SOLID PRINCIPLES

1.Single Responsibility Principles

* The **Single Responsibility Principle (SRP)** is one of the SOLID principles of object-oriented design. It states that a class should have only **one reason to change**, meaning it should have **only one job or responsibility**.

Code : (INVOICE)

public class Main {

public static void main(String[] args) {

Invoice invoice = new Invoice("Laptop", 1, 1200.00);

InvoicePrinter printer = new InvoicePrinter();

InvoiceRepository repository = new InvoiceRepository();

printer.print(invoice);

repository.save(invoice);

}

}

class Invoice {

private String item;

private int quantity;

private double price;

public Invoice(String item, int quantity, double price) {

this.item = item;

this.quantity = quantity;

this.price = price;

}

public double calculateTotal() {

return quantity \* price;

}

public String getItem() {

return item;

}

public int getQuantity() {

return quantity;

}

public double getPrice() {

return price;

}

}

class InvoicePrinter {

public void print(Invoice invoice) {

System.out.println("------- Invoice -------");

System.out.println("Item: " + invoice.getItem());

System.out.println("Quantity: " + invoice.getQuantity());

System.out.println("Price: $" + invoice.getPrice());

System.out.println("Total: $" + invoice.calculateTotal());

System.out.println("------------------------");

}

}

class InvoiceRepository {

public void save(Invoice invoice) {

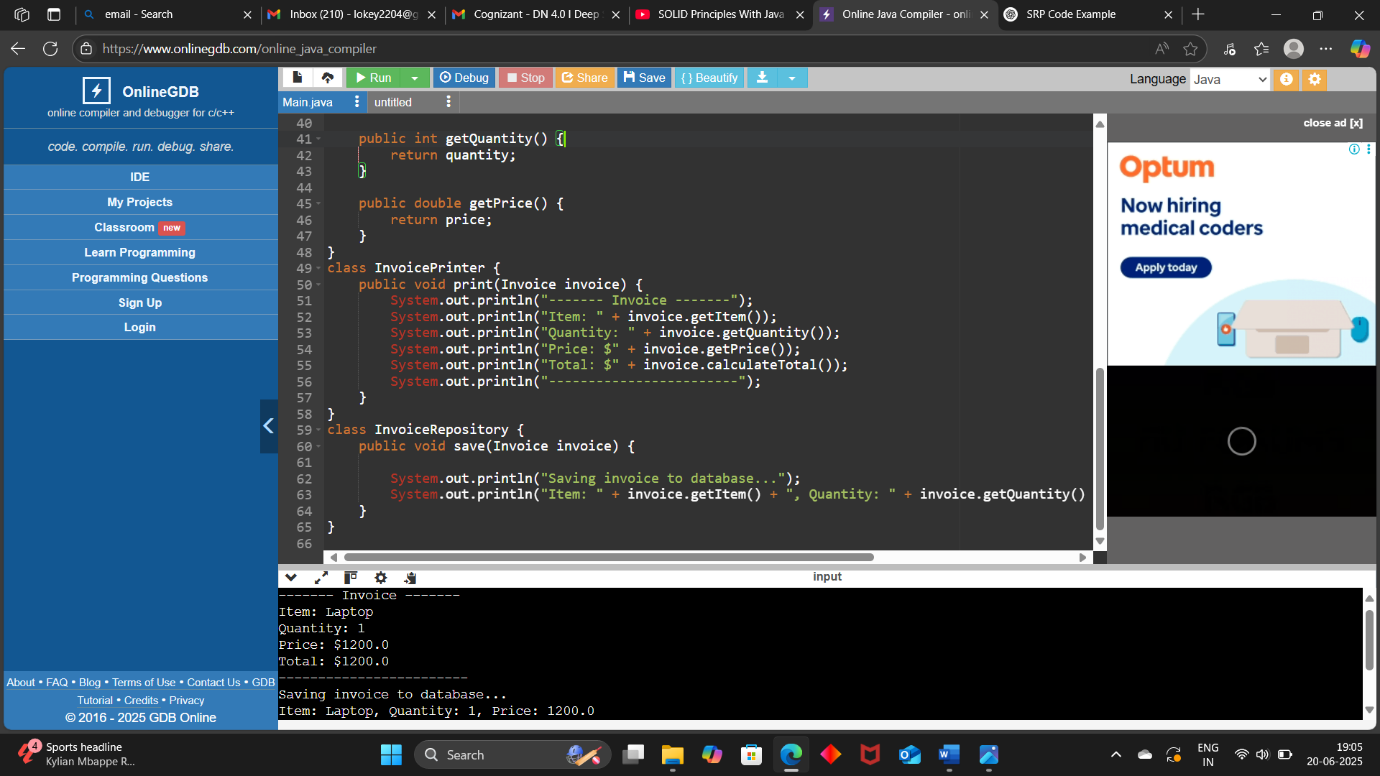
System.out.println("Saving invoice to database...");

