Object Oriented Programming Concepts (OOPS)

1. What is Object Oriented Programming?

- When a complete software works as a bunch of **objects talking** to **each other**
- Entire program is organized using objects

2. What is a Class?

- **Blueprint** for objects having **common** properties
- User defined data type that contains
  - Properties (attributes)
  - Behaviors (methods)

3. What is an Object?

- An **instance** of the class
- It’s a real world entity
- Need to create an object of the class in order to use the attributes and methods of the class

4. Static Methods VS Non-Static Methods

- In object-oriented programming (OOP), **functions associated** with a class or object can be categorized as following:
  - Static Methods
  - Non-static Methods (also known as instance methods)

Static Method

- Associated with the **class** rather than the instance of the class
- Called on the class itself, **not** on an **object** of the class
- **Don't** have access to the i**nstance-specific data** of the class
- Declared using the **static** keyword
- Can be called using the scope resolution operator **‘::’** without creating an object of the class

```
class MathOperations {
public:
    static int add(int a, int b) {
        return a + b;
    }
    static double multiply(double a, double b) {
        return a * b;
    }
};

int main() {
    // Calling static methods without creating an object
    int result1 = MathOperations::add(5, 3);
    double result2 = MathOperations::multiply(2.5, 4.0);
}
```

Non-Static Method

- Associated with **instances (objects)** of the class
- Called on the **objects** of the class
- They **have** access to the **instance-specific data** of the class
- **No** special keyword is needed for non-static methods
- `calculateArea` is called on an object (`myCircle`) of the class and has access to the instance-specific data (the `radius` attribute)

```
class Circle {
private:
    double radius;
public:
    // Constructor
    Circle(double r) : radius(r) {}
    // Non-static method
    double calculateArea() {
        return 3.14 * radius * radius;
    }
};

int main() {
    // Creating an object of the Circle class
    Circle myCircle(5.0);
    // Calling the non-static method on the object
    double area = myCircle.calculateArea();
}
```

5. What is a Constructor?

- A block of code that **initializes** the **newly created object**
  - Its purpose is to initialize the object's data members and perform any necessary setup
- In `python`, a constructor is named **__init__**
- In `C++` and `Java`, the constructor is named the **same** as the **class name**

Different types of Constructors in C++

- **Default** Constructor
  - A constructor with **no** parameters
  - Called **automatically** when an object is created if no other constructor is specified
  - If a class does not explicitly define any constructor, the compiler generates a default constructor
- **Parameterized** Constructor
  - Accepts **parameters** to initialize the data members of an object
- **Copy** Constructor
  - Initializes a **new** object using an **existing** object of the same class
  - It is called when an object is passed by value or returned by value
- **Constructor** Overloading
  - Constructor overloading involves defining **multiple** constructors in a class with **different parameters**
  - Allows creating objects with different initialization options

Default Constructor

```
class MyClass {
public:
    // Default constructor
    MyClass() {
        cout << "Default constructor called" << endl;
    }
};

int main() {
    // Creation of the object invokes the default
       constructor
    MyClass obj;
}
```

Parameterized Constructor

```
class Point {
public:
    // Data members
    int x;
    int y;
    // Parameterized constructor
    Point(int a, int b) : x(a), y(b) {
        cout << "Parameterized constructor called" << endl;
    }
};

int main() {
    // Creating an object with parameters
    Point myPoint(3, 4);
}
```

Copy Constructor

```
class ComplexNumber {
public:
    // Data members
    double real;
    double img;
    // Default constructor
    ComplexNumber(double r, double i): real(r), img(i){}
    // Copy constructor
    ComplexNumber(const ComplexNumber& other){
        real = other.real;
        img = other.img;
        cout << "Copy constructor called" << endl;
    }
};

int main() {
    // Creating an object
    ComplexNumber num1(5, 10);
    // Creating another object as a copy of the first one
    ComplexNumber num2 = num1;
}
```

Constructor Overloading

```
class Rectangle {
public:
    // Data members
    int width;
    int height;
    // Default constructor
    Rectangle() : width(0), height(0) {
        cout << "Default constructor" << endl;
    }
    // Parameterized constructor
    Rectangle(int w, int h) : width(w), height(h) {
        cout << "Parameterized constructor" << endl;
    }
};

int main() {
    Rectangle rect1;          // Default constructor
    Rectangle rect2(4, 6);    // Parameterized constructor
}
```

