Exercises (Question + rephrased question by AI)

9.3 (Using Composition Rather Than Inheritance) Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class BasePlusCommissionEmployee (Fig. 9.11) of the CommissionEmployee–BasePlusCommissionEmployee hierarchy to use composition rather than inheritance.

Exercise 9.3: Using Composition Rather Than Inheritance To rewrite the `BasePlusCommissionEmployee` class using composition, you would create a separate class to hold the "base salary" attribute and then include an instance of that class within the `BasePlusCommissionEmployee` class. This way, you use composition to combine the functionality of the two classes without inheritance.

9.4 (Opportunities for Defects) Every additional line of code is an opportunity for a defect. Discuss the ways in which inheritance promotes defect reduction.

Exercise 9.4: Opportunities for Defects Inheritance can help reduce defects by promoting code reuse. If you have a well-designed superclass and subclasses, you can implement common functionality in the superclass, reducing code duplication. This also allows you to make changes or fixes in one place (the superclass) rather than in multiple subclasses. However, misuse of inheritance can lead to defects when it doesn't properly model the relationship between classes or when it introduces unwanted dependencies.

9.5 (Student Inheritance Hierarchy) Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. 9.2. Use Student as the superclass of the hierarchy, then extend Student with classes UndergraduateStudent and GraduateStudent. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, Freshman, Sophomore, Junior and Senior might extend UndergraduateStudent, and DoctoralStudent and MastersStudent might be subclasses of GraduateStudent. After drawing the hierarchy, discuss the relationships that exist between the classes. [Note: You do not need to write any code for this exercise.]

Exercise 9.5: Student Inheritance Hierarchy Create an inheritance hierarchy with `Student` as the superclass. Subclasses could include `UndergraduateStudent` and `GraduateStudent`. Further subclasses could include `Freshman`, `Sophomore`, `Junior`, `Senior` under `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` under `GraduateStudent`.

9.6 (Shape Inheritance Hierarchy) The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 9.3. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete Shape hierarchy with as many levels as possible. Your hierarchy should have class Shape at the top. Classes TwoDimensionalShape and ThreeDimensionalShape should extend Shape. Add additional subclasses, such as Quadrilateral and Sphere, at their correct locations in the hierarchy as necessary.

Exercise 9.6: Shape Inheritance Hierarchy Create a hierarchy with `Shape` as the superclass. Subclasses can include `TwoDimensionalShape` and `ThreeDimensionalShape`. Continue to create additional subclasses such as `Quadrilateral`, `Triangle`, `Rectangle`, `Circle`, `Sphere`, `Cylinder`, etc., as needed.

9.7 (protected vs. private) Some programmers prefer not to use protected access, because they believe it breaks the encapsulation of the superclass. Discuss the relative merits of using protected access vs. using private access in superclasses.

Exercise 9.7: `protected` vs. `private` Discuss the merits of using `protected` access vs. `private` access in superclasses. `protected` allows access by subclasses, promoting code reuse and extension. `private` restricts access to the class where the member is defined, enhancing encapsulation but not allowing for subclass access.

9.8 (Quadrilateral Inheritance Hierarchy) Write an inheritance hierarchy for classes Quadrilateral, Trapezoid, Parallelogram, Rectangle and Square. Use Quadrilateral as the superclass of the hierarchy. Create and use a Point class to represent the points in each shape. Make the hierarchy as deep (i.e., as many levels) as possible. Specify the instance variables and methods for each class. The private instance variables of Quadrilateral should be the x-y coordinate pairs for the four endpoints of the Quadrilateral. Write a program that instantiates objects of your classes and outputs each object's area (except Quadrilateral).

Exercise 9.8: Quadrilateral Inheritance Hierarchy Design an inheritance hierarchy with `Quadrilateral` as the superclass. Subclasses might include `Trapezoid`, `Parallelogram`, `Rectangle`, and `Square`. Use a `Point` class to represent the coordinates of the four endpoints.

9.9 (What Does Each Code Snippet Do?)
a) Assume that the following method call is located in an overridden earnings method in a subclass:
super.earnings()
b)Assume that the following line of code appears before a method declaration:
@Override
c)Assume that the following line of code appears as the first statement in a constructor's body:
super(firstArgument, secondArgument);

Exercise 9.9: `super` keyword usages Explain two usages of the `super` keyword, including its advantages. `super` is used to call a superclass's method (e.g., `super.earnings()`), or to invoke a superclass's constructor (e.g., `super(firstArgument, secondArgument)`). It's advantageous because it allows you to access and reuse the behavior or constructor of the superclass in a subclass.

