Name : Satyam Sahu

1) What are the four main principles of Object-Oriented Programming? Explain each with an example.

The four main principles of OOP are encapsulation, inheritance, polymorphism, and abstraction. These principles form the foundation of object-oriented design and help create more maintainable and reusable code.

1. Encapsulation

Encapsulation is about bundling data and methods that operate on that data within a single unit (class) and restricting access to the internal state. It's like putting related stuff in a box and controlling who gets to look inside or change things.

Example:

```
public class BankAccount {
  private double balance;
  private String accountNumber;
  public void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
      System.out.println(amount + " deposited successfully");
    }
  }
  public void withdraw(double amount) {
    if (amount <= balance && amount > 0) {
      balance -= amount;
      System.out.println(amount + " withdrawn successfully");
    } else {
      System.out.println("Insufficient funds or invalid amount");
    }
  }
  public double getBalance() {
    return balance;
  }
}
```

Here, the account balance is encapsulated - you can't directly change it, but must use the deposit and withdraw methods which enforce business rules (can't withdraw more than available, can't deposit negative amounts).

Name : Satyam Sahu

2. Inheritance

Inheritance allows a class to inherit properties and behaviors from another class. It promotes code reuse and establishes an "is-a" relationship between classes.

Example:

```
public class Vehicle {
  protected String make;
  protected String model;
  protected int year;

  public void start() {
    System.out.println("Engine started");
  }

  public void stop() {
    System.out.println("Engine stopped");
  }
}
public class Car extends Vehicle {
  private int numberOfDoors;

  public void honk() {
    System.out.println("Beep beep!");
  }
  @Override
  public void start() {
    System.out.println("Car engine started with key");
  }
}
```

The Car class inherits make, model, year and the methods start() and stop() from Vehicle, while adding its own property numberOfDoors and method honk().

3. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. The most common use is when a parent class reference is used to refer to a child class object.

Example:

```
public class Animal {
  public void makeSound() {
    System.out.println("Some generic sound");
```

```
  }
}

public class Dog extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Woof woof!");
  }
}

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

public class Main {
  public static void main(String[] args) {
    Animal myDog = new Dog();
    Animal myCat = new Cat();

    myDog.makeSound();
    myCat.makeSound();

    Animal[] animals = {new Dog(), new Cat(), new Dog()};
    for (Animal animal : animals) {
      animal.makeSound();
    }
  }
}
```

Here, even though we're using Animal references, the correct makeSound() method is called based on the actual object type.

4. Abstraction

Abstraction means hiding complex implementation details and showing only the necessary features of an object. It lets you focus on what an object does rather than how it does it.

Example:

```
public abstract class Database {
```

```
  public abstract void connect();
  public abstract void disconnect();

  public void executeQuery(String query) {
    connect();
    System.out.println("Executing query: " + query);
    disconnect();
  }
}

public class MySQLDatabase extends Database {
  @Override
  public void connect() {
    System.out.println("Connecting to MySQL database...");
  }

  @Override
  public void disconnect() {
    System.out.println("Disconnecting from MySQL database...");
  }
}

public class Main {
  public static void main(String[] args) {
    Database db = new MySQLDatabase();
    db.executeQuery("SELECT * FROM users");
  }
}
```

The Database class abstracts the concept of database operations. Users of this code don't need to understand the connection details - they just call executeQuery() and the database handles the rest.

These four principles work together to create code that's organized, reusable, and easy to maintain as your projects grow in complexity.

2) 2. How does inheritance work in OOP? Write a sample program to demonstrate single and multilevel inheritance.

Name : Satyam Sahu

Inheritance in OOP allows a class to inherit attributes and methods from another class. The class that inherits is called a subclass (or derived class, child class), and the class being inherited from is called a superclass (or base class, parent class).

Here's a sample program demonstrating both single inheritance and multilevel inheritance in

```
class Animal {
  String name;

  public Animal(String name) {
    this.name = name;
  }

  void eat() {
    System.out.println(name + " is eating");
  }
}

class Dog extends Animal {
  String breed;

  public Dog(String name, String breed) {
    super(name);
    this.breed = breed;
  }

  void bark() {
    System.out.println(name + " is barking");
  }
}

class Puppy extends Dog {
  int age;

  public Puppy(String name, String breed, int age) {
    super(name, breed);
```

```
    this.age = age;
  }

  void play() {
    System.out.println(name + " is playing");
  }
}

public class InheritanceDemo {
  public static void main(String[] args) {
    Animal animal = new Animal("Generic Animal");
    animal.eat();

    Dog dog = new Dog("Max", "Labrador");
    dog.eat();
    dog.bark();

    Puppy puppy = new Puppy("Buddy", "Beagle", 3);
    puppy.eat();
    puppy.bark();
    puppy.play();
  }
}
```

3) What is the difference between method overloading and method overriding? Provide code examples.

