CMPE 103
Module 6
ABSTRACTION AND ENCAPSULATION

Four Main OOP Principles (Pillars of OOP)

INHERITANCE	Process where one object acquires the properties of another. Information is made manageable in a hierarchical order.
POLYMORPHISM	Ability of an object to take on many forms, such that it can be used by different objects.
DATA ABSTRACTION	Ability to make a class abstract, one that cannot be instantiated.
ENCAPSULATION	Ability or technique of making the fields in a class private and providing access to the fields via public methods.

Abstraction

- **Abstraction** refers to the act of representing essential features without including the background details or explanations.
- **Abstraction** provides you a generalized view of your classes or object by providing relevant information.
- **Abstraction** is the process of hiding the working style of an object, and showing the information of an object in understandable manner.

Abstraction Example

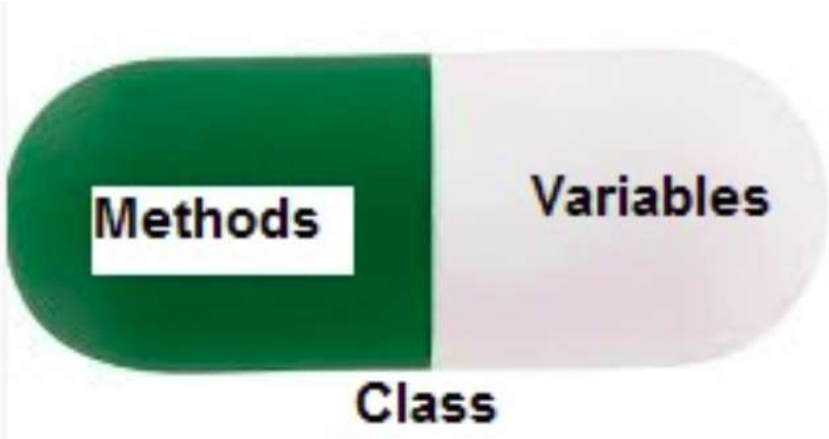
Abstract information (Necessary and Common Information) for the object "Mobile Phone" is make a call to any number and can send SMS."

Nokia 1400	Nokia 2700	Nokia 1400
Features: **Calling** **SMS**	Features: **Calling** **SMS** FM Radio MP3 Camera	Features: **Calling** **SMS** FM Radio MP3 Camera Video Recording Reading E-mails

```
1   from tkinter import *
2   from tkinter import ttk
3
4   root = Tk()
5   frm = ttk.Frame(root, padding=10)
6   frm.grid()
7
8   ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
9   ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
10
11  root.mainloop()
```

Encapsulation

- The wrapping up of data and functions into a single unit is known as **encapsulation**
- The insulation of the data from direct access by the program is called **data hiding** or **information hiding**.
- It is the process of enclosing one or more details from outside world through access right.



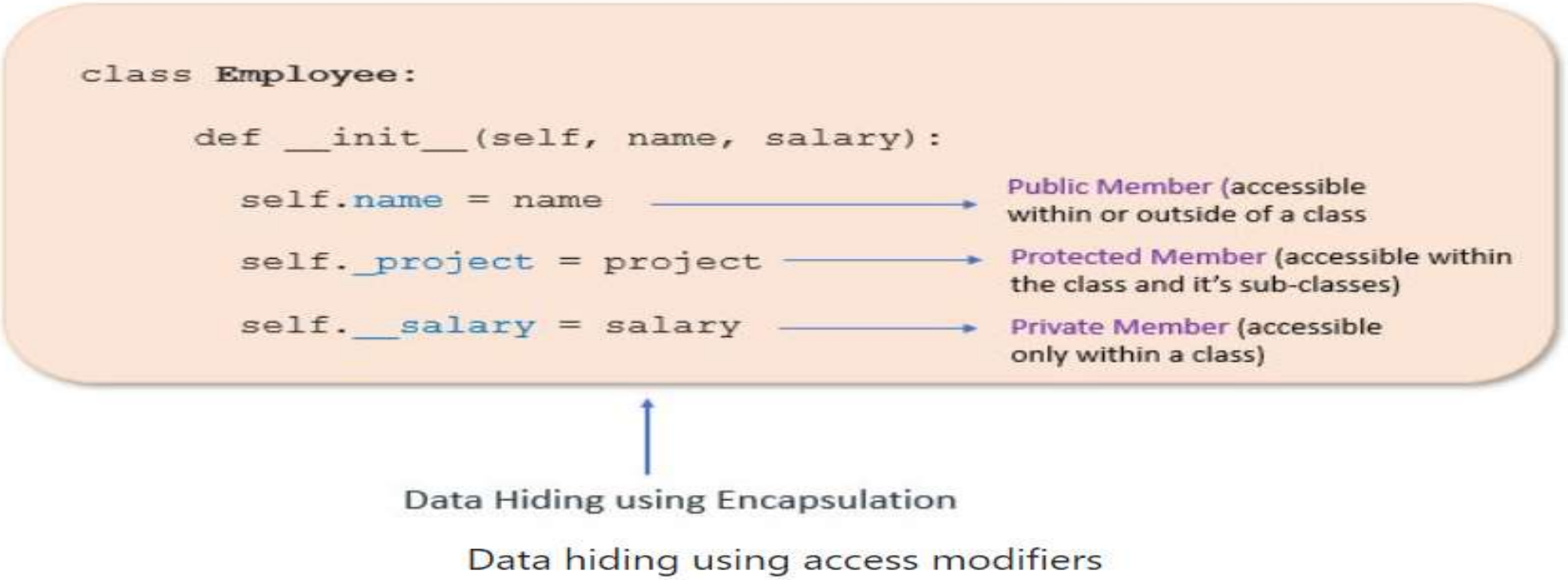
```
class Employee:
    # constructor
    def __init__(self, name, salary, project):
        # data members
        self.name = name
        self.salary = salary
        self.project = project

    # method
    # to display employee's details
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

    # method
    def work(self):
        print(self.name, 'is working on', self.project)

# creating object of a class
emp = Employee('Jessa', 8000, 'NLP')

# calling public method of the class
emp.show()
emp.work()
```

Encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.

