Al-Mustaqbal University
College of Sciences
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم
قسم الانظمة الطبية الذكية

Lecture: (7)

Encapsulation and Data Hiding in OOP

Subject: Object oriented programming I

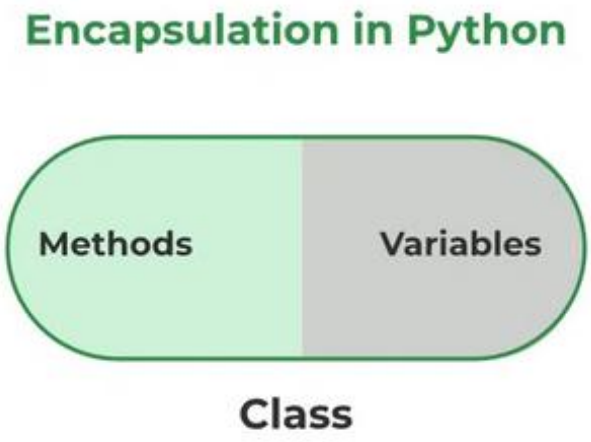
Class: Second

Dr. Maytham N. Meqdad

Encapsulation and Data Hiding in OOP

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as **private variables.**

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc. The goal of information hiding is to ensure that an object's state is always valid by controlling access to attributes that are hidden from the outside world.



Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when due to some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are

wrapped under a single name “sales section”. Using encapsulation also hides the data. In this example, the data of the sections like sales, finance, or accounts are hidden from any other section.

Protected members

Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore “_”**.

Although the protected variable can be accessed out of the class as well as in the derived class (modified too in derived class), it is customary(convention not a rule) to not access the protected out the class body.

```
# Python program to
# demonstrate protected members

# Creating a base class
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",   self._a)

        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class: ",
              self._a)


obj1 = Derived()
obj2 = Base()
```

```
# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```

Output:

```
Calling protected member of base class:  2
Calling modified protected member outside class:  3
Accessing protected member of obj1:  3
Accessing protected member of obj2:  2
```

EX: This program simulates a patient management system for a medical clinic, where sensitive patient data (like medical history and diagnosis) is protected using encapsulation. This allows access only through specific methods, ensuring that data remains secure and cannot be accidentally modified.

```
class Patient:
    def __init__(self, name, age, medical_history):
        self.name = name            # Public attribute
        self.age = age              # Public attribute
        self._medical_history = medical_history  # Protected attribute
        self.__diagnosis = None     # Private attribute

    # Method to access medical history
    def get_medical_history(self):
        return self._medical_history

    # Method to update medical history
    def add_medical_history(self, record):
        self._medical_history.append(record)

    # Method to set a diagnosis (only accessible within the class)
    def set_diagnosis(self, diagnosis):
        self.__diagnosis = diagnosis

    # Method to get the diagnosis (only accessible within the class)
    def get_diagnosis(self):
```

```
        return self.__diagnosis

    # Display patient information
    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")
        print("Medical History:", self.get_medical_history())
        print("Diagnosis:", self.get_diagnosis())

# Simulating the medical system
# Creating a patient with some initial medical history
patient1 = Patient("John Doe", 45, ["High Blood Pressure", "Diabetes"])

# Accessing protected medical history via method
print("Patient's Initial Medical History:", patient1.get_medical_history())

# Adding a new record to the medical history
patient1.add_medical_history("Cholesterol Issue")

# Setting a diagnosis using the encapsulated method
patient1.set_diagnosis("Stable Condition")

# Displaying patient info (using the encapsulated display method)
patient1.display_info()

# Accessing the private diagnosis directly (should not be done)
# print(patient1.__diagnosis)  # This will raise an AttributeError
```

https://www.programiz.com/online-compiler/1VrTcOTxZGKcg

Explanation

1. **Encapsulation**:
   - `_medical_history` is a **protected** attribute, meaning it should not be accessed directly outside the class. Access to it is provided through the methods `get_medical_history` and `add_medical_history`.

- `__diagnosis` is a **private** attribute, meaning it's intended to be used only within the `Patient` class and should not be accessed or modified directly from outside the class. Instead, the methods `set_diagnosis` and `get_diagnosis` are used to interact with it.

2. **Methods**:
   - `get_medical_history()` returns the protected medical history of the patient.
   - `add_medical_history(record)` adds a new entry to the medical history.
   - `set_diagnosis(diagnosis)` and `get_diagnosis()` provide a controlled way to set and retrieve the diagnosis.
3. **Usage**:
   - The program creates a patient and interacts with their medical information only through designated methods, demonstrating how encapsulation helps secure and control access to sensitive data.

Output

