Object-Oriented Programming (OOP)

1. What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to represent data and methods to manipulate that data.

2. What is a Class and Object?

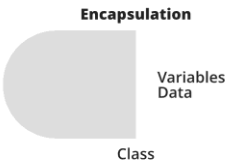
- **Class: A blueprint/plan/template describing an object's details. (Like the design of a house.)**

- **Object: An instance of a class. (Taking the design and building the house.)**

3. Basic Concepts of OOP

1. **Encapsulation**
2. **Abstraction**
3. **Inheritance**
4. **Polymorphism**

4. What is Encapsulation?



Encapsulation is an attribute of an object that contains all hidden data. The user interacts with the object through a defined interface (e.g., getter and setter methods). It emphasizes the protection of the system from invalid states and unauthorized access.

```
Copy code

class Person {
  private:
    int age;  // Private attribute

  public:
    void setAge(int a) {
        age = a;
    }

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

5. What is Inheritance?

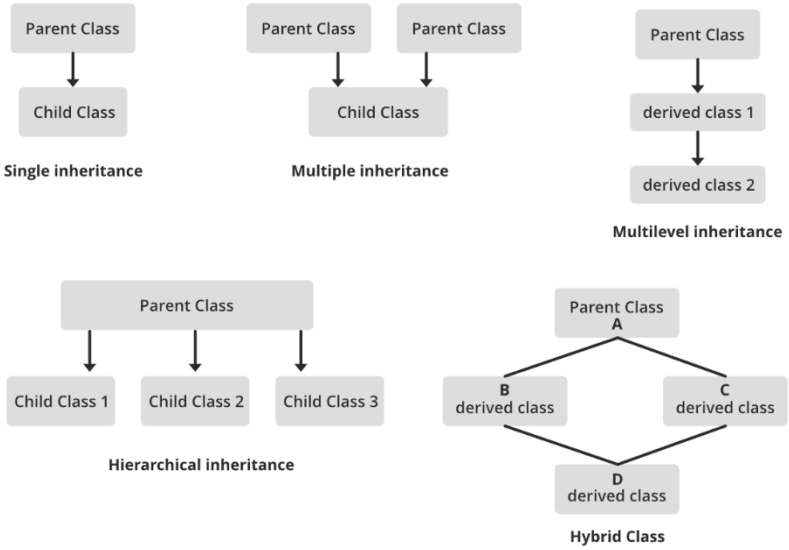
Inheritance allows a new class (called a derived class) to inherit properties and behaviors (methods) from an existing class (called a base class). This enables the subclass to reuse the code from the superclass without rewriting it.

(All of them inherit characteristics from their parent but with some differences.)

```
class Vehicle {
  public:
    int wheels;
    void drive() {
        cout << "Vehicle is moving" << endl;
    }
};

class Car : public Vehicle {
  public:
    string brand;
};
```

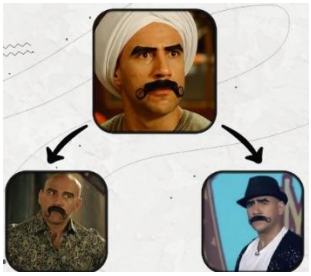
Types of Inheritance:



6. What is Polymorphism?

Polymorphism refers to the ability of a variable, function, or object to take on multiple forms.

(El Keber can take on different shapes.)



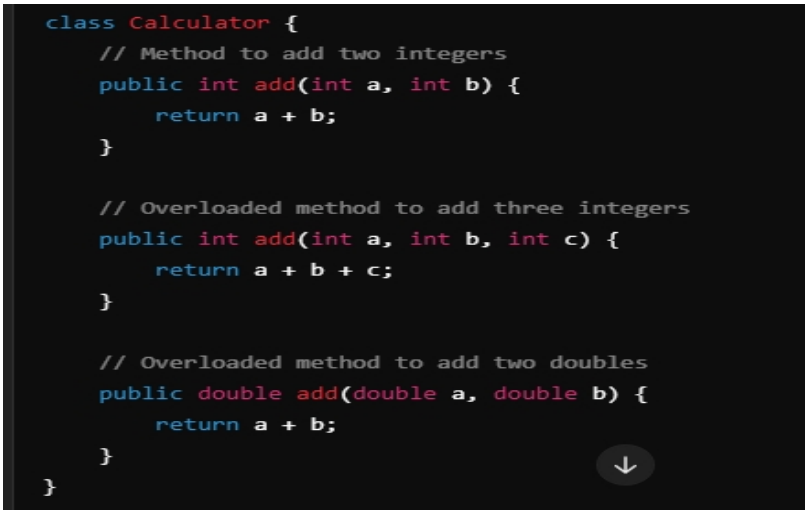
```cpp
class Shape {
  public:
    virtual void draw() {
        cout << "Drawing Shape" << endl;
    }
};

