deletion.
- No need to allocate large memory blocks upfront.

5. Disadvantages:

- No random access; you must traverse nodes sequentially.
- Extra memory required for storing pointers.
- More complex data structure, more pointer manipulations.

Stack

1. What is a stack?

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. The last element added to the stack is the first one to be removed.

Key operations:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove the top element.
- **Peek/Top:** View the top element without removing it.
- **isEmpty:** Check if the stack is empty.

---

2. Where are stacks used in real life?

Stacks are used in:

- Undo/Redo functionality in editors
- Browser history navigation
- Expression evaluation (infix to postfix conversion)
- Syntax parsing in compilers
- Function call stacks in recursion

---

3. What are the advantages of stacks?

- Easy to implement and use.
- Useful for solving recursive and backtracking problems.
- Requires minimal memory overhead when implemented using arrays or linked lists.

---

4. What are the limitations of stacks?

- In array implementation, the size is fixed (stack overflow can occur).
- Not suitable for accessing random elements directly.
- Only the top element is accessible (not flexible for other types of access).

---

5. How is a stack implemented?

- **Array-based implementation:** Uses a fixed-size array and a pointer (top) to track the last pushed element.
- **Linked-list implementation:** Each node contains data and a reference to the next node.

Array stack (fixed size):

```
int stack[100];
int top = -1;
```

Linked list stack:

```
cpp
CopyEdit
struct Node {
   int data;
   Node* next;
};
```

---

6. What is stack overflow and underflow?

- **Stack Overflow:** Trying to push an element when the stack is already full (mostly in array implementation).
- **Stack Underflow:** Trying to pop an element when the stack is empty.

---

7. What is the time complexity of stack operations?

All basic operations – push, pop, peek, and isEmpty – run in **O(1) time**, making stacks efficient.

---

8. What is the difference between stack and queue?

Feature	Stack (LIFO)	Queue (FIFO)
Insertion	At top	At rear
Deletion	From top	From front
Access Pattern	Last inserted = first out	First inserted = first out

---

Queue

1. What is a Queue?

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. The first element added is the first one to be removed.

Key Operations:

- **Enqueue:** Add an element at the rear.
- **Dequeue:** Remove the element from the front.
- **Front/Peek:** Get the front element without removing it.
- **isEmpty:** Check if the queue is empty.

---

2. Where are queues used in real life?

Queues are used in:

- Printer job scheduling
- CPU scheduling in operating systems
- Handling requests in web servers

- Customer service lines
- Breadth-First Search (BFS) in graphs

---

3. What are the advantages of queues?

- Maintains processing order.
- Supports asynchronous data transfer (e.g., data buffering).
- Efficient for breadth-based traversal in algorithms.

---

4. What are the limitations of queues?

- In array-based implementation, space can be wasted unless a circular approach is used.
- Random access is not possible — only front and rear operations are allowed.

---

5. How is a queue implemented?

Queues can be implemented in several ways:

- **Array-based:** Fixed-size; front and rear pointers manage the queue.
- **Linked List-based:** Dynamic size; each node holds data and a pointer to the next.
- **Circular Queue:** Uses array in a circular fashion to reuse space.

Example Conceptual Operations:

enqueue(x): rear++

dequeue(): front++

---

6. What is a circular queue?

A **circular queue** reuses space by connecting the last index back to the first.
This prevents the need for shifting elements and avoids "false full" conditions in a static array.

Condition for full queue:
(rear + 1) % size == front

---

7. What is the time complexity of queue operations?

All basic operations – enqueue, dequeue, peek, isEmpty – take **O(1)** time.

---

8. What is the difference between queue and stack?

Feature	Queue (FIFO)	Stack (LIFO)
Insertion	At rear	At top
Deletion	From front	From top
Access	First inserted = first out	Last inserted = first out

---

9. What are types of queues?

1. **Simple Queue:** Basic FIFO queue.
2. **Circular Queue:** Reuses array space.
3. **Deque (Double-Ended Queue):** Insertion and deletion at both ends.
4. **Priority Queue:** Elements are served based on priority, not insertion order.

