Lecture slides - Week 8

OOP - Classes, Objects and Encapsulation

Dr. Aamir Akbar

Director of both AWKUM AI Lab and AWKUM Robotics, Final Year Projects (FYPs) coordinator, and lecturer at the department of Computer Science
Abdul Wali Khan University, Mardan (AWKUM)

Contents

A Class in OOP

What is a Class, and how to choose a Class?

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will have.

- A class must represent a single concept from the problem domain
- Name for a class should be a noun that describes concept
- A noun represents a person, place, thing, or idea. When naming a class in OOP, a noun typically describes a concept or an entity. It signifies what the object (an instance of that class) is or represents.

Examples: Customer, Car, BankAccount, Book, CompactDisc, Library.

Each of these nouns signifies a distinct entity that can have attributes and behaviors associated with it. In OOP, these nouns are often used as class names to represent the blueprint for objects that share common characteristics.

How to design a Class?

1. A class should encapsulate a concept.

A class is like a box that holds together all the information and actions related to a particular idea or thing. It encapsulates (or wraps up) everything about that concept, making it easy to manage and work with.

How to design a Class?

1. A class should encapsulate a concept.

A class is like a box that holds together all the information and actions related to a particular idea or thing. It encapsulates (or wraps up) everything about that concept, making it easy to manage and work with.

2. The attributes of the class stored information about this concept.

In a class, the attributes are like containers that hold the details and facts about that idea. They store all the important information related to the concept the class represents.

How to design a Class?

1. A class should encapsulate a concept.

A class is like a box that holds together all the information and actions related to a particular idea or thing. It encapsulates (or wraps up) everything about that concept, making it easy to manage and work with.

2. The attributes of the class stored information about this concept.

In a class, the attributes are like containers that hold the details and facts about that idea. They store all the important information related to the concept the class represents.

3. Methods manipulates the information in the class

Methods in a class are like the tools that work with the stored information. They change, use, or do things with the data (i.e., attributes) stored in the class to perform specific actions.

How to create a Class in Python?

This following example demonstrates a Car class with attributes (make, model, and year) and a method (start_engine).

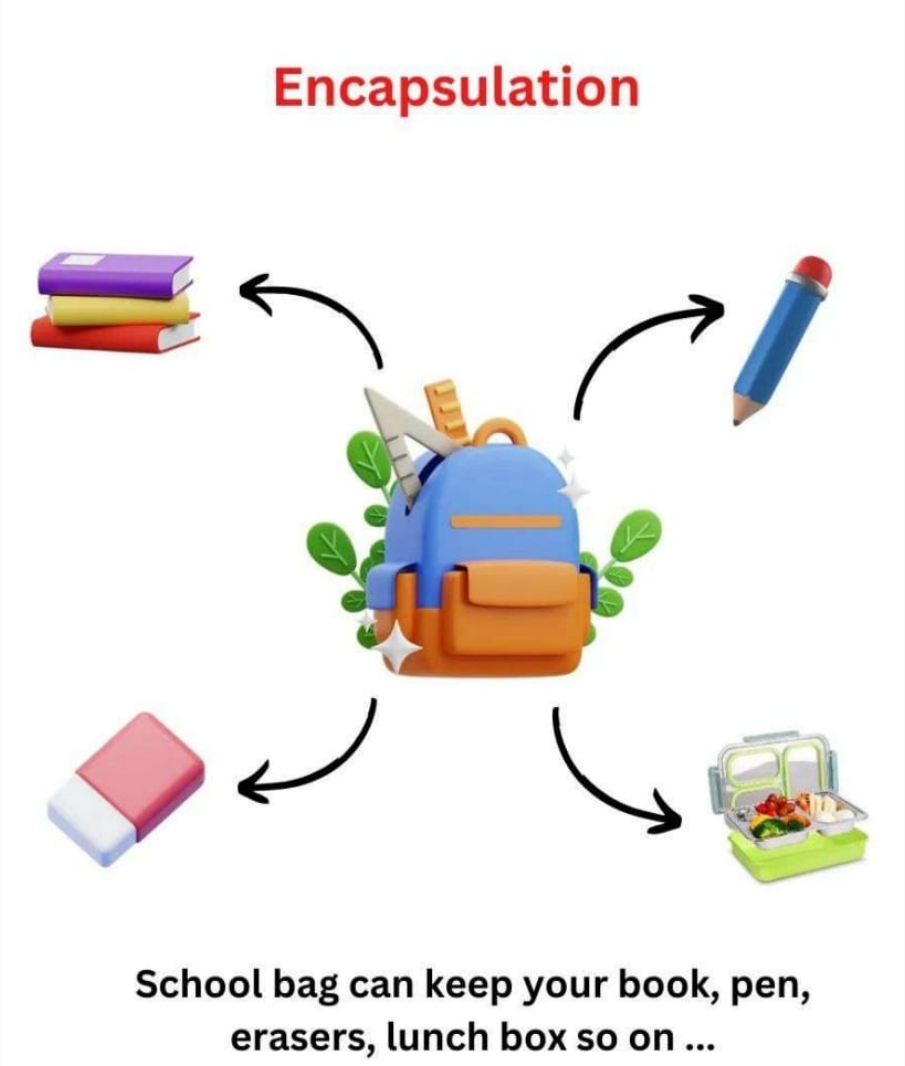
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        # Method to start the car's engine
        print(f"Engine started for {self.make} {self.model} {self.year}")
```

Constructor of a Class:

In Python, the `__init__` method is a special method, often referred to as the constructor. It's automatically called when an object (or an instance) of a class is created.

Encapsulation

What is Encapsulation? i



What is Encapsulation? ii

In object-oriented programming, encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit, i.e., a class.

- Data Hiding: Encapsulation hides the internal state of objects from the outside and only exposes what's necessary.
- Access Control: It enables controlling access to certain parts of the code, protecting data from unwanted interference.
- Enhanced Modularity: Encapsulation promotes modular design, where changes within the class don't affect the external code.

Example: In the Car class, encapsulation is demonstrated by bundling the data (attributes like make, model, and year) and methods (like start_engine) into a single unit.

What about the `self` keyword?

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        # Method to start the car's engine
        print(f"Engine started for {self.make} {self.model} {self.year}")
```

