Pemrograman Berorientasi Objek

Encapsulation (Data Hiding)

Bayu Widodo[1]

13 September 2022

[1] Teknik Komputer, SV IPB

Sekolah Vokasi
School of Vocational Studies

Enkapsulasi

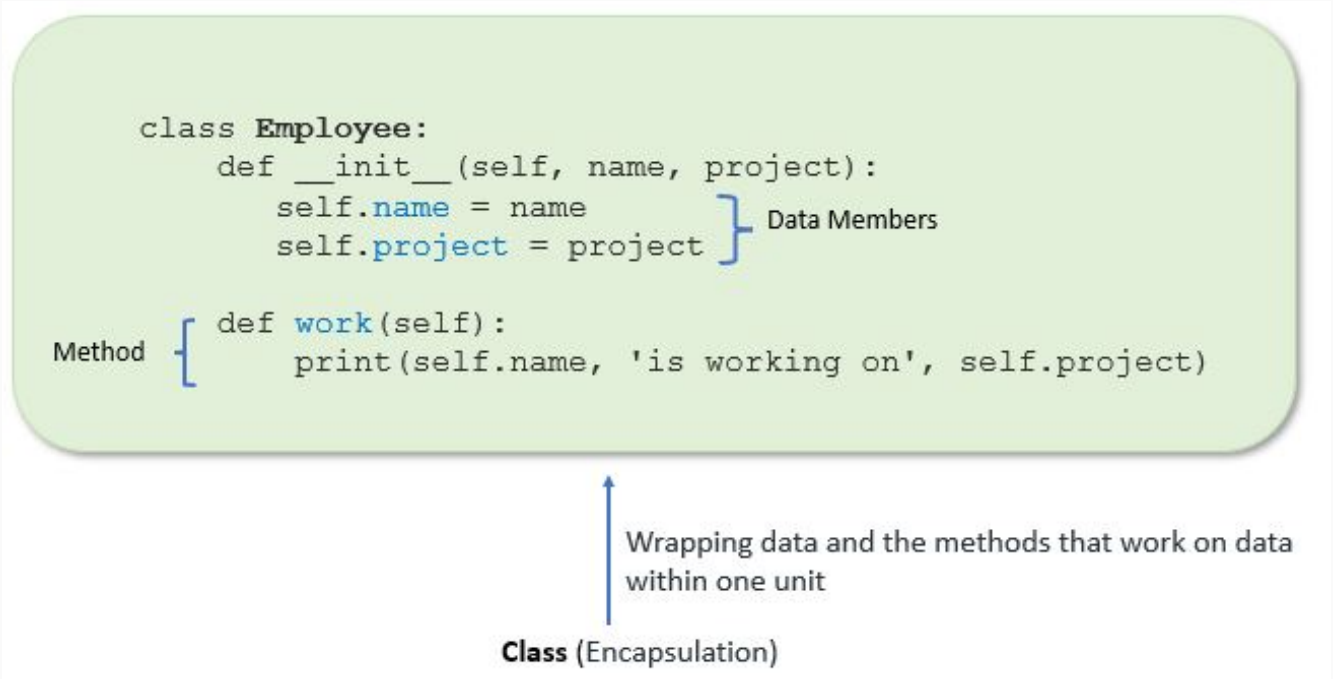
1. Python adalah bahasa pemrograman berorientasi objek. Tidak seperti pada pemrograman berorientasi prosedural, di mana penekanan utama adalah pada fungsi(walaupun Python dapat melakukannya). Sedangkan pemrograman berorientasi objek lebih menekankan pada suatu objek.
2. Objek adalah suatu kumpulan data (variabel) dan metode (fungsi) yang bekerja pada data tersebut. Sedangkan, class adalah blueprint dari suatu objek.
3. Class memiliki suatu atribut, yaitu data yang terdapat di dalamnya (variable class dan instance variable) serta metode yang dapat di akses dengan tanda titik (.)
4. Pemrograman berorientasi objek mencakup empat pillar utama, yaitu **Encapsulation**, Inheritance, Polymorphism dan Abstraction.

1. Class variable yaitu variabel yang dapat di akses oleh semua objek/ instance pada kelas yang berbeda. Variabel kelas didefinisikan di dalam kelas, tapi di luar metode-metode yang ada dalam kelas tersebut.
2. Instance variable ialah variabel khusus untuk setiap instance yang didefinisikan di dalam suatu metode dan hanya dimiliki oleh instance kelas yang digunakan pada saat