Tree

1. What is a Tree in Data Structures?

A **tree** is a hierarchical data structure consisting of nodes connected by edges. It starts with a **root** node and expands into **child** nodes.

Each node may have:

- A **value**
- Zero or more **child nodes**
- One **parent** (except the root)

It is a **non-linear** structure used to represent hierarchical relationships.

---

2. What are the different types of trees?

- **Binary Tree:** Each node has at most two children.
- **Binary Search Tree (BST):** A binary tree with the left child < parent < right child.
- **AVL Tree:** A self-balancing BST.
- **Red-Black Tree:** A balanced BST with coloring rules for balancing.
- **N-ary Tree:** Each node can have N children.
- **Trie (Prefix Tree):** Used for storing strings efficiently.

---

3. What is the difference between a Binary Tree and Binary Search Tree (BST)?

Feature	Binary Tree	Binary Search Tree (BST)
Child nodes	≤ 2 children per node	Same
Node ordering	No specific order	Left < Root < Right
Usage	General purpose	Fast search operations

---

4. What are tree traversal methods?

1. **Inorder (Left, Root, Right)** – Used in BST to get sorted order.
2. **Preorder (Root, Left, Right)** – Used to create a copy of the tree.
3. **Postorder (Left, Right, Root)** – Used to delete the tree.
4. **Level-order (BFS)** – Uses a queue to traverse level by level.

---

5. What is a balanced binary tree?

A binary tree where the height difference between left and right subtrees of any node is at most 1.

Balanced trees improve time complexity:

- Search: O(log n) instead of O(n) in skewed trees.

Examples: AVL Tree, Red-Black Tree

---

6. What is the height of a tree?

The height of a tree is the **number of edges on the longest path** from the root to a leaf node.

For a tree with only one node (root), height = 0.

---

7. What is the difference between full, complete, and perfect binary trees?

Type	Description
Full Binary Tree	Every node has 0 or 2 children
Complete Binary Tree	All levels are full except possibly the last, which is filled left to right
Perfect Binary Tree	All internal nodes have 2 children and all leaves are at the same level

---

8. What are leaf, internal, and root nodes?

- **Root node:** The top node with no parent.
- **Leaf node:** A node with no children.

- **Internal node:** A node with at least one child.

---

9. What is a Binary Heap?

A **binary heap** is a complete binary tree used to implement priority queues.

- **Min-Heap:** Parent ≤ children.
- **Max-Heap:** Parent ≥ children.
  Operations like insertion and deletion take O(log n) time.

---

10. What are common tree applications?

- Representing hierarchical data (file systems, XML)
- Databases and indexing (B-trees, B+ trees)
- Priority Queues (Heaps)
- Routing algorithms (Trie, Decision Trees)
- AI (game trees, decision trees)

Graph

1. What is a graph in data structures?

A **graph** is a non-linear data structure consisting of **vertices (nodes)** and **edges (connections)** that link pairs of nodes. Graphs can represent various real-world problems like networks, maps, and relationships.

Types:

- **Directed vs Undirected**
- **Weighted vs Unweighted**

---

2. What are the types of graphs?

Type	Description
Directed Graph	Edges have direction (u → v)
Undirected Graph	Edges have no direction (u — v)
Weighted Graph	Each edge has a weight or cost
Unweighted Graph	All edges are equal

Type	Description
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles
Connected Graph	All vertices are reachable from any vertex (for undirected graphs)
Disconnected Graph	At least one node is not reachable from others
DAG (Directed Acyclic Graph)	Used in scheduling, compilers (no cycles allowed)

---

3. How are graphs represented in memory?

1. **Adjacency Matrix** (2D array):
    - Size = V x V (V = number of vertices)
    - matrix[i][j] = 1 if there's an edge from i to j
    - Efficient for dense graphs, but uses more space.
2. **Adjacency List** (Array of lists):
    - Each node stores a list of its neighbors
    - Space-efficient for sparse graphs

---

4. What is a traversal in a graph?

Traversal means visiting all the nodes of a graph.

