Chapter 7: Model Solutions

Below, you'll find sample solutions to the lab exercises in the book.

Lab Exercises 7.1

In this solution, we define a class called BankAccount with the specified attributes and methods. The __init__ method serves as the constructor and initializes the account attributes.

The deposit method adds the given amount to the account balance, while the withdraw method subtracts the given amount from the account balance if the balance is sufficient. If the balance is not sufficient, an error message is displayed.

The display_info method prints the account details: account number, account holder's name, and account balance.

We create an instance of the BankAccount class, account1, and test the methods by depositing, withdrawing, and displaying the account information.

```
class BankAccount:
    def __init__(self, account_number, account_holder,
                 initial_balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.account_balance = initial_balance

    def deposit(self, amount):
        self.account_balance += amount

    def withdraw(self, amount):
        if self.account_balance >= amount:
            self.account_balance -= amount
        else:
            print("Insufficient balance. Withdrawal denied.")

    def display_info(self):
        print("Account Number:", self.account_number)
        print("Account Holder:", self.account_holder)
        print("Account Balance:", self.account_balance)
```

```
# Create objects and test the methods
account1 = BankAccount("123456789", "John Doe", 1000)
account1.display_info()

account1.deposit(500)
account1.display_info()

account1.withdraw(200)
account1.display_info()

account1.withdraw(1500)  # insufficient balance
account1.display_info()
```

Lab Exercises 7.2

In this solution, we define a parent class called Shape with the given attributes and a method called display_info().

The Rectangle class and Circle class inherit from the Shape class using the super() function. They have additional attributes (width and height for Rectangle, radius for Circle) and a method called calculate_area() to calculate the area of the shape.

We create objects rectangle and circle, and test the methods by displaying the shape information and calculating the area.

You can create additional instances of Rectangle and Circle classes, and test the methods further as needed.

```
import math


class Shape:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def display_info(self):
        print("Shape:", self.name)
        print("Color:", self.color)

class Rectangle(Shape):
    def __init__(self, name, color, width, height):
        super().__init__(name, color)
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, name, color, radius):
        super().__init__(name, color)
        self.radius = radius
```

```
    def calculate_area(self):
        return math.pi * self.radius ** 2

# Create objects and test the methods
rectangle = Rectangle("Rectangle", "Blue", 4, 6)
rectangle.display_info()
print("Area:", rectangle.calculate_area())

circle = Circle("Circle", "Red", 3)
circle.display_info()
print("Area:", circle.calculate_area())
```

Lab Exercises 7.3

In this solution, we define a parent class called Vehicle with the attributes brand and year. It also has the methods display_info() to print the brand and year, and start_engine() to print a generic message for starting the engine.

The Car and Motorcycle classes inherit from the Vehicle class and override the start_engine() method to print specific messages for starting a car engine and a motorcycle engine, respectively.

We create objects of the classes and test the methods by displaying the vehicle information and starting the engine.

```
class Vehicle:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def display_info(self):
        print("Brand:", self.brand)
        print("Year:", self.year)

    def start_engine(self):
        print("Starting the engine of the vehicle.")

class Car(Vehicle):
    def start_engine(self):
        print("Starting the car engine.")

class Motorcycle(Vehicle):
    def start_engine(self):
        print("Starting the motorcycle engine.")

# Create objects and test the methods
vehicle = Vehicle("Generic Brand", 2022)
vehicle.display_info()
vehicle.start_engine()

car = Car("Toyota", 2019)
```

```
car.display_info()
car.start_engine()

motorcycle = Motorcycle("Honda", 2020)
motorcycle.display_info()
motorcycle.start_engine()
```

Lab Exercises 7.4

1. Object-oriented programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes. OOP focuses on creating objects that have their own state (attributes) and behavior (methods), and allows for code reuse and modularity. In OOP, the emphasis is on modeling real-world entities as objects and interacting between them.

   Procedural programming, focuses on writing procedures or functions that manipulate data. It is based on a step-by-step execution of instructions. Procedural programming does not involve the concept of objects and classes.

2. The purpose of creating a parent class (such as Shape) in object-oriented programming is to provide a blueprint or template that can be used to create derived or child classes. The parent class contains common attributes and behaviors that are shared among the child classes. It helps in achieving code reusability and promotes a hierarchical structure for organizing and managing classes.

3. Inheritance is a fundamental concept in object-oriented programming that allows a child class to inherit properties (attributes) and behaviors (methods) from a parent class. The child class is said to inherit the characteristics of the parent class. It enables code reuse and promotes the concept of "is-a" relationship.

   By inheriting from a parent class, the child class automatically gets access to the attributes and methods defined in the parent class. This means the child class can use and override those attributes and methods without needing to redefine them. Inheritance allows for the extension and specialization of classes, as child classes can add new attributes and methods or override the existing ones.

4. Shape with child classes

```
import math


class Shape:
    def area(self):
        pass


class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius


    def area(self):
        return math.pi * self.radius**2


class Square(Shape):
    def __init__(self, side):
```

```
        self.side = side

    def area(self):
        return self.side**2

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return (self.base * self.height) / 2

# Create objects of child classes
circle = Circle(5)
square = Square(4)
triangle = Triangle(3, 6)

# Calculate and print the areas
print("Circle Area:", circle.area())
print("Square Area:", square.area())
print("Triangle Area:", triangle.area())
```

5. See number 4
6. See number 4
7. Encapsulation is one of the fundamental principles of object-oriented programming. It refers to the bundling of data (attributes) and methods (behaviors) within a class. Encapsulation allows for the hiding of internal details and provides controlled access to the data and methods through the class's public interface.

   By encapsulating data, we can ensure data integrity and prevent direct access or modification from outside the class. The attributes are typically made private or protected, and access to them is provided through getter and setter methods. Encapsulation helps in maintaining data consistency, improving code maintainability, and promoting code reusability.

8. Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as objects of a common parent class. It allows the same method or operator to behave differently depending on the object on which it is called.

   Polymorphism enables code to be written in a more generic and flexible manner, as it allows for the substitution of objects without affecting the behavior of the program. It promotes code reusability and modularity.

   Polymorphism can be achieved through method overriding (providing different implementations of a method in child classes) and method overloading (defining multiple methods with the same name but different parameter lists).

9. Method overriding is a feature in object-oriented programming that allows a child class to provide a different implementation of a method defined in its parent class. When a child class overrides a method, it provides a specialized implementation of that method specific to its own behavior.

   In method overriding, both the parent and child classes have methods with the same name and the same number and type of parameters. When the method is called on an object of the child class, the overridden method in the child class is executed instead of the parent class method.

   Method overriding allows for the customization and extension of inherited methods. It is an essential aspect of polymorphism in object-oriented programming.