1. **Create a class with properties and a method to display details**

```
 class Student {
int rollNumber;
String name;
String course;

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

public void displayDetails() {
        System.out.println("Student Details:");
        System.out.println("Roll Number: " + rollNumber);
        System.out.println("Name: " + name);
        System.out.println("Course: " + course);
    }
}

class StudentTest {
    public static void main(String[] args) {

            Student s1 = new Student(101, "Anjali", "Computer Science");
            s1.displayDetails();
    }
}
```

2. **Demonstrate class instantiation and method invocation.**

```
class Car {
    String brand;
    String color;

public Car(String brand, String color) {
this.brand = brand;
this.color = color;
    }
 public void displayInfo() {
System.out.println("Brand: " + brand);
        System.out.println("Color: " + color);
    }

    // Method to start the car
public void start() {
        System.out.println(brand + " is starting...");
    }
}
```

```
// Main class to demonstrate public
class CarDemo {
   public static void main(String[] args) {
      // Class instantiation
      Car myCar = new Car("Toyota", "Red");

      // Method invocation
myCar.displayInfo();   // Calls displayInfo method
myCar.start();         // Calls start method
   }
}
```

3. **Use getters and setters to access private data members.**

```
// Employee class with private members class
Employee
private int id;
private String name;
private double salary;

   // Getter for id
public int getId() {
return id;
   }

   // Setter for id     public
void setId(int id) {
this.id = id;
   }

   // Getter for name
public String getName() {
return name;
   }

   // Setter for name     public void
setName(String name) {
this.name = name;
   }

   // Getter for salary
public double getSalary() {
return salary;
   }

   // Setter for salary
```

```
public void setSalary(double salary) {
this.salary = salary;
    }

   // Method to display details
public void display() {
        System.out.println("Employee ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Salary: ₹" + salary);
    }
}

// Main class public class
EmployeeDemo {
 public static void main(String[] args) {
// Create an Employee object
        Employee emp = new Employee();

        // Set values using setters
emp.setId(101);
emp.setName("Ravi Kumar");
emp.setSalary(55000.75);

        // Access values using getters
        System.out.println("Accessing through getters:");
        System.out.println("ID: " + emp.getId());
        System.out.println("Name: " + emp.getName());
        System.out.println("Salary: ₹" + emp.getSalary());

        // Display full details using method
        System.out.println("\nUsing display method:");
emp.display();
    }
}
```

4. **Build a Book class and create objects to store different book information.**

```
class Book {
    String title;
    String author;
 String isbn;     double
 price;
 public Book(String title, String author, String isbn, double price) {
 this.title = title;
 this.author = author;
 this.isbn = isbn;
 this.price = price;
    }
```

```
    // Method to display book details
public void displayDetails() {
        System.out.println("Title : " + title);
        System.out.println("Author: " + author);
        System.out.println("ISBN  : " + isbn);
        System.out.println("Price : ₹" + price);
        System.out.println("--------------------------");
}
}

// Main class to test Book objects public
class BookStore {
public static void main(String[] args) {
        // Create Book objects
        Book book1 = new Book("Java Basics", "Anjali Sharma", "ISBN001", 399.50);
        Book book2 = new Book("Data Structures", "Ravi Mehra", "ISBN002", 499.99);
        Book book3 = new Book("OOP Concepts", "Meena Rao", "ISBN003", 299.00);

        // Display book information
        System.out.println(" Book Details:");
book1.displayDetails();
book2.displayDetails();
book3.displayDetails();
    }
 }
```

5. **Create a BankAccount class and show deposit and withdrawal actions**.

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

    // Method to deposit money
 public void deposit(double amount) {
if (amount > 0) {
balance += amount;
            System.out.println("Deposited ₹" + amount + ". New Balance: ₹" + balance);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }
```

```
   // Method to withdraw money
public void withdraw(double amount) {
if (amount > 0 && amount <= balance) {
balance -= amount;
          System.out.println("Withdrew ₹" + amount + ". Remaining Balance: ₹" + balance);
       } else {
          System.out.println("Insufficient balance or invalid amount.");
       }
   }

   // Method to display account details
public void displayAccountInfo() {
       System.out.println("Account Holder : " + accountHolder);
       System.out.println("Account Number : " + accountNumber);
       System.out.println("Balance        : ₹" + balance);
       System.out.println("-----------------------------");
   }
}

// Main class to test BankAccount public
class BankApp {
public static void main(String[] args) {
       // Create a BankAccount object
       BankAccount acc1 = new BankAccount("Ravi Kumar", "ACC12345", 1000.0);

       // Display initial account details
acc1.displayAccountInfo();

       // Perform deposit and withdrawal
acc1.deposit(500.0);       // Depositing ₹500
acc1.withdraw(300.0);      // Withdrawing ₹300
acc1.withdraw(1500.0);     // Attempt to overdraw         //
Final account state         acc1.displayAccountInfo();
   }
}
```

1. Create a Student class with name, age, and grade attributes.