9.10 (Write a Line of Code, or Two) Write up to two lines of code that perform each of the following tasks:
a) Specify that class Orange inherits from class Fruit.
b) Declare that you are going to override the toString method from inside the Orange class.
c) Call superclass Fruit's constructor from subclass Orange's constructor. Assume that the superclass constructor receives two Strings, one for shape, another for color; and an integer for the calories.

Exercise 9.10: Writing code lines Here are the lines of code for each task:

a) `class Orange extends Fruit;` b) `@Override` (inside the `Orange` class when you intend to override `toString`). c) `super(shape, color, calories);`

Exercise 9.10: Writing code lines Here are the lines of code for each task:

a) `class Orange extends Fruit;` b) `@Override` (inside the `Orange` class when you intend to override `toString`). c) `super(shape, color, calories);`

9.11 (Super) Explain two usages of the super keyword, and state some of the advantages of each type of usage.

Exercise 9.11: `super` keyword usages Explain two usages of the `super` keyword and their advantages.

1. Calling a superclass constructor with `super(...)` allows you to initialize the superclass's attributes or perform common setup. It ensures that the superclass's constructor is executed before the subclass's constructor.
2. Calling a superclass method with `super.methodName()` allows you to invoke and reuse the behavior of a method defined in the superclass. This promotes code reuse and can help avoid code duplication.

9.12 (Using super) A method decode() is declared in a parent class, and also the child. How can you access the decode() of the parent from within the child?

Exercise 9.12: Accessing the parent's method You can access the `decode()` method of the parent class from within the child class using `super.decode()`.

9.13 (Calling get Methods in a Class’s Body) In Figs. 9.10–9.11 methods earnings and toString each call various get methods within the same class. Explain the benefits of calling these get methods within the classes.

Exercise 9.13: Calling get methods within a class's body Calling get methods within a class's body can provide the following benefits:

- Encapsulation: It allows you to encapsulate the internal state of an object and provides controlled access to it, making the class more maintainable and secure.
- Code Reusability: By using get methods to access the object's state, you can easily reuse the same methods in different parts of the class or in subclasses, promoting code reuse and reducing redundancy.

9.14 (Employee Hierarchy) In this chapter, you studied an inheritance hierarchy in which class BasePlusCommissionEmployee inherited from class CommissionEmployee. However, not all types of

employees are CommissionEmployees. In this exercise, you’ll create a more general Employee superclass
that factors out the attributes and behaviors in class CommissionEmployee that are common to all Employees. The common attributes and behaviors for all Employees are firstName, lastName, socialSecurityNumber, getFirstName, getLastName, getSocialSecurityNumber and a portion of method toString. Create a new superclass Employee that contains these instance variables and methods and a constructor. Next, rewrite class CommissionEmployee from Section 9.4.5 as a subclass of Employee. Class CommissionEmployee should contain only the instance variables and methods that are not declared in superclass Employee. Class CommissionEmployee’s constructor should invoke class Employee’s
constructor and CommissionEmployee’s toString method should invoke Employee’s toString method. Once you’ve completed these modifications, run the CommissionEmployeeTest and BasePlusCommissionEmployeeTest apps using these new classes to ensure that the apps still display the same results for a CommissionEmployee object and BasePlusCommissionEmployee object, respectively.

Exercise 9.14: Employee Hierarchy Create a new superclass `Employee` with common attributes and behaviors. Rewrite `CommissionEmployee` as a subclass of `Employee`, invoking the superclass's constructor. Ensure that the test apps still work as expected.

9.15 (Creating a New Subclass of Employee) Other types of Employees might include SalariedEmployees who get paid a fixed weekly salary, PieceWorkers who get paid by the number of pieces they produce or HourlyEmployees who get paid an hourly wage with time-and-a-half—1.5 times the hourly wage—for hours worked over 40 hours.
Create class HourlyEmployee that inherits from class Employee (Exercise 9.14) and has instance variable hours (a double) that represents the hours worked, instance variable wage (a double) that represents the wages per hour, a constructor that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked, set and get methods for manipulating the hours and wage, an earnings method to calculate an HourlyEmployee ’s earnings based on the hours worked and a toString method that returns the HourlyEmployee ’s String representation. Method setWage should ensure that wage is nonnegative, and setHours should ensure that the value of hours is between 0 and 168 (the total number of hours in a week). Use class HourlyEmployee in a test program that’s similar to the one in Fig. 9.5.