5. What is a Destructor?

- **Automatically** called when the object is made **out of scope** or **destroyed**
- A destructor **cannot** be **overloaded**
  - There can only be one destructor
- In **C++** → The destructor name is the same as the class name but with the (**~**) tilde symbol as the prefix
  - If a class **does not** allocate resources manually (new, file handles,..) the compiler **automatically** provides a destructor
  - If a class **does** allocate resources, define a **custom** destructor to release them properly to avoid memory leaks
- In **Python** → The destructor is called **__del__**
  - Python has **automatic** garbage collection
    - Python will automatically **reclaim memory** when the object is **no** longer **referenced**
  - However, if a class manages **external** resources (files, database connections,..) defining a destructor (__del__) is useful
    - **Without** __del__, you'd rely on Python's garbage collector, which might **delay** cleanup
- In **Java** → The garbage collector automatically deletes the useless objects
  - Similar to python but have 2 options close() & finalise()
  - **close()** → **Releases** resources (files, network connections, database connections..)
    - Must be called **manually** or used inside a **try-with-resources** block
  - **finalise()** → Meant to be called by the **garbage collector** before an object is destroyed
    - **Unreliable** because garbage collection does **not** guarantee when finalize() will run
    - **Deprecated** in Java 9+ and completely removed in Java 18

Example

```
class MyClass {
public:
    // Constructor
    MyClass() {
        cout << "Constructor called." << endl;
    }
    // Destructor
    ~MyClass() {
        cout << "Destructor called." << endl;
    }
};

int main() {
    MyClass obj;
    // Object goes out of scope, destructor is automatically called
    cout << "Exiting main function." << endl;
    return 0;
}
```

5. What are the Access Specifiers?

- Keywords that define the **visibility** and **accessibility** of attributes and methods
- Three types of access specifiers:
  - Public
  - Private
  - Protected

Public

- Accessible from **any** part of the program
  - i.e From within the class and outside the class
- Can be **freely accessed** and **modified** by any code that uses the class

```
class MyClass {
   public:
        int publicVar;
        void publicMethod() {}
};
```

Private

- Only accessible within the **same** class
- **Hidden** from the outside world, and access to them is **restricted** to the methods of the **same** class
- Private members enhance security by preventing external code from directly accessing or modifying sensitive data

```
class MyClass {
   private:
        int privateVar;
        void privateMethod() {
        }
   public:
        void publicMethod() {
            // Can access 'privateVar' and 'privateMethod'
        }
   };
```

Protected

- Accessible within the **same** class and its **derived** classes

```
class MyBaseClass {
   protected:
        int Var;
};

class MyDerivedClass : public MyBaseClass {
   public:
        void accessProtected() {
            Var = 42;  // Accessing protected member from the derived class
        }
};
```

6. Virtual Function

- Member function of a class that can be **overridden** by a function with the same signature in a **derived** class
- Virtual functions are essential for providing abstraction where different **derived** classes can provide their **own** implementations of a common interface defined in the **base** class
- In C++ → A virtual function is declared using the **virtual** keyword
    - **@override** annotation is used to explicitly indicate method overriding
- In Java → Every public, non-static, and non-final method is a virtual function
- In Python → Methods are always virtual
- With non-virtual function → The function call is determined at **compile-time** based on the **static type** of the pointer or reference
    - Also known as early-binding
- With virtual function → The function call is determined at **runtime** based on the **actual type** of the object being pointed to
    - Also known as late-binding
- More info on the above 2 points → https://www.geeksforgeeks.org/vtable-and-vptr-in-cpp/

C++

```cpp
// Base class with a virtual function
class Shape {
public:
    virtual void draw() const {
        cout << "Drawing a shape." << endl;
    }
};

// Derived class with a specific implementation of the virtual function
class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle." << endl;
    }
};
```

Java

```java
class Shape {
    void draw() {
        System.out.println("Drawing a shape.");
    }
}

class Circle extends Shape {
    // Overrides the draw method in the base class
    void draw() {
        System.out.println("Drawing a circle.");
    }
}
```

Python

```python
class Shape:
    # Virtual function in Python is achieved without any special keyword
    def draw(self):
        print("Drawing a shape.")

class Circle(Shape):
    # Overrides the draw method in the base class
    def draw(self):
        print("Drawing a circle.")
```

7. Pure Virtual Function

- A Function that **doesn't** contain **any** statements
- This function is **defined** in the **derived** class if needed
- In C++ → A pure virtual function is marked as 'pure' using the **= 0** syntax
- In Java → **Abstract** classes are used to define abstract methods (pure virtual functions)
    - Abstract classes are defined using the abstract keyword is used
- In Python → the **abc** module is used to create abstract base classes with abstract methods (pure virtual functions)
    - The **@abstractmethod** decorator is used to declare an abstract method