class Circle : public Shape {
  public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};

int main() {
  Shape* shape = new Circle();
  shape->draw();  // Calls Circle's draw()
}
```

Types of Polymorphism:

- **Method Overloading: When two or more methods in the same class share the same name but have different function signatures.**

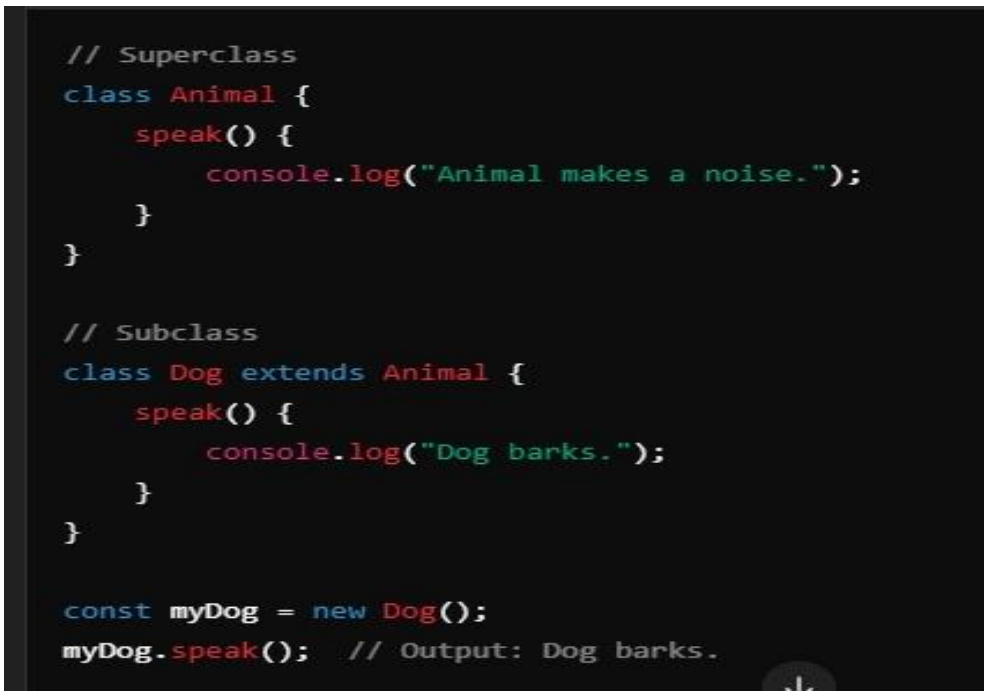


```
class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

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

    // Overloaded method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}
```

- **Method Overriding: Occurs when the derived class provides a specific implementation of a method already defined in its base class. The derived class overrides the behavior of the inherited method to provide a more specific or specialized implementation.**



```
// Superclass
class Animal {
    speak() {
        console.log("Animal makes a noise.");
    }
}