- **Breadth-First Search (BFS):**
    - Level-wise traversal using a queue
    - Used in shortest path algorithms (like in unweighted graphs)
- **Depth-First Search (DFS):**
    - Goes deep along one branch before backtracking
    - Uses recursion or a stack
    - Used in cycle detection, topological sort

---

5. What is the difference between BFS and DFS?

Feature	BFS	DFS
Data structure	Queue	Stack or Recursion

Feature	BFS	DFS
Approach	Level-order	Depth-first
Shortest Path	Yes (in unweighted graph)	No
Space	More	Less

---

6. What are applications of graphs?

- Social Networks
- Google Maps (Shortest Path)
- Web Crawlers
- Dependency resolution (e.g., packages, build systems)
- Network Routing
- Recommendation Systems

---

7. What is a cycle in a graph?

A **cycle** is a path where the first and last vertices are the same, and no edge is repeated.

- **Cycle detection** is important in many applications like deadlock detection and verifying DAGs.
- Can be detected using DFS.

---

8. What is a connected component?

In an undirected graph, a **connected component** is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

In DFS/BFS, we count how many times we start a new traversal to determine the number of components.

---

9. What is a weighted graph and where is it used?

In a **weighted graph**, each edge has a weight (or cost). It's used when there's a cost associated with moving from one node to another.

Used in:

- Shortest path algorithms like Dijkstra, Bellman-Ford
- Network routing

- Flight cost calculations

---

10. What are shortest path algorithms?

These algorithms find the minimum cost path between nodes.

- **Dijkstra's Algorithm:** Works with non-negative weights.
- **Bellman-Ford Algorithm:** Works with negative weights.
- **Floyd-Warshall Algorithm:** All pairs shortest path.
- *A Search:** Uses heuristics for optimal path finding.

Set & Map

1. What is a Set in data structures?

A **Set** is a collection of **unique elements** with **no duplicates** and typically **no defined order**.

Key Properties:

- Duplicate elements are not allowed.
- Mostly implemented using **Hash Tables**, **Trees**, or **Bit Vectors**.
- Common operations: insert(), remove(), contains().

Example Use Cases:

- Removing duplicates from a collection
- Checking membership in O(1) time using hash-based sets

---

2. What is a Map (or Dictionary)?

A **Map** stores data in **key-value pairs**, where each key maps to a specific value.

Key Properties:

- Keys are **unique**, but values can be duplicated.
- Provides **fast lookup** by key.
- Common operations: put(key, value), get(key), remove(key).

Example Use Cases:

- Caching
- Counting word frequencies
- Storing configuration settings

---

3. What is the difference between Set and Map?

Feature	Set	Map
Structure	Collection of unique items	Key-value pairs
Keys	Not applicable	Keys must be unique
Values	Elements themselves	Values associated with keys
Example	{1, 2, 3}	{1: "A", 2: "B"}

---

4. What are the types of sets and maps?

In most languages (like C++, Java, Python):

Set Variants:

- **HashSet / Unordered Set:** No order, O(1) access.
- **TreeSet / Ordered Set:** Sorted order, O(log n) access.

Map Variants:

- **HashMap / Unordered Map / Dictionary:** Fast access, no order.
- **TreeMap / Ordered Map:** Keys sorted, slower than hash-based.

---

5. How is a Set implemented internally?

- **HashSet/Unordered Set:** Uses a **hash table** for constant-time operations.
- **TreeSet:** Uses a **self-balancing binary search tree (e.g., Red-Black Tree)** for ordered access.

