Course Title: Object - Oriented Programming II LAB

Course Code: CSE 2110

Submitted by	Submitted to
Name: Muhammad Shabab Sayem ID: 11230321377 Department: CSE Section: 3J	Name: SHOVON MANDAL Designation: Lecturer Dept. of Computer Science Engineering Northern University of Business & Technology, Khulna

Submission Date: 03/02/25

Problem 1:

Question:

Write a Java program to create a superclass Animal with properties name and age. Create a subclass Dog that extends Animal and adds a new method bark(). Demonstrate how to create an object of Dog and access both superclass and subclass methods

Objective:

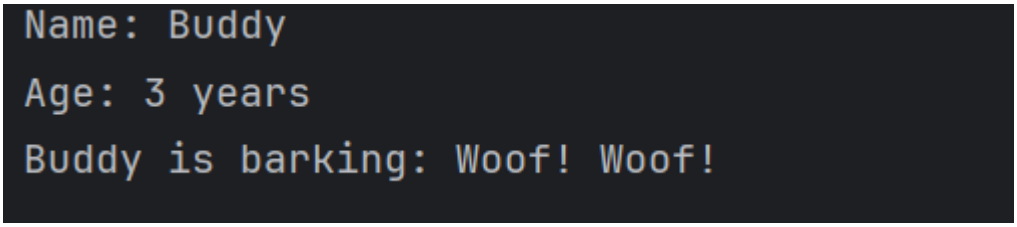
The objective of this Java program is to demonstrate **inheritance**, a fundamental Object-Oriented Programming (OOP) concept, where a subclass (Dog) inherits attributes and methods from a superclass (Animal). This allows code reusability and logical structuring of related objects.

Lab Work:

```
class Animal {
    String name;
    int age;


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

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age + " years");
    }
}

class Dog extends Animal {
    public Dog(String name, int age) {
        super(name, age); // Calling superclass constructor
    }

    public void bark() {
        System.out.println(name + " is barking: Woof! Woof!");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", 3);

        myDog.displayInfo();
```

```
        myDog.bark();
    }
}
```

Output:



Result analysis:

✅ Encapsulation & Reusability: Attributes and methods from Animal are reused in Dog, avoiding redundancy.
✅ Inheritance in Action: Dog extends Animal to gain its properties.
✅ Method Overloading or Overriding Not Used Yet: The subclass doesn’t modify the displayInfo() method, but it could be overridden for a more customized output.

Problem 2:

Question:

Explain the purpose of the extends keyword in Java inheritance. Write a program that demonstrates how a child class inherits properties and methods from a parent class.

Objective:

The objective of this program is to demonstrate inheritance in Java using the extends keyword. Specifically, it shows how a child class (Dog) inherits properties and methods from a parent class (Animal), overrides a method, and introduces its own unique method.

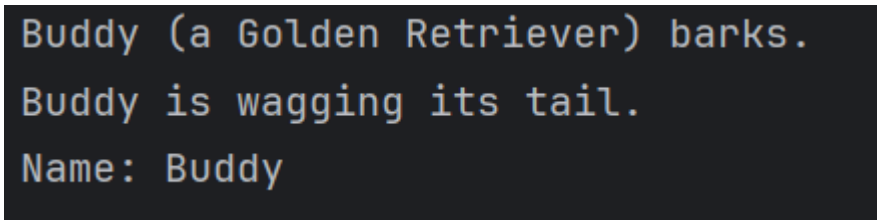
The extends keyword in Java is used for class inheritance. It allows a child (subclass) to inherit the properties and methods of a parent (superclass). This promotes code reusability and enables polymorphism.

Lab work:

```
class Animal {
    String name;

    // Constructor
    Animal(String name) {
        this.name = name;
    }

    // Method to display general animal behavior
    void makeSound() {
        System.out.println(name + " makes a sound.");
    }
}

class Dog extends Animal {
    String breed;

    Dog(String name, String breed) {
        super(name); // Calls the parent class constructor
        this.breed = breed;
    }

    @Override
    void makeSound() {
        System.out.println(name + " (a " + breed + ") barks.");
    }

    void wagTail() {
        System.out.println(name + " is wagging its tail.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", "Golden Retriever");

        myDog.makeSound();
        myDog.wagTail();

        // Demonstrating access to parent class members
        System.out.println("Name: " + myDog.name);
    }
}
```

Output:



Result analysis:

This program effectively illustrates how Java inheritance works: ✅ **Code reusability**: Dog reuses Animal properties and methods.
✅ **Method overriding**: Dog customizes inherited behavior.
✅ **Extensibility**: New features can be added to subclasses without modifying the parent class.