```
public class Student {
        //   Attributes
      String name;
      int age;
      char grade;

         // Constructor
      public Student(String name, int age, char grade) {
      this.name = name;
      this.age = age;
```

```
this.grade = grade;
    }

    // Method to display details
 public void displayDetails() {
System.out.println("Name : " + name);
        System.out.println("Age  : " + age);
        System.out.println("Grade: " + grade);
        System.out.println("----------------------");
    }
}
```

2. **Define a method inside a class to display details.**

```
public class MainClass {
public static void main(String[] args) {
        // Create multiple Student objects
        Student s1 = new Student("Anjali", 16, 'A');
        Student s2 = new Student("Rohan", 17, 'B');
        Student s3 = new Student("Meena", 16, 'A');

        // Display each student's details
s1.displayDetails();
s2.displayDetails();
s3.displayDetails();
    }
}
```

3. **Instantiate multiple objects and show their data.**

```
class Student {
    // Attributes
    String name;
    int age;
char grade;

    // Constructor
 public Student(String name, int age, char grade) {
this.name = name;
this.age = age;
this.grade = grade;
    }

    // Method to display student details
public void displayDetails() {
        System.out.println("Name : " + name);
        System.out.println("Age  : " + age);
        System.out.println("Grade: " + grade);
        System.out.println("----------------------");
```

```
    }
}

// Main class to run the program public
class StudentDemo {
public static void main(String[] args) {
// Instantiate multiple Student objects
        Student student1 = new Student("Anjali", 16, 'A');
        Student student2 = new Student("Ravi", 17, 'B');
        Student student3 = new Student("Meena", 15, 'A');

        // Show their data
student1.displayDetails();
student2.displayDetails();
student3.displayDetails();
    }
}
```

4. **Create a class Employee with a method to calculate salary based on hours worked.**

```
class Employee {
    //          Attributes
String name;
int hoursWorked;
double hourlyRate;

    // Constructor
public Employee(String name, int hoursWorked, double hourlyRate) {
this.name = name;
this.hoursWorked = hoursWorked;
this.hourlyRate = hourlyRate;
    }

    // Method to calculate salary
public double calculateSalary() {
return hoursWorked * hourlyRate;
    }

    // Method to display employee details and salary
public void displaySalaryDetails() {
        System.out.println("Employee Name : " + name);
        System.out.println("Hours Worked  : " + hoursWorked);
        System.out.println("Hourly Rate   : ₹" + hourlyRate);
        System.out.println("Total Salary  : ₹" + calculateSalary());
        System.out.println("--------------------------");
    }
}
```

```
// Main class to run the program public
class SalaryCalculator {
public static void main(String[] args) {
        // Create Employee objects
        Employee emp1 = new Employee("Ravi Kumar", 40, 250.0);
        Employee emp2 = new Employee("Anjali Mehra", 35, 300.0);

        // Display their salary details
emp1.displaySalaryDetails();
emp2.displaySalaryDetails();
    }
}
```

5. Write a Car class and display its specifications using a method.

```
// Car class definition class Car {
    // Attributes
    String brand;
String model;
int year;
    double engineCapacity;

    // Constructor
public Car(String brand, String model, int year, double engineCapacity) {
this.brand = brand;
this.model = model;
this.year = year;
this.engineCapacity = engineCapacity;
    }

    // Method to display car specifications
public void displaySpecifications() {
System.out.println("Car Specifications:");
        System.out.println("Brand : " + brand);
        System.out.println("Model  : " + model);
        System.out.println("Year : " + year);
        System.out.println("Engine Capacity: " + engineCapacity + "L");
        System.out.println("-----------------------------");
    }
}

// Main class to test Car objects public
class CarDemo {
public static void main(String[] args) {
        // Create Car objects
        Car car1 = new Car("Toyota", "Innova", 2022, 2.4);
        Car car2 = new Car("Hyundai", "Creta", 2023, 1.5);
```

```
      // Display car specifications
car1.displaySpecifications();
car2.displaySpecifications();
   }
}
```

1. **Demonstrate method overloading with different parameters.**

```
class Calculator {

   // Method to add two integers
public int add(int a, int b) {
return a + b;
   }

   // Method to add three integers
public int add(int a, int b, int c) {
return a + b + c;
   }

   // Method to add two doubles
public double add(double a, double b) {
return a + b;
   }

   // Method to add an int and a double
public double add(int a, double b) {
return a + b;
   }

   // Method to add a double and an int
public double add(double a, int b) {
return a + b;
   }
}

// Main class to test overloading public
class OverloadingDemo {
public static void main(String[] args) {
      Calculator calc = new Calculator();

      // Call overloaded methods
      System.out.println("add(10, 20) = " + calc.add(10, 20));
      System.out.println("add(10, 20, 30) = " + calc.add(10, 20, 30));
      System.out.println("add(10.5, 20.3) = " + calc.add(10.5, 20.3));
      System.out.println("add(10, 20.5) = " + calc.add(10, 20.5));
      System.out.println("add(15.2, 5) = " + calc.add(15.2, 5));
   }
}
```

2. Show method overriding in child class

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

// Child class
class Cat extends Animal {
   // Overriding the parent class method
   @Override
   public void makeSound() {
      System.out.println("Cat meows");
   }
}

// Main class to test overriding public class OverridingDemo {
public static void main(String[] args) {
Animal genericAnimal = new Animal();
genericAnimal.makeSound();
      Cat kitty = new Cat();
kitty.makeSound();  // Output: Cat meows

      // Polymorphism: parent reference, child object
Animal animalRef = new Cat();
animalRef.makeSound();
   }
}
```

3. Use instanceof to check the type at runtime.

```
class Animal {
public void speak() {
      System.out.println("Animal speaks");
   }
}

// Derived class class Dog
extends Animal {
public void bark() {
      System.out.println("Dog barks");
   }
}

// Derived class class Cat
extends Animal {
public void meow() {
```

```
        System.out.println("Cat meows");
    }
}

// Main class public class
InstanceOfExample {
public static void main(String[] args) {
        Animal a1 = new Dog();
        Animal a2 = new Cat();
        Animal a3 = new Animal();

        checkType(a1);
checkType(a2);
checkType(a3);
    }

    // Method to check type using instanceof
public static void checkType(Animal a) {
if (a instanceof Dog) {
            System.out.println("This is a Dog.");
            ((Dog) a).bark(); // downcast to Dog
        } else if (a instanceof Cat) {
            System.out.println("This is a Cat.");
            ((Cat) a).meow(); // downcast to Cat
        } else if (a instanceof Animal) {
            System.out.println("This is a generic Animal.");
a.speak();
        }
    }
}
```