```
Patient's Initial Medical History: ['High Blood Pressure', 'Diabetes']
Name: John Doe, Age: 45
Medical History: ['High Blood Pressure', 'Diabetes', 'Cholesterol Issue']
Diagnosis: Stable Condition
```

Note: Attempting to access `__diagnosis` directly from outside the class

(e.g. `patient1.__diagnosis`) will result in an `AttributeError`, demonstrating the encapsulation of the private attribute.

-

- Example that focuses on a hospital management system where patient information is encapsulated. In this example, we'll have a Doctor class that manages patients' sensitive information such as prescriptions and medical notes. This data will be protected, allowing only specific access methods.

```
class Doctor:
    def __init__(self, name, specialty):
        self.name = name                    # Public attribute
        self.specialty = specialty          # Public attribute
        self._patients = []                 # Protected attribute to store patient info

    # Method to add a new patient
    def add_patient(self, patient_name, age, ailment):
        patient = {
            "name": patient_name,
            "age": age,
            "ailment": ailment,
            "_prescription": None,          # Protected attribute for prescription
            "_notes": []                    # Protected attribute for doctor's notes
        }
        self._patients.append(patient)
        print(f"Patient {patient_name} added.")

    # Method to prescribe medication to a specific patient
    def prescribe_medication(self, patient_name, prescription):
        for patient in self._patients:
            if patient["name"] == patient_name:
                patient["_prescription"] = prescription
                print(f"Prescription for {patient_name} updated.")
                return
        print(f"Patient {patient_name} not found.")

    # Method to add a medical note for a specific patient
    def add_medical_note(self, patient_name, note):
        for patient in self._patients:
            if patient["name"] == patient_name:
                patient["_notes"].append(note)
                print(f"Note added for {patient_name}.")
```

```
                return
        print(f"Patient {patient_name} not found.")

    # Method to display patient information (without direct access to
protected data)
    def display_patient_info(self, patient_name):
        for patient in self._patients:
            if patient["name"] == patient_name:
                print(f"\nPatient Name: {patient['name']}")
                print(f"Age: {patient['age']}")
                print(f"Ailment: {patient['ailment']}")
                print(f"Prescription: {patient['_prescription']}")
                print(f"Medical Notes: {patient['_notes']}")
                return
        print(f"Patient {patient_name} not found.")

# Simulating the hospital system
doctor = Doctor("Dr. Smith", "Cardiology")

# Adding patients to the doctor's list
doctor.add_patient("Alice Brown", 30, "Arrhythmia")
doctor.add_patient("Bob White", 45, "Hypertension")

# Prescribing medication and adding medical notes for patients
doctor.prescribe_medication("Alice Brown", "Beta Blockers")
doctor.add_medical_note("Alice Brown", "Patient should reduce caffeine
intake.")
doctor.add_medical_note("Alice Brown", "Recommended follow-up in 3
months.")

doctor.prescribe_medication("Bob White", "ACE Inhibitors")
doctor.add_medical_note("Bob White", "Patient advised to exercise
regularly.")

# Displaying patient information
doctor.display_patient_info("Alice Brown")
doctor.display_patient_info("Bob White")
```

https://www.programiz.com/online-compiler/3vCtIyZPAawEK

Output

```
Patient Alice Brown added.
Patient Bob White added.
Prescription for Alice Brown updated.
Note added for Alice Brown.
Note added for Alice Brown.
Prescription for Bob White updated.
Note added for Bob White.

