**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:

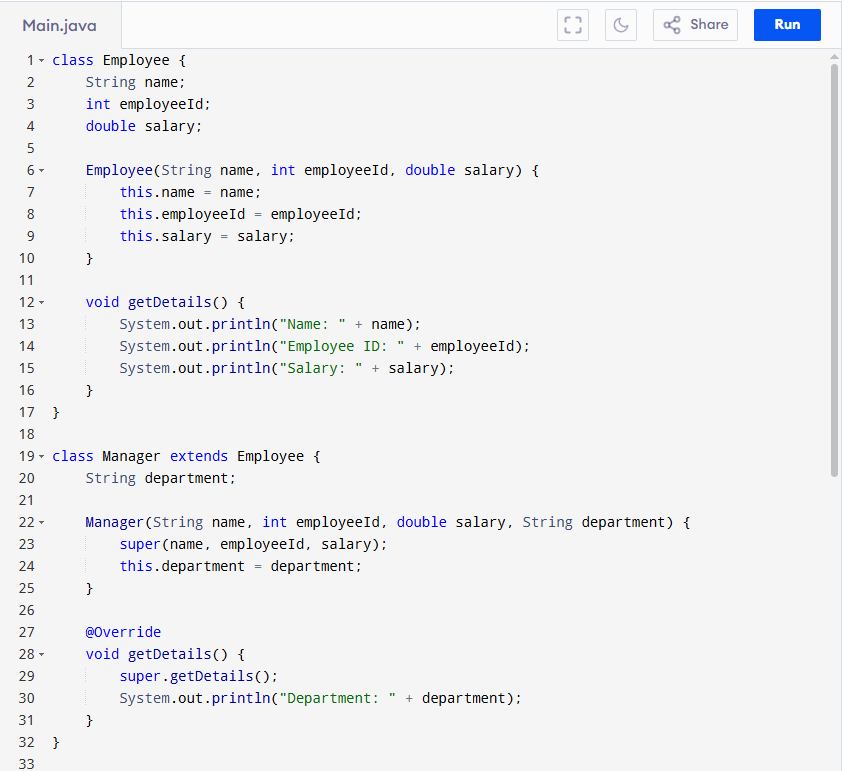
Create Manager object and set values

Call getDetails()

Create Developer object and set values

Call getDetails()

**4. Program Code**

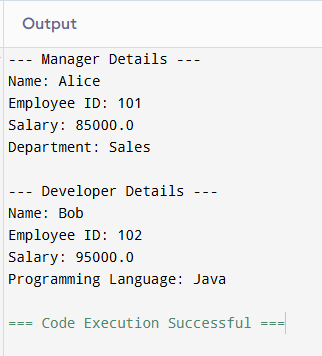


****

**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**

****

**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:

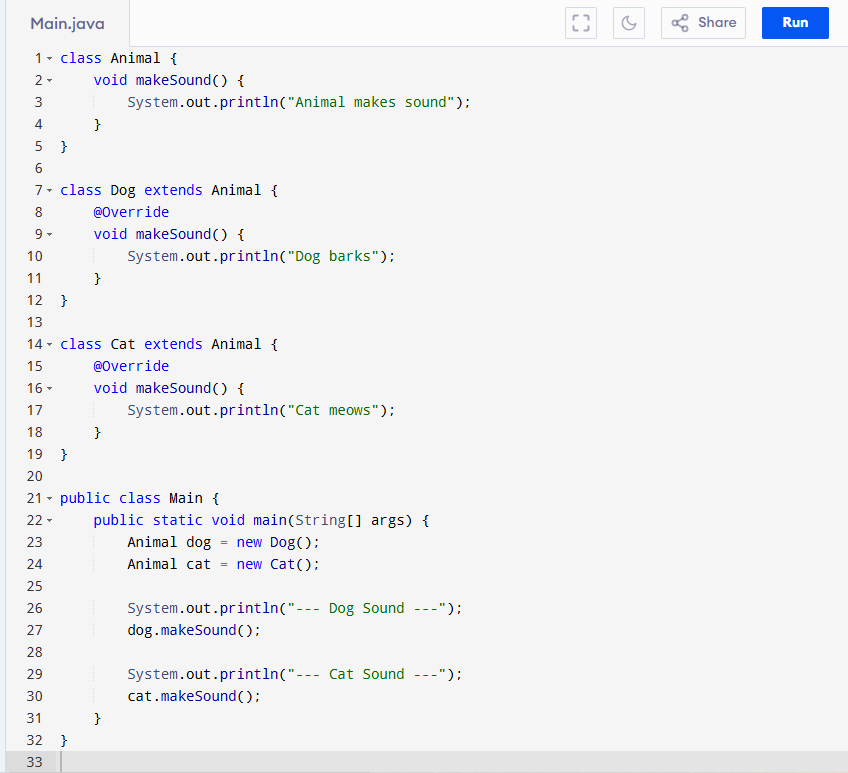
Create Dog object

Call makeSound()

