Learner Assignment Submission Format

Learner Details

- **Name: Shrayanth S**
- **Enrollment Number: SU625MR011**
- **Batch / Class: June 2025**
- **Assignment: (Bridge Course Day 6)**
- **Date of Submission: 01/07/2025**

Problem Solving Activity 1.1

1. Program Statement

Problem 1.1: Employee Hierarchy
Create a base class Employee with:

- Attributes: name, employeeId, salary
- Method: getDetails()

Subclass Manager:

- Attribute: department
- Override getDetails() to include department

Subclass Developer:

- Attribute: programmingLanguage
- Override getDetails()

.

---

2. Algorithm

Define the base class Employee with:

- Attributes: name, employeeId, salary
- Method: getDetails() to print these attributes

Define subclass Manager:

- Additional attribute: department
- Override getDetails() to also print department

Define subclass Developer:

- Additional attribute: programmingLanguage
- Override getDetails() to also print programming language

Create objects for Manager and Developer

Call getDetails() on each object

---

3. Pseudocode

```
Class Employee:
   Attributes: name, employeeId, salary
   Method getDetails():
      Print name, employeeId, salary

Class Manager extends Employee:
   Attribute: department
   Method getDetails():
      Call super.getDetails()
      Print department

Class Developer extends Employee:
   Attribute: programmingLanguage
   Method getDetails():
      Call super.getDetails()
      Print programmingLanguage

Main:
```

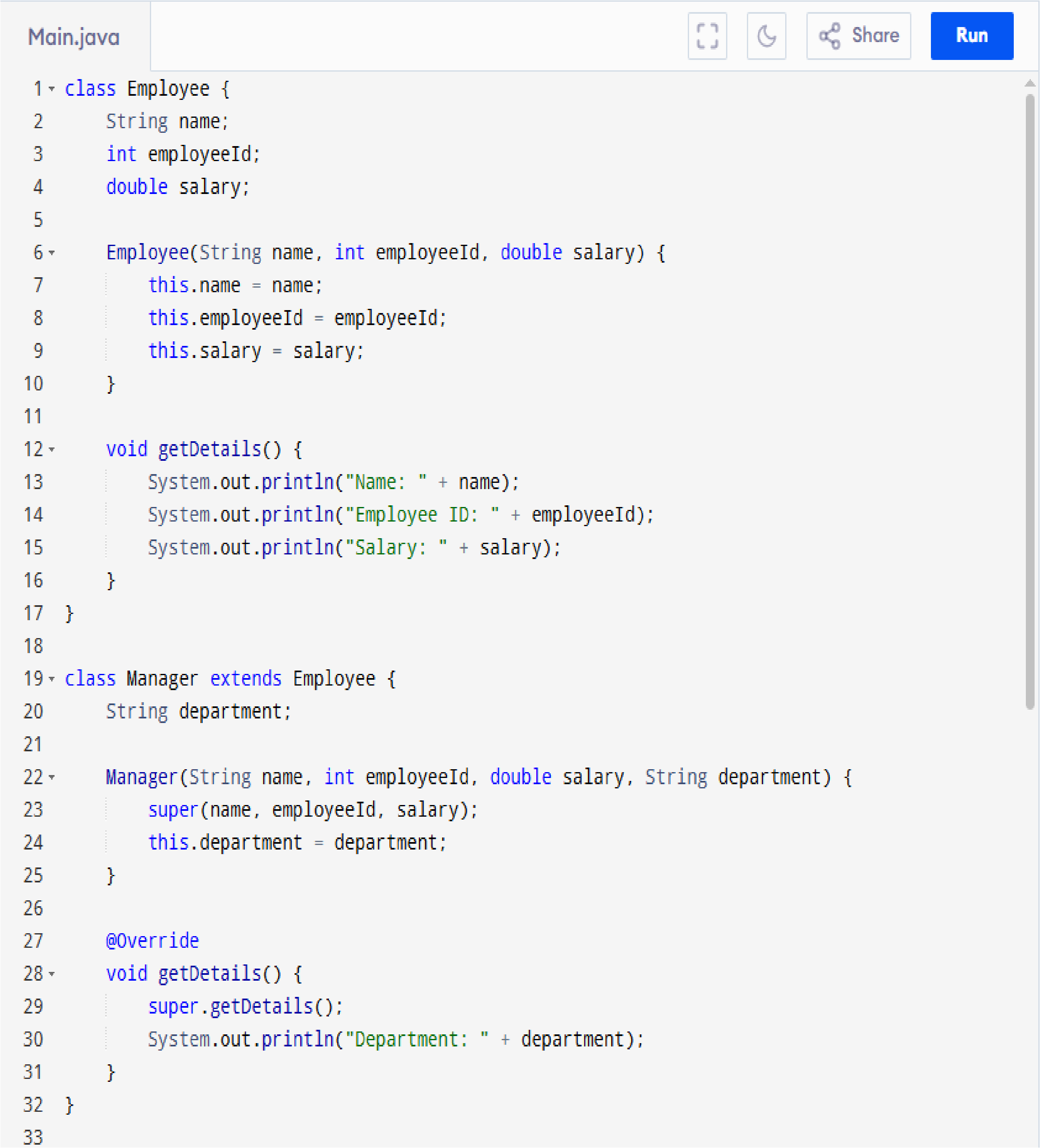
Create Manager object and set values

Call getDetails()

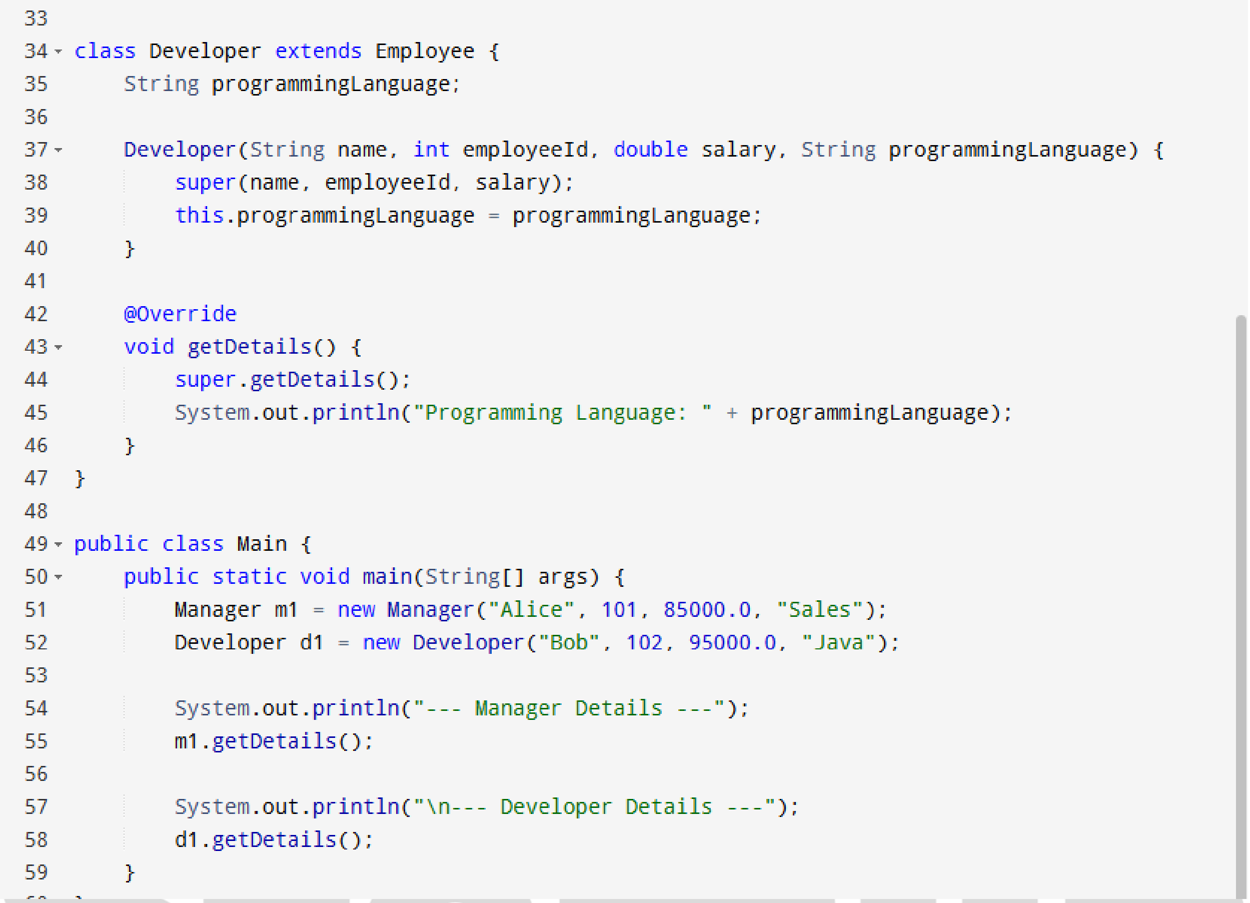
Create Developer object and set values

Call getDetails()

4. Program Code



```java
class Employee {
    String name;
    int employeeId;
    double salary;

    Employee(String name, int employeeId, double salary) {
        this.name = name;
        this.employeeId = employeeId;
        this.salary = salary;
    }

    void getDetails() {
        System.out.println("Name: " + name);
        System.out.println("Employee ID: " + employeeId);
        System.out.println("Salary: " + salary);
    }
}

class Manager extends Employee {
    String department;

    Manager(String name, int employeeId, double salary, String department) {
        super(name, employeeId, salary);
        this.department = department;
    }

    @Override
    void getDetails() {
        super.getDetails();
        System.out.println("Department: " + department);
    }
}

```

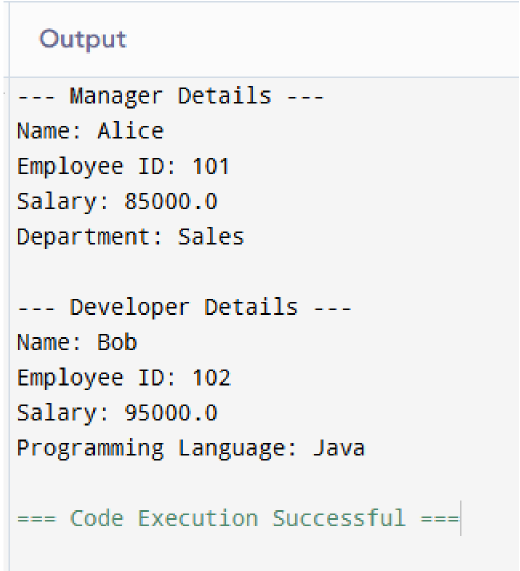


```java

class Developer extends Employee {
    String programmingLanguage;

    Developer(String name, int employeeId, double salary, String programmingLanguage) {
        super(name, employeeId, salary);
        this.programmingLanguage = programmingLanguage;
    }

    @Override
    void getDetails() {
        super.getDetails();
        System.out.println("Programming Language: " + programmingLanguage);
    }
}

public class Main {
    public static void main(String[] args) {
        Manager m1 = new Manager("Alice", 101, 85000.0, "Sales");
        Developer d1 = new Developer("Bob", 102, 95000.0, "Java");

        System.out.println("--- Manager Details ---");
        m1.getDetails();

        System.out.println("\n--- Developer Details ---");
        d1.getDetails();
    }
```

5. Test Cases

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	Manager("Alice", 101, 85000.0, "Sales")	Developer("Bob", 102, 95000.0, "Java")	As Expected	Pass
2	Developer("Bob", 102, 95000.0, "Java")	Name: Bob, Employee ID: 102, Salary: 95000.0, Programming Language: Java	As Expected	Pass

6. Screenshots of Output



Output

```
--- Manager Details ---
Name: Alice
Employee ID: 101
Salary: 85000.0
Department: Sales

--- Developer Details ---
Name: Bob
Employee ID: 102
Salary: 95000.0
Programming Language: Java

=== Code Execution Successful ===
```

7. Observation / Reflection

1. **Challenges Faced**: Managing constructor chaining and overriding getDetails() properly.
2. **What I Learned**: Understanding how polymorphism and inheritance work in Java.
3. **Improvements**: Add more employee types or features like user input and validations in the future.