Method overloading occurs when multiple methods in the same class have the same name but different parameters (different number, type, or order of parameters). This is resolved at compile time (static binding).

```
public class Calculator {

    public int add(int a, int b) {

        return a + b;
```

```
    }

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

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String a, String b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println(calc.add(5, 10));          // Calls first method: 15
        System.out.println(calc.add(5, 10, 15));      // Calls second method: 30
        System.out.println(calc.add(5.5, 10.5));      // Calls third method: 16.0
        System.out.println(calc.add("Hello ", "World")); // Calls fourth method: Hello World
    }
}
```

Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its parent class. This is resolved at runtime (dynamic binding).

Name : Satyam Sahu

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

    public void eat() {
        System.out.println("Animal eats food");
    }
}

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

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

    @Override
    public void eat() {
        System.out.println("Cat eats fish");
    }
```

```
Name : Satyam Sahu

}


public class OverridingDemo {
   public static void main(String[] args) {
      Animal myAnimal = new Animal();
      Animal myDog = new Dog();
      Animal myCat = new Cat();


      myAnimal.makeSound();
      myDog.makeSound();
      myCat.makeSound();


      myAnimal.eat();
      myDog.eat();
      myCat.eat();
   }
}
```

4) . What is encapsulation and how does it help in software development? Show how it is implemented in code.

Encapsulation in Software Development

Encapsulation is one of the four fundamental principles of object-oriented programming. It refers to the bundling of data (attributes) and methods (functions) that operate on that data within a single unit (class), while restricting direct access to some of the object's components.

How Encapsulation Helps in Software Development

1. **Data Hiding**: It prevents direct access to data, protecting it from unintended modifications.
2. **Controlled Access**: It provides controlled access to data through public methods.

3. **Flexibility**: Implementation details can be changed without affecting the code that uses the class.
4. **Maintainability**: Code becomes easier to maintain as changes in one part won't affect other parts.
5. **Reduced Complexity**: It simplifies the interface for using the object by hiding implementation details.

Implementation of Encapsulation in Code

```
public class BankAccount {
  private String accountNumber;
  private String accountHolderName;
  private double balance;
  private String pin;
  public BankAccount(String accountNumber, String accountHolderName, double initialBalance, String pin) {
    this.accountNumber = accountNumber;
    this.accountHolderName = accountHolderName;
    this.balance = initialBalance;
    this.pin = pin;
  }
  public String getAccountNumber() {
    return accountNumber;
  }

  public String getAccountHolderName() {
    return accountHolderName;
  }

  public void setAccountHolderName(String accountHolderName) {
    this.accountHolderName = accountHolderName;
  }

  public double getBalance() {
    return balance;
  }
```

Name : Satyam Sahu

```
  // No direct setter for balance - it can only be modified through defined operations
  public void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
      System.out.println("Deposit successful. New balance: " + balance);
    } else {
      System.out.println("Invalid amount for deposit.");
    }
  }

  public boolean withdraw(double amount, String providedPin) {
    if (!this.pin.equals(providedPin)) {
      System.out.println("Incorrect PIN. Transaction declined.");
      return false;
    }

    if (amount > 0 && amount <= balance) {
      balance -= amount;
      System.out.println("Withdrawal successful. New balance: " + balance);
      return true;
    } else {
      System.out.println("Invalid amount or insufficient funds.");
      return false;
    }
  }

  // No getter for PIN - security by design
  public void changePin(String oldPin, String newPin) {
    if (this.pin.equals(oldPin)) {
      this.pin = newPin;
      System.out.println("PIN changed successfully.");
    } else {
      System.out.println("Incorrect old PIN. PIN change failed.");
    }
```

Name : Satyam Sahu

```
  }
}


public class EncapsulationDemo {
  public static void main(String[] args) {
    BankAccount account = new BankAccount("123456789", "John Doe", 1000.0, "1234");

    System.out.println("Account Holder: " + account.getAccountHolderName());
    System.out.println("Account Balance: $" + account.getBalance());

    account.deposit(500.0);
    account.withdraw(200.0, "1234");

    account.setAccountHolderName("John Smith");
    System.out.println("Updated Account Holder: " + account.getAccountHolderName());

    account.withdraw(100.0, "9999");
    account.changePin("1234", "5678");

  }
}
```