Create Cat object

Call makeSound()

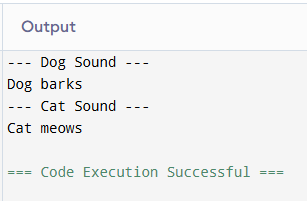
**4. Program Code**



**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 |

1. **Screenshots of Output**



**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

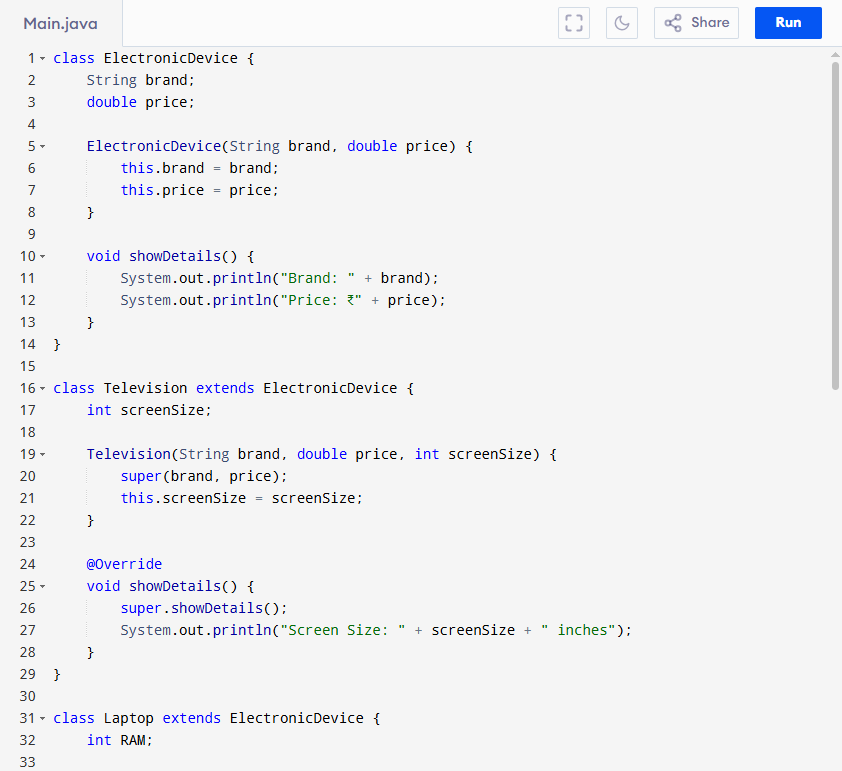
Main method:

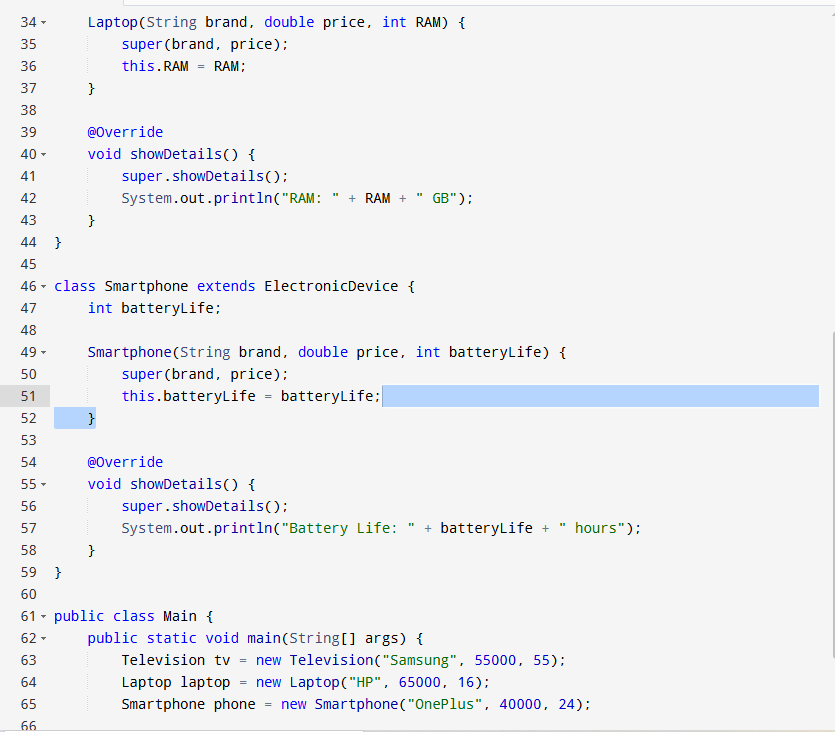
Create object of Television

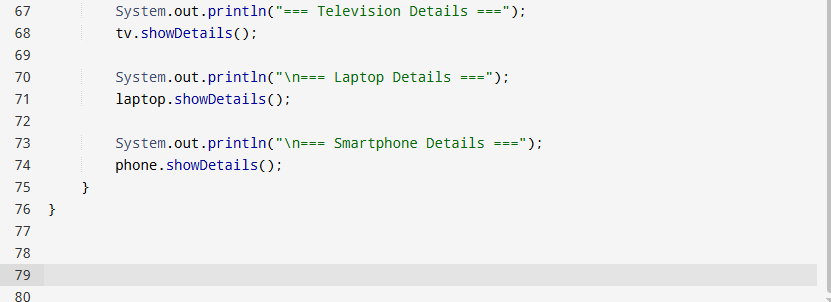
Create object of Laptop

Create object of Smartphone

Call showDetails() on each

1. **Program Code**



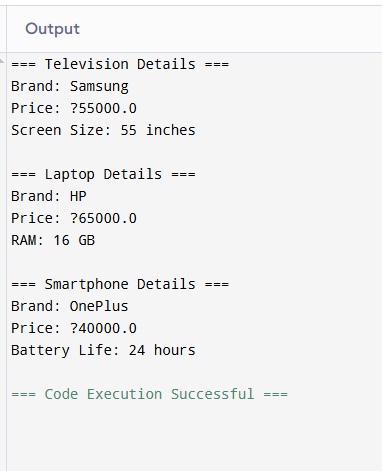


**5. Test Cases**

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**

****

**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:

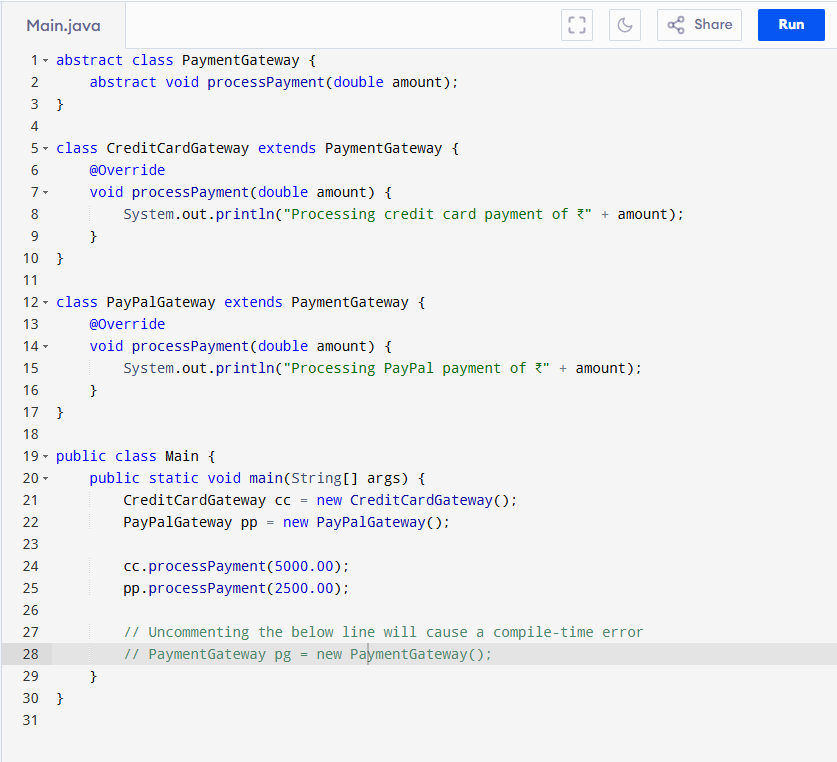
Create CreditCardGateway object

Call processPayment()