Problem Solving Activity 1.2

1. Program Statement

Problem 1.2: Animal Kingdom

Base class: Animal with method makeSound() Subclasses: Dog and Cat,

override the method Create and test objects

2. Algorithm

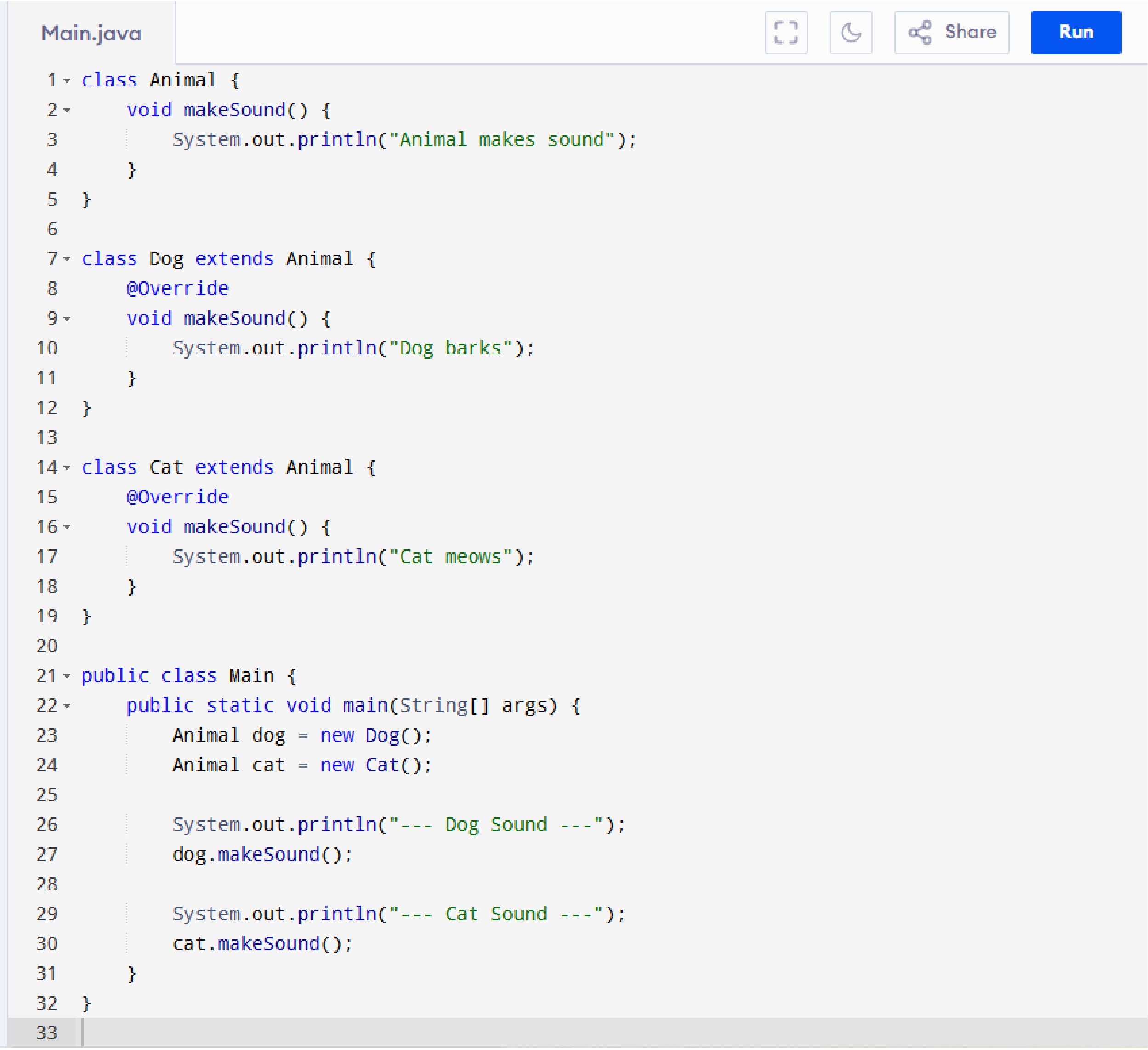
1. Create a base class Animal with a method makeSound().
2. Create a subclass Dog that overrides makeSound() to print "Dog barks".
3. Create a subclass Cat that overrides makeSound() to print "Cat meows".

4. Create objects of Dog and Cat.
5. Call makeSound() on each object to verify correct behavior.

3. Pseudocode

```
Class Animal:
    Method makeSound():
        Print "Animal makes sound"

Class Dog extends Animal:
    Method makeSound():
        Print "Dog barks"

Class Cat extends Animal:
    Method makeSound():
        Print "Cat meows"

Main:
    Create Dog object
    Call makeSound()

    Create Cat object
    Call makeSound()
```

4. Program Code

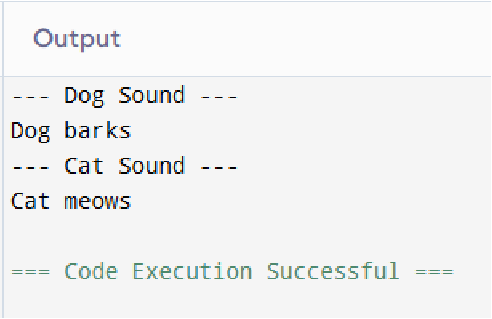


```java
class Animal {
    void makeSound() {
        System.out.println("Animal makes sound");
    }
}

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

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

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

        System.out.println("--- Dog Sound ---");
        dog.makeSound();

        System.out.println("--- Cat Sound ---");
        cat.makeSound();
    }
}

```

5. Test Cases

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	Dog object	Dog barks	Dog barks	Pass
2	Cat object	Cat meows	Cat meows	Pass

6. Screenshots of Output



Output

```
--- Dog Sound ---
Dog barks
--- Cat Sound ---
Cat meows

=== Code Execution Successful ===
```

---

7. Observation / Reflection

1. **Challenges**: Understanding how method overriding works across base and derived classes.
2. **Learning**: This task strengthened my grasp on polymorphism and the use of inheritance in Java.
3. **Improvements**: Add more animal types or use an array of Animal objects to loop through sounds.

Problem Solving Activity 1.3

1. Program Statement

Activity 1.3: Design an Inheritance Tree

Base: ElectronicDevice Subclasses: Television, Laptop, Smartphone List

attributes and methods per subclass

---

2. Algorithm

Create base class ElectronicDevice with:

- Attributes: brand, price
- Method: showDetails() to display basic device details

Subclass Television:

- Additional attribute: screenSize
- Override showDetails() to include screen size

Subclass Laptop:

- Additional attribute: RAM
- Override showDetails() to include RAM

Subclass Smartphone:

- Additional attribute: batteryLife
- Override showDetails() to include battery life

In the main() method:

- Create objects for all three subclasses
- Call showDetails() on each object

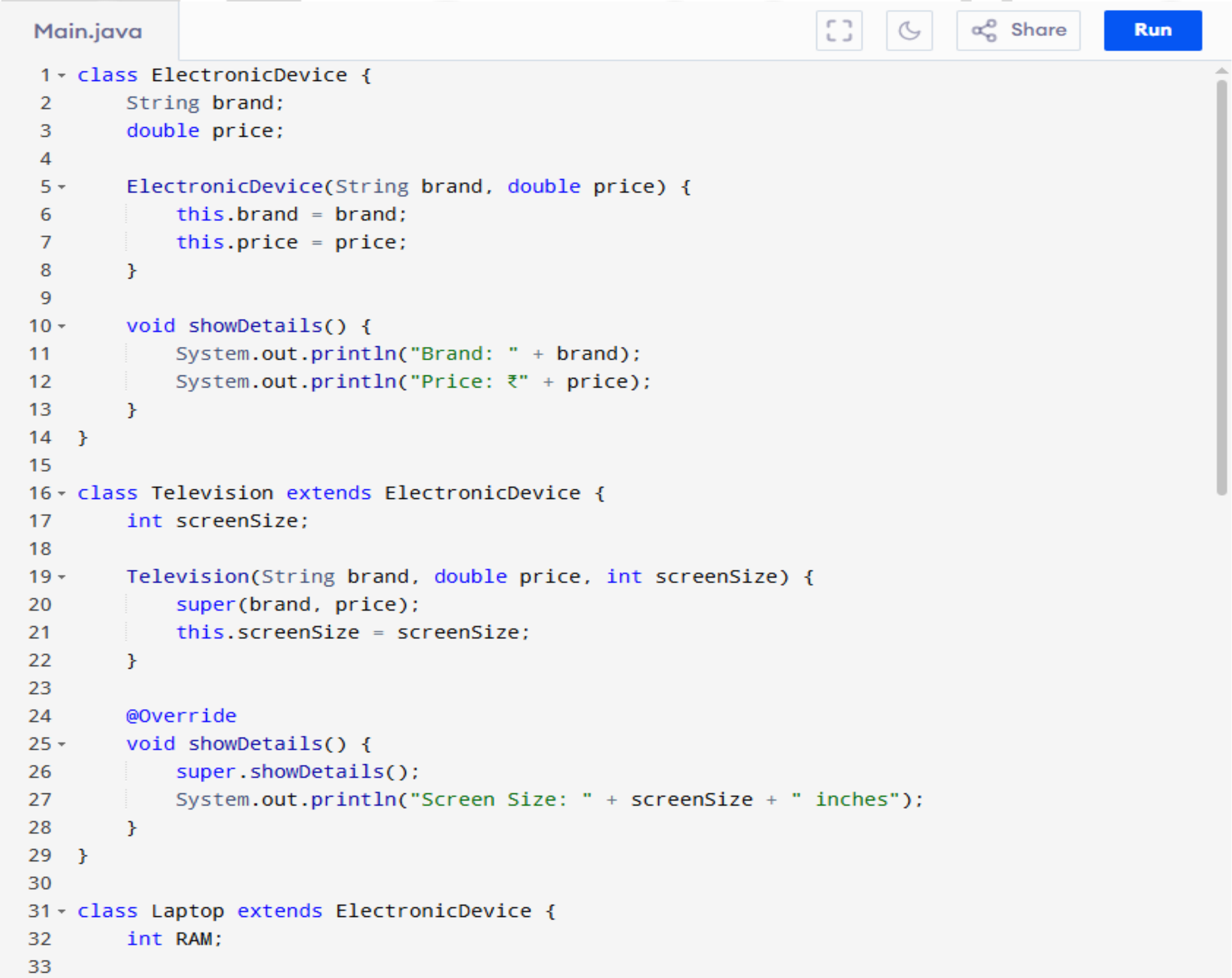
---

3. Pseudocode

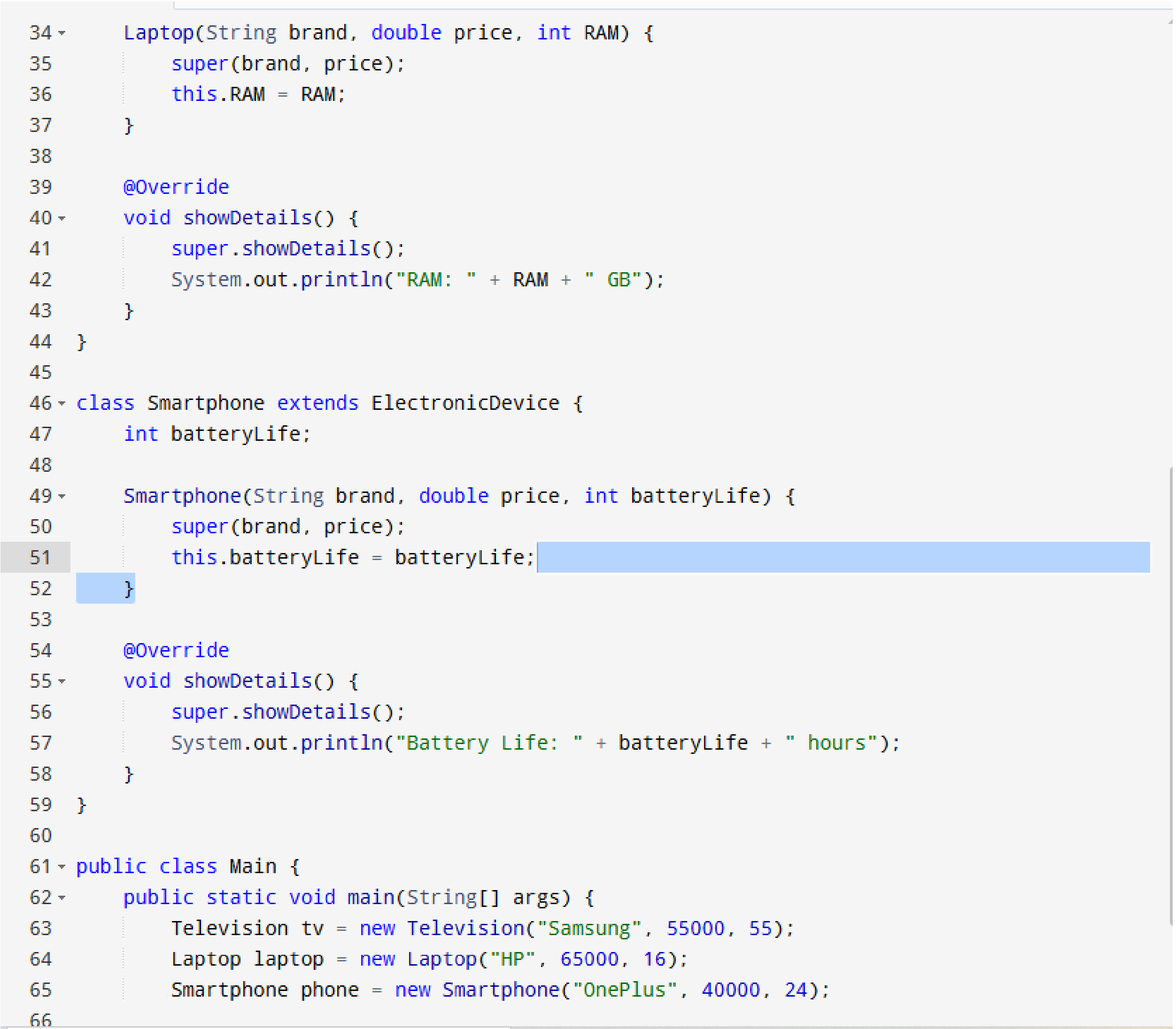
```
Class ElectronicDevice:
  Attributes: brand, price
  Method showDetails():
    Print brand and price

Class Television inherits ElectronicDevice:
  Attribute: screenSize
  Method showDetails():
    Call super.showDetails()
    Print screenSize


Class Laptop inherits ElectronicDevice:
  Attribute: RAM
  Method showDetails():
    Call super.showDetails()
    Print RAM
```

```
Class Smartphone inherits ElectronicDevice:
    Attribute: batteryLife
    Method showDetails():
        Call super.showDetails()
        Print batteryLife

Main method:
    Create object of Television
    Create object of Laptop
    Create object of Smartphone
    Call showDetails() on each
```

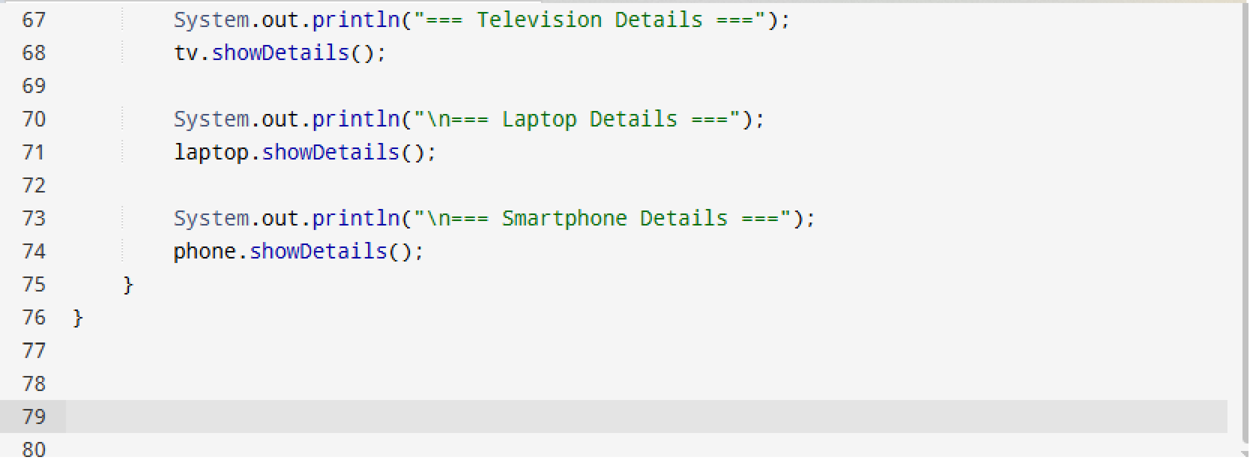
4. Program Code



```java
class ElectronicDevice {
    String brand;
    double price;

    ElectronicDevice(String brand, double price) {
        this.brand = brand;
        this.price = price;
    }

    void showDetails() {
        System.out.println("Brand: " + brand);
        System.out.println("Price: ₹" + price);
    }
}

class Television extends ElectronicDevice {
    int screenSize;

    Television(String brand, double price, int screenSize) {
        super(brand, price);
        this.screenSize = screenSize;
    }

    @Override
    void showDetails() {
        super.showDetails();
        System.out.println("Screen Size: " + screenSize + " inches");
    }
}

class Laptop extends ElectronicDevice {
    int RAM;

```



```java
    Laptop(String brand, double price, int RAM) {
        super(brand, price);
        this.RAM = RAM;
    }

    @Override
    void showDetails() {
        super.showDetails();
        System.out.println("RAM: " + RAM + " GB");
    }
}

class Smartphone extends ElectronicDevice {
    int batteryLife;

    Smartphone(String brand, double price, int batteryLife) {
        super(brand, price);
        this.batteryLife = batteryLife;
    }

    @Override
    void showDetails() {
        super.showDetails();
        System.out.println("Battery Life: " + batteryLife + " hours");
    }
}

public class Main {
    public static void main(String[] args) {
        Television tv = new Television("Samsung", 55000, 55);
        Laptop laptop = new Laptop("HP", 65000, 16);
        Smartphone phone = new Smartphone("OnePlus", 40000, 24);

```



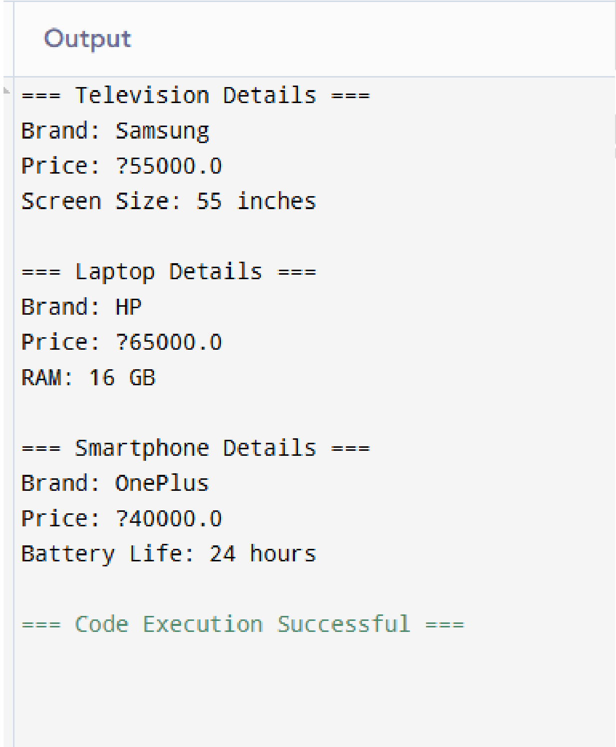
```java
        System.out.println("=== Television Details ===");
        tv.showDetails();

        System.out.println("\n=== Laptop Details ===");
        laptop.showDetails();

        System.out.println("\n=== Smartphone Details ===");
        phone.showDetails();
    }
}
```

5. Test Cases

Present a table of test cases you used to validate your program. Include a mix of regular, boundary, and edge cases.