Encapsulation is a way can restrict access to methods and variables from outside of class. Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

- **Public Member**: Accessible anywhere from outside class.
- **Private Member**: Accessible within the class
- **Protected Member**: Accessible within the class and its sub-classes



Public Member

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

Private Member

We can protect variables in the class by marking them private. To define a private variable, add two underscores as a prefix at the start of a variable name. Private members are accessible only within the class, and we can't access them directly from the class objects.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
```

Output:

```
AttributeError: 'Employee' object has no attribute '__salary'
```

We can access private members from outside of a class using the following two approaches

- Create public method to access private members
- Use name mangling

Public method to access private members

Access Private member outside of a class using an instance method

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

    # public instance methods
    def show(self):
        # private members are accessible from a class
        print("Name: ", self.name, 'Salary:', self.__salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# calling public method of the class
emp.show()
```

Name Mangling to access private members

We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this `_classname__dataMember`, where `classname` is the current class, and data member is the private variable name.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

print('Name:', emp.name)
# direct access to private member using name mangling
print('Salary:', emp._Employee__salary)
```

Protected Member

Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member's name with a single underscore _. Protected data members are used when you implement inheritance and want to allow data members access to only child classes.

```
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)

c = Employee("Jessa")
c.show()

# Direct access protected data member
print('Project:', c._project)
```

Getters and Setters in Python

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members. In Python, private variables are not hidden fields like in other programming languages.

The getters and setters methods are often used when:

- When we want to avoid direct access to private variables
- To add validation logic for setting a value

```
class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

Output:

```
Name: Jessa 14
Name: Jessa 16
```

Information Hiding and conditional logic for setting an object attributes

```
class Student:
    def __init__(self, name, stud_no, age):
        # private member
        self.name = name
        # private members to restrict access
        # avoid direct data modification
        self.__stud_no = stud_no
        self.__age = age

    def show(self):
        print('Student Details:', self.name, self.__stud_no)

    # getter methods
    def get_stud_no(self):
        return self.__stud_no

    # setter method to modify data member
    # condition to allow data modification with rules
    def set_stud_no(self, number):
        if number > 50:
            print('Invalid stud no. Please set correct student number')
        else:
            self.__stud_no = number

jessa = Student('Jessa', 10, 15)

# before Modify
jessa.show()
# changing student number using setter
jessa.set_stud_no(120)


jessa.set_stud_no(25)
jessa.show()
```

Output:

```
Student Details: Jessa 10
Invalid stud no. Please set correct student number

Student Details: Jessa 25
```

Advantages of Encapsulation

•**Security**: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.

•**Data Hiding**: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.

•**Simplicity**: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.

•**Aesthetics**: Bundling data and methods within a class makes code more readable and maintainable

Try and run this programming problem

(The Fan class). Design a class named Fan to represent a fan. The class contains:

- Three constants named SLOW, MEDIUM, and FAST with the values 1, 2, and 3 to denote the fan speed.
- A private int data field named speed that specifies the speed of the fan.
- A private bool data field named on that specifies whether the fan is on (the default is False).
- A private float data field named radius that specifies the radius of the fan.
- A private string data field named color that specifies the color of the fan.
- The accessor(getters) and mutator(setters) methods for all four data fields.
- A constructor that creates a fan with the specified speed (default SLOW), radius (default 5), color (default blue), and on (default False).

Write a test program named TestFan that creates two Fan objects. For the first object, assign the maximum speed, radius 10, color yellow, and turn it on. Assign medium speed, radius 5, color blue, and turn it off for the second object. Display each object's speed, radius, color, and on properties.

Programming Exercise:

A. Car Class

Write a class named Car that has the following data attributes:

- _ _year_model (for the car's year model)
- _ _make (for the make of the car)
- _ _speed (for the car's current speed)

The Car class should have an _ _init_ _ method that accepts the car's year model and make as arguments. These values should be assigned to the object's _ _year_model and _ _make data attributes. It should also assign 0 to the _ _speed data attribute.

The class should also have the following methods:

- accelerate()

The accelerate method should add 5 to the speed data attribute each time it is called.

- brake()

The brake method should subtract 5 from the speed data attribute each time it is called.

- get_speed()

The get_speed method should return the current speed.

Next, design a program that creates a Car object then calls the accelerate method five times. After each call to the accelerate method, get the current speed of the car and display it. Then call the brake method five times. After each call to the brake method, get the current speed of the car and display it.

B. Pet Class

Write a class named Pet, which should have the following data attributes:

- _ _name (for the name of a pet)
- _ _animal_type (for the type of animal that a pet is. Example values are 'Dog', 'Cat', and 'Bird')
- _ _age (for the pet's age)

The Pet class should have an _ _init_ _ method that creates these attributes. It should also have the following methods:

- set_name()

This method assigns a value to the _ _name field.

- set_animal_type()

This method assigns a value to the _ _animal_type field.

- set_age()

This method assigns a value to the _ _age field.

- get_name()

This method returns the value of the _ _ name field.

- get_animal_type()

This method returns the value of the _ _animal_type field.

- get_age()

This method returns the value of the _ _age field.

Once you have written the class, write a program that creates an object of the class and prompts the user to enter the name, type, and age of his or her pet. This data should be stored as the object's attributes. Use the object's accessor methods to retrieve the pet's name, type, and age and display this data on the screen.