Create PayPalGateway object

Call processPayment()

Try creating PaymentGateway object (should fail)

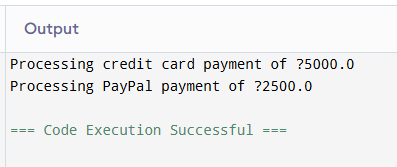
**4. Program Code**

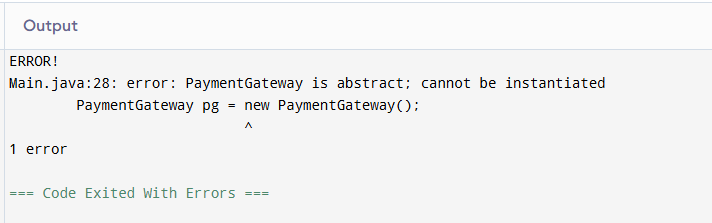
**5. Test Cases**

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**





**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()

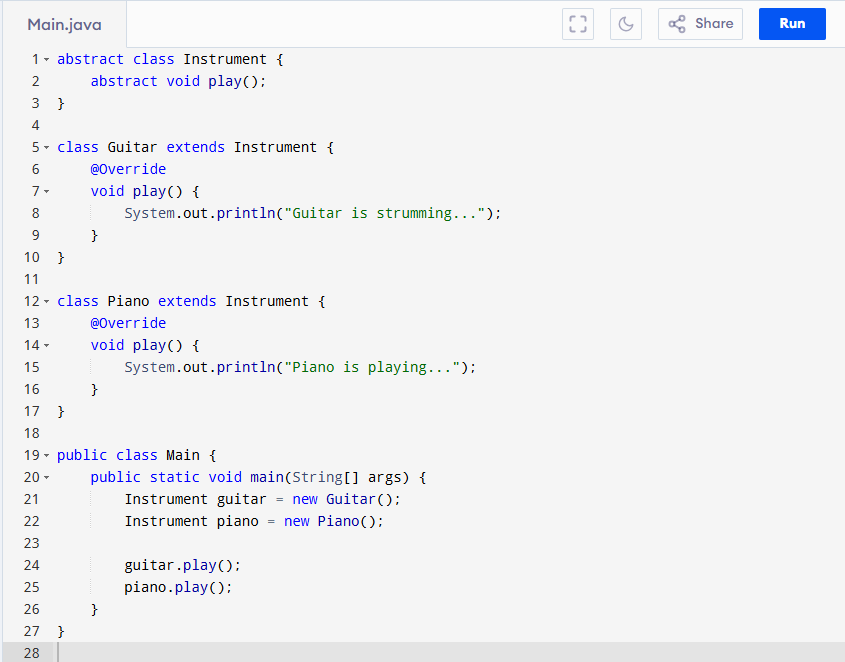
Class Piano extends Instrument:

Implement play()

Main:

Create Guitar and Piano objects

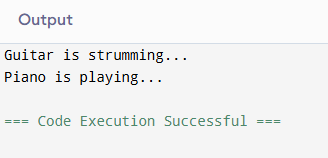
Call play() on each

1. **Program Code**

**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**



**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()

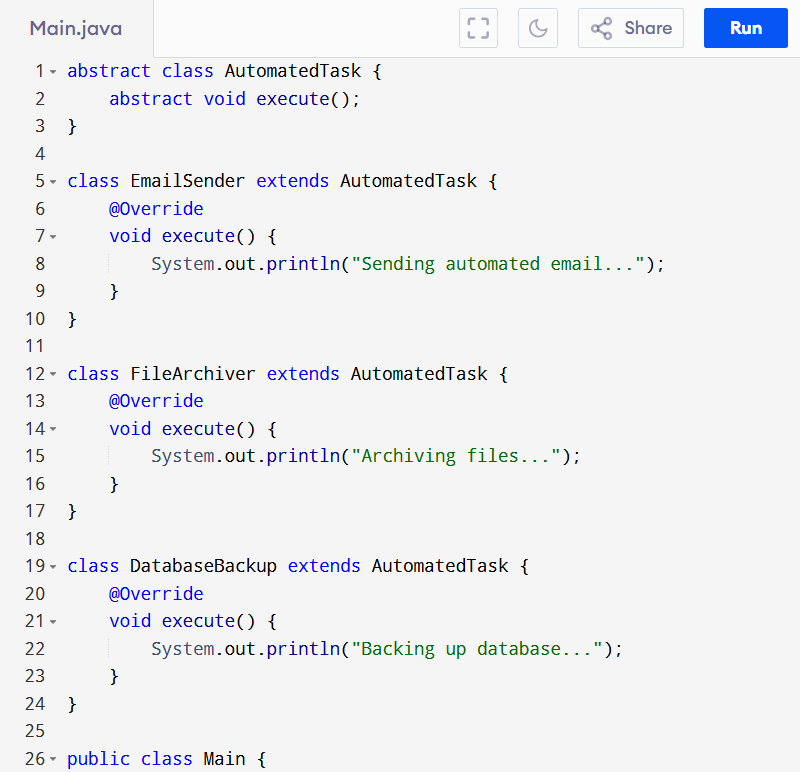
Class DatabaseBackup extends AutomatedTask:

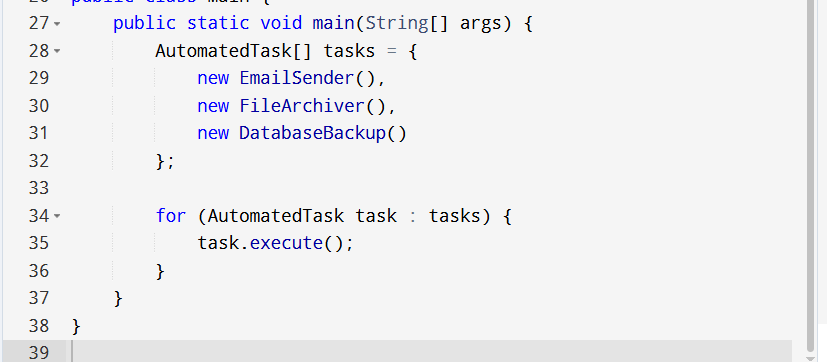
Implement execute()

Main:

Create array of AutomatedTask objects

Loop through and call execute()

**4. Program Code**

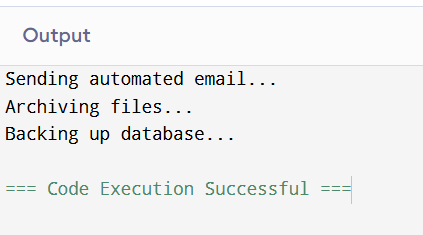
****

**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**



**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:

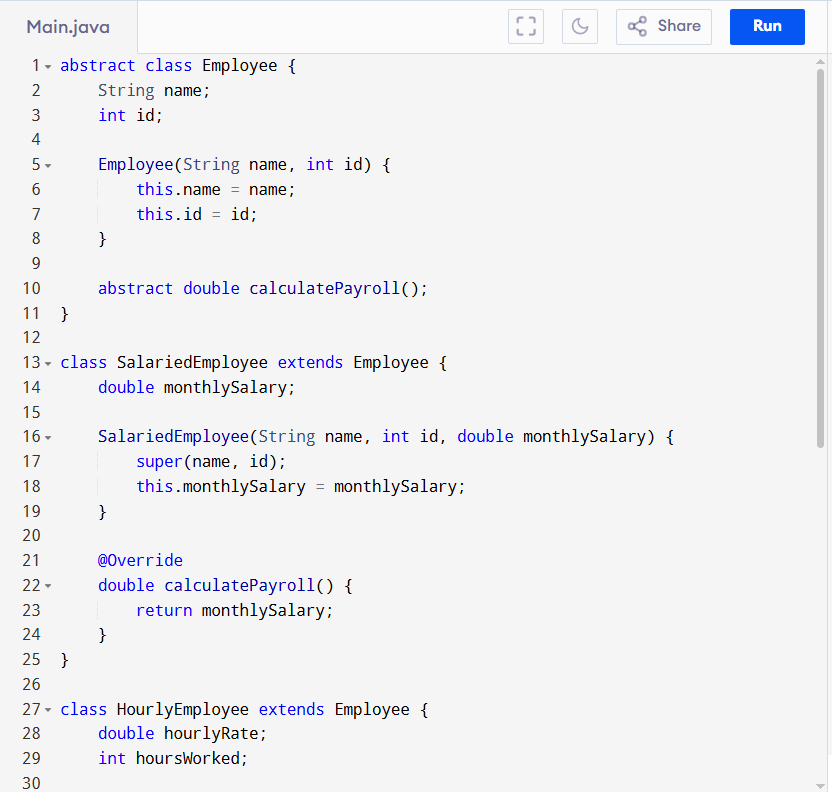
Create array of Employee objects

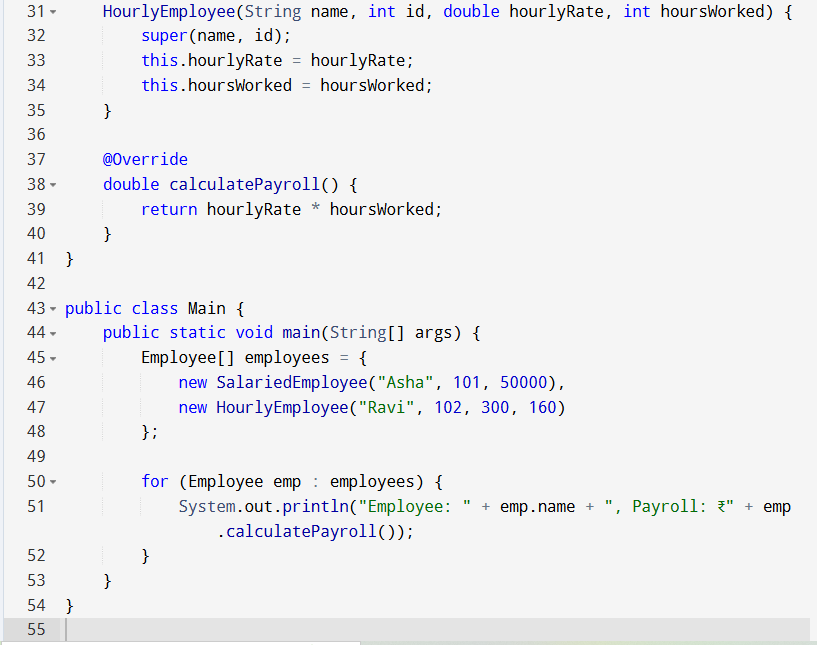
Add SalariedEmployee("Asha", 101, 50000)

Add HourlyEmployee("Ravi", 102, 300, 160)

For each Employee:

print name and calculatePayroll()

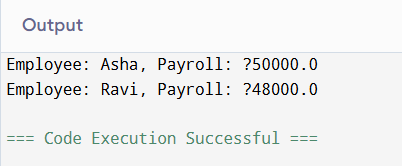
**4. Program Code**

****

**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**

**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 π × r²
* 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 × radius²

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:

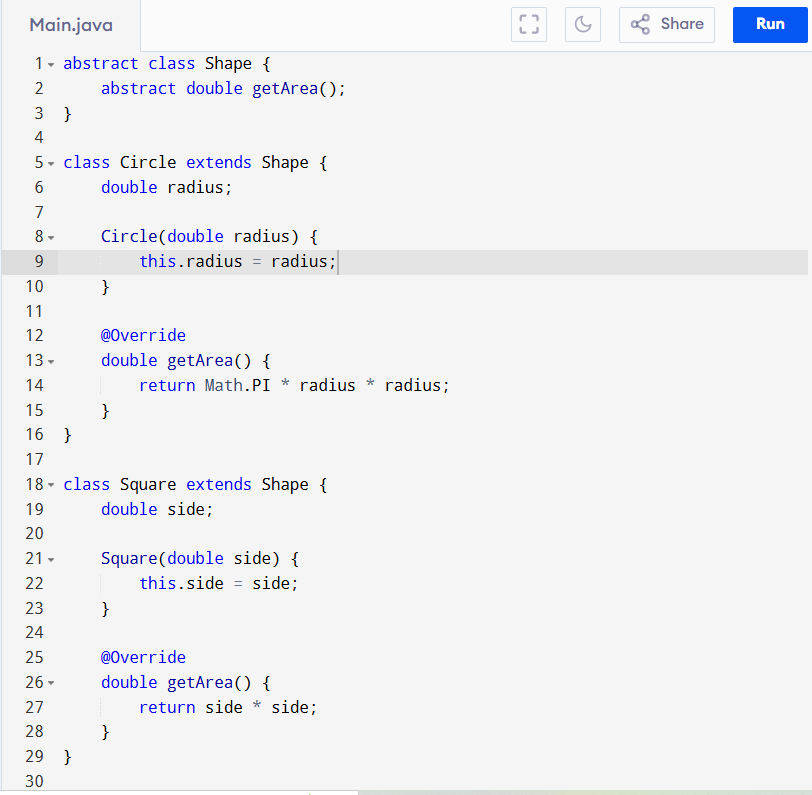
Create array of Shape

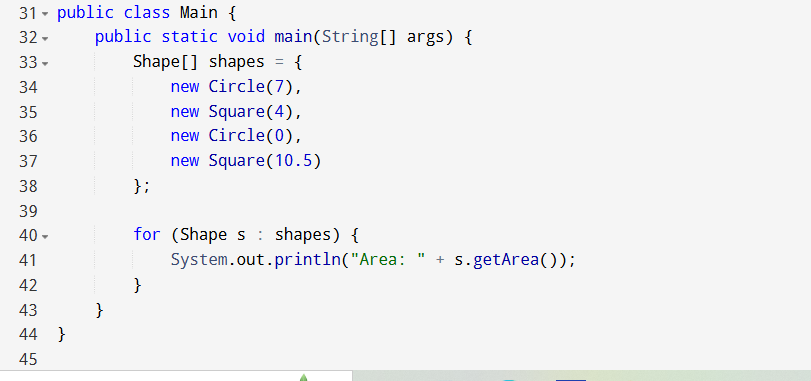
Add Circle(7)

Add Square(4)

For each shape in array:

Print shape.getArea().

**4. Program Code**

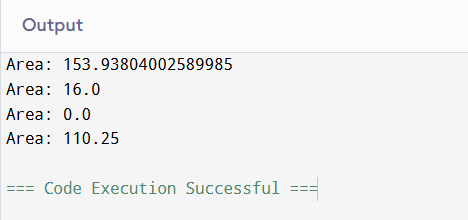
****

**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**

****