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	Television("Samsung", 55000, 55)	Brand: Samsung, Price: ₹55000, Screen Size: 55	As Expected	Pass
2	Laptop("HP", 65000, 16)	Brand: HP, Price: ₹65000, RAM: 16 GB	As Expected	Pass
3	Smartphone("OnePlus", 40000, 24)	Brand: OnePlus, Price: ₹40000, Battery Life: 24	As Expected	Pass

6. Screenshots of Output



Output

```
=== Television Details ===
Brand: Samsung
Price: ?55000.0
Screen Size: 55 inches

=== Laptop Details ===
Brand: HP
Price: ?65000.0
RAM: 16 GB

=== Smartphone Details ===
Brand: OnePlus
Price: ?40000.0
Battery Life: 24 hours

=== Code Execution Successful ===
```

7. Observation / Reflection

1. **Challenges Faced**: Structuring the classes to avoid redundancy and understanding the use of super() in constructors.
2. **What I Learned**: Clear understanding of inheritance and how overriding methods allows each class to customize behavior.
3. **What I Would Improve**: Add interfaces for more flexibility, or a dynamic array to store different devices and loop through them.

Problem Solving Activity 2.1

1. Program Statement

Payment Gateway

Abstract class: PaymentGateway with abstract processPayment(double amount) Subclasses: CreditCardGateway, PayPalGateway Attempt to instantiate abstract class (should fail)

2. Algorithm

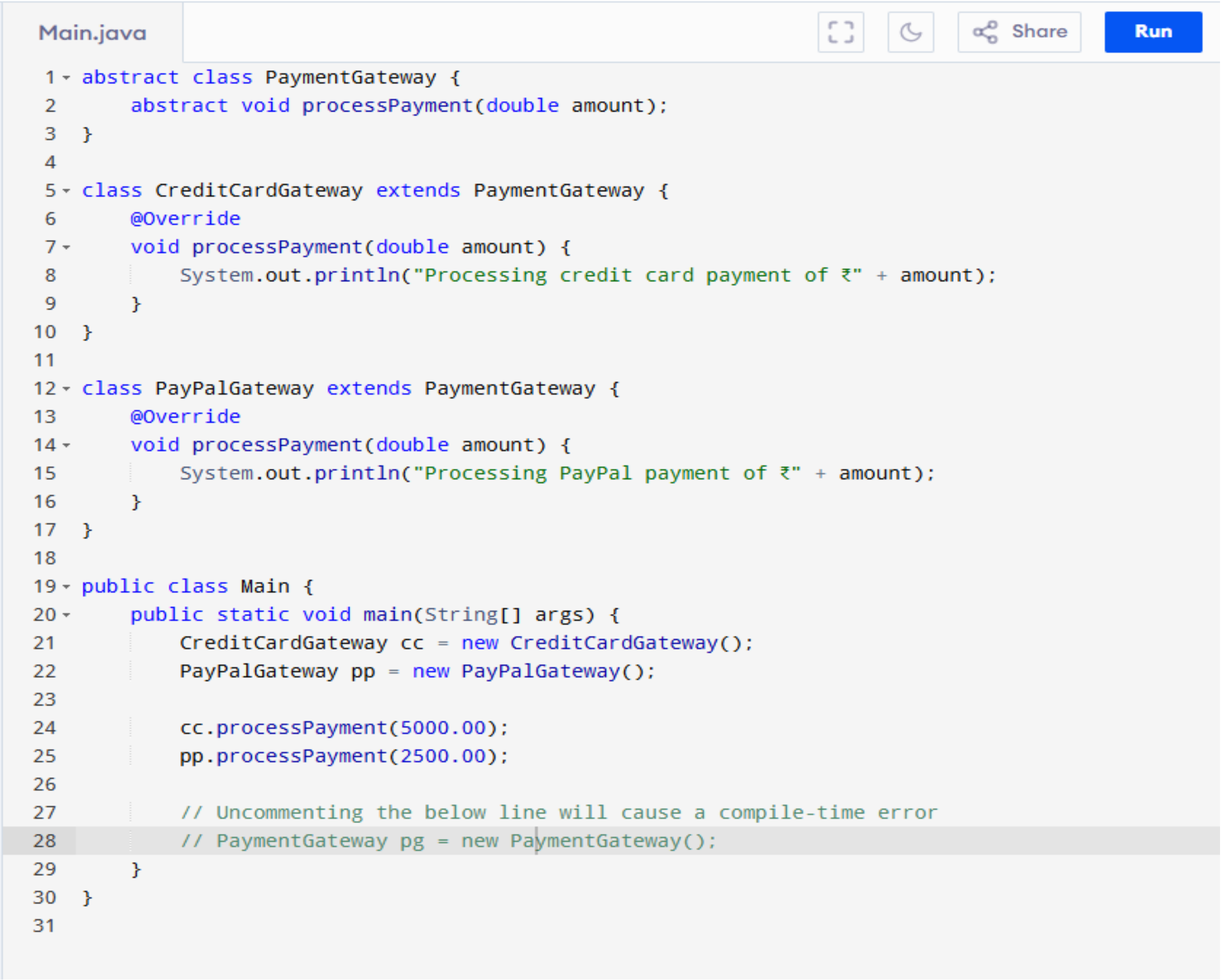
1. Create an abstract class PaymentGateway with processPayment(double amount).
2. Create subclass CreditCardGateway and implement processPayment().
3. Create subclass PayPalGateway and implement processPayment().
4. In main(), create objects of both subclasses and call their method.
5. Attempt to instantiate PaymentGateway (should cause a compilation error).

3. Pseudocode

```
Abstract Class PaymentGateway:
    Abstract method: processPayment(double amount)


Class CreditCardGateway inherits PaymentGateway:
    Implement processPayment()
```

```
Class PayPalGateway inherits PaymentGateway:
    Implement processPayment()


Main:
    Create CreditCardGateway object
    Call processPayment()


    Create PayPalGateway object
    Call processPayment()


    Try creating PaymentGateway object (should fail)
```

4. Program Code



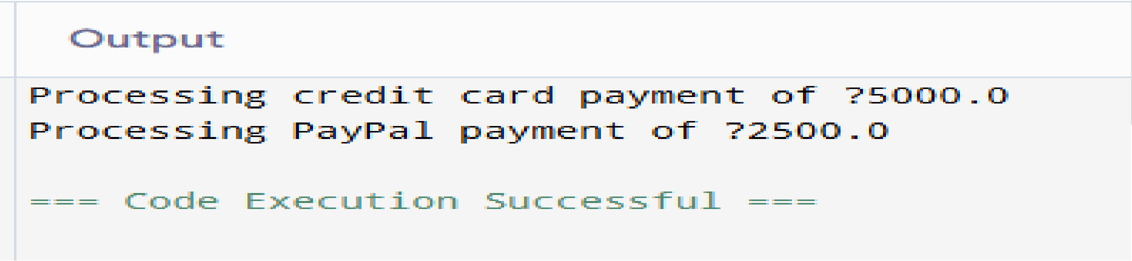
```java
abstract class PaymentGateway {
    abstract void processPayment(double amount);
}

class CreditCardGateway extends PaymentGateway {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing credit card payment of ₹" + amount);
    }
}

class PayPalGateway extends PaymentGateway {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing PayPal payment of ₹" + amount);
    }
}

public class Main {
    public static void main(String[] args) {
        CreditCardGateway cc = new CreditCardGateway();
        PayPalGateway pp = new PayPalGateway();

        cc.processPayment(5000.00);
        pp.processPayment(2500.00);

        // Uncommenting the below line will cause a compile-time error
        // PaymentGateway pg = new PaymentGateway();
    }
}
```

5. Test Cases

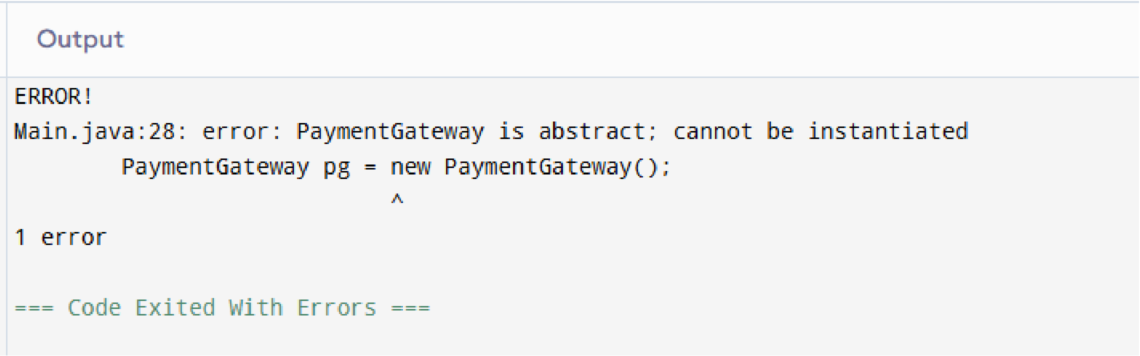
Present a table of test cases you used to validate your program. Include a mix of regular, boundary, and edge cases.

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	CreditCardGateway.process(5000)	Processing credit card payment of ₹5000.00	Processing credit card payment of ₹5000.00	Pass
2	PayPalGateway.process(2500)	Processing PayPal payment of ₹2500.00	Processing PayPal payment of ₹2500.00	Pass
3	PaymentGateway obj = new PaymentGateway();	Compilation error	Compilation error	Fail

6. Screenshots of Output



```
Processing credit card payment of ?5000.0
Processing PayPal payment of ?2500.0

=== Code Execution Successful ===
```



```
ERROR!
Main.java:28: error: PaymentGateway is abstract; cannot be instantiated
        PaymentGateway pg = new PaymentGateway();
                            ^
1 error

=== Code Exited With Errors ===
```

7. Observation / Reflection

1. **Challenge**: Remembering that abstract classes cannot be instantiated.
2. **Learning**: How to use abstraction for different payment behaviors.
3. **Improvement**: Add user input or support for multiple currencies.

Problem Solving Activity 2.2

1. Program Statement

Instrument Sounds

Abstract class: Instrument with abstract play() Subclasses: Guitar, Piano

Implement and test

2. Algorithm

1. Define abstract class Instrument with play().
2. Define Guitar subclass and implement play().
3. Define Piano subclass and implement play().
4. In main(), create one object of each subclass and call play().