// Subclass
class Dog extends Animal {
    speak() {
        console.log("Dog barks.");
    }
}

const myDog = new Dog();
myDog.speak();  // Output: Dog barks.
```

Overloading vs. Overriding:

Feature	Overloading	Overriding
Definition	Same method name with different parameters.	Same method name and parameters but different implementation.
Where	Occurs within the same class.	Occurs between superclass and subclass.
Parameters	Must be different (number/type).	Must be exactly the same.
Return Type	Can be different.	Must be the same.
Purpose	Provides multiple ways to call the same method based on parameters.	Changes the behavior of the inherited method.
Polymorphism Type	Compile-time polymorphism (early binding).	Runtime polymorphism (late binding).

7. What is Abstraction?

Abstraction shows only the necessary details to the client of an object, hiding the inner workings.

(For example, when using a television, you don't need to know how electricity flows inside; you just want it to work.)

```
class Animal {
  public:
    virtual void sound() = 0;  // Abstract method (pure virtual)
};

class Dog : public Animal {
  public:
    void sound() override {
        cout << "Bark" << endl;
    }
};
```

8. What are Access Modifiers?

Access modifiers control the level of access to class members (attributes and methods):

- **Public: Accessible from anywhere.**
- **Private: Accessible only within the class itself.**
- **Protected: Accessible within the class and by derived classes.**

9.What is a Constructor?

A constructor is a special member function that is automatically called when an object of a class is created. Its primary purpose is to initialize the object's attributes or allocate resources.

It has the same name as the class, No Return Type, and Overloading.

Types of Constructors:

1. **Default Constructor: A constructor that takes no parameters.**
2. **Parameterized Constructor: A constructor that takes parameters to initialize an object with specific values.**

3. **Copy Constructor: A constructor that initializes an object using another object of the same class.**

10. What is a Destructor?

A destructor is a special member function that is automatically called when an object goes out of scope or is explicitly deleted. Its primary purpose is to free up resources that the object may have acquired during its lifetime.

It has the same name as the class, No Return Type, and is Called Automatically.

```
class Rectangle {
public:
    int width, height;

    // Constructor
    Rectangle(int w, int h) : width(w), height(h) {
        std::cout << "Constructor called!" << std::endl;
    }

    // Destructor
    ~Rectangle() {
        std::cout << "Destructor called!" << std::endl;
    }

    void display() {
        std::cout << "Width: " << width << ", Height: " << height << std::endl;
    }
};
```

Feature	Constructor	Destructor
Name	Same as class name	Same as class name preceded by ~
Parameters	Can take parameters (overloaded)	Cannot take parameters
Return Type	No return type (not even void)	No return type
Purpose	Initialize object and allocate resources	Free resources and perform cleanup
Calls	Called when an object is created	Called when an object is destroyed

11. What is an Abstract Class, Interface, and the Difference Between an Interface and an Abstract Class?

. Abstract Class:

class that cannot be instantiated directly. can contain both fully implemented methods and abstract methods (methods without any implementation that must be implemented by derived classes).

```
abstract class Animal {
    String name;

    // Abstract method (must be implemented by subclasses)
    abstract void makeSound();

    // Non-abstract method
    public void sleep() {
        System.out.println("Sleeping...");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Bark");
    }
}
```

. **Interface:**

An interface defines a contract or a set of methods that a class must implement. Unlike abstract classes, interfaces cannot have any implemented methods.

```java
interface Animal {
    void makeSound();
    void sleep();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }

    public void sleep() {
        System.out.println("Sleeping...");
    }
}
```

Interface	Abstract class
Interface support multiple implementations.	Abstract class does not support multiple inheritance.
Interface does not contain Data Member	Abstract class contains Data Member
Interface does not contain Constructors	Abstract class contains Constructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

12. Destructors Cannot Have Overload?

True, Because They Don’t Take Parameters The purpose is to clean up resources like memory

13. What is the Difference Between Reference Types and Value Types?

Value Types:

- **Store the actual data.**
- **When you assign one value type to another, a copy of the data is made.**
- **Changes made to one value type do not affect the other.**