C++

```cpp
class Shape {
public:
    // Pure virtual function
    virtual void draw() const = 0;
};

// Derived class implementing the pure virtual function
class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle." << endl;
    }
};
```

Java

```java
abstract class Shape {
    // Pure virtual function (abstract method)
    abstract void draw();
}
// Derived class implementing the pure virtual function
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle.");
    }
}
```

Python

```python
from abc import ABC, abstractmethod
# Abstract base class with a pure virtual function
class Shape(ABC):
    # Pure virtual function
    @abstractmethod
    def draw(self):
        pass
# Derived class implementing the pure virtual function
class Circle(Shape):
    def draw(self):
        print("Drawing a circle.")
```

7. Overloading VS Overriding

1. Function Overloading

- Refers to defining multiple functions in the **same scop**e with the **same name** but **different parameters** (either in terms of the number or types of parameters)
- Binding of the function call to its code happens at **compile-time**
    - Also known as compile-time polymorphism or static polymorphism
- The compiler determines which function to call based on the number and types of arguments provided during the function call

2. Function Overriding

- Involves providing a **specific** implementation for a function in a **derived** class that is already defined in its base class
- The **base** class must declare the function as **virtual**, and the derived class must use the **‘override’** keyword to indicate that it intends to override that function
- The function signature (name, return type, and parameters) in the derived class must match the function signature in the base class
- Binding of the function call to its code happens at **runtime**
    - Also known as runtime polymorphism or dynamic polymorphism

Overloading

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

int main() {
    MathOperations math;
    // Calls int add(int, int)
    cout << math.add(3, 4) << endl;
    // Calls double add(double, double)
    cout << math.add(2.5, 3.7) << endl;
}
```

Overriding

```
class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing a shape." << std::endl;
    }
};

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

int main() {
    Shape* shape1 = new Shape();
    Shape* shape2 = new Circle();
    shape1->draw();  // Calls Shape::draw()
    shape2->draw();  // Calls Circle::draw()
}
```

8. Main OOPS Features

1. **Encapsulation** → Involves **bundling** the **data** (attributes or properties) and **methods** (functions) that operate on the data within a **single unit**, known as a **class**
    - The data is kept private from the outside world, and access to it is controlled through public methods (getters and setters). This helps in hiding the internal details of the implementation
2. **Abstraction** → **Hiding** the **unnecessary** details
3. **Inheritence** → Allows a new class to **inherit** properties and behaviors from an existing class
4. **Polymorphism** → Property of the code to behave **differently** in **different contexts**

9. Encapsulation

- Involves **bundling** the **data** (attributes or properties) and **methods** (functions) that operate on the data within a **single unit**, known as a **class**
- The data is kept private from the outside world, and access to it is controlled through public methods (**getters** and **setters**)
    - This helps in hiding the internal details of the implementation

```
class Student {
private:
    // Private members are not directly accessible
       outside the class
    string name;
    float gpa;

public:
    // Public methods provide controlled access to
       private members
    void setName(const string& n) name = n;
    void setGPA(float g) {
        if (g >= 0.0 && g <= 4.0) {
            gpa = g;
        } else {
            cout << "Invalid GPA" << std::endl;
            gpa = 0.0;
        }
    }
    // Public methods to retrieve private data
    string getName() const {return name};
    float getGPA() const {return gpa};
};
```

```
int main() {
    // Create an object of the Student class
    Student student1;
    // Use public methods to set private data
    student1.setName("John Doe");
    student1.setAge(20);
    student1.setGPA(3.5);
    // Use public methods to retrieve private data
    cout << "Student Name: " << student1.getName() << endl;
    cout << "Student Age: " << student1.getAge() << endl;
    cout << "Student GPA: " << student1.getGPA() << endl;
}
```

10. Abstraction

- **Hiding** the **unnecessary** or irrelevant details
- Implemented using **‘classes’** and **‘interfaces’**
- **Abstract Class**
  - A class that **cannot** be instantiated on its own and is meant to serve as a base for other classes
  - It may have **abstract** methods, which are declared but **not** implemented in the abstract class Subclasses **must** provide concrete implementations for these methods
  - Abstract classes may also have **concrete** methods with fully implemented functionality

JAVA

```java
abstract class Animal {
    Animal() { // Constructor
      System.out.println("An Animal is going to be created");
    }
    abstract void walk();  // Abstract method
    void eat() {
      System.out.println("The animal is eating");
    }

}

class Cow extends Animal {
    Cow() {  // Constructor
      System.out.println("You have created a Cow");
    }
    // Implementation of the abstract method
    void walk() {
       System.out.println("Cow is walking");
    }
}
```

C++