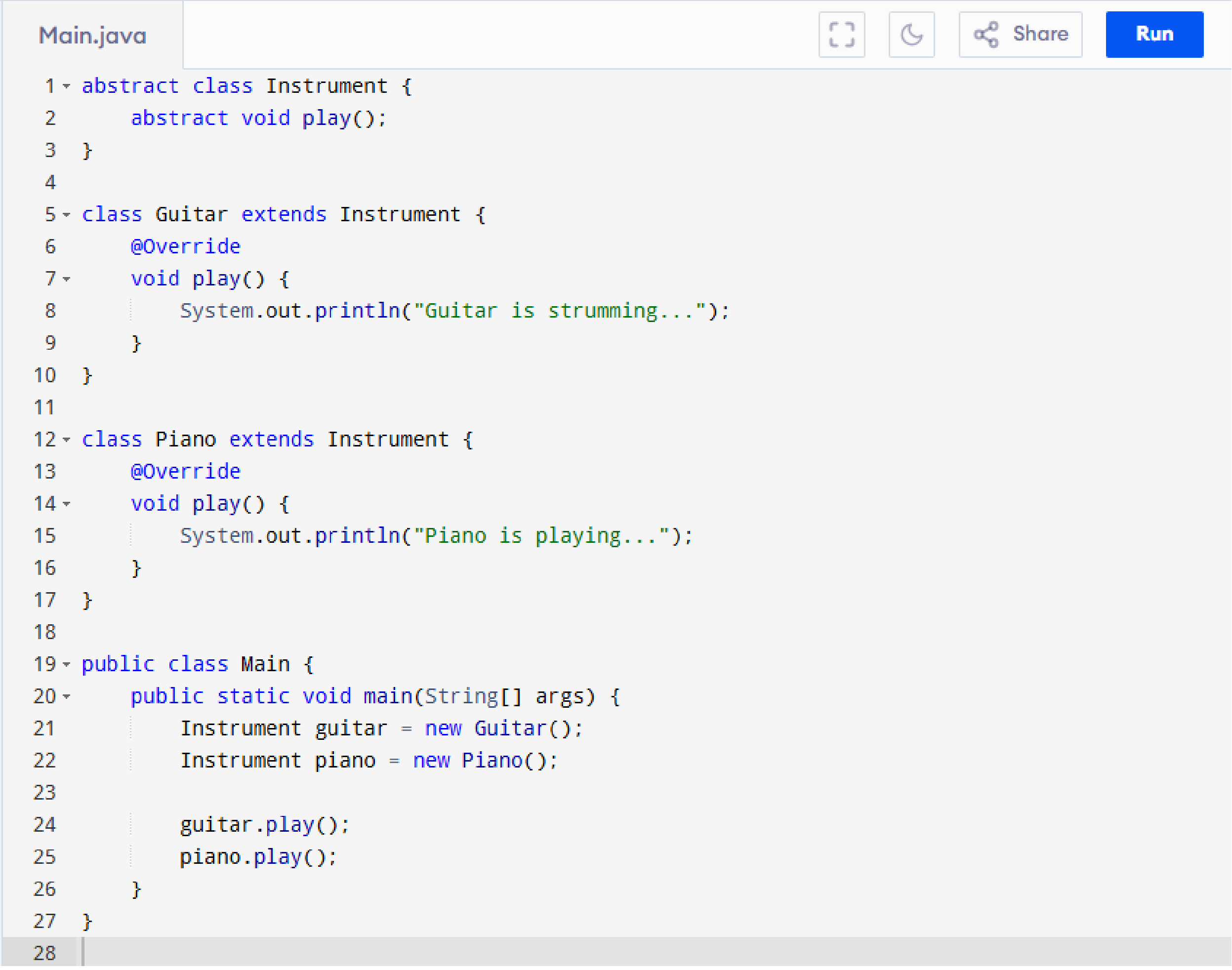
3. Pseudocode

```
Abstract Class Instrument:
   Abstract method: play()


Class Guitar extends Instrument:
   Implement play()


Class Piano extends Instrument:
   Implement play()


Main:
```

Create Guitar and Piano objects

Call play() on each

5. Program Code



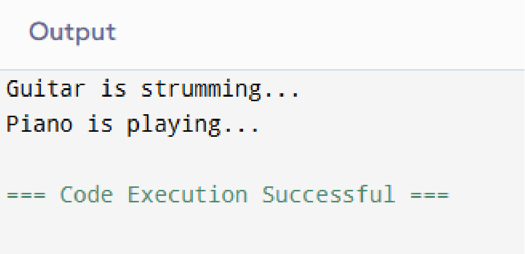
```java
abstract class Instrument {
    abstract void play();
}

class Guitar extends Instrument {
    @Override
    void play() {
        System.out.println("Guitar is strumming...");
    }
}

class Piano extends Instrument {
    @Override
    void play() {
        System.out.println("Piano is playing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Instrument guitar = new Guitar();
        Instrument piano = new Piano();

        guitar.play();
        piano.play();
    }
}

```

5. Test Cases

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	Guitar.play()	Guitar is strumming...	Guitar is strumming...	Pass

2	Piano.play()	Piano is playing...	Piano is playing...	Pass

6. Screenshots of Output



```
Guitar is strumming...
Piano is playing...

=== Code Execution Successful ===
```

7. Observation / Reflection

1. **Challenge**: None, conceptually straightforward.
2. **Learning**: How to implement abstract behavior for musical instruments.
3. **Improvement**: Add more instruments and a user menu to choose.

Problem Solving Activity 2.3

1. Program Statement

Activity 2.3: Abstracting a Task

Base: AutomatedTask, method execute() Subclasses: EmailSender, FileArchiver, DatabaseBackup Use abstraction to simplify the execution of tasks

2. Algorithm

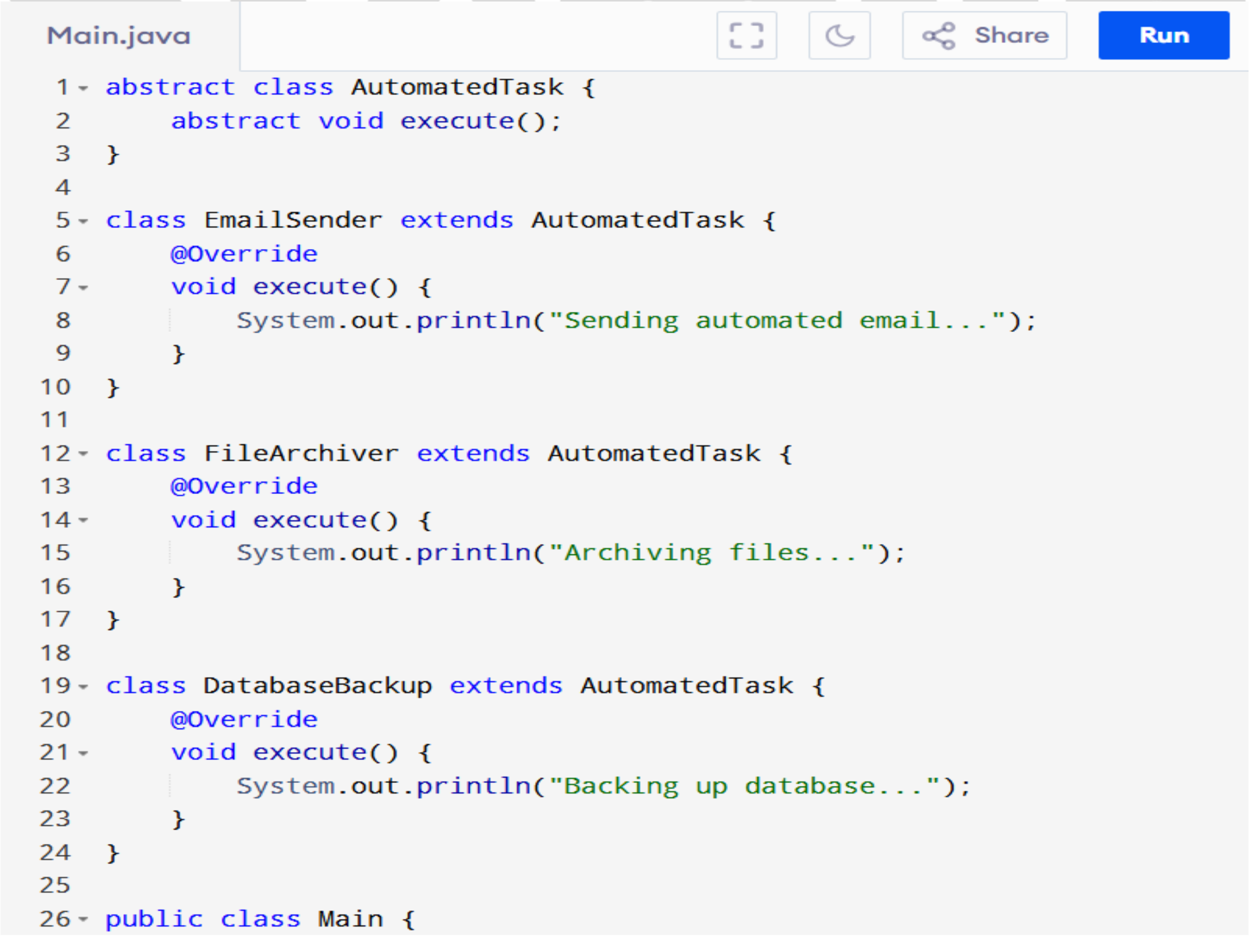
1. Create abstract class AutomatedTask with method execute().
2. Define 3 subclasses:
   - EmailSender: prints sending email
   - FileArchiver: prints archiving files
   - DatabaseBackup: prints backing up DB
3. Create an array of AutomatedTask references.