Exercise 9.15: Creating HourlyEmployee class Create a `HourlyEmployee` class that inherits from `Employee` and includes attributes like `hours` and `wage`. Implement methods for calculating earnings, enforcing non-negative wages, and hours within a valid range. Test the `HourlyEmployee` class.

Excercise (Base solution – GPT 3.5)

Exercise 9.3: Using Composition Rather Than Inheritance To rewrite the `BasePlusCommissionEmployee` class using composition, create a new class to hold the "base salary" attribute and include an instance of that class within the `BasePlusCommissionEmployee` class. This is an example:

```java
java

class BaseSalary {
    private double baseSalary;

    public BaseSalary(double baseSalary) {
        this.baseSalary = baseSalary;
    }

    public double getBaseSalary() {
        return baseSalary;
    }

    public void setBaseSalary(double baseSalary) {
        this.baseSalary = baseSalary;
    }
}

class BasePlusCommissionEmployee {
    private CommissionEmployee commissionEmployee;
    private BaseSalary baseSalary;

    public BasePlusCommissionEmployee(String firstName, String lastName, String
socialSecurityNumber,
            double grossSales, double commissionRate, double baseSalary) {
        this.commissionEmployee = new CommissionEmployee(firstName, lastName,
socialSecurityNumber, grossSales, commissionRate);
        this.baseSalary = new BaseSalary(baseSalary);
    }

    public double earnings() {
        return commissionEmployee.earnings() + baseSalary.getBaseSalary();
    }

    // Include other methods as needed.
}
```

In this solution, we created a `BaseSalary` class to manage the base salary attribute. The `BasePlusCommissionEmployee` class uses composition by including an instance of `BaseSalary` and `CommissionEmployee`.

Exercise 9.4: Opportunities for Defects Inheritance can promote defect reduction by:

- Code Reusability: Inheritance allows you to create a common base class with shared functionality, reducing code duplication.
- Centralized Code: Common code is defined in one place (the superclass), making it easier to maintain and update.

- Polymorphism: Inheritance allows you to treat different subclasses as instances of the superclass, making it easier to work with a group of related objects.

However, misuse of inheritance can lead to defects when:

- The inheritance hierarchy is too deep or complex, leading to confusion.
- Subclasses override methods in a way that violates the Liskov Substitution Principle, causing unexpected behavior.
- Superclass changes affect multiple subclasses, potentially introducing defects in all of them.

Exercise 9.5: Student Inheritance Hierarchy Create an inheritance hierarchy for students at a university. Here's a simplified representation:

```java
class Student {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

class UndergraduateStudent extends Student {
    public UndergraduateStudent(String name) {
        super(name);
    }
}

class GraduateStudent extends Student {
    public GraduateStudent(String name) {
        super(name);
    }
}

class Freshman extends UndergraduateStudent {
    public Freshman(String name) {
        super(name);
    }
}

class Sophomore extends UndergraduateStudent {
    public Sophomore(String name) {
        super(name);
    }
}

// Add more subclasses as needed (e.g., Junior, Senior, DoctoralStudent,
MastersStudent)
```

In this hierarchy, `Student` is the superclass, and it has two main subclasses, `UndergraduateStudent` and `GraduateStudent`. You can continue to add more levels of subclasses as per your requirements, such as `Freshman`, `Sophomore`, `Junior`, `Senior`, `DoctoralStudent`, and `MastersStudent`.

Exercise 9.6: Shape Inheritance Hierarchy Create an inheritance hierarchy for shapes, including two-dimensional and three-dimensional shapes. This is a simplified representation:

```java
class Shape {
    // Common properties and methods for all shapes.
}

class TwoDimensionalShape extends Shape {
    // Common properties and methods for 2D shapes.
}

class ThreeDimensionalShape extends Shape {
    // Common properties and methods for 3D shapes.
}

class Quadrilateral extends TwoDimensionalShape {
    // Properties and methods specific to quadrilaterals.
}

class Triangle extends TwoDimensionalShape {
    // Properties and methods specific to triangles.
}

class Circle extends TwoDimensionalShape {
    // Properties and methods specific to circles.
}

class Sphere extends ThreeDimensionalShape {
    // Properties and methods specific to spheres.
}

class Cylinder extends ThreeDimensionalShape {
    // Properties and methods specific to cylinders.
}

// Add more subclasses as necessary (e.g., Rectangle, Square, etc.)
```

This hierarchy defines a basic structure for shapes with `Shape` as the top-level superclass, and subclasses for two-dimensional and three-dimensional shapes. You can add more specific shapes and properties as needed.