In Python, the `self` keyword refers to the instance of the class itself. `self` is important in encapsulation as it refers to the instance's own attributes and methods, enabling controlled access to them.

It keeps the instance-specific data within the scope of that instance, preventing unwanted interference from outside the class. For example, `self.make` refers to the `make` attribute of the specific instance.

Objects

Creating an object or instance of the Car class i

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        # Method to start the car's engine
        print("Engine started for", self.make, self.model, self.year)

# Creating an instance of the Car class
my_car = Car("Toyota", "Corolla", 2022)

# Accessing attributes and calling methods
print(my_car.make)  # Output: Toyota
print(my_car.model)  # Output: Corolla
my_car.start_engine()  # Output: Engine started for Toyota Corolla 2022
```

Creating an object or instance of the Car class ii

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        # Method to start the car's engine
        print("Engine started for", self.make, self.model, self.year)

# Creating two instances of the Car class
car1 = Car("Toyota", "Corolla", 2022)
car2 = Car("Honda", "Civic", 2023)

# Accessing attributes and calling methods for car1
print(car1.make)  # Output: Toyota
print(car1.model)  # Output: Corolla
car1.start_engine()  # Output: Engine started for Toyota Corolla 2022

# Accessing attributes and calling methods for car2
print(car2.make)  # Output: Honda
print(car2.model)  # Output: Civic
car2.start_engine()  # Output: Engine started for Honda Civic 2023
```

Object State: Car Instances

Car 1 State

Attribute	Value
Make	Toyota
Model	Corolla
Year	2022

Car 2 State

Attribute	Value
Make	Honda
Model	Civic
Year	2023

Modified Car Class

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.engine_started = False

    def start_engine(self):
        self.engine_started = True
        print("Engine started for", self.make, self.model, self.year)