**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:

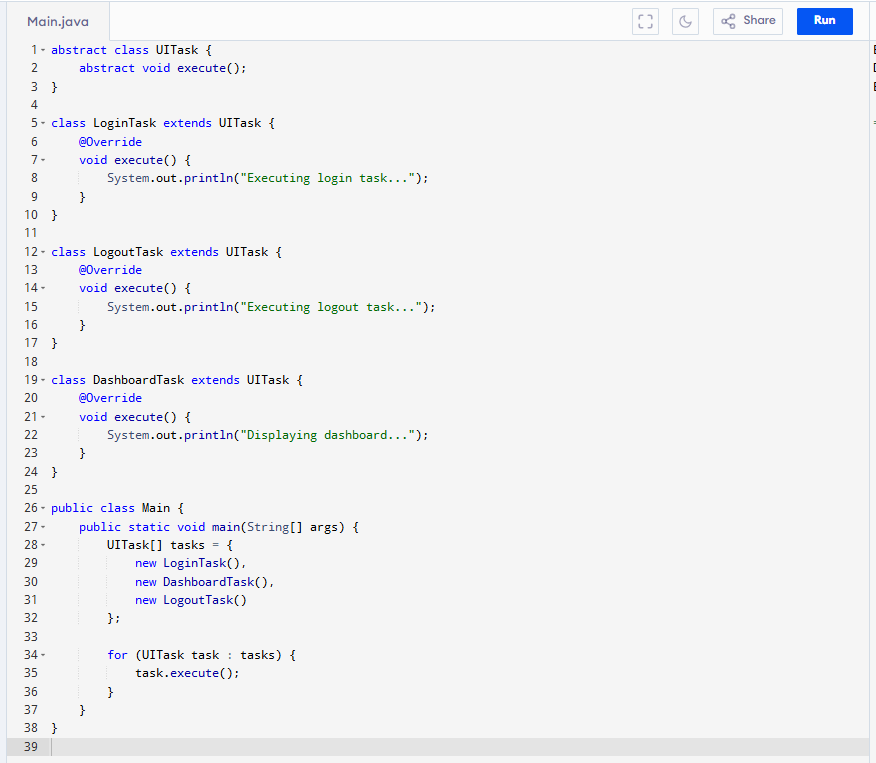
- LoginTask

- LogoutTask

- DashboardTask

For each task in array:

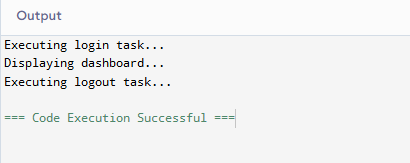
Call execute()

**4. Program Code**

**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**

****

**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.