Exercise 9.7: `protected` vs. `private` The choice between using `protected` and `private` access depends on the design and requirements:

- `protected`: Use `protected` when you want to provide access to the member within the class and its subclasses. It's useful for sharing implementation details with subclasses, enabling them to extend or override methods while maintaining some level of encapsulation.

- `private`: Use `private` when you want to restrict access to the member to only the class where it's defined. It enforces strong encapsulation by hiding the implementation details from both subclasses and external classes.

The relative merits depend on the specific design goals and the level of control you need over access to class members. Use `protected` when you intend to design for extensibility, and `private` when you want to tightly control access and hide implementation details.

Exercise 9.8: Quadrilateral Inheritance Hierarchy Here's a simplified inheritance hierarchy for quadrilaterals:

```
java

class Quadrilateral {
    private Point[] coordinates = new Point[4];

    public Quadrilateral(Point p1, Point p2, Point p3, Point p4) {
        coordinates[0] = p1;
        coordinates[1] = p2;
        coordinates[2] = p3;
        coordinates[3] = p4;
    }

    public Point[] getCoordinates() {
        return coordinates;
    }

    public double area() {
        // Calculate and return the area of the quadrilateral.
        return 0.0; // Placeholder value.
    }
}

class Trapezoid extends Quadrilateral {
    public Trapezoid(Point p1, Point p2, Point p3, Point p4) {
        super(p1, p2, p3, p4);
    }

    @Override
    public double area() {
        // Calculate and return the area of the trapezoid.
        return 0.0; // Placeholder value.
    }
}

class Parallelogram extends Quadrilateral {
    public Parallelogram(Point p1, Point p2, Point p3, Point p4) {
        super(p1, p2, p3, p4);
    }

    @Override
    public double area() {
        // Calculate and return the area of the parallelogram.
        return 0.0; // Placeholder value.
    }
}
```

```java
class Rectangle extends Parallelogram {
    public Rectangle(Point p1, Point p2, Point p3, Point p4) {
        super(p1, p2, p3, p4);
    }

    @Override
    public double area() {
        // Calculate and return the area of the rectangle.
        return 0.0; // Placeholder value.
    }
}

class Square extends Rectangle {
    public Square(Point p1, Point p2, Point p3, Point p4) {
        super(p1, p2, p3, p4);
    }

    @Override
    public double area() {
        // Calculate and return the area of the square.
        return 0.0; // Placeholder value.
    }
}
```

In this hierarchy, `Quadrilateral` is the superclass, and subclasses include `Trapezoid`, `Parallelogram`, `Rectangle`, and `Square`. Each subclass overrides the `area` method to calculate its specific area.

Exercise 9.9: `super` keyword usages a) `super.earnings()`: This line, when located in an overridden `earnings` method in a subclass, calls the `earnings` method of the superclass. It allows you to reuse the earnings calculation of the superclass and potentially add additional logic in the subclass.

b) `@Override`: This annotation is placed before a method declaration in a subclass to indicate that you intend to override a method from the superclass. It helps ensure that you're correctly overriding a method, reducing the risk of accidental mistakes.

Exercise 9.10: Writing code lines Here are the lines of code for each task:

a) `class Orange extends Fruit;`: This line specifies that the `Orange` class inherits from the `Fruit` class.

b) `@Override`: This annotation is used inside the `Orange` class when you intend to override the `toString` method.

c) `super(shape, color, calories);`: This line calls the superclass constructor from the `Orange` class's constructor, passing the required arguments.

Exercise 9.11: Super Keyword Usages and Advantages The `super` keyword is used in two main contexts:

1. Calling Superclass Constructor: `super(firstArgument, secondArgument);`

- Usage: This is used within a subclass's constructor to call the constructor of its superclass (parent class).
- Advantages:
  - Ensures that the initialization code in the superclass constructor is executed, setting up the initial state of the object.
  - Allows the subclass to perform additional initialization specific to itself after the superclass's constructor has run.

