Scenario 1: Vehicle Hierarchy

Detailed Explanation

In this scenario, we have a base class `Vehicle` that contains common properties and methods for all types of vehicles. The derived classes `Car`, `Truck`, and `Motorcycle` inherit from `Vehicle` and add their own specific properties and methods. The `displayInfo()` method in the base class is used to display common information, while each derived class has its own method to display additional details specific to that type of vehicle. This design allows us to maintain a common interface for all vehicles while still providing specific functionality for each type.

Base Class: Vehicle

- **Properties**: `make`, `model`, `year`
- **Methods**:
  - `displayInfo()`: Displays basic information about the vehicle.

Derived Class: Car

- **Properties**: `numberOfDoors`
- **Methods**:
  - `displayCarInfo()`: Displays information specific to cars.

Derived Class: Truck

- **Properties**: `cargoCapacity`
- **Methods**:
  - `displayTruckInfo()`: Displays information specific to trucks.

Derived Class: Motorcycle

- **Properties**: `hasSidecar`
- **Methods**:
  - `displayMotorcycleInfo()`: Displays information specific to motorcycles.

Expected Input and Output

```
Vehicle car = new Car("Toyota", "Corolla", 2022, 4);
car.displayInfo(); // Calls the method from the base class
((Car) car).displayCarInfo(); // Calls the method from the Car class
// Expected Output:
// Toyota Corolla 2022
// Number of Doors: 4

Vehicle truck = new Truck("Ford", "F-150", 2021, 1000);
truck.displayInfo(); // Calls the method from the base class
((Truck) truck).displayTruckInfo(); // Calls the method from the Truck class
// Expected Output:
// Ford F-150 2021
// Cargo Capacity: 1000 kg

Vehicle motorcycle = new Motorcycle("Harley-Davidson", "Street Glide", 2023, true);
motorcycle.displayInfo(); // Calls the method from the base class
((Motorcycle) motorcycle).displayMotorcycleInfo(); // Calls the method from the
Motorcycle class
// Expected Output:
```

```
// Harley-Davidson Street Glide 2023
// Has Sidecar: true
```

Scenario 2: Employee Hierarchy

Detailed Explanation

In this scenario, the `Employee` class provides a basic structure for all employees, including a method to calculate the annual salary based on the base salary. Each derived class (`Manager`, `Engineer`, `Salesperson`) adds specific properties and methods to calculate the salary considering additional factors like bonuses or commissions. This design allows each type of employee to have its own way of calculating the salary while still adhering to the common interface provided by the base class.

Base Class: Employee

- **Properties**: `name`, `baseSalary`
- **Methods**:
  - `calculateAnnualSalary()`: Calculates the annual salary based on the base salary.

Derived Class: Manager

- **Properties**: `bonus`, `numberOfEmployeesManaged`
- **Methods**:
  - `calculateManagerSalary()`: Calculates the annual salary including the bonus.

Derived Class: Engineer

- **Properties**: `projectCount`
- **Methods**:
  - `calculateEngineerSalary()`: Calculates the annual salary including a bonus based on the number of projects completed.

Derived Class: Salesperson

- **Properties**: `commissionRate`, `totalSales`
- **Methods**:
  - `calculateSalespersonSalary()`: Calculates the annual salary including a commission based on total sales.

Expected Input and Output

```
Employee manager = new Manager("Alice", 50000, 10000, 5);
System.out.println(manager.calculateAnnualSalary()); // Calls the method from the base class
System.out.println(((Manager) manager).calculateManagerSalary()); // Calls the method from the Manager class
// Expected Output:
// 60000.0
// 60000.0

Employee engineer = new Engineer("Bob", 70000, 5);
System.out.println(engineer.calculateAnnualSalary()); // Calls the method from the base class
System.out.println(((Engineer) engineer).calculateEngineerSalary()); // Calls the
```

```
method from the Engineer class
// Expected Output:
// 75000.0
// 75000.0

Employee salesperson = new Salesperson("Charlie", 40000, 0.1, 200000);
System.out.println(salesperson.calculateAnnualSalary()); // Calls the method from the
base class
System.out.println(((Salesperson) salesperson).calculateSalespersonSalary()); // Calls
the method from the Salesperson class
// Expected Output:
// 60000.0
// 60000.0
```

Scenario 3: Shape Hierarchy

Detailed Explanation

In this scenario, the `Shape` class provides a basic structure for all shapes, including a method to calculate the area. Each derived class (`Circle`, `Rectangle`, `Triangle`) adds specific properties and methods to calculate the area based on the shape's dimensions. This design allows each shape to have its own way of calculating the area while still adhering to the common interface provided by the base class. This is a classic example of using inheritance to provide a common interface while allowing for specific implementations in derived classes.

Base Class: Shape

- **Properties**: None
- **Methods**:
  - `calculateArea()`: Calculates the area of the shape.

Derived Class: Circle

- **Properties**: `radius`
- **Methods**:
  - `calculateCircleArea()`: Calculates the area of the circle.

Derived Class: Rectangle

- **Properties**: `length`, `width`
- **Methods**:
  - `calculateRectangleArea()`: Calculates the area of the rectangle.