Problem 3:

Question:

Define a superclass Vehicle with a method move(). Create a subclass Car that overrides the move() method to provide a more specific implementation. Demonstrate method overriding in action.

Objective:

The objective of this program is to demonstrate method overriding in Java. It showcases how a subclass (Car) can provide a specific implementation of a method (move()) that is already defined in its superclass (Vehicle). Additionally, it highlights runtime polymorphism, where a Vehicle reference points to a Car object, but the overridden method in Car is executed.

Lab Work:

```
class Vehicle {
    void move() {
        System.out.println("Vehicle is moving.");
    }
}

class Car extends Vehicle {
    @Override
    void move() {
        System.out.println("Car is driving on the road.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.move();

        Car c = new Car();
        c.move();

        Vehicle vc = new Car();
        vc.move();
    }
}
```

Output:

```
Vehicle is moving.
Car is driving on the road.
Car is driving on the road.
```

Result analysis:

✅ **Method Overriding:** The Car class overrides the move() method from Vehicle to provide a more specific behavior.
✅ **Polymorphism:** When a Vehicle reference holds a Car object, the overridden method in Car is executed.
✅ **Code Reusability & Extensibility:** The Car class inherits common behavior from Vehicle while modifying it as needed.

Problem 4:

Question:

Create a class Parent with three instance variables: one private, one protected, and one public. Create a subclass Child and attempt to access these variables. Explain which variables can be accessed and why.

Objective:

The goal of this program is to demonstrate how access modifiers (private, protected, and public) work in Java. Specifically, it shows:
The visibility of different types of instance variables (with different access levels) in a **parent class** (Parent) and a **child class** (Child).
How these variables can be accessed within the **same class**, **subclass**, and **outside the class**.

Lab work:

```
class Parent {
    private int privateVar = 10;
    protected int protectedVar = 20;
    public int publicVar = 30;

    void display() {
        System.out.println("Private: " + privateVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Public: " + publicVar);
    }
}

class Child extends Parent {
    void show() {
        // System.out.println("Private: " + privateVar); // Not accessible
        System.out.println("Protected: " + protectedVar);
        System.out.println("Public: " + publicVar);
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.show();

        Parent p = new Parent();
        // System.out.println(p.privateVar); // Not accessible
        // System.out.println(p.protectedVar); // Not accessible outside subclass
        System.out.println(p.publicVar);
    }
}
```

Output:

```
Protected: 20
Public: 30
30
```

Result analysis:

This program effectively illustrates the use of **access modifiers** in Java:
private: Restricted to the class.
protected: Accessible within the class, subclass, and package.
public: Accessible from anywhere.

Problem 5:

Question:

Write a Java program to show how the super keyword is used to call a superclass method and constructor. Create a superclass Person with a constructor and method, then create a subclass Employee that calls the superclass constructor and method using super.

Objective:

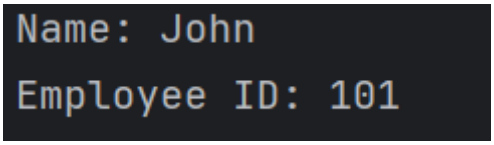
The objective of this Java program is to demonstrate the use of the super keyword to:
Call a superclass constructor: The Employee class calls the Person class constructor using super(name).
Call a superclass method: The Employee class calls the display() method of the superclass Person using super.display().
This showcases how the super keyword can be used to:
Initialize properties from a superclass.
Access methods from a superclass.