4. Create a Payment class and override pay() in CreditCard, Cash subclasses.

```
// Base class class Payment {
    public void pay(double amount) {
        System.out.println("Paying ₹" + amount + " using a generic payment
method.");
    }
}

// Subclass - CreditCard class
CreditCard extends Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paying ₹" + amount + " using Credit Card.");
    }
}

// Subclass - Cash class Cash
extends Payment {
```

```
    @Override
    public void pay(double amount) {
        System.out.println("Paying ₹" + amount + " using Cash.");
    }
}

// Main class to test public class
PaymentDemo {
public static void main(String[] args) {
        Payment p1 = new CreditCard();
        Payment p2 = new Cash();

        p1.pay(1500.50);  // Output: Paying ₹1500.5 using Credit Card.
p2.pay(500);      // Output: Paying ₹500.0 using Cash.
    }
}
```

4. Build a Notification class and implement different behaviors for Email, SMS, Push.

```
class Notification {
public void notifyUser() {
        System.out.println("Sending a generic notification");
    }
}

// Subclass for Email class EmailNotification
extends Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending Email notification to user");
    }
}

// Subclass for SMS class SMSNotification
extends Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending SMS notification to user");
    }
}

// Subclass for Push class PushNotification
extends Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending Push notification to user");
    }
}
```

```
// Main class to test public class
NotificationDemo {
public static void main(String[]
args) {
Notification n1 = new
EmailNotification();
      Notification n2 = new SMSNotification();
      Notification n3 = new PushNotification();

      n1.notifyUser();  // Output: Sending Email notification to user
n2.notifyUser();  // Output: Sending SMS notification to user
n3.notifyUser();  // Output: Sending Push notification to user
   }
}
```

1. **Create an abstract class Shape with an abstract method draw().**

```
class Shape {
public abstract void draw();

   // Concrete method
public void displayType() {
      System.out.println("This is a shape.");
   }
}

// Concrete subclass - Circle class
Circle extends Shape {
   @Override
public void draw() {
      System.out.println("Drawing a Circle");
   }
}

// Concrete subclass - Rectangle
class Rectangle extends Shape {
   @Override
public void draw() {
      System.out.println("Drawing a Rectangle");
   }
}

// Main class to test public
class ShapeDemo {
   public static void main(String[] args) {
      Shape s1 = new Circle();
      Shape s2 = new Rectangle();
```

```
        s1.displayType();   // Inherited concrete method
s1.draw();          // Circle's implementation
```

```
    s2.displayType();   // Inherited concrete method s2.draw();
    // Rectangle's implementation

    }
```

2. Implement the abstract class in Circle and Square.

```
class Shape {
    public abstract void draw();
    }

    // Subclass: Circle class
    Circle extends Shape {
      @Override
    public void draw() {
        System.out.println("Drawing a Circle with a compass.");
      }
    }

    // Subclass: Square class
    Square extends Shape {
      @Override
    public void draw() {
        System.out.println("Drawing a Square with a ruler.");
      }
    }

    // Main class to test public
    class ShapeTest {
      public static void main(String[] args) {
        Shape circle = new Circle();
        Shape square = new Square();

        circle.draw();  // Output: Drawing a Circle with a compass.
    square.draw();  // Output: Drawing a Square with a ruler.
      }
    }
```

3. Show partial abstraction using non-abstract and abstract methods

```
class Vehicle {

      // Abstract method (no body)
    public abstract void start();

      // Concrete method
    public void fuelType() {
        System.out.println("This vehicle uses petrol or diesel.");
      }
```

```
}

// Subclass
class Car extends Vehicle {

   @Override
public void start() {
      System.out.println("Car starts with key ignition.");
   }
}

// Main class to test public class
PartialAbstractionDemo {
public static void main(String[] args) {
      Vehicle v = new Car();

      v.start();       // Abstract method overridden by Car
      v.fuelType();    // Concrete method from abstract class
   }
}
```

4. Define a class Employee with abstract method calculateSalary() and implement in FullTime and PartTime subclasses.

```
// Abstract class abstract
class Employee {
String name;
int id;

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

   // Abstract method
public abstract double calculateSalary();

   // Concrete method
public void displayInfo() {
      System.out.println("Employee ID: " + id);
      System.out.println("Employee Name: " + name);
   }
}

// FullTime subclass class FullTime
extends Employee {
double monthlySalary;
```

```
    FullTime(String name, int id, double monthlySalary) {
        super(name, id); this.monthlySalary
        = monthlySalary;


    @Override
public double calculateSalary() {
return monthlySalary;
    }
}

// PartTime subclass class PartTime
extends Employee {
int hoursWorked;
double hourlyRate;

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

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

// Main class to test public
class EmployeeTest {
    public static void main(String[] args) {
        Employee ft = new FullTime("Alice", 101, 50000);
        Employee pt = new PartTime("Bob", 102, 80, 300);

        ft.displayInfo();
        System.out.println("Monthly Salary: ₹" + ft.calculateSalary());
        System.out.println();

        pt.displayInfo();
        System.out.println("Monthly Salary: ₹" + pt.calculateSalary());
    }
}
```