4. Loop through and call execute() on each object.

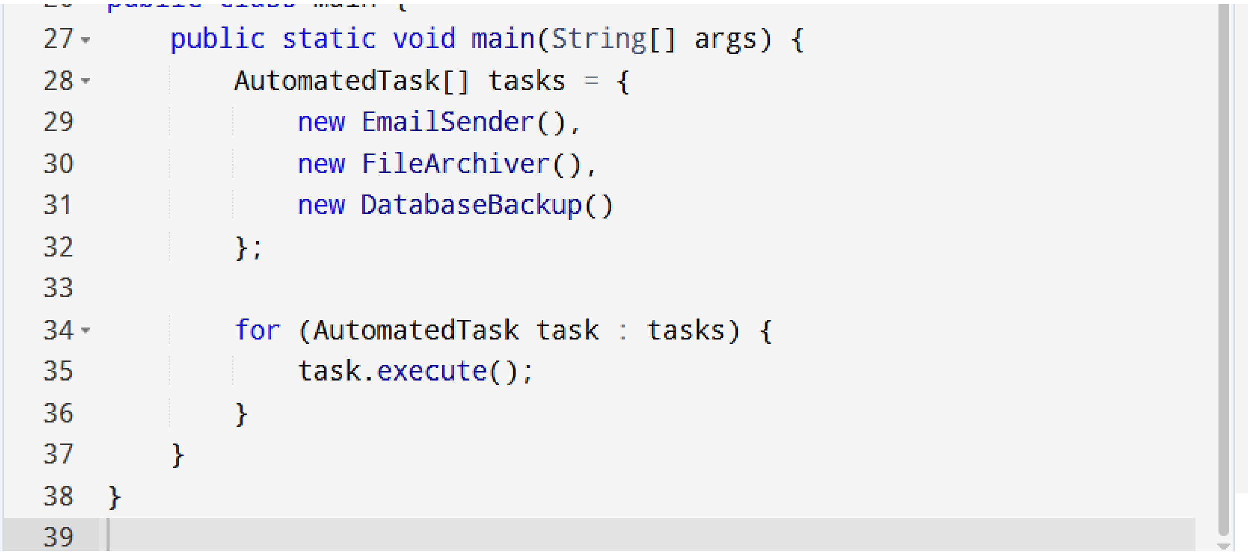
3. Pseudocode

```
Abstract Class AutomatedTask:
    Abstract method: execute()
Class EmailSender extends AutomatedTask:
    Implement execute()
Class FileArchiver extends AutomatedTask:
    Implement execute()
Class DatabaseBackup extends AutomatedTask:
    Implement execute()
Main:
    Create array of AutomatedTask objects
    Loop through and call execute()
```

4. Program Code



```java
abstract class AutomatedTask {
    abstract void execute();
}

class EmailSender extends AutomatedTask {
    @Override
    void execute() {
        System.out.println("Sending automated email...");
    }
}

class FileArchiver extends AutomatedTask {
    @Override
    void execute() {
        System.out.println("Archiving files...");
    }
}

class DatabaseBackup extends AutomatedTask {
    @Override
    void execute() {
        System.out.println("Backing up database...");
    }
}

public class Main {
```



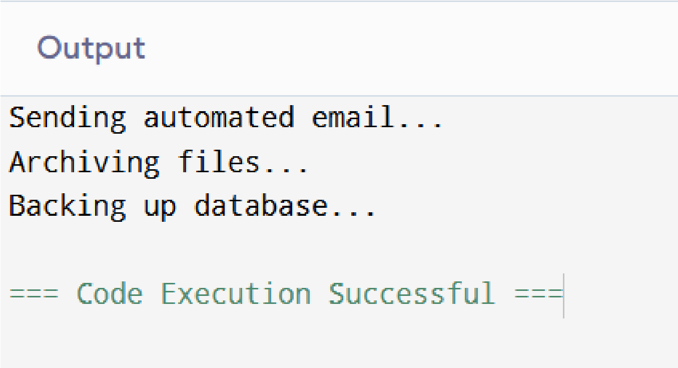
```java
public class Main {
    public static void main(String[] args) {
        AutomatedTask[] tasks = {
            new EmailSender(),
            new FileArchiver(),
            new DatabaseBackup()
        };

        for (AutomatedTask task : tasks) {
            task.execute();
        }
    }
}

```

5. Test Cases

Present a table of test cases you used to validate your program. Include a mix of regular, boundary, and edge cases.

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	EmailSender.execute()	Sending automated email...	Sending automated email...	Pass
2	FileArchiver.execute()	Archiving files...	Archiving files...	Pass
3	DatabaseBackup.execute()	Backing up database...	Backing up database...	Pass

6. Screenshots of Output



Output

```
Sending automated email...
Archiving files...
Backing up database...

=== Code Execution Successful ===
```

7. Observation / Reflection

1. **Challenge**: None; abstraction is applied effectively.
2. **Learning**: Efficient way to handle related tasks using polymorphism.
3. **Improvement**: Add user input for task scheduling or logs.

Problem Solving Activity 3.1

1. Program Statement

Employee Payroll

Base: Employee, abstract method calculatePayroll() Subclasses:

SalariedEmployee, HourlyEmployee Implement payroll logic and process

list of employees

Create an abstract class Employee with:

- Attributes: name, id
- Abstract method: calculatePayroll()

Subclasses:

- **SalariedEmployee**: with monthlySalary, overrides calculatePayroll()
- **HourlyEmployee**: with hourlyRate, hoursWorked, overrides calculatePayroll()

In main():

- Create a list of Employee references
- Add objects of both subclasses
- Use polymorphism to call calculatePayroll() on each object

2. Algorithm

1. Define abstract class Employee with attributes name and id, and abstract method calculatePayroll().

2. Create subclass SalariedEmployee with monthlySalary; override calculatePayroll() to return monthlySalary.
3. Create subclass HourlyEmployee with hourlyRate and hoursWorked; override calculatePayroll() to return hourlyRate * hoursWorked.
4. In main(), create an array of Employee references.
5. Add instances of SalariedEmployee and HourlyEmployee to the array.
6. Iterate through the array and call calculatePayroll() on each object.
7. Print the result.

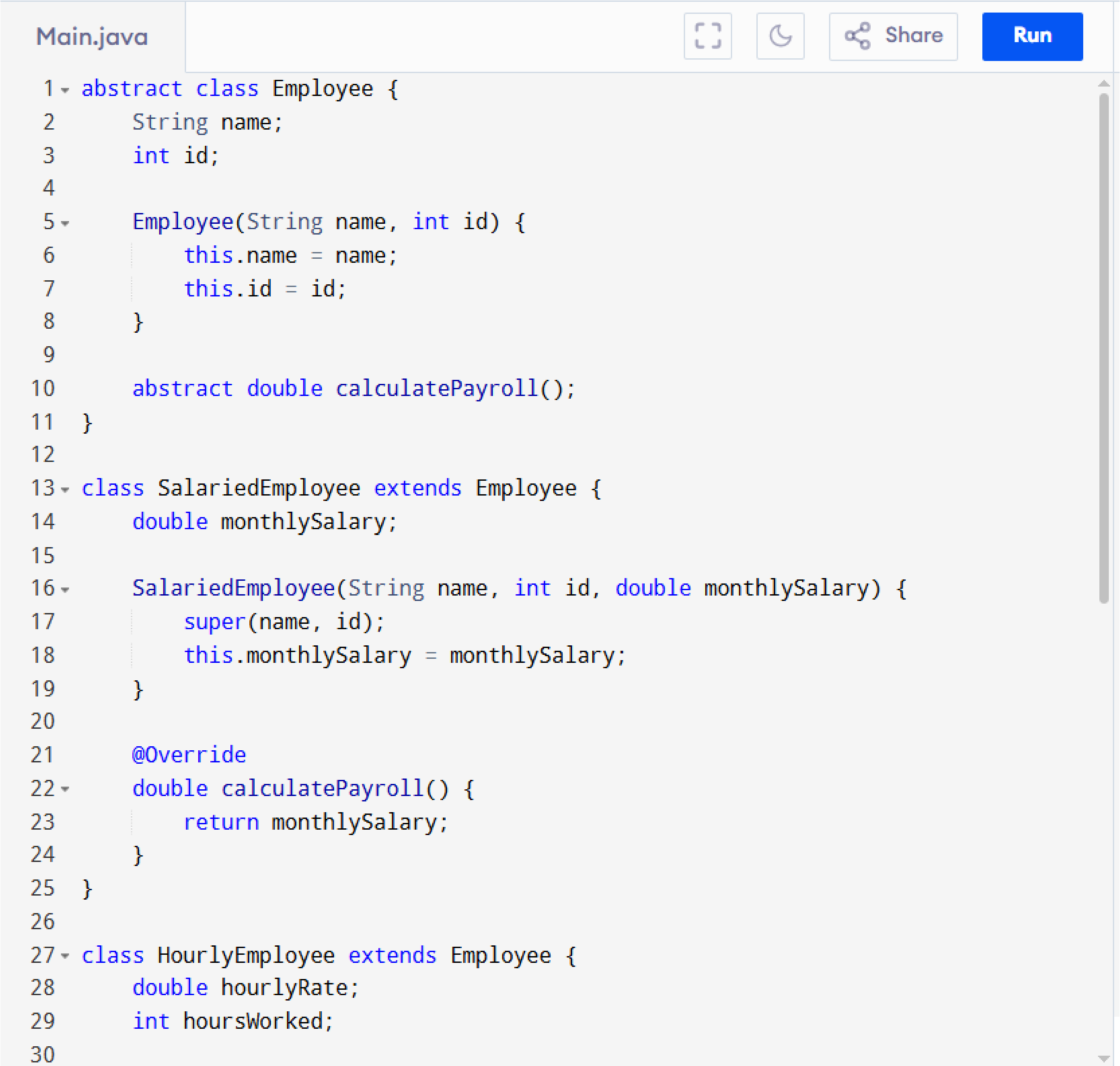
---

3. Pseudocode

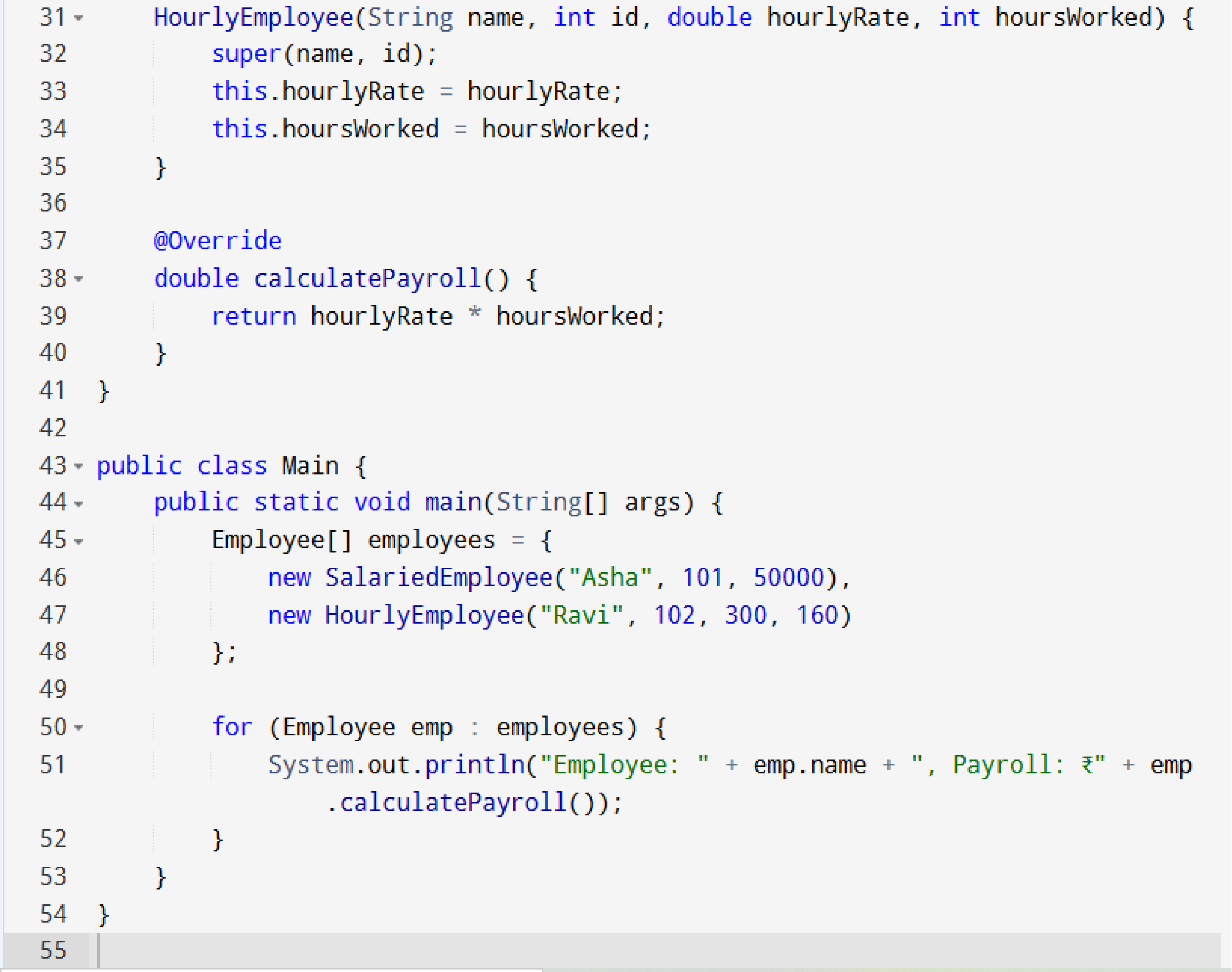
```
Abstract class Employee:
   Attributes: name, id
   Abstract method: calculatePayroll()

Class SalariedEmployee extends Employee:
   Attribute: monthlySalary
   Method calculatePayroll():
      return monthlySalary

Class HourlyEmployee extends Employee:
   Attributes: hourlyRate, hoursWorked
   Method calculatePayroll():
      return hourlyRate * hoursWorked

Main:
   Create array of Employee objects
   Add SalariedEmployee("Asha", 101, 50000)
   Add HourlyEmployee("Ravi", 102, 300, 160)
   For each Employee:
```