Patient Name: Alice Brown
Age: 30
Ailment: Arrhythmia
Prescription: Beta Blockers
Medical Notes: ['Patient should reduce caffeine intake.', 'Recommended follow-up in 3 months.']

Patient Name: Bob White
Age: 45
Ailment: Hypertension
Prescription: ACE Inhibitors
Medical Notes: ['Patient advised to exercise regularly.']
```

Data Hiding is an essential concept in object-oriented programming (OOP) that helps protect the internal object data from unauthorized access or modification. By concealing certain details of an object, data hiding allows developers to safeguard an object's state and integrity, only exposing data and methods necessary for other parts of the program to function.

Key Points of Data Hiding

1. **Restriction of Access**: Data hiding restricts direct access to the attributes and methods of a class. By doing so, it ensures that only authorized methods within the class can access or modify the hidden data. This is typically achieved through the use of access modifiers like `private` or `protected`.
2. **Use of Private Variables**: In many languages (e.g., Python, C++, Java), you can make an attribute private (i.e., only accessible within the class) by prefixing it with certain symbols:
    - In **Python**, prefixing a variable name with __ (double underscore) makes it private.
    - In **Java** and **C++**, using the `private` keyword restricts access to within the class.
3. **Controlled Access Through Methods**: Data hidden through encapsulation can be accessed only by designated methods (getters and setters). This approach provides controlled and secure access, allowing changes to the internal data only in a regulated way.
4. **Improved Code Security and Integrity**: Data hiding prevents external code from interfering with an object's internal data, which helps maintain the integrity of the data. This is especially important for applications that involve sensitive data or require stability, such as financial and healthcare systems.
5. **Implementation Independence**: By hiding internal data, a class can be modified or updated without affecting other parts of the program that rely on it. External code interacts only with the exposed methods, so internal changes do not impact how the class is used.

Example of Data Hiding in Python

Here's a Python program that demonstrates data hiding in a banking system. In this example, the `BankAccount` class has a private balance attribute, and access to it is restricted by `deposit()` and `withdraw()` methods. Direct access to balance is not allowed.

```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        self.account_holder = account_holder   # Public attribute
        self.__balance = initial_balance       # Private attribute, hidden from outside

    # Method to get the balance
    def get_balance(self):
        return self.__balance
    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"{amount} deposited. New balance: {self.__balance}")
        else:
            print("Invalid deposit amount.")
    # Method to withdraw money
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"{amount} withdrawn. New balance: {self.__balance}")
        else:
            print("Invalid withdrawal amount or insufficient funds.")

# Simulating bank account operations
account = BankAccount("Alice", 1000)
# Trying to deposit and withdraw money
account.deposit(500)       # Valid operation
account.withdraw(200)      # Valid operation
# Trying to access the private balance attribute directly
# print(account.__balance)  # This will raise an AttributeError
# Accessing the balance through the designated method
print("Current balance:", account.get_balance())
```

https://www.programiz.com/online-compiler/6rlj2Z6rD4Vn5

Explanation

1. **Private Data Attribute**:
   - `__balance` is a private attribute. This data cannot be accessed directly from outside the class, ensuring the account balance remains secure.
2. **Controlled Access**:
   - Only the `deposit()` and `withdraw()` methods can modify the balance, and the `get_balance()` method provides controlled access to the balance. This ensures that any changes to the balance follow business rules (e.g., no negative deposits or over-withdrawals).
3. **Encapsulation and Data Hiding**:
   - The `BankAccount` class hides the `__balance` attribute from the outside world. Any interaction with the balance occurs through carefully controlled methods, preventing accidental or unauthorized modifications.

Benefits of Data Hiding in the Example

- **Security**: The balance is protected from being changed directly, reducing the risk of accidental or unauthorized modifications.
- **Integrity**: Only the `deposit()` and `withdraw()` methods can alter the balance, ensuring that balance updates follow specific conditions.
- **Maintainability**: Internal details of how `balance` is managed are hidden, so any changes made to the class's internal structure will not affect external code that uses the `BankAccount` class.