5. Create an abstract Appliance class and implement it for Fan and AC.

```
class Appliance {
```

```
    String brand;

    // Constructor
    Appliance(String brand) {
        this.brand = brand;
    }

    // Abstract method
public abstract void turnOn();

    // Concrete method
public void showBrand() {
        System.out.println("Appliance Brand: " + brand);
    }
}

// Subclass: Fan class
Fan extends Appliance {

    Fan(String brand) {
super(brand);
    }

    @Override
public void turnOn() {
        System.out.println("Fan is now spinning.");
    }
}

// Subclass: AC class AC
extends Appliance {

    AC(String brand) {
super(brand);
    }

    @Override     public
void turnOn() {
        System.out.println("AC is now cooling the room.");
    }
}

// Main class to test public class
ApplianceTest {
public static void main(String[] args) {
Appliance fan = new Fan("Havells");
        Appliance ac = new AC("LG");
```

```
        fan.showBrand();
fan.turnOn();
        System.out.println();

        ac.showBrand();
        ac.turnOn();

}
```

1. Create a class with private fields and public getters/setters

```
class Person {
    // Private fields
private String name;
private int age;

    // Public setter for name
public void setName(String name) {
this.name = name;
    }

    // Public getter for name
public String getName() {
return name;
    }

    // Public setter for age
public void setAge(int age) {
if(age > 0) {
this.age = age;
        } else {
            System.out.println("Age must be positive.");
        }
    }

    // Public getter for age
public int getAge() {
return age;
    }
}

// Main class to test public class
EncapsulationDemo {
public static void main(String[] args) {
        Person p = new Person();

        p.setName("John Doe");
```

```
        p.setAge(25);

        System.out.println("Name: " + p.getName());
        System.out.println("Age: " + p.getAge());
    }
}
```

2. **Demonstrate how encapsulation protects data.**

```
class BankAccount {
    private String accountHolder;
private double balance;

    // Constructor
public BankAccount(String accountHolder, double initialDeposit) {
this.accountHolder = accountHolder;
if (initialDeposit >= 0) {
this.balance = initialDeposit;
        }
else {
            System.out.println("Initial deposit cannot be negative.");
        }
    }

    // Public getter for account holder
public String getAccountHolder() {
return accountHolder;
    }

    // Public getter for balance
public double getBalance() {
return balance;
    }

    // Public method to deposit money (controlled access)
public void deposit(double amount) {
if (amount > 0) {
balance += amount;
            System.out.println("Deposited: ₹" + amount);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }

    // Public method to withdraw money (controlled access)
public void withdraw(double amount) {
if (amount > 0 && amount <= balance) {
balance -= amount;
```

```
            System.out.println("Withdrawn: ₹" + amount);
        } else {
            System.out.println("Invalid or insufficient funds.");
        }
    }
}
```

```
// Main class to test public class
EncapsulationProtection {
public static void main(String[] args) {
        BankAccount account = new BankAccount("Ravi", 1000);

        // Access via methods only (data is protected)
System.out.println("Account Holder: " + account.getAccountHolder());
        System.out.println("Initial Balance: ₹" + account.getBalance());

        account.deposit(500);
account.withdraw(300);
account.withdraw(2000);

        // Direct access like account.balance = -1000; is not possible
        System.out.println("Final Balance: ₹" + account.getBalance());
    }
}
```

3. Prevent setting negative values to age field using validation in setter

```
class Person {
private String name;
private int age;

    // Setter for name
public void setName(String name) {
this.name = name;
    }

    // Getter for name
public String getName() {
return name;
    }
public void setAge(int age) {
 if (age >= 0) {
this.age = age;
        }
else {
            System.out.println("Age cannot be negative. Value not set.");
        }
    }

    // Getter for age
public int getAge() {
return age;
    }
}
```

```
// Main class to test public class
AgeValidationTest {
public static void main(String[] args) {
        Person person = new Person();

        person.setName("Anjali");
person.setAge(25);  // Valid
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());

        person.setAge(-5);  // Invalid
        System.out.println("Age after invalid update attempt: " +
person.getAge());
    }
}
```

a. **Create a Patient class that restricts direct access to medicalHistory** //

```
class Patient {
private String name;
private int age;
        private String medicalHistory;

    // Constructor
public Patient(String name, int age, String medicalHistory) {
this.name = name;
this.age = age;
this.medicalHistory = medicalHistory;
    }

    // Getters for name and age
(public)
 public String getName() {
return name;
    }

    public int getAge() {
return age;
    }

    // Getter for medicalHistory (controlled
access)
public String getMedicalHistory(String role) {
if (role.equalsIgnoreCase("Doctor") ||
role.equalsIgnoreCase("Admin")) {
return medicalHistory;
        }
 else {
return "Access Denied: Unauthorized role.";
```

```
        }
    }

    // Setter for medicalHistory (restricted)
public void setMedicalHistory(String history, String role) {
if (role.equalsIgnoreCase("Doctor")) {
this.medicalHistory = history;
            System.out.println("Medical history updated.");
        } else {
            System.out.println("Access Denied: Only doctors can update
medical history.");
        }
    }
}

// Main class to test public class
PatientAccessControl {
public static void main(String[] args) {
        Patient p = new Patient("Ravi", 40, "Diabetes, High BP");

        // Public access
        System.out.println("Patient Name: " + p.getName());
        System.out.println("Patient Age: " + p.getAge());

        // Trying to access medical history as a Nurse
System.out.println("Medical History (Nurse): " +
p.getMedicalHistory("Nurse"));

        // Trying as Doctor
        System.out.println("Medical History (Doctor): " +
p.getMedicalHistory("Doctor"));

        // Attempt to update medical history as Admin
        p.setMedicalHistory("Updated History", "Admin");

        // Update by Doctor
        p.setMedicalHistory("Recovered from Diabetes", "Doctor");
System.out.println("Updated Medical History: " +
p.getMedicalHistory("Doctor"));
    }
}
```