System.out.println("Item: " + invoice.getItem() + ", Quantity: " + invoice.getQuantity() + ", Price: " + invoice.getPrice());

}

}

OUTPUT : (ONLINE GDB COMPILER)



2. Open/Closed Principle

* This means you should be able to **add new behavior** (extend) **without changing existing code** (modification).

CODE : (Area Calculator)

public class Main {

public static void main(String[] args) {

Shape circle = new Circle(5);

Shape rectangle = new Rectangle(4, 6);

Shape triangle = new Triangle(3, 4);

AreaCalculator calculator = new AreaCalculator();

System.out.println("Circle Area: " + calculator.calculateArea(circle));

System.out.println("Rectangle Area: " + calculator.calculateArea(rectangle));

System.out.println("Triangle Area: " + calculator.calculateArea(triangle));

}

}

interface Shape {

double area();

}

class Circle implements Shape {

private double radiu

public Circle(double radius) {

this.radius = radius;

}

public double area() {

return Math.PI \* radius \* radius;

}

}

class Rectangle implements Shape {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

public double area() {

return length \* width;

}

}

class Triangle implements Shape {

private double base;

private double height;

public Triangle(double base, double height) {

this.base = base;

this.height = height;

}

public double area() {

return 0.5 \* base \* height;

}

}

class AreaCalculator {

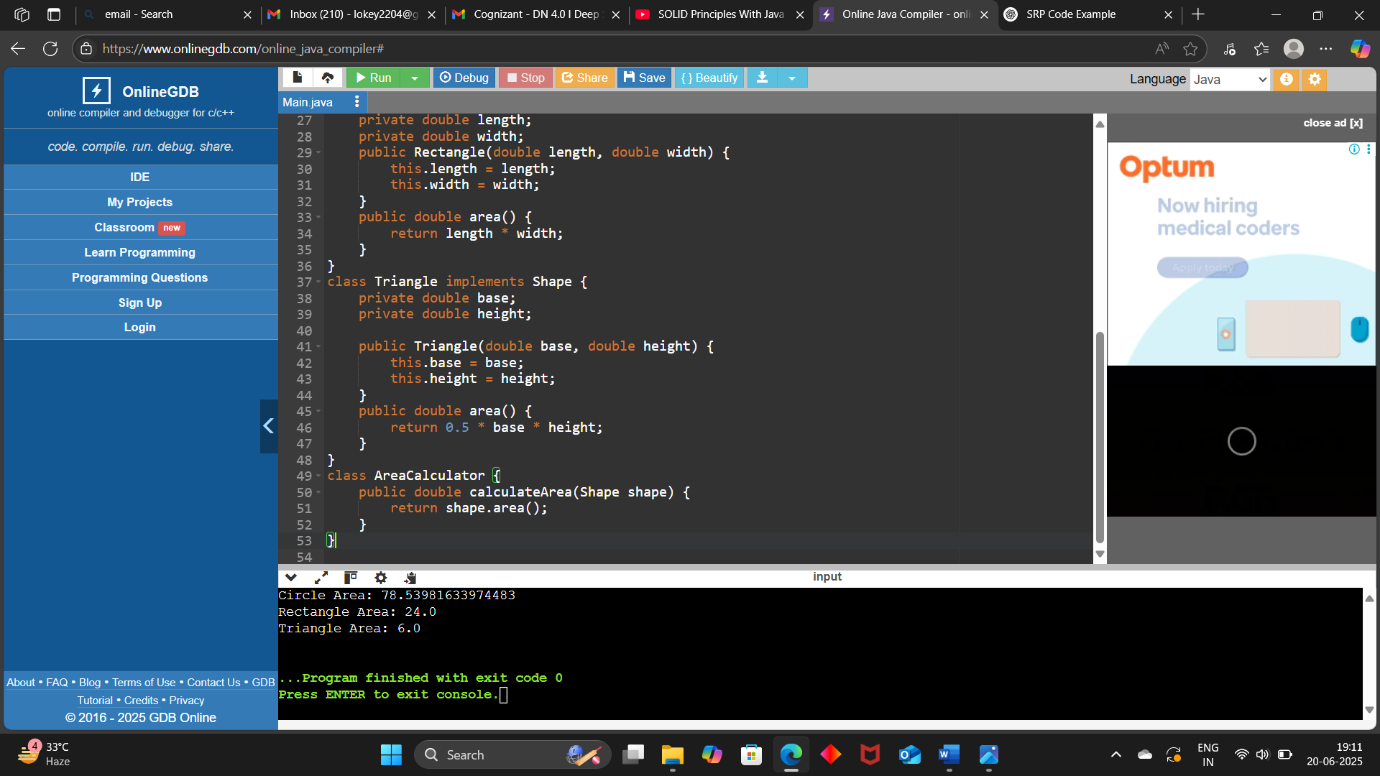
public double calculateArea(Shape shape) {