    def stop_engine(self):
        self.engine_started = False
        print("Engine stopped for", self.make, self.model, self.year)
```

Object State: Car Instances

Before Method Calls

Car 1 State

Attribute	Value
Make	Toyota
Model	Corolla
Year	2022
Engine Started	False

Car 2 State

Attribute	Value
Make	Honda
Model	Civic
Year	2023
Engine Started	False

After start_engine() Method Calls

Car 1 State

Attribute	Value
Make	Toyota
Model	Corolla
Year	2022
Engine Started	True

Car 2 State

Attribute	Value
Make	Honda
Model	Civic
Year	2023
Engine Started	False

Case Study: Bank Account

Case Study: Bank Accounts i

Class Design:

We are going to create a program that simulates transactions in a bank account. A bank account has a balance and an interest rate and we can deposit money, withdraw money, get the balance of the bank account or calculate the interest of the deposited money.

Activity 1: Identify class name, attributes and methods

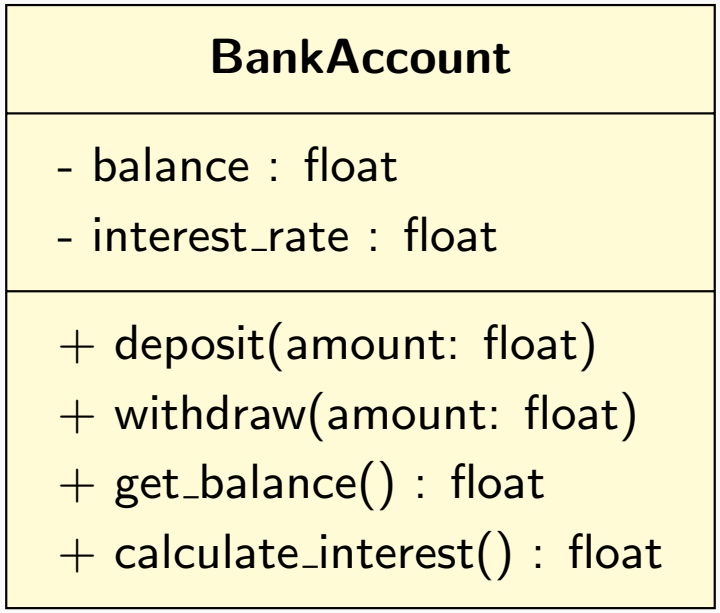
Case Study: Bank Accounts ii

Class Design:

We are going to create a program that simulates transactions in a bank account. A bank account has a `balance` and an `interest rate` and we can `deposit money`, `withdraw money`, `get the balance` of the bank account or `calculate the interest` of the deposited money.

Activity 2: Draw a UML class diagram of the bank account.

UML Class Diagram: BankAccount i



Encapsulation: Data hiding and Access control

- + (Public Access): Attributes or methods marked with a + symbol are public. They can be accessed and utilized from outside the class, providing a clear interface for interaction. `deposit`, `withdraw`, `get_balance`, and `calculate_interest` are marked as public methods, implying that they are accessible and usable from outside the BankAccount class.

UML Class Diagram: BankAccount ii

- - (Private Access): Attributes or methods marked with a - symbol are private. They are accessible and modifiable only from within the class itself. `balance` and `interest_rate` are marked as private attributes, indicating that they are not directly accessible from outside the BankAccount class.

Activity 3: create the `BankAccount` class in Python

BankAccount Class

```
1  class BankAccount:
2      def __init__(self, bal, int_rate):
3          self.__balance = bal
4          self.__interest_rate = int_rate
5
6      def deposit(self, amount):
7          self.__balance += amount
8
9      def withdraw(self, amount):
10         if self.__balance >= amount:
11             self.__balance -= amount
12         else:
13             print("Insufficient funds")
14
15     def get_balance(self):
16         return self.__balance
17
18     def calculate_interest(self):
19         return self.__balance * self.__interest_rate
```

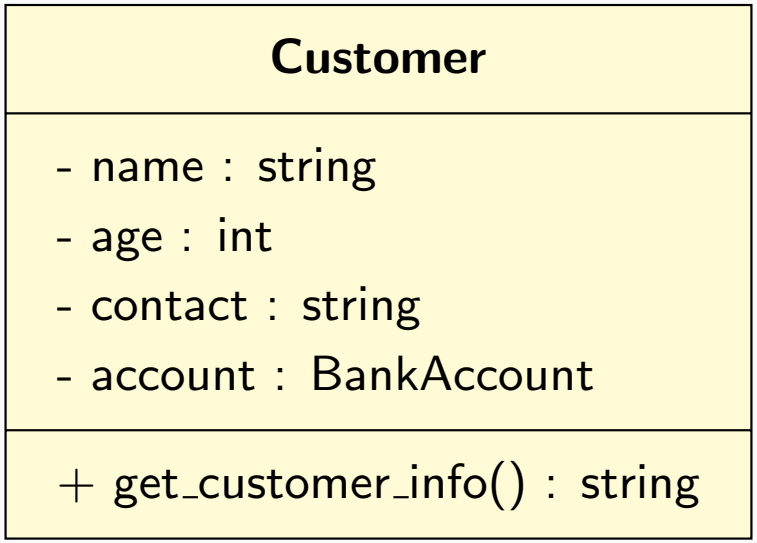
Activity 4: create objects of the BankAccount class (Demo)

Account holder

Now, we need a class for an individual with attributes related to personal details and a linked BankAccount. This class can maintain information such as name, age, and contact details. The person can access functionalities related to their associated BankAccount enabling actions like depositing money, withdrawing funds, obtaining account balance, and calculating interest.

Activity 5: create UML class diagram and Implement the class in Python.

UML Diagram of Customer



Customer Class ii

```
class Customer:
    def __init__(self, name, age, contact, account):
        self.__name = name
        self.__age = age
        self.__contact = contact
        self.__account = account

    def get_customer_info(self):
        return f"Name: {self.__name}, Age: {self.__age}, Contact: {self.__contact}"
```

Download Source Codes

Download Source Codes

You can download the source codes used in the previous slides from the following link:

▸ Download Source Codes

You can import the codes into your preferred development environment, such as Visual Studio Code (VS Code), to practice and explore further.