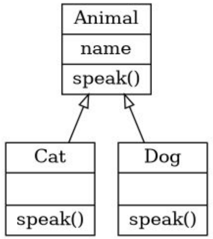
Principles of Software Engineering and Data Bases

Davide Yi Xian Hu
Email: davideyi.hu@polimi.it
Date: 04 December 2024
Exercise Lecture: 07 - Object Oriented Programming

POLITECNICO
MILANO 1863

Exercise 1 - Inheritance

```
class Animal:

    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):
    def speak(self):
        return f"{self.name} barks."

class Cat(Animal):
    def speak(self):
        return f"{self.name} meows."
```

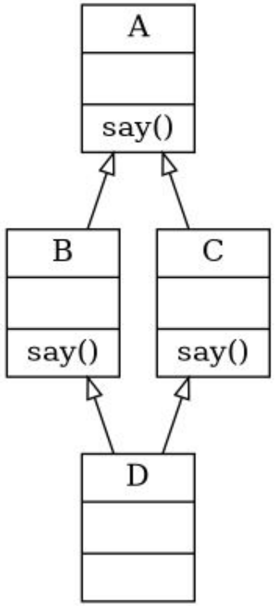


```
# Example Usage
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak())
print(cat.speak())
```

Exercise 2 - Multiple Inheritance

What does this code prints?

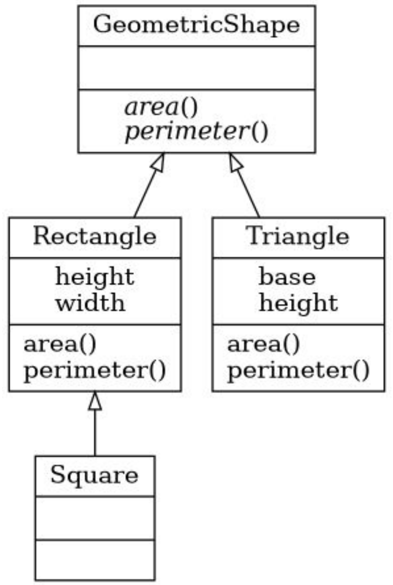
```
class A:
    def say(self):
        return "A"

class B(A):
    def say(self):
        return "B"

class C(A):
    def say(self):
        return "C"

class D(B, C):
    pass

d = D()
print(d.say())
```



Exercise 3 - Geometric Figures

Create an abstract class GeometricShape with two methods: area() and perimeter().

Implement subclasses Rectangle, Square, and Triangle that calculate their respective areas and perimeters.



Exercise 4 - Instance and Subclass

What does this code prints?

```
class Animal:
    pass

class Dog(Animal):
    pass


dog = Dog()
print(isinstance(dog, Dog))
print(isinstance(dog, Animal))
print(isinstance(Dog, Animal))
print(issubclass(Dog, Animal))
print(issubclass(Animal, Dog))
print(issubclass(dog, Animal))
```

Exercise 5 - Array of Integers

Create a class IntegerArray to represent an array of integers.
Override the __add__ and __sub__ methods to support:

Addition (+):
Combine two IntegerArray objects
by adding their respective elements.
If the arrays have different lengths, fill missing values with zeros.
Subtraction (-):
Subtract the elements of one
IntegerArray from another.
Similar to addition, fill missing values with
zeros if the arrays have different lengths.

Exercise 6 - Vehicle Management

Design a system to represent vehicles using classes.

Implement a Vehicle base class with subclasses **Car**, **Motorcycle**, and **Truck**.

Each vehicle has an **Engine** object and
defines methods to calculate maximum speed and fuel efficiency.

Theory - SOLID Principles

Single Responsibility Principle
A class should have only one reason to change.

Open/Closed Principle
Software entities should be open for extension but closed for modification.

Liskov Substitution Principle
Subtypes must be substitutable for their base types.

Interface Segregation Principle
Clients should not be forced to depend on interfaces they do not use.

Dependency Inversion Principle
Depend on abstractions, not on concrete implementations.