return shape.area();

}

}

OUTPUT:



3. Liskov Substitution Principle

* **Objects of a superclass should be replaceable with objects of a subclass without breaking the application.**
* In simpler terms:
* If class B is a subclass of class A, then we should be able to **use B in place of A without unexpected behavior**.

CODE : ( Area of SHAPES)

public class Main {

public static void main(String[] args) {

Shape rect = new Rectangle(4, 5);

Shape square = new Square(4);

printArea(rect);

printArea(square);

}

public static void printArea(Shape shape) {

System.out.println("Area: " + shape.getArea());

}

}

interface Shape {

double getArea();

}

class Rectangle implements Shape {

protected double width;

protected double height;

public Rectangle(double width, double height) {

this.width = width;

this.height = height;

}

public double getArea() {

return width \* height;

}

}

class Square implements Shape {

private double side;

public Square(double side) {

this.side = side;

}

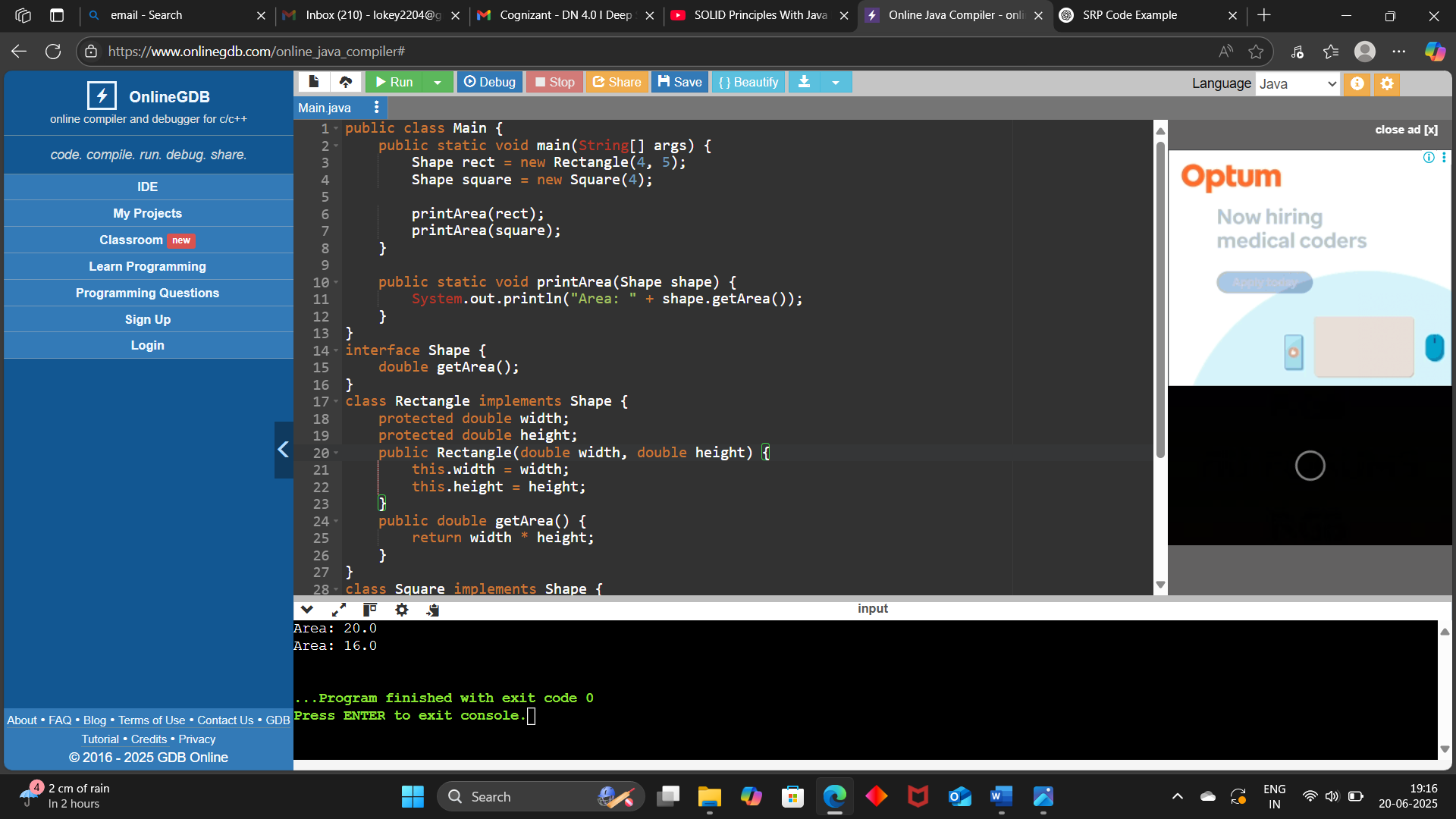
public double getArea() {

return side \* side;

}

}

OUTPUT :



4. ***Interface Segregation Principle (ISP)***:

* **"Clients should not be forced to depend upon interfaces that they do not use."**
* This means instead of having one fat interface, we should have multiple smaller, specific interfaces.

CODE :

interface Workable {

void work();

}

interface Eatable {

void eat();

}

class HumanWorker implements Workable, Eatable {

@Override

public void work() {

System.out.println("Human is working...");

}

@Override

public void eat() {

System.out.println("Human is eating...");

}

}

class RobotWorker implements Workable {

@Override

public void work() {

System.out.println("Robot is working...");

}

}

public class InterfaceSegregationExample {

public static void main(String[] args) {

Workable human = new HumanWorker();

Workable robot = new RobotWorker();

Eatable eater = new HumanWorker();

human.work();

