Encapsulation

1.Student with Grade Validation & Configuration Ensure marks are always valid and immutable once set.

•Create a Student class with private fields: name, rollNumber, and marks

. •Use a constructor to initialize all values and enforce marks to be between 0 and 100; invalid values reset to 0.

•Provide getter methods, but no setter for marks (immutable after object creation).

•Add displayDetails() to print all fields. In future versions, you might allow updating marks only via a special inputMarks(int newMarks) method that has stricter logic (e.g. cannot reduce marks). Design accordingly

ANSWER

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

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

      if (marks >= 0 && marks <= 100) {
         this.marks = marks;
      } else {
         System.out.println("Invalid marks for " + name + ". Setting default = 0.");
         this.marks = 0;
      }
   }
   public String getName() {
      return name;
```

```
    }

    public int getRollNumber() {
        return rollNumber;
    }

    public int getMarks() {
        return marks; // immutable by default
    }
    public void inputMarks(int newMarks) {
        if (newMarks >= marks && newMarks <= 100) {
            this.marks = newMarks;
            System.out.println("Marks updated for " + name + " → " + newMarks);
        } else {
            System.out.println("Invalid update! Marks unchanged for " + name);
        }
    }
    public void displayDetails() {
        System.out.println(" Student Details:");
        System.out.println("   Name " + name);
        System.out.println("   Roll No  : " + rollNumber);
        System.out.println("   Marks     " + marks);
        System.out.println("-");
    }
}
```

2. Rectangle Enforced Positive Dimensions Encapsulate validation and provide derived calculations.

- Build a Rectangle class with private width and height

. •Constructor and setters should reject or correct non-positive values (e.g., use default or throw an exception).

•Provide getArea() and getPerimeter() methods.

•Include displayDetails() method.

ANSWER

```
class Rectangle {
   private double width;
   private double height;

   public Rectangle(double width, double height) {
      setWidth(width);
      setHeight(height);
   }

   public void setWidth(double width) {
      this.width = (width > 0) ? width : 1;
   }

   public void setHeight(double height) {
      this.height = (height > 0) ? height : 1;
   }

   public double getArea() { return width * height; }
   public double getPerimeter() { return 2 * (width + height); }

   public void displayDetails() {
      System.out.println("Width: " + width + ", Height: " + height +
            ", Area: " + getArea() + ", Perimeter: " + getPerimeter());
```

```
    }
}
```

3. Advanced: Bank Account with Deposit/Withdraw Logic Transaction validation and encapsulation protection.

- Create a BankAccount class with private accountNumber, accountHolder, balance.
- Provide: odeposit(double amount) — ignores or rejects negative. owithdraw(double amount) — prevents overdraft and returns a boolean success. oGetter for balance but no setter.
- Optionally override toString() to display masked account number and details.
- Track transaction history internally using a private list (or inner class for transaction object).
- Expose a method getLastTransaction() but do not expose the full internal list.

ANSWER

```
import java.util.ArrayList;
import java.util.List;

class BankAccount {
    private String accountNumber;
    private String accountHolder;
    private double balance;
    private List<String> transactionHistory = new ArrayList<>();

    public BankAccount(String accountNumber, String accountHolder, double balance) {
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        this.balance = balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
```

```
        balance += amount;
        transactionHistory.add("Deposited: " + amount);
    }
  }

  public boolean withdraw(double amount) {
    if (amount > 0 && balance >= amount) {
        balance -= amount;
        transactionHistory.add("Withdrawn: " + amount);
        return true;
    }
    return false;
  }

  public double getBalance() { return balance; }

  public String getLastTransaction() {
    return transactionHistory.isEmpty() ? "No transactions yet" :
          transactionHistory.get(transactionHistory.size() - 1);
  }

  @Override
  public String toString() {
    return "Account[****" + accountNumber.substring(accountNumber.length() - 4) +
          ", Holder=" + accountHolder + ", Balance=" + balance + "]";
  }
}
```

4. Inner Class Encapsulation: Secure Locker Encapsulate helper logic inside the class

. •Implement a class Locker with private fields such as lockerId, isLocked, and passcode.

•Use an inner private class SecurityManager to handle passcode verification logic.

•Only expose public methods: lock(), unlock(String code), isLocked().

•Password attempts should not leak verification logic externally—only success/failure.

•Ensure no direct access to passcode or the inner SecurityManager from outside.

ANSWER

```
class Locker {
   private String lockerId;
   private boolean isLocked;
   private String passcode;

   public Locker(String lockerId, String passcode) {
      this.lockerId = lockerId;
      this.passcode = passcode;
      this.isLocked = true;
   }

   private class SecurityManager {
      boolean verify(String code) {
         return passcode.equals(code);
      }
   }

   public void lock() { isLocked = true; }
   public boolean unlock(String code) {
      SecurityManager sm = new SecurityManager();
      if (sm.verify(code)) {
```

```
        isLocked = false;
        return true;
      }
      return false;
    }
    public boolean isLocked() { return isLocked; }
}
```

5. Builder Pattern & Encapsulation: Immutable Product Use Builder design to create immutable class with encapsulation.

- Create an immutable Product class with private final fields such as name, code, price, and optional category.
- Use a static nested Builder inside the Product class. Provide methods like withName(), withPrice(), etc., that apply validation (e.g. non-negative price).
- The outer class should have only getter methods, no setters.
- The builder returns a new Product instance only when all validations succeed.

ANSWER

```
class Product {
    private final String name;
    private final String code;
    private final double price;
    private final String category;

    private Product(Builder builder) {
        this.name = builder.name;
        this.code = builder.code;
        this.price = builder.price;
        this.category = builder.category;
    }
```

```
    public String getName() { return name; }
    public String getCode() { return code; }
    public double getPrice() { return price; }
    public String getCategory() { return category; }

    public static class Builder {
        private String name;
        private String code;
        private double price;
        private String category;

        public Builder withName(String name) { this.name = name; return this; }
        public Builder withCode(String code) { this.code = code; return this; }
        public Builder withPrice(double price) {
            if (price < 0) throw new IllegalArgumentException("Price cannot be negative");
            this.price = price; return this;
        }
        public Builder withCategory(String category) { this.category = category; return this; }

        public Product build() { return new Product(this); }
    }
}
```