Exercise 7 - Single Responsibility

```
class Report:
    def generate_data(self):
        # Simulate data generation
        self.data = {"sales": 1000, "profit": 200}

    def format_as_json(self):
        import json
        return json.dumps(self.data)

    def save_to_file(self, filename):
        with open(filename, "w") as file:
            file.write(self.format_as_json())
```

Solution 7 - Single Responsibility

```
class DataGenerator:
    def generate_data(self):
        return {"sales": 1000, "profit": 200}

class DataFormatter:
    def format_as_json(self, data):
        import json
        return json.dumps(data)

class FileSaver:
    def save_to_file(self, content, filename):
        with open(filename, "w") as file:
            file.write(content)

class Report:
    def __init__(self):
        self.data_generator = DataGenerator()
        self.data_formatter = DataFormatter()
        self.file_saver = FileSaver()

    def create_report(self, filename):
        data = self.data_generator.generate_data()
        formatted_data = self.data_formatter.format_as_json(data)
        self.file_saver.save_to_file(formatted_data, filename)
```

Exercise 8 - Open/Closed

```
class Discount:
    def apply_discount(self, type, amount):
        if type == "student":
            return amount * 0.9
        elif type == "senior":
            return amount * 0.8
        elif type == "regular":
            return amount
        else:
            raise ValueError("Unknown discount type")
```

Exercise 8 - Open/Closed

```
from abc import ABC, abstractmethod

class Discount(ABC):
    @abstractmethod
    def apply_discount(self, amount):
        pass

class StudentDiscount(Discount):
    def apply_discount(self, amount):
        return amount * 0.9

class SeniorDiscount(Discount):
    def apply_discount(self, amount):
        return amount * 0.8

class RegularDiscount(Discount):
    def apply_discount(self, amount):
        return amount

def calculate_price(discount: Discount, amount):
    return discount.apply_discount(amount)
```

Exercise 9 - Liskov Substitution

```
class Payment:
    def process_payment(self, amount):
        raise NotImplementedError

class CreditCardPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

class PayPalPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing PayPal payment of ${amount}")

    def login_to_paypal_account(self, email, password):
        print("Logging into PayPal...")

def process_all_payments(payments, amount):
    for payment in payments:
        payment.process_payment(amount)

payments = [
    CreditCardPayment(),
    PayPalPayment()
]

payments[1].login_to_paypal_account("user@example.com", "password")
process_all_payments(payments, 100)
```

Solution 9 - Liskov Substitution

```
class Payment:
    def process_payment(self, amount):
        raise NotImplementedError

class CreditCardPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

class PayPalPayment(Payment):
    def process_payment(self, amount):
        self._login_to_paypal_account()
        print(f"Processing PayPal payment of ${amount}")

    def _login_to_paypal_account(self):
        print("Logging into PayPal...")
```

Exercise 10 - Interface Segregation

```
# Large interface
class Machine:
    def print(self, document):
        raise NotImplementedError

    def scan(self, document):
        raise NotImplementedError

    def fax(self, document):
        raise NotImplementedError


# Printer class forced to implement methods it doesn't use
class Printer(Machine):
    def print(self, document):
        print(f"Printing: {document}")

    def scan(self, document):
        raise NotImplementedError("Printer cannot scan")

    def fax(self, document):
        raise NotImplementedError("Printer cannot fax")
```

Solution 10 - Interface Segregation

```python
# Smaller, specific interfaces
class Printable:
    def print(self, document):
        raise NotImplementedError

class Scannable:
    def scan(self, document):
        raise NotImplementedError

class Faxable:
    def fax(self, document):
        raise NotImplementedError
```

```python
class Printer(Printable):
    def print(self, document):
        print(f"Printing: {document}")

class Scanner(Scannable):
    def scan(self, document):
        print(f"Scanning: {document}")

class MultiFunctionPrinter(Printable,
Scannable, Faxable):
    def print(self, document):
        print(f"Printing: {document}")

    def scan(self, document):
        print(f"Scanning: {document}")

    def fax(self, document):
        print(f"Faxing: {document}")
```

Exercise 11 - Dependency Inversion

```
class EmailService:
    def send(self, message):
        print(f"Sending email: {message}")

class Notification:
    def __init__(self):
        self.email_service = EmailService()
    def notify(self, message):
        self.email_service.send(message)
```

What if we want to add SMS Service?

Solution 11 - Dependency Inversion

```python
from abc import ABC, abstractmethod

class NotificationService(ABC):
    @abstractmethod
    def send(self, message):
        pass

class EmailService(NotificationService):
    def send(self, message):
        print(f"Sending email: {message}")

class SMSService(NotificationService):
    def send(self, message):
        print(f"Sending SMS: {message}")

class Notification:
    def __init__(self, service: NotificationService):
        self.service = service  # Inject dependency

    def notify(self, message):
        self.service.send(message)
```