5. Build a LoanAccount class and encapsulate the balance with setter restrictions.

```
class LoanAccount {
 private String accountHolder;
private double balance;
```

```
   // Constructor
   public LoanAccount(String accountHolder, double initialLoanAmount)
{
this.accountHolder = accountHolder;
if (initialLoanAmount > 0) {
this.balance = initialLoanAmount;
       }
else {
          System.out.println(" Loan amount must be positive. Account not
initialized properly.");
       }
   }

   // Getter for account holder
public String getAccountHolder() {
return accountHolder;
   }

   // Getter for balance (read-only
access)
public double getBalance() {         return
balance;
   }

   // Repayment method (controlled way to reduce
balance)
public void repay(double amount) {
if (amount > 0) {
if (amount <= balance) {
balance -= amount;
             System.out.println(" Repaid " + amount + ". Remaining loan: "
+ balance);
          } else {
             System.out.println(" Repayment exceeds current loan
balance.");
          }
       } else {
          System.out.println(" Repayment amount must be positive.");
       }
   }

private void setBalance(double balance) {         t
his.balance = balance;
   }

   public void addLoan(double extraAmount) {
```

```
 if (extraAmount > 0) {
balance += extraAmount;
          System.out.println(" Additional loan of " + extraAmount + "
approved. New balance: " + balance);
       } else {
          System.out.println(" Invalid loan amount.");
       }
   }
} //Main class public class
LoanAccountTest {
public static void main(String[] args) {
       // Creating a loan account with an initial loan amount
       LoanAccount loan = new LoanAccount("Rahul", 50000);

       System.out.println("Account Holder: " + loan.getAccountHolder());
       System.out.println("Current Loan Balance: " + loan.getBalance());

       // Trying to repay part of the loan
 loan.repay(10000);  // Valid repayment
loan.repay(60000);  // Invalid repayment (exceeds balance)
loan.repay(-500);   // Invalid repayment (negative amount)

loan.addLoan(20000);        //    Valid
loan.addLoan(-3000);  // Invalid
   }
}
```

1. **Create an interface Drawable with method draw().**

```
interface Drawable {
   void draw();  // abstract method
}

// Class implementing Drawable
class Circle implements Drawable {
public void draw() {
       System.out.println("Drawing a Circle");
   }
}

// Another class implementing Drawable
class Rectangle implements Drawable {
public void draw() {
       System.out.println("Drawing a Rectangle");
   }
}
```

```
// Test class with main method
public class InterfaceDemo {
    public static void main(String[] args) {
        Drawable d1 = new Circle();     // Polymorphism: Drawable
reference to Circle
        Drawable d2 = new Rectangle();  // Polymorphism: Drawable
reference to Rectangle

        d1.draw();  // Output: Drawing a Circle
d2.draw();  // Output: Drawing a Rectangle
    }
}
```

2. Implement the interface in class

Circle.

```
interface Drawable {
    void draw();  // abstract method
}
 class Circle implements Drawable {

    @Override
public void draw() {
        System.out.println(" Drawing a Circle");
    }
}

// Step 3: Main class to test public class
InterfaceImplementationDemo {
public static void main(String[] args) {
        Circle c = new Circle();
        c.draw();  // Call the draw method
    }
}
```

2. Create another interface Colorable and implement both interfaces in a single class.

```
interface Drawable {
void draw();  // Abstract method to draw a shape
}

// Second interface interface Colorable {
void fillColor(String color);  // Abstract method to fill color
}

// Class implementing both interfaces class
Circle implements Drawable, Colorable {
@Override
public void draw() {
        System.out.println(" Drawing a Circle");
```

```
    }

    @Override
    public void fillColor(String color) {
        System.out.println("Filling Circle with color: " + color);
    }
}

// Main class to test public class
MultipleInterfaceDemo {
public static void main(String[] args) {
        Circle c = new Circle();
        c.draw();                    // Call draw method
        c.fillColor("Blue");         // Call fillColor method
    }
}
```

4. Design an interface Payable and implement it in classes Invoice and Employee.

```
interface Payable {
double getPaymentAmount();  // abstract method
}

class Invoice implements Payable {
private String item;
private int quantity;
private double pricePerItem;

    public Invoice(String item, int quantity, double pricePerItem)
{
this.item = item;
this.quantity = quantity;
this.pricePerItem = pricePerItem;
    }

    @Override     public double
getPaymentAmount() {
return quantity * pricePerItem;
    }

    public void display() {
        System.out.println(" Invoice for " + item + " | Qty: " + quantity
+
" | Total: ₹" + getPaymentAmount());
    }
}
```

```
class Employee implements Payable {
private String name;

 private double monthlySalary;

   public Employee(String name, double monthlySalary)
{
this.name = name;
this.monthlySalary = monthlySalary;
   }

   @Override
public double getPaymentAmount() {
return monthlySalary;
   }

   public void display() {
      System.out.println(" Employee: " + name + " | Monthly Salary: "
+ getPaymentAmount());
   }
}

// Main class to test public
class PayableTest {
   public static void main(String[] args) {
      Invoice invoice = new Invoice("Laptop", 2, 50000);
      Employee employee = new Employee("Anjali", 60000);

      invoice.display();     // Display invoice payment info
employee.display();    // Display employee salary info
   }
}
```

5. Create an interface Database with connect() method and implement

```
interface Database {
   void connect();  // abstract method
}

class MySQL implements Database {
   @Override
public void connect() {
      System.out.println("  Connecting to MySQL Database...");
   }
}

class Oracle implements Database {
```

```
    @Override
public void connect() {
        System.out.println("  Connecting to Oracle Database...");
    }
}

// Test class with main method
public class DatabaseTest {
    public static void main(String[] args) {
        Database db1 = new MySQL();   // Using interface reference
        Database db2 = new Oracle();

        db1.connect();  // Connects to MySQL
db2.connect();  // Connects to Oracle
    }
}
```

1. **Create a class Engine and use it in class Car.**

```
class Engine {
private String type;
private int horsepower;

    public Engine(String type, int horsepower) {
this.type = type;
        this.horsepower = horsepower;
    }

    public void start() {
        System.out.println("Engine started. Type: " + type + ",
Horsepower: " + horsepower);
    }
}

class Car {

private String model;
private Engine engine;

    public Car(String model, Engine engine)
{
this.model = model;        t
his.engine = engine;
    }

    public void startCar() {
```

```
        System.out.println("Starting car: " + model);
engine.start();  // Delegate to Engine
    }
}

// Main class to test public class
CarEngineDemo {
public static void main(String[] args) {
Engine e = new Engine("V8", 400);
        Car car = new Car("Mustang", e);

        car.startCar();
    }
}
```

2. **Use composition to build a Computer with Processor, RAM, and HardDrive objects.**

```
class Processor {
private String brand;
private double speedGHz;

    public Processor(String brand, double speedGHz)
{
this.brand = brand;
this.speedGHz = speedGHz;
    }

    public void displayInfo() {
        System.out.println("Processor: " + brand + " @ " +
speedGHz + "GHz");
    }
}

// Component 2: RAM class
class RAM {
    private int sizeGB;

    public RAM(int sizeGB) {
this.sizeGB = sizeGB;
    }

    public void displayInfo() {
        System.out.println(" RAM: " + sizeGB + "GB");
    }
}

// Component 3: HardDrive
class class HardDrive {
```

```
private int capacityGB;
private String type;

   public HardDrive(int capacityGB, String type)
{        this.capacityGB = capacityGB;
this.type = type;
   }

   public void displayInfo() {
      System.out.println(" Hard Drive: " + capacityGB + "GB " +
type);
   }
}

// Main class: Computer
(Composition) class Computer {
private Processor processor;
private RAM ram;
private HardDrive hardDrive;

   public Computer(Processor processor, RAM ram,
HardDrive hardDrive) {
this.processor = processor;
this.ram = ram;
this.hardDrive = hardDrive;
   }

   public void showSpecs() {
      System.out.println(" Computer
Specifications:");
processor.displayInfo();
ram.displayInfo();
hardDrive.displayInfo();
   }
}

// Test class
public class ComputerDemo {
public static void main(String[] args) {
      Processor p = new Processor("Intel i7", 3.8);
RAM r = new RAM(16);
      HardDrive h = new HardDrive(512, "SSD");

      Computer myPC = new Computer(p, r, h);
myPC.showSpecs();
   }
}
```

3. Demonstrate "has-a" relationship using class Library with Book objects.

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

// Book class
class Book {
   private String title;
private String author;

   public Book(String title, String author)
{
this.title = title;
this.author = author;
   }

   public void displayInfo() {
      System.out.println(" Title: " + title + ", Author: " + author);
   }
}

// Library class (has-a relationship with Book) class Library {
private List<Book> books;  // Library has a list of Book objects

   public Library() {
books = new ArrayList<>();
   }

   public void addBook(Book book) {
books.add(book);
   }

   public void showLibrary() {
      System.out.println(" Library contains the following books:");
for (Book book : books) {
 book.displayInfo();
      }
   }
}

// Main class to test public
class LibraryDemo {
   public static void main(String[] args) {
      Book b1 = new Book("The Alchemist", "Paulo Coelho");
      Book b2 = new Book("To Kill a Mockingbird", "Harper
```

```
Lee");

        Library library = new Library();
        library.addBook(b1);
        library.addBook(b2);

        library.showLibrary();
    }
}
```

3. **Model a Student having an Address and IDCard as composed objects.**

```
class Address {
private String street;
private String city;
private String zipCode;

   public Address(String street, String city, String zipCode)
{
this.street = street;
this.city = city;
this.zipCode = zipCode;
   }

   public void displayAddress() {
      System.out.println(" Address: " + street + ", " + city + " - " +
zipCode);
   }
}

// IDCard class class
IDCard {
private String idNumber;
private String issueDate;

   public IDCard(String idNumber, String issueDate)
{
this.idNumber = idNumber;
 this.issueDate = issueDate;
   }

   public void displayIDCard() {
      System.out.println("ID Card Number: " + idNumber + ", Issued on:
" + issueDate);
   }
}
```

```
// Student class (has-a Address and
IDCard) class Student {
private String name;
private Address address;
private IDCard idCard;

   public Student(String name, Address address, IDCard idCard)
{
this.name = name;
this.address = address;
this.idCard = idCard;
   }

   public void displayStudentInfo() {
      System.out.println(" Student Name: " + name);
idCard.displayIDCard();
address.displayAddress();
   }
}

// Main class to test public
class StudentDemo {
   public static void main(String[] args) {
      Address addr = new Address("123 Main St", "Pune", "411001");
      IDCard card = new IDCard("STU2025", "01-Jul-2025");

      Student student = new Student("Ananya Sharma", addr, card);

      student.displayStudentInfo();
   }
}
```

4. Create a House class that has Room and Kitchen as components.

```
// Room class
class Room {
private int roomNumber;
private String purpose;

   public Room(int roomNumber, String purpose) {
this.roomNumber = roomNumber;
this.purpose = purpose;
   }

   public void displayRoomInfo() {
      System.out.println(" Room " + roomNumber + ": " +
purpose);
   }
}
```

```
// Kitchen class class Kitchen {
private boolean hasMicrowave;
private String style;

   public Kitchen(boolean hasMicrowave, String style)
{
 this.hasMicrowave = hasMicrowave;
this.style = style;
   }

   public void displayKitchenInfo() {
      System.out.println(" Kitchen Style: " + style);
System.out.println("Microwave Available: " +
(hasMicrowave ? "Yes" : "No"));
   }
}

// House class (has-a Room and
Kitchen) class House {
private Room room;
private Kitchen kitchen;

   public House(Room room, Kitchen kitchen)
{
this.room = room;
this.kitchen = kitchen;
   }

   public void displayHouseInfo() {
System.out.println("House Details:");
room.displayRoomInfo();
kitchen.displayKitchenInfo();
   }
}

// Test class public class
HouseDemo {
   public static void main(String[] args) {
      Room room = new Room(1, "Bedroom");
      Kitchen kitchen = new Kitchen(true, "Modern");

      House house = new House(room, kitchen);
house.displayHouseInfo();
   }
}
```

1. Overload a method add() with two and three parameters.

```
public class Calculator {

    // Method with two parameters
public int add(int a, int b) {
return a + b;
    }

    // Overloaded method with three
parameters
public int add(int a, int b, int c) {
 return a + b + c;
    }

    // Main method to test
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        int sumTwo = calc.add(10, 20);
int sumThree = calc.add(5, 15, 25);

        System.out.println("Sum of 2 numbers: " + sumTwo);      // 30
        System.out.println("Sum of 3 numbers: " + sumThree);    // 45
    }
}
```

2. Override toString() method of a custom class.

```
    // Custom class
class Person {
private String name;
private int age;
private String city;

    // Constructor
public Person(String name, int age, String city) {
this.name = name;
this.age = age;
this.city = city;
    }

    // Override toString() method
    @Override
public String toString() {
return " Person{name='" + name + "', age=" + age + ", city='" + city + "'}";
    }
}
```

```
// Main class to test public
class ToStringDemo {
    public static void main(String[] args) {
        Person person1 = new Person("Aarav", 25, "Mumbai");
        Person person2 = new Person("Diya", 30, "Delhi");

        // Print objects directly
        System.out.println(person1);  // toString() is automatically called
        System.out.println(person2);
    }
}
```

3. Override a display() method from base class in child class

```
class Animal {
    public void display() {
        System.out.println("This is an animal.");
    }
}

// Derived class
class Dog extends Animal {
    @Override
    public void display() {
        System.out.println("This is a dog.");
    }
}

// Main class to test public
class OverrideDemo {
    public static void main(String[] args) {
        Animal a = new Animal(); // Base class reference and object
a.display(); // Output: This is an animal.

        Dog d = new Dog(); // Child class reference and object
d.display(); // Output: This is a dog.

        Animal ad = new Dog(); // Base class reference, child class object
(runtime polymorphism)
ad.display(); // Output: This is a dog.
    }
}
```

4. Create a Logger class with overloaded log() methods for different data types.

```
public class Logger {

    // Log a String message
public void log(String message) {
        System.out.println("String log: " + message);
```

```
    }

    // Log an integer value
  public void log(int number) {
        System.out.println("Integer log: " + number);
    }

    // Log a double value
   public void log(double value) {
        System.out.println("Double log: " + value);
    }

    // Log a boolean value
  public void log(boolean flag) {
        System.out.println("Boolean log: " + flag);
    }

    // Main method to test
    public static void main(String[] args) {
        Logger logger = new Logger();

        logger.log("System started
  successfully.");
  logger.log(404);
  logger.log(3.14159);        logger.log(true);
    }
  }
```

5. Build a Vehicle class with overridden move() in Car and Bike subclasses.

```
class Vehicle {
public void move() {
        System.out.println("The vehicle is moving...");
    }
  }

  // Subclass Car
  class Car extends Vehicle {
    @Override
  public void move() {
        System.out.println(" The car drives smoothly on the road.");
    }
  }

  // Subclass Bike class Bike
  extends Vehicle {
    @Override
  public void move() {
```

```
        System.out.println(" The bike zooms through traffic.");
    }
}

// Main class to test the behavior public
class VehicleTest {
    public static void main(String[] args) {
        Vehicle genericVehicle = new Vehicle();
        Vehicle car = new Car();
        Vehicle bike = new Bike();

        genericVehicle.move(); // Base class method
car.move();                    // Car's overridden method
bike.move();           // Bike's overridden method
    }
}
```

1. Create Department and Student classes showing aggregation.

```
class Student {
private String name;

private int rollNo;

        // Constructor     p
     ublic Student(String name, int rollNo) {
     this.name = name;
this.rollNo = rollNo;
        }

        // Method to display student info
     public void displayStudent() {
            System.out.println("Student Name: " + name + ", Roll No: " + rollNo);
        }
     }

     // Department class containing Student (Aggregation)
     class Department {
     private String deptName;
        private Student student; // Aggregation

        // Constructor
     public Department(String deptName, Student student) {
     this.deptName = deptName;
     this.student = student;
```

```
    }

    // Method to display department and student details
public void showDetails() {
        System.out.println("Department: " + deptName);
student.displayStudent();
    }
}

// Main class to run the program public
class AggregationDemo {
public static void main(String[] args) {
        Student s1 = new Student("Anjali", 101);
        Department d1 = new Department("Computer Science", s1);

        d1.showDetails();
    }
}
```

2. Model a Team that contains players as aggregated objects.

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

// Player class class
Player {
private String name;
private String position;

    // Constructor
public Player(String name, String position) {
this.name = name;
this.position = position;
    }

    // Display player details
public void displayInfo() {
        System.out.println("Player Name: " + name + ", Position: " + position);
    }
}

// Team class that aggregates Player objects
class Team {
private String teamName;
private List<Player> players; // Aggregation

    // Constructor
```

```
public Team(String teamName) {
this.teamName = teamName;
this.players = new ArrayList<>();
   }

   // Add player to team
public void addPlayer(Player player) {
players.add(player);
   }

   // Display team and player details
public void displayTeam() {
       System.out.println("Team: " + teamName);
System.out.println("Players:");
for (Player p : players) {
           p.displayInfo();
       }
   }
}

// Main class to test public class
AggregationTeamDemo {
public static void main(String[] args) {
// Create players
       Player p1 = new Player("Ravi", "Striker");
       Player p2 = new Player("Amit", "Goalkeeper");
       Player p3 = new Player("Suresh", "Defender");

       // Create team and add players
Team team = new Team("Thunderbolts");
team.addPlayer(p1);
team.addPlayer(p2);
team.addPlayer(p3);

       // Display team details
team.displayTeam();
   }
}
```

3. Illustrate aggregation with Teacher and Subject.

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

// Subject class class
Subject {
private String name;
```

```
    public Subject(String name) {
this.name = name;
    }

    public void display() {
        System.out.println("Subject: " + name);
    }
}

// Teacher class that aggregates Subject
class Teacher {
private String teacherName;
    private List<Subject> subjects; // Aggregation

    public Teacher(String teacherName) {
this.teacherName = teacherName;
this.subjects = new ArrayList<>();
    }

    public void addSubject(Subject subject) {
subjects.add(subject);
    }

    public void displayDetails() {
        System.out.println("Teacher: " + teacherName);
System.out.println("Subjects taught:");
for (Subject subject : subjects) {
subject.display();
        }
    }
}

// Main class public class
AggregationExample {
public static void main(String[] args) {
        // Create subjects
        Subject math = new Subject("Mathematics");
        Subject physics = new Subject("Physics");
        Subject cs = new Subject("Computer Science");

        // Create teacher and assign subjects
        Teacher teacher = new Teacher("Mrs. Sharma");
teacher.addSubject(math);
teacher.addSubject(physics);
teacher.addSubject(cs);
```

```
        // Display teacher and subject details
    teacher.displayDetails();
    }
}
```

4. Build a University class that aggregates multiple College objects.

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

// College class class
College {
 private String name;
private String dean;

   public College(String name, String dean) {
this.name = name;
this.dean = dean;
   }

   public void displayDetails() {
      System.out.println("College Name: " + name + ", Dean: " + dean);
   }
}

// University class that aggregates College
class University {
private String universityName;
private List<College> colleges;

   public University(String universityName) {
this.universityName = universityName;
 this.colleges = new ArrayList<>();
   }

   // Add college to the university     public
void addCollege(College college) {
colleges.add(college);
   }

   // Display university and its colleges
public void showUniversityDetails() {
      System.out.println("University: " + universityName);
System.out.println("Colleges under this University:");
for (College c : colleges) {
c.displayDetails();
      }
   }
```

```
}

// Main class to test aggregation public
class AggregationUniversityDemo {
public static void main(String[] args) {
        // Create college objects
        College c1 = new College("Engineering College", "Dr. Mehta");
        College c2 = new College("Arts College", "Prof. Sharma");
        College c3 = new College("Science College", "Dr. Iyer");

        // Create university and add colleges
        University university = new University("Global University");
university.addCollege(c1);
university.addCollege(c2);
 university.addCollege(c3);

        // Display full details
        university.showUniversityDetails();
    }
}
```

5. Create a Hospital with a list of Doctor objects (not tightly bound).

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

// Doctor class class Doctor {
private String name;
private String specialization;
public Doctor(String name,
String specialization) {
this.name = name;
this.specialization =
specialization;
    }

    public void displayDetails() {
        System.out.println("Doctor Name: " + name + ", Specialization: " + specialization);
    }
}

// Hospital class that aggregates Doctor objects class Hospital {
private String hospitalName;
private List<Doctor> doctors; // Aggregation (not tightly bound)

    public Hospital(String hospitalName) {
this.hospitalName = hospitalName;
this.doctors = new ArrayList<>();
```

```
    }

    public void addDoctor(Doctor doctor) {
doctors.add(doctor);
    }

    public void displayHospitalDetails() {
        System.out.println("Hospital: " + hospitalName);
System.out.println("List of Doctors:");
for (Doctor doc : doctors) {
doc.displayDetails();
        }
    }
}

// Main class to test aggregation public
class AggregationHospitalDemo {
public static void main(String[] args) {
        // Doctors created independently
        Doctor d1 = new Doctor("Dr. Anjali", "Cardiologist");
        Doctor d2 = new Doctor("Dr. Kumar", "Orthopedic");
        Doctor d3 = new Doctor("Dr. Nisha", "Neurologist");

        // Hospital aggregates doctors
        Hospital hospital = new Hospital("City Care Hospital");
hospital.addDoctor(d1);
hospital.addDoctor(d2);
hospital.addDoctor(d3);

        // Display hospital and doctor details
        hospital.displayHospitalDetails();
    } }
```