Lab work:

```
class Person {
    String name;

    Person(String name) {
        this.name = name;
    }

    void display() {
        System.out.println("Name: " + name);
    }
}

class Employee extends Person {
    int id;

    Employee(String name, int id) {
        super(name);
        this.id = id;
    }

    void display() {
        super.display();
        System.out.println("Employee ID: " + id);
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        Employee e = new Employee("John", 101);
        e.display();
    }
}
```

Output:



Result discussion:

This program demonstrates how the super keyword works in Java to:
Invoke the superclass constructor.
Call the superclass method.

Problem 6:

Question:

o Implement a program with three levels of inheritance:

♣ Animal (Superclass)

♣ Mammal (Intermediate Class)

♣ Dog(Subclass)

Show how properties are inherited across multiple levels

Objective:

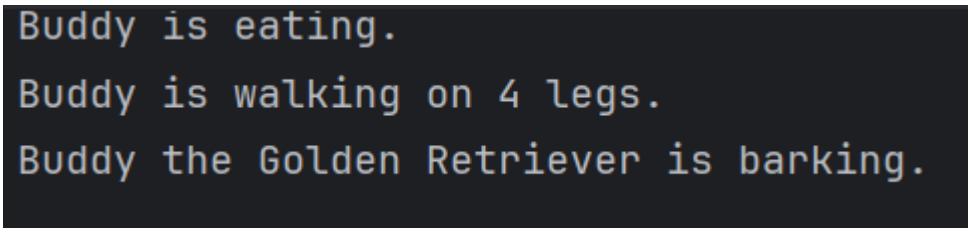
The primary objective is to show how inheritance allows a subclass (like Mammal or Dog) to reuse methods and properties from its superclass (Animal and Mammal), and to extend or override functionality as needed. This also illustrates the concept of method overriding (for custom behaviors) and method inheritance (reusing behaviors).

Lab Work:

```
class Animal {
    String name;

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

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

class Mammal extends Animal {
    int legs;

    Mammal(String name, int legs) {
        super(name);
        this.legs = legs;
    }

    void walk() {
        System.out.println(name + " is walking on " + legs + " legs.");
    }
}

class Dog extends Mammal {
    String breed;

    Dog(String name, int legs, String breed) {
        super(name, legs);
        this.breed = breed;
```

```
    }

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", 4, "Golden Retriever");

        myDog.eat();
        myDog.walk();
        myDog.bark();
    }
}
```

Output:



Result discussion:

 Inheritance: The Dog class inherits from Mammal, which in turn inherits from Animal. This means the Dog class has access to all public/protected methods and fields of its parent classes (like eat() from Animal and walk() from Mammal).
 Method Overriding: The methods eat(), walk(), and bark() are defined in such a way that the Dog object can use all of them. However, none of the methods are overridden, so each class's version of the method is called in the hierarchy.
 Constructors: Each class's constructor ensures proper initialization by calling the constructor of the parent class using super() (which is essential for proper initialization in the inheritance chain).

Problem: 7

Question:

Create a superclass Shape with a method draw(). Create two subclasses Circle and Rectangle that override draw(). Write a Java program to demonstrate dynamic method dispatch using a reference variable of type Shape

Objective:

The objective of this program is to demonstrate dynamic method dispatch in Java, where:
A reference variable of type Shape is used to refer to objects of its subclasses (Circle and Rectangle).
The appropriate overridden method (draw()) is invoked based on the actual object type at runtime (i.e., Circle or Rectangle), rather than the reference type (Shape).

Lab Work:

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

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape s1 = new Circle();
        Shape s2 = new Rectangle();

        s1.draw();
        s2.draw();
    }
}
```

Output:

```
Drawing a Circle
Drawing a Rectangle
```

Result discussion:

- Dynamic Method Dispatch:

The variable s1 is a reference of type Shape but is assigned an instance of Circle. When s1.draw() is called, the draw() method of Circle is invoked, because the actual object is of type Circle.
Similarly, s2 is a reference of type Shape, but assigned an instance of Rectangle. When s2.draw() is called, the draw() method of Rectangle is invoked, because the actual object is of type Rectangle.

- Overriding:

Both Circle and Rectangle override the draw() method from Shape. This is an example of method overriding where the subclass provides its own implementation of a method that is already defined in the superclass.

- Runtime Polymorphism:

This program demonstrates runtime polymorphism, where the method that gets executed is determined at runtime based on the actual object type (Circle or Rectangle), even though the reference is of type Shape.

Problem 8:

Question:

Define an abstract class BankAccount with an abstract method calculateInterest(). Create two subclasses SavingsAccount and CurrentAccount that implement this method differently. Write a Java program to demonstrate abstract classes.

Objective:

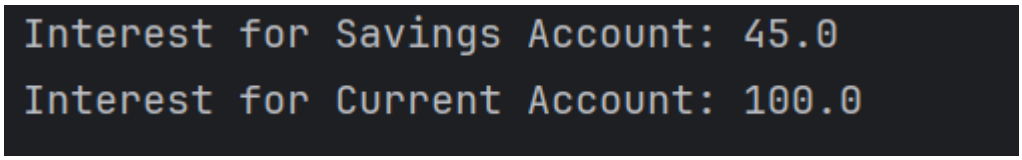
The objective of this program is to demonstrate the use of **abstract classes** and **abstract methods** in Java. In this program:
The BankAccount class is abstract and defines an abstract method calculateInterest().
The SavingsAccount and CurrentAccount classes extend BankAccount and provide their own implementations for calculateInterest().
This demonstrates the concept of **polymorphism** by calling the overridden calculateInterest() method on objects of the subclasses.

Lab work:

```
abstract class BankAccount {
    double balance;

    BankAccount(double balance) {
        this.balance = balance;
    }

    abstract void calculateInterest();
}

class SavingsAccount extends BankAccount {
    double interestRate;

    SavingsAccount(double balance, double interestRate) {
        super(balance);
        this.interestRate = interestRate;
    }

    @Override
    void calculateInterest() {
        double interest = balance * interestRate / 100;
        System.out.println("Interest for Savings Account: " + interest);
    }
}

class CurrentAccount extends BankAccount {
    double overdraftLimit;

    CurrentAccount(double balance, double overdraftLimit) {
        super(balance);
        this.overdraftLimit = overdraftLimit;
```

```
    }

    @Override
    void calculateInterest() {
        double interest = balance * 0.05; // Fixed interest rate for current accounts
        System.out.println("Interest for Current Account: " + interest);
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount savings = new SavingsAccount(1000, 4.5);
        savings.calculateInterest();

        BankAccount current = new CurrentAccount(2000, 500);
        current.calculateInterest();
    }
}
```

Output:



Result discussion:

- Abstract Class (BankAccount):

The BankAccount class is abstract, meaning it cannot be instantiated directly. It has an abstract method calculateInterest(), which must be implemented by any subclass.
It also contains a regular method and a constructor that initializes the balance.

- Subclasses (SavingsAccount and CurrentAccount):

Both SavingsAccount and CurrentAccount are concrete classes that inherit from BankAccount and provide their own implementation for calculateInterest().
SavingsAccount calculates interest based on a provided interest rate.
CurrentAccount calculates interest using a fixed rate of 5%.

- In the Main class:

Instances of SavingsAccount and CurrentAccount are created, and their calculateInterest() methods are called.
Despite being referred to by the BankAccount type, each subclass provides its specific implementation of the calculateInterest() method, showcasing polymorphism.

Problem 9:

Question:

Explain the role of the final keyword in preventing inheritance and method overriding. Write a Java program to demonstrate how declaring a class or method as final affects inheritance.

Objective:

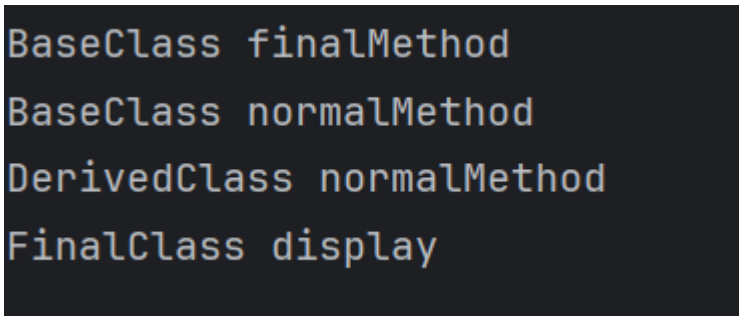
To demonstrate the effect of the final keyword on inheritance and method overriding in Java. Specifically, to show how declaring a class as final prevents inheritance, and declaring a method as final prevents method overriding.

Lab Work:

```
class BaseClass {
    final void finalMethod() {
        System.out.println("BaseClass finalMethod");
    }

    void normalMethod(){
        System.out.println("BaseClass normalMethod");
    }
}

final class FinalClass {
    void display() {
        System.out.println("FinalClass display");
    }
}

// class DerivedClass extends FinalClass { // This will cause a compile-time error
// }

class DerivedClass extends BaseClass {
    // void finalMethod() { // This will cause a compile-time error
    //     System.out.println("DerivedClass finalMethod");
    // }

    @Override
    void normalMethod(){
        System.out.println("DerivedClass normalMethod");
    }
}

public class FinalKeywordDemo {
    public static void main(String[] args) {
        BaseClass base = new BaseClass();
        base.finalMethod();
```

```
        base.normalMethod();

        DerivedClass derived = new DerivedClass();
        derived.normalMethod();

        FinalClass finalClass = new FinalClass();
        finalClass.display();
    }
}
```

Output:



Result discussion:

The output demonstrates that the finalMethod of the BaseClass is called (and cannot be overridden), while the normalMethod is overridden in DerivedClass. It also shows that the display method of the FinalClass can be called, but the class itself cannot be extended. This behavior illustrates the two key uses of the final keyword: preventing inheritance and preventing method overriding.

Problem 10:

Question:

Demonstrate how all Java classes inherit from Object by creating a class Student and overriding the toString() method. Show how an object of Student can use the equals() and toString() methods inherited from Object.

Objective:

To demonstrate inheritance in Java, specifically how all classes implicitly inherit from the Object class. This is achieved by creating a custom class Student and overriding the toString() and equals() methods, which are originally defined in the Object class. The program then creates Student objects and utilizes these overridden methods, along with the getClass() method inherited from Object, to illustrate the concepts of inheritance and object equality.

Lab work:

```
class Student {
    private String name;
    private int rollNumber;

    public Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }

    @Override
    public String toString() {
        return "Name: " + name + ", Roll Number: " + rollNumber;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Student student = (Student) obj;
        return rollNumber == student.rollNumber && name.equals(student.name);
    }


    public static void main(String[] args) {
        Student student1 = new Student("Alice", 123);
        Student student2 = new Student("Alice", 123);
        Student student3 = new Student("Bob", 456);

        System.out.println(student1.toString());
        System.out.println(student2.toString());
        System.out.println(student3.toString());
```

```
        System.out.println("student1 equals student2: " + student1.equals(student2));
        System.out.println("student1 equals student3: " + student1.equals(student3));

        Object obj = new Student("Charlie",789);
        System.out.println(obj.toString());
        System.out.println(student1.getClass());
        System.out.println(obj.getClass());

    }
}'
```

Output:

```
Name: Alice, Roll Number: 123
Name: Alice, Roll Number: 123
Name: Bob, Roll Number: 456
student1 equals student2: true
student1 equals student3: false
Name: Charlie, Roll Number: 789
class Student
class Student
```

Result discussion:

 The toString() method, when overridden, provides a meaningful string representation of the Student object, containing the name and roll number. Without overriding, the default toString() from Object would print a less informative representation (e.g., the class name followed by a hash code).
 The equals() method, when overridden, allows for a custom comparison of Student objects based on their content (name and roll number). The example shows that student1 and student2 are considered equal because they have the same name and roll number, while student1 and student3 are not equal. The default equals() from Object would only check for reference equality (whether two variables refer to the same object in memory).
 The example demonstrates the inheritance from Object class by creating a reference of Object class and assigning a Student object to it. The toString() method is called on the object reference. It prints the string representation of the Student object. This shows that the Student class inherits the toString() method from the Object class. Also, the getClass() method is called on both student1 and obj. Both return the class of their respective objects. This

confirms that student1 is an instance of Student class and obj is also an instance of Student class. Both inherit the getClass() method from the Object class.