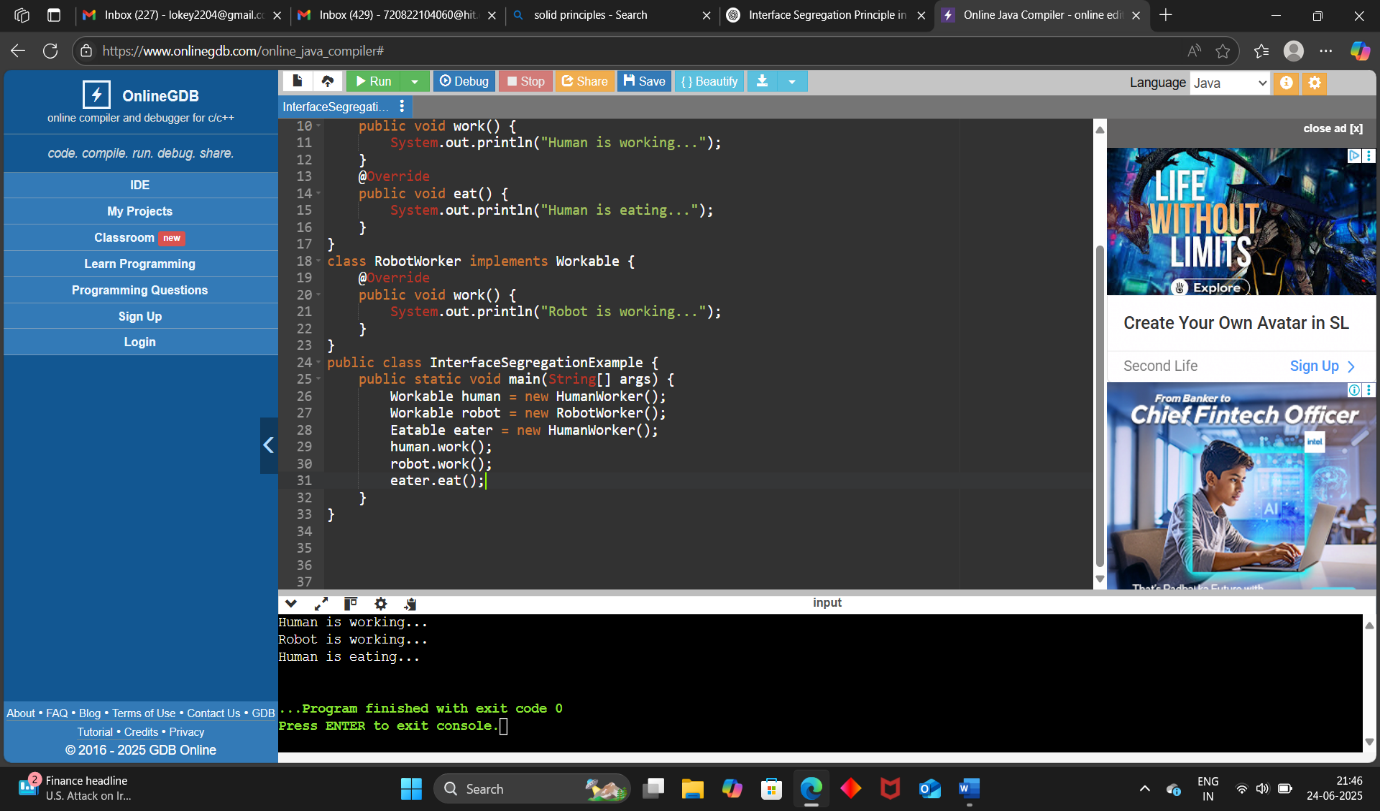
robot.work();

eater.eat();

}

}

OUTPUT :



5. Dependency Inversion Principle

* High-level modules should not depend on low-level modules. Both should depend on abstractions.  
  Also: Abstractions should not depend on details. Details should depend on abstractions.

CODE :

interface SwitchableDevice {

void turnOn();

void turnOff();

}

class LightBulb implements SwitchableDevice {

@Override

public void turnOn() {

System.out.println("LightBulb: turned ON");

}

@Override

public void turnOff() {

System.out.println("LightBulb: turned OFF");

}

}

class Fan implements SwitchableDevice {

@Override

public void turnOn() {

System.out.println("Fan: turned ON");

}

@Override

public void turnOff() {

System.out.println("Fan: turned OFF");

}

}

class ElectricSwitch {

private final SwitchableDevice device;

private boolean on;

public ElectricSwitch(SwitchableDevice device) {

this.device = device;

this.on = false;

}

public void press() {

if (on) {

device.turnOff();

on = false;

} else {

device.turnOn();

on = true;

}

}

}

public class DependencyInversionExample {

public static void main(String[] args) {

SwitchableDevice bulb = new LightBulb();

SwitchableDevice fan = new Fan();

ElectricSwitch bulbSwitch = new ElectricSwitch(bulb);

ElectricSwitch fanSwitch = new ElectricSwitch(fan);

System.out.println("Toggling Bulb Switch:");

bulbSwitch.press();

bulbSwitch.press();

System.out.println("\nToggling Fan Switch:");

fanSwitch.press();

fanSwitch.press();

}

}

OUTPUT :