Derived Class: Triangle

- **Properties**: `base`, `height`
- **Methods**:
  - `calculateTriangleArea()`: Calculates the area of the triangle.

Expected Input and Output

```
Shape circle = new Circle(5);
System.out.println(circle.calculateArea()); // Calls the method from the base class
System.out.println(((Circle) circle).calculateCircleArea()); // Calls the method from
the Circle class
// Expected Output:
```

```
// 78.53981633974483
// 78.53981633974483

Shape rectangle = new Rectangle(4, 6);
System.out.println(rectangle.calculateArea()); // Calls the method from the base class
System.out.println(((Rectangle) rectangle).calculateRectangleArea()); // Calls the
method from the Rectangle class
// Expected Output:
// 24.0
// 24.0

Shape triangle = new Triangle(3, 7);
System.out.println(triangle.calculateArea()); // Calls the method from the base class
System.out.println(((Triangle) triangle).calculateTriangleArea()); // Calls the method
from the Triangle class
// Expected Output:
// 10.5
// 10.5
```

Scenario 4: Animal Hierarchy

Detailed Explanation

In this scenario, the `Animal` class provides a basic structure for all animals, including a method to display the habitat. Each derived class (`Mammal` , `Bird` , `Reptile`) adds specific properties and methods to display additional information related to the type of animal. This design allows each animal to have its own way of displaying information while still adhering to the common interface provided by the base class. This is useful for scenarios where different types of animals have unique characteristics but share some common attributes.

Base Class: Animal

- **Properties**: `name` , `habitat`
- **Methods**:
  - `displayHabitat()` : Displays the habitat of the animal.

Derived Class: Mammal

- **Properties**: `isWarmBlooded`
- **Methods**:
  - `displayMammalInfo()` : Displays information specific to mammals.

Derived Class: Bird

- **Properties**: `canFly`
- **Methods**:
  - `displayBirdInfo()` : Displays information specific to birds.

Derived Class: Reptile

- **Properties**: `isColdBlooded`
- **Methods**:
  - `displayReptileInfo()` : Displays information specific to reptiles.

Expected Input and Output

```
Animal mammal = new Mammal("Lion", "Savannah", true);
mammal.displayHabitat(); // Calls the method from the base class
((Mammal) mammal).displayMammalInfo(); // Calls the method from the Mammal class
// Expected Output:
// Habitat: Savannah
// Is Warm Blooded: true

Animal bird = new Bird("Eagle", "Forest", true);
bird.displayHabitat(); // Calls the method from the base class
((Bird) bird).displayBirdInfo(); // Calls the method from the Bird class
// Expected Output:
// Habitat: Forest
// Can Fly: true

Animal reptile = new Reptile("Snake", "Desert", true);
reptile.displayHabitat(); // Calls the method from the base class
((Reptile) reptile).displayReptileInfo(); // Calls the method from the Reptile class
// Expected Output:
// Habitat: Desert
// Is Cold Blooded: true
```

Scenario 5: Student Hierarchy

Detailed Explanation

In this scenario, the `Student` class provides a basic structure for all students, including a method to display basic information. Each derived class (`Undergraduate`, `Graduate`, `PhDStudent`) adds specific properties and methods to display additional information related to the type of student. This design allows each student to have its own way of displaying information while still adhering to the common interface provided by the base class. This is useful for scenarios where different types of students have unique characteristics but share some common attributes.

Base Class: Student

- **Properties**: `name`, `studentId`
- **Methods**:
  - `displayStudentInfo()`: Displays basic information about the student.

Derived Class: Undergraduate

- **Properties**: `major`
- **Methods**:
  - `displayUndergraduateInfo()`: Displays information specific to undergraduates.

Derived Class: Graduate

- **Properties**: `thesisTopic`
- **Methods**:
  - `displayGraduateInfo()`: Displays information specific to graduates.

Derived Class: PhDStudent

- **Properties**: `researchArea`
- **Methods**:

- `displayPhDStudentInfo()`: Displays information specific to PhD students.

Expected Input and Output

```
Student undergraduate = new Undergraduate("Alice", "U12345", "Computer Science");
undergraduate.displayStudentInfo(); // Calls the method from the base class
((Undergraduate) undergraduate).displayUndergraduateInfo(); // Calls the method from the Undergraduate class
// Expected Output:
// Name: Alice, Student ID: U12345
// Major: Computer Science

Student graduate = new Graduate("Bob", "G67890", "Machine Learning");
graduate.displayStudentInfo(); // Calls the method from the base class
((Graduate) graduate).displayGraduateInfo(); // Calls the method from the Graduate class
// Expected Output:
// Name: Bob, Student ID: G67890
// Thesis Topic: Machine Learning

Student phdStudent = new PhDStudent("Charlie", "P11223", "Artificial Intelligence");
phdStudent.displayStudentInfo(); // Calls the method from the base class
((PhDStudent) phdStudent).displayPhDStudentInfo(); // Calls the method from the PhDStudent class
// Expected Output:
// Name: Charlie, Student ID: P11223
// Research Area: Artificial Intelligence
```