2. Calling Superclass Method: `super.earnings();`

- Usage: This is used within a subclass's method to call a method of the superclass.
- Advantages:
  - Enables reuse of the behavior defined in the superclass. The subclass can extend or modify this behavior as needed.
  - Helps avoid code duplication by centralizing common functionality in the superclass.

Exercise 9.12: Accessing Parent's Method To access the `decode()` method of the parent class from within the child class, you can use the `super` keyword followed by the method name, like `super.decode()`. This calls the parent class's `decode()` method, allowing you to reuse or extend its behavior in the child class.

Exercise 9.13: Calling Get Methods Within a Class's Body Calling get methods within a class's body provides several benefits:

- **Encapsulation**: It enforces the encapsulation principle, where class attributes are private and can only be accessed and modified through controlled get and set methods. This helps protect the internal state of the class from unintended modifications.
- **Code Reusability**: By using get methods, you can access the object's attributes consistently throughout the class's methods. This promotes code reusability, as the same get methods can be used in multiple places within the class.
- **Maintainability**: Using get methods provides a clear and consistent way to access the class's attributes, making the code more maintainable. If you need to change the internal representation of an attribute, you only need to update the get method, and the rest of the code remains unaffected.

Exercise 9.14: Employee Hierarchy To create a more general `Employee` superclass, you can factor out common attributes and methods from the `CommissionEmployee` class. Here's a basic example:

```java
class Employee {
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;

    public Employee(String firstName, String lastName, String socialSecurityNumber)
{
```

```
        this.firstName = firstName;
        this.lastName = lastName;
        this.socialSecurityNumber = socialSecurityNumber;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getSocialSecurityNumber() {
        return socialSecurityNumber;
    }

    @Override
    public String toString() {
        return "Name: " + firstName + " " + lastName + "\nSSN: " +
socialSecurityNumber;
    }
}

class CommissionEmployee extends Employee {
    private double grossSales;
    private double commissionRate;

    public CommissionEmployee(String firstName, String lastName, String
socialSecurityNumber,
        double grossSales, double commissionRate) {
        super(firstName, lastName, socialSecurityNumber);
        this.grossSales = grossSales;
        this.commissionRate = commissionRate;
    }

    // ... Implement earnings method and other specific methods
}
```

In this example, `Employee` is a superclass that contains common attributes and methods shared by all employees. `CommissionEmployee` is a subclass that extends `Employee` and adds attributes specific to commission-based employees.

Exercise 9.15: Creating HourlyEmployee Class Create the `HourlyEmployee` class by inheriting from the `Employee` class. Here's an example implementation:

```java
class HourlyEmployee extends Employee {
    private double hours;
    private double wage;

    public HourlyEmployee(String firstName, String lastName, String
socialSecurityNumber,
                          double hours, double wage) {
```

```java
        super(firstName, lastName, socialSecurityNumber);
        setHours(hours);
        setWage(wage);
    }

    public double getHours() {
        return hours;
    }

    public void setHours(double hours) {
        if (hours >= 0 && hours <= 168) {
            this.hours = hours;
        } else {
            throw new IllegalArgumentException("Hours must be between 0 and 168.");
        }
    }

    public double getWage() {
        return wage;
    }

    public void setWage(double wage) {
        if (wage >= 0) {
            this.wage = wage;
        } else {
            throw new IllegalArgumentException("Wage must be non-negative.");
        }
    }

    public double earnings() {
        if (hours <= 40) {
            return hours * wage;
        } else {
            return (40 * wage) + ((hours - 40) * wage * 1.5);
        }
    }

    @Override
    public String toString() {
        return super.toString() + "\nHours: " + hours + "\nWage: " + wage;
    }
}

public class HourlyEmployeeTest {
    public static void main(String[] args) {
        HourlyEmployee hourlyEmployee = new HourlyEmployee("John", "Doe", "123-45-
6789", 45, 12.50);
        System.out.println(hourlyEmployee);

        System.out.println("Earnings: $" + hourlyEmployee.earnings());
    }
}
```

In this example, `HourlyEmployee` is a subclass of the `Employee` class and includes attributes for hours worked and wage per hour. The `earnings` method calculates the earnings based on the number of hours worked and the wage. The `setHours` and `setWage` methods ensure that the values are within valid ranges. The `toString` method provides a textual representation of the `HourlyEmployee`.

The `HourlyEmployeeTest` class demonstrates the usage of the `HourlyEmployee` class.

I hope this helps!

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts.
ChatGPT September 25 Version