MODULE: Java Programming

© 2026 Busycoder Academy. All rights reserved.

This assignment material is the intellectual property of Busycoder Academy and is provided strictly for classroom training conducted by the trainer.

This content is not intended for self-study distribution or public reuse.

ASSIGNMENT 2 – OOP, Classes, Inheritance, Polymorphism & Interfaces

Learning Objectives

By completing this assignment, learners will:

- Understand class design using attributes and behaviors
- Create objects and use constructors effectively
- Apply inheritance and method overriding
- Implement polymorphism through base-class references
- Model real-world systems using OOP principles
- Work with interfaces to support multiple behaviors
- Gain hands-on practice with UML interpretation

General Instructions

1. Each exercise should be implemented in **separate classes**, organized meaningfully.
2. Follow **Java naming conventions** for classes, methods, and variables.
3. Avoid code duplication — use inheritance or helper methods correctly.
4. Ensure all user-facing messages are clear and concise.
5. Validate user inputs where appropriate.
6. Demonstrate object behavior through a clean, readable **main()** test class.
7. Use encapsulation: all fields **private**, methods **public** unless otherwise needed.

Estimated Time & Difficulty

Question	Time	Difficulty
Q1	20–30 min	Beginner
Q2	60–90 min	Intermediate
Q3	90–120 min	Intermediate → Advanced
Q4	60–90 min	Intermediate
Q5	45–60 min	Intermediate

rguptatech@gmail.com

Evaluation Rubric

Criteria	Weight
Correctness & Functionality	50%
Class Design & OOP Principles	25%
Code Quality & Modularity	15%
Output Formatting & Clarity	10%

Q1. Implement UML for Circle Class & Test It

Requirements

1. Using the provided UML (circle class diagram), implement:
    - fields
    - constructors
    - methods (getters/setters or described methods)
2. Create a **test class** that:
    - creates Circle objects
    - displays radius, area, and circumference
    - tests multiple input values

Notes

- Use `Math.PI` for calculations.
- Validate radius must be positive.

Q2. Book Store Management (Objects, Arrays, Encapsulation)

Real-World Context

Simulate a small bookstore inventory system that can add, sell, and display books.

Part A: Book Class

Create a class **Book** with:

Attribute	Type
bookTitle	String
author	String
ISBN	String
numOfCopies	int

Requirements

- Parameterized constructor
- Method `display()` to print:

```
Title – Author – ISBN – Quantity
```

- Validate quantity must be non-negative.

rgupta.mtech@gmail.com

Part B: BookStore Class

BookStore manages an array of **maximum 10 books**.

Required Methods

1. sell(String title, int count)

- Search by **bookTitle**.
- If found → decrease copies.
- If not found → display "Book not found."
- If insufficient copies → display "Not enough stock."

2. order(String isbn, int count)

- Search by **ISBN**.
- If book exists → increase quantity.
- If not found → **create a new Book object** (ask user for title & author).
- Add the new Book if space is available.

3. display()

- Print all books using Book's `display()` method.

Part C: BookStoreApp (main class)

Demonstrate the following:

- Add sample books
- Order new & existing books
- Sell books
- Display inventory after each operation

Q3. Banking System Using Inheritance & Overriding

Real-World Context

Simulate behavior of different bank account types with customized rules.

Base Class: Account

Attributes:

- `name`
- `accountNumber`
- `accountBalance`

Methods:

- `deposit(double amount)` → increases balance
- `withdraw(double amount)` → to be overridden

rgupta.mtech@gmail.com

- Getters/setters as required

SavingsAccount (extends Account)

Additional attributes:

- `interest = 5` (fixed 5%)
- `maxWithdrawLimit = accountBalance`
- Minimum balance allowed: **₹5000**

Required Methods

getBalance()

- Must calculate:

```
return accountBalance + (accountBalance * interest / 100)
```

- Must NOT change `accountBalance`.

withdraw(double amount)

Allow withdrawal only if:

- amount ≤ maxWithdrawLimit
- balance after withdrawal ≥ **5000**
  Else print meaningful error messages.

CurrentAccount (extends Account)

Additional attributes:

- `tradeLicenseNumber`
- `overdraft` (allow negative balance until overdraft limit)

Required Methods

getBalance()

Return the current balance without interest.

withdraw(double amount)

Allow if:

```
amount <= accountBalance + overdraft
```

Else display “Withdrawal exceeds overdraft limit.”

Test Class (BankApp)

Demonstrate:

- Creating Savings and Current accounts
- Deposits & withdrawals
- Balance checks
- Overdraft behavior

rgupta.mtech@gmail.com

- Interest behavior

Q4. Employee Payment System Using Polymorphism

Real-world Context

A company needs to compute weekly salary for different types of employees.

Employee Types

1. SalariedEmployee

- Paid a **fixed weekly salary**.

2. HourlyEmployee

- Paid:

```
hoursWorked * hourlyRate
```

3. CommissionEmployee

- Paid:

```
(commissionRate% * totalSales) / 100
```

Requirements

Part A: Create an abstract or base `Employee` class

Include:

- Common attributes: name, employeeId
- Abstract `getPayment()` or overridden method in subclasses

Part B: Implement Each Employee Type

Each must override `getPayment()`.

Part C: Use ArrayList<Employee>

- Add at least one of each employee type
- Loop and print weekly salary using polymorphism

Q5. Unified Payment System Using Interface (Payable)

Real-world Context

Companies often need to pay two unrelated entities:

- Employees
- Vendors (through Invoice)

Both should be processed uniformly.

Interface: Payable

Define:

```
double getPayment();
```

rgupta.mtech@gmail.com

Invoice Class

Attributes:

- invoiceId
- itemDescription
- quantity
- pricePerUnit

Payment:

`quantity * pricePerUnit`

Update Employee Classes

- Implement Payable interface
- Ensure getPayment() returns correct values

Application Class (PaymentApp)

Demonstrate:

- Create objects: invoices + different employees
- Store them in a single `ArrayList<Payable>`
- Loop through and call `getPayment()`
- Display payments uniformly

BONUS CHALLENGES (Optional)

Bonus 1

Rebuild BookStore using **ArrayList<Book>** instead of array.

Bonus 2

Add search methods:

```
searchByAuthor()
searchByISBN()
```

Bonus 3

Add transaction logs to Bank System (deposit/withdraw history).

Bonus 4

Print Employee payment summary sorted by highest salary.

Reflection Questions

1. How does inheritance reduce code duplication in the banking system?
2. Why is polymorphism essential for the payment system?
3. What real-world business rules did you model in the BookStore?

rgupta.mtech@gmail.com

4. How do interfaces help unify unrelated objects?
5. Which OOP principle did you struggle with the most and why?

rgupta.mtech@gmail.com