```cpp
class Animal {
public:
    Animal() {
        cout << "Animal is going to be created" << endl;
    }
    // Pure virtual function makes the class abstract
    virtual void walk() = 0;
    void eat() {
        cout << "The animal is eating." << endl;
    }
};

// Derived class Cow
class Cow : public Animal {
public:
    Cow() {
        cout << "You have created a Cow" << endl;
    }
    // Implementation of the pure virtual function
    void walk() override {
        cout << "Cow is walking." << endl;
    }
};
```

11. Inheritence

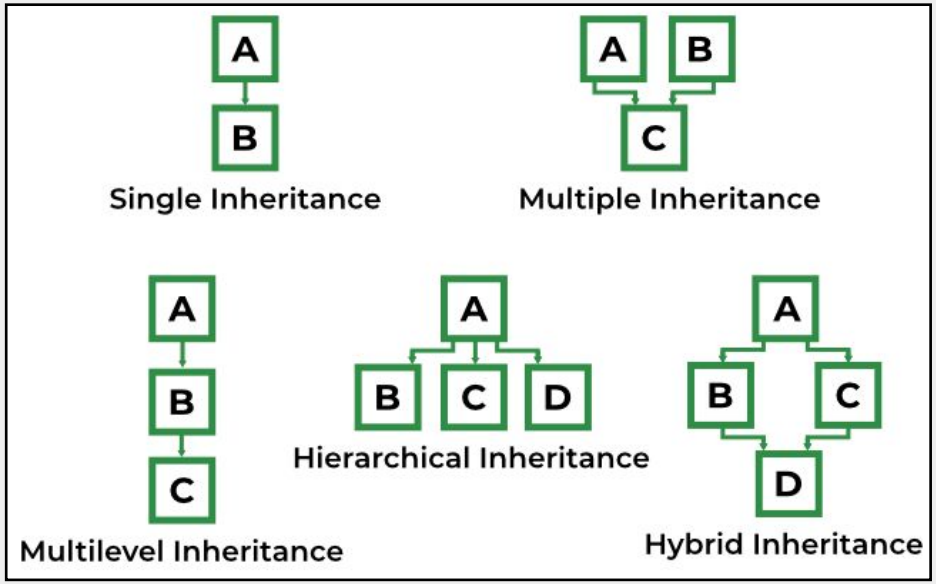
- Allows a **new** class to **inherit** properties and behaviors from an **existing** class
  - The existing class is called the **base** class or parent class
  - The new class is called the **derived** class or child class.
- Inheritance provides a way to model an "is-a" relationship between classes, where a derived class is a **specialized** version of the base class
- Inheritance promotes code **reuse** and helps create a hierarchical structure in the code
- In the following example:
  - ‘Animal’ is the base class with attributes and methods common to all animals
  - ‘Dog’ and ‘Cat’ are derived classes that inherit from the ‘Animal’ class
  - The ‘Dog’ and ‘Cat’ classes have their own specific methods (‘bark’ and ‘meow’) in addition to the inherited methods from the base class

```cpp
// Base class
class Animal {
protected: // To allow derived classes to access the `name` attribute
    string name;
public:
    Animal(const std::string& n) : name(n) {}
    void eat() {
        cout << name << " is eating." << endl;
    }
    void sleep() {
        cout << name << " is sleeping." << endl;
    }

// Derived class 1
class Dog : public Animal {
public:
    Dog(const std::string& n) : Animal(n) {}
    void bark() {
        cout << name << " is barking." << endl;
    }
};

// Derived class 2
class Cat : public Animal {
public:
    Cat(const std::string& n) : Animal(n) {}
    void meow() {
        cout << name << " is meowing." << endl;
    }
};
```

```cpp
int main() {
    // Creating objects of derived classes
    Dog myDog("Buddy");
    Cat myCat("Whiskers");
    // Using inherited methods
    myDog.eat();
    myDog.sleep();
    myDog.bark();
    myCat.eat();
    myCat.sleep();
    myCat.meow();
}
```

Types of Inheritence

- **Single** Inheritance**:** Child class derived directly from the base class
- **Multiple** Inheritance**:** Child class derived from multiple base classes
- **Multilevel** Inheritance**:** Child class derived from the class which is also derived from another base class and so on…
- **Hierarchical** Inheritance**:** Multiple child classes derived from a single base class
- **Hybrid** Inheritance**:** Inheritance consisting of multiple inheritance types of the above specified



12. Polymorphism

- Property of the code to behave **differently** in **different contexts**
- Polymorphism can be classified in 2 types:
  - Compile-time (or static) polymorphism → Achieved through function overloading
  - Runtime (or dynamic) polymorphism → Achieved through function overriding