Operations like insertion, deletion, and lookup take:

- **O(1)** on average in hash-based sets
- **O(log n)** in tree-based sets

---

6. How is a Map implemented internally?

- **HashMap/Dictionary:** Uses a **hash table** to map keys to values.
- **TreeMap:** Uses a **binary search tree** (like Red-Black Tree) for sorted key storage.

Collision handling in hash maps is done using **chaining** or **open addressing**.

---

7. What is hashing and how does it relate to maps/sets?

Hashing is a technique to convert keys into array indices using a hash function.

- It's the backbone of hash-based data structures like HashMap and HashSet.
- A good hash function reduces **collisions** and ensures **uniform distribution**.

---

8. What are common operations on Set and Map and their complexities?

Operation	HashSet/HashMap	TreeSet/TreeMap
Insert	O(1) average	O(log n)
Delete	O(1) average	O(log n)
Search	O(1) average	O(log n)
Ordered Traversal	Not possible	Yes (in order)

---

9. When would you use a Set vs Map?

- Use a **Set** when you need to **track unique items** or perform quick membership tests.
- Use a **Map** when you need to **associate values with keys** for fast retrieval.

Examples:

- Checking if a number is a duplicate → Set
- Counting frequency of elements → Map

---

10. What are common use cases of maps and sets in programming?

Sets:

- Removing duplicates from a list
- Finding union, intersection, difference of datasets
- Checking presence of a visited node (e.g., in DFS/BFS)

Maps:

- Caching recently computed values
- Counting occurrences (e.g., word frequency)
- Storing user settings, metadata, and key-value configs

All Time Complexities

1. Data Structures – Time Complexity

Data Structure	Operation	Average Case	Worst Case
Array	Access	O(1)	O(1)
	Search	O(n)	O(n)
	Insert (end)	O(1)	O(n) (resize)
	Delete	O(n)	O(n)
Linked List	Access	O(n)	O(n)
	Search	O(n)	O(n)
	Insert/Delete (head)	O(1)	O(1)
Stack / Queue	Push/Enqueue	O(1)	O(1)
	Pop/Dequeue	O(1)	O(1)
	Search	O(n)	O(n)
Hash Table (Map/Set)	Search	O(1)	O(n)
	Insert/Delete	O(1)	O(n)
Binary Search Tree (BST)	Search	O(log n)	O(n)
	Insert/Delete	O(log n)	O(n)
AVL Tree / Red-Black Tree	Search	O(log n)	O(log n)
	Insert/Delete	O(log n)	O(log n)
Heap (Min/Max)	Insert/Delete	O(log n)	O(log n)
	Search	O(n)	O(n)
	Peek (min/max)	O(1)	O(1)
Trie (Prefix Tree)	Insert/Search	O(m)	O(m)
	m = length of key		
Graph (Adjacency List)	Add Vertex	O(1)	O(1)
	Add Edge	O(1)	O(1)

Data Structure	Operation	Average Case	Worst Case
	Search Edge	O(V)	O(V)

---

2. Sorting Algorithms – Time Complexity

Algorithm	Best Case	Average Case	Worst Case	Space
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	$O(n^2)$	O(n)

---

3. Search Algorithms – Time Complexity

Algorithm	Time Complexity	Space Complexity
Linear Search	O(n)	O(1)
Binary Search	O(log n)	O(1) or O(log n) (recursive)

Note: Binary search only works on sorted data.

---

4. Graph Algorithms – Time Complexity

Algorithm	Time Complexity	Description
BFS / DFS	O(V + E)	Graph traversal
Dijkstra's	O((V + E) log V)	Shortest path (no negative weights)
Bellman-Ford	O(VE)	Works with negative weights
Floyd-Warshall	$O(V^3)$	All pairs shortest paths

Algorithm	Time Complexity	Description
Kruskal's MST	O(E log E)	Minimum spanning tree
Prim's MST (Heap)	O((V + E) log V)	Minimum spanning tree
Topological Sort	O(V + E)	DAGs only
Union-Find	O(α(n)) per operation	Disjoint sets (α is inverse Ackermann)