print name and calculatePayroll()

4. Program Code



```java
abstract class Employee {
    String name;
    int id;

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

    abstract double calculatePayroll();
}

class SalariedEmployee extends Employee {
    double monthlySalary;

    SalariedEmployee(String name, int id, double monthlySalary) {
        super(name, id);
        this.monthlySalary = monthlySalary;
    }

    @Override
    double calculatePayroll() {
        return monthlySalary;
    }
}

class HourlyEmployee extends Employee {
    double hourlyRate;
    int hoursWorked;

```



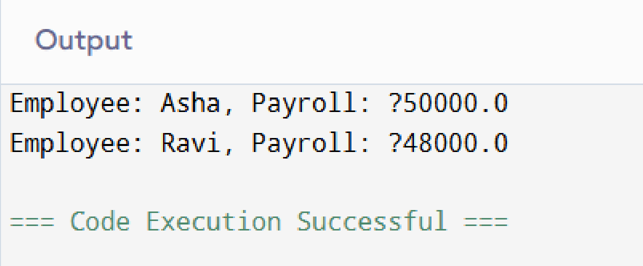
```java
    HourlyEmployee(String name, int id, double hourlyRate, int hoursWorked) {
        super(name, id);
        this.hourlyRate = hourlyRate;
        this.hoursWorked = hoursWorked;
    }

    @Override
    double calculatePayroll() {
        return hourlyRate * hoursWorked;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee[] employees = {
            new SalariedEmployee("Asha", 101, 50000),
            new HourlyEmployee("Ravi", 102, 300, 160)
        };

        for (Employee emp : employees) {
            System.out.println("Employee: " + emp.name + ", Payroll: ₹" + emp.calculatePayroll());
        }
    }
}

```

5. Test Cases

Present a table of test cases you used to validate your program. Include a mix of regular, boundary, and edge cases.

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	SalariedEmployee("Asha", 101, 50000)	Payroll: ₹50000.0	Payroll: ₹50000.0	Pass
2	HourlyEmployee("Ravi", 102, 300, 160)	Payroll: ₹48000.0 (300 × 160)	Payroll: ₹48000.0	Pass

6. Screenshots of Output



```
Employee: Asha, Payroll: ?50000.0
Employee: Ravi, Payroll: ?48000.0

=== Code Execution Successful ===
```

7. Observation / Reflection

1. **Challenges Faced:** Managing inheritance, constructor chaining, and abstract method implementation.
2. **What I Learned:** Clear understanding of abstract classes, polymorphism, and payroll logic through method overriding.
3. **Improvements for Future:** Add employee types like freelancer or consultant; also add input from user or file.

Problem Solving Activity 3.2

1. Program Statement

Problem 3.2: Geometric Shapes

Abstract base: Shape with getArea() Subclasses: Circle, Square Create

polymorphic list and calculate areas.

Create an abstract class Shape with:

- Abstract method: getArea()

Subclasses:

- Circle with radius as attribute, and overridden getArea() using formula $\pi \times r^2$
- Square with side as attribute, and overridden getArea() using formula side × side

In main():

- Create an array of Shape references (polymorphism)

- Add Circle and Square objects
- Call getArea() on each and display the result

2. Algorithm

Define an abstract class Shape with abstract method getArea().

Create subclass Circle:

- Attribute: radius
- Override getArea() to return π × radius^2

Create subclass Square:

- Attribute: side
- Override getArea() to return side × side

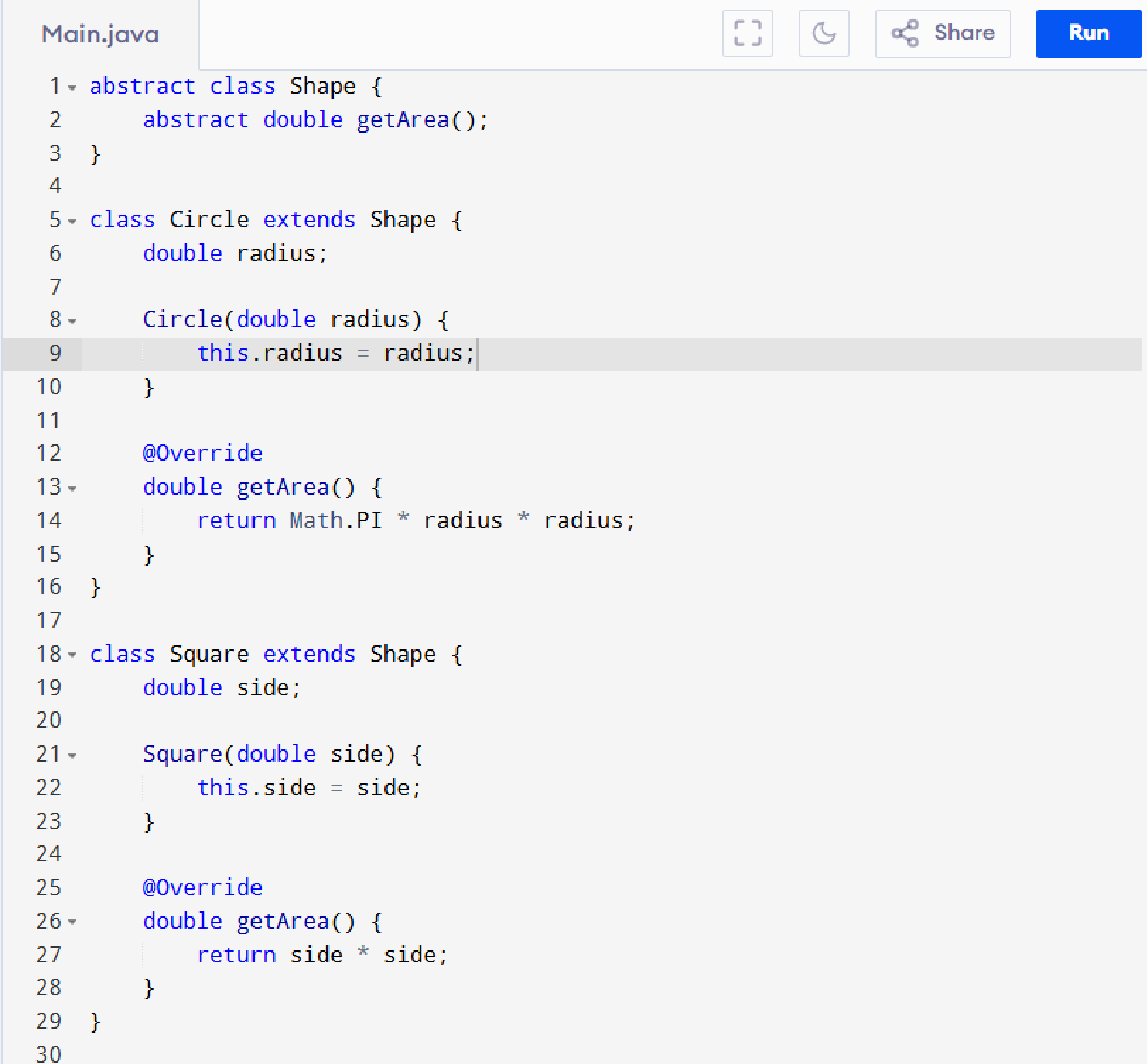
In the main method:

- Create an array of Shape references
- Add Circle and Square objects
- Loop through and call getArea() on each shape
- Print the result

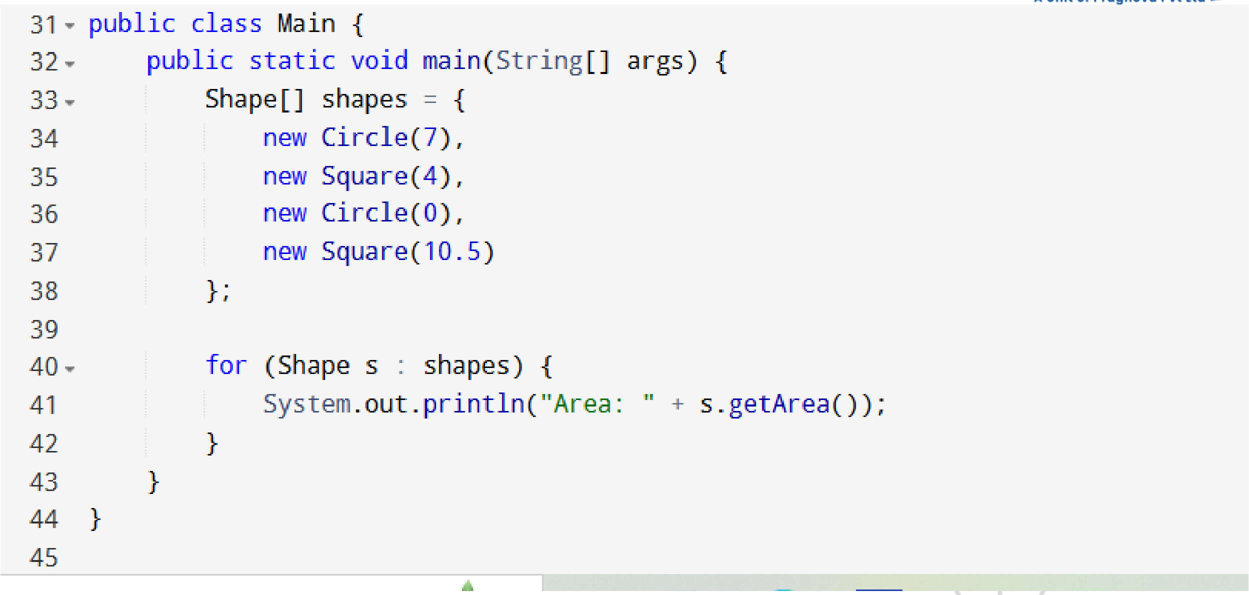
3. Pseudocode

```
Abstract class Shape:
    Abstract method getArea()


Class Circle extends Shape:
    Attribute: radius
    Method getArea():
        return PI * radius * radius


Class Square extends Shape:
    Attribute: side
    Method getArea():
        return side * side

Main:
    Create array of Shape
    Add Circle(7)
    Add Square(4)
    For each shape in array:
        Print shape.getArea().
```