```
int a = 10;
int b = a;  // b gets a copy of the value of a
```

Reference Types:

- **Store a reference (or pointer) to the actual data.**
- **When you assign one reference type to another, both refer to the same object.**
- **Changes made through one reference will affect the other reference as they both point to the same object.**

```javascript
let obj1 = { name: 'John' };
let obj2 = obj1;  // obj2 points to the same object as obj1
obj2.name = 'Doe';
console.log(obj1.name);  // Output: Doe
```

14. What is an Abstract Method?

An abstract method is a method that is declared in an abstract class or interface without any implementation. It only defines the method signature (name, return type, and parameters).

```cpp
class Shape {
    public:
    // Abstract method - no implementation
    virtual void draw() = 0;
};
```

15. Accessing Members in Inheritance?

Inheritance Mode	Public Members in Base Class	Protected Members in Base Class	Private Members in Base Class
Public	Public in derived class	Protected in derived class	Not accessible in derived class
Protected	Protected in derived class	Protected in derived class	Not accessible in derived class
Private	Private in derived class	Private in derived class	Not accessible in derived class

16. What is pure virtual function?

is a virtual function that does not have any implementation in the base class and must be overridden in derived classes. It is used to create abstract classes, which cannot be instantiated directly. Any class that contains at least one pure virtual function is considered an abstract class.

```
class Animal {
public:
    // Pure virtual function (no implementation)
    virtual void makeSound() = 0;

    // Non-pure virtual function (optional to override)
    virtual void sleep() {
        cout << "Animal is sleeping..." << endl;
    }
};

// Derived class implementing the pure virtual function
class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Bark" << endl;
    }
};
```

17. What is a Virtual Function?

A virtual function is a member function in a base class that you expect to override in derived classes. It allows for dynamic polymorphism, meaning the function that gets called is determined at runtime based on the type of the object, rather than at compile time.

```cpp
class Base {
public:
    virtual void show() {  // Virtual function
        cout << "Base class show function called" << endl;
    }

    void display() {  // Non-virtual function
        cout << "Base class display function called" << endl;
    }
};

class Derived : public Base {
public:
    void show() override {  // Overriding virtual function
        cout << "Derived class show function called" << endl;
    }

    void display() {  // Overriding non-virtual function
        cout << "Derived class display function called" << endl;
    }
};
```

18. What is a Member Function?

A member function is defined inside a class and can operate on the data members (attributes) of that class. These functions are used to manipulate the data within an object.

```cpp
private:
    int length;
    int width;

public:
    // Constructor to initialize the rectangle
    Rectangle(int l, int w) : length(l), width(w) {}

    // Member function to calculate area
    int calculateArea() {
        return length * width;
    }

    // Member function to display the dimensions
    void displayDimensions() {
        cout << "Length: " << length << ", Width: " << width << endl;
    }
};
```

Types of Member Functions:

1. **Simple Member Functions: Functions that operate on data members or perform some task.**
2. **Constructor: A special member function used to initialize objects of a class.**
3. **Destructor: A member function used to clean up resources when an object is destroyed.**
4. **Static Member Functions: These belong to the class rather than any specific object, and they can be called without creating an instance of the class.**
5. **Virtual Member Functions: These are used to achieve runtime polymorphism, allowing derived classes to override base class functions.**
6. **Inline Member Functions: Functions defined inside the class body are treated as inline functions, and the compiler attempts to expand them in-place for performance.**

19. What is a Friend Function?

Is not a member of a class but is allowed to access the class's private and protected members. Friend functions are useful when you want an external function to have special access to the internal data of a class without being a member of the class.

```cpp
class Box {
private:
    double length;

public:
    Box() : length(0) {}  // Constructor

    // Declare friend function
    friend void setLength(Box &b, double len);

    void displayLength() {
        cout << "Length of box: " << length << endl;
    }
};

// Friend function definition
void setLength(Box &b, double len) {
    b.length = len;  // Accessing private member directly
}
```