4. Program Code



```java
abstract class Shape {
    abstract double getArea();
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double getArea() {
        return Math.PI * radius * radius;
    }
}

class Square extends Shape {
    double side;

    Square(double side) {
        this.side = side;
    }

    @Override
    double getArea() {
        return side * side;
    }
}

```



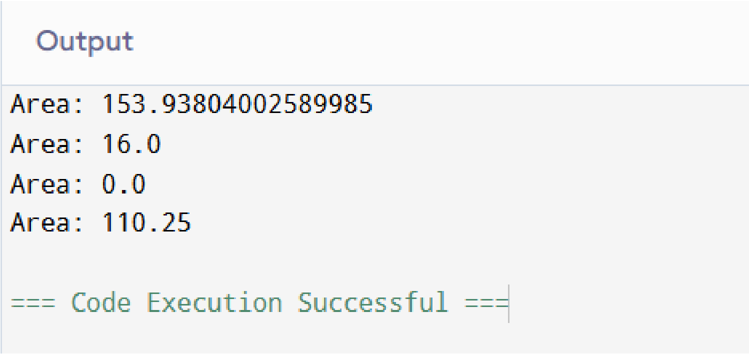
```java
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = {
            new Circle(7),
            new Square(4),
            new Circle(0),
            new Square(10.5)
        };

        for (Shape s : shapes) {
            System.out.println("Area: " + s.getArea());
        }
    }
}

```

5. Test Cases

Present a table of test cases you used to validate your program. Include a mix of regular, boundary, and edge cases.

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	Circle(7)	Area: 153.93804002589985	Area: 153.93804002589985	Pass
2	Square(4)	Area: 16.0	Area: 16.0	Pass
3	Circle(0)	Area: 0.0	Area: 0.0	Pass
4	Square(10.5)	Area: 110.25	Area: 110.25	Pass

6. Screenshots of Output



Output

```
Area: 153.93804002589985
Area: 16.0
Area: 0.0
Area: 110.25

=== Code Execution Successful ===
```

7. Observation / Reflection

1. **Challenges Faced**: Minor syntax issues while defining abstract methods and overriding them correctly.
2. **What I Learned**: Abstract classes help in defining a common template, while subclasses give specific implementations.
3. **Improvements**: Add more shape types like Rectangle, Triangle, and use dynamic input or GUI to display shapes.

Problem Solving Activity 3.3

1. Program Statement

Polymorphism in UI

Base: Tool, method draw() Subclasses: PenTool, EraserTool, LineTool

Demonstrate polymorphism using a collection

Create an abstract class UITask with an abstract method execute().
Subclasses:

- LoginTask: prints "Executing login task..."
- LogoutTask: prints "Executing logout task..."
- DashboardTask: prints "Displaying dashboard..."

In main():

- Create an array of UITask references
- Add objects of each subclass
- Loop through the array and call execute() on each

2. Algorithm

Define abstract class UITask with abstract method execute().

Create subclass LoginTask that overrides execute() to print "Executing login task..."

Create subclass LogoutTask that overrides execute() to print "Executing logout task..."

Create subclass DashboardTask that overrides execute() to print "Displaying dashboard..."

In the main() method:

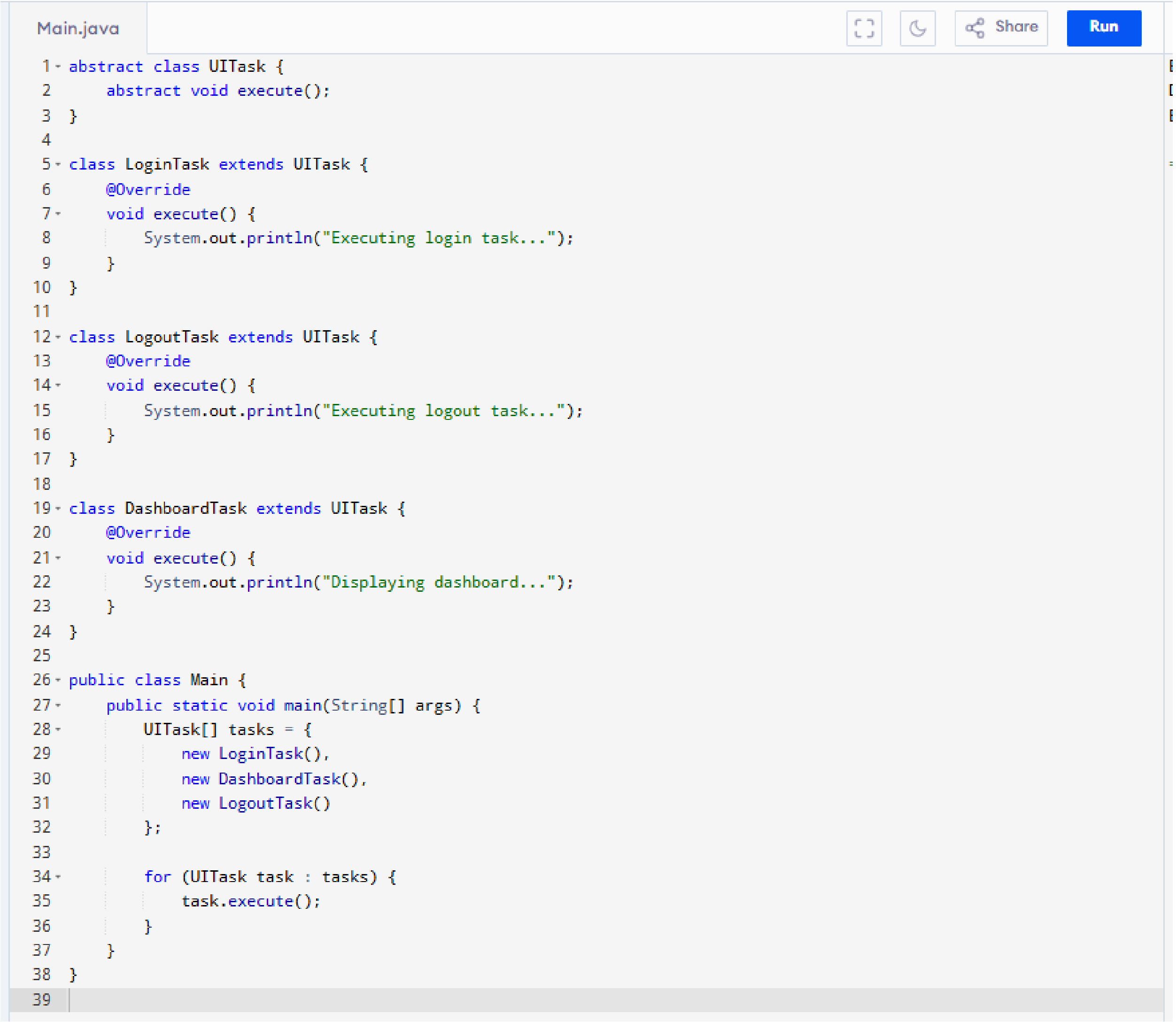
- Create an array of UITask references
- Add objects of LoginTask, LogoutTask, and DashboardTask
- Iterate through the array and call execute() on each object

3. Pseudocode

```
Abstract class UITask:
  Abstract method execute()

Class LoginTask extends UITask:
  Method execute():
    Print "Executing login task..."

Class LogoutTask extends UITask:
  Method execute():
    Print "Executing logout task..."

Class DashboardTask extends UITask:
  Method execute():
    Print "Displaying dashboard..."
```

```
Main:
    Create array of UITask:
        - LoginTask
        - LogoutTask
        - DashboardTask
    For each task in array:
        Call execute()
```

4. Program Code

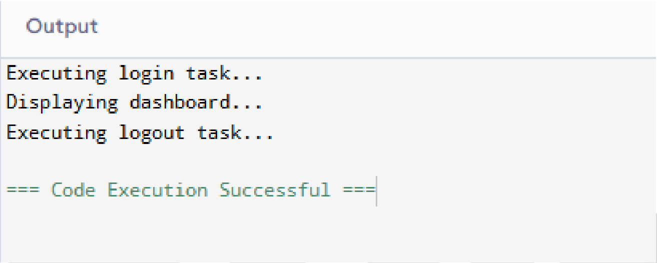


```java
abstract class UITask {
    abstract void execute();
}

class LoginTask extends UITask {
    @Override
    void execute() {
        System.out.println("Executing login task...");
    }
}

class LogoutTask extends UITask {
    @Override
    void execute() {
        System.out.println("Executing logout task...");
    }
}

class DashboardTask extends UITask {
    @Override
    void execute() {
        System.out.println("Displaying dashboard...");
    }
}

public class Main {
    public static void main(String[] args) {
        UITask[] tasks = {
            new LoginTask(),
            new DashboardTask(),
            new LogoutTask()
        };

        for (UITask task : tasks) {
            task.execute();
        }
    }
}
```

5. Test Cases

Test Case No.	Input	Expected Output	Actual Output	Status (Pass/Fail)
1	LoginTask	Executing login task...	Executing login task...	Pass
2	DashboardTask	Displaying dashboard...	Displaying dashboard...	Pass
3	LogoutTask	Executing logout task...	Executing logout task...	Pass

6. Screenshots of Output



```
Executing login task...
Displaying dashboard...
Executing logout task...

=== Code Execution Successful ===
```

7. Observation / Reflection

1. **Challenges Faced**: None; this problem was straightforward and helped reinforce abstract method execution.
2. **What I Learned**: How to use abstraction and polymorphism to simplify repeated task executions in UI systems.
3. **Improvements**: Future enhancements could include adding parameters to execute(), or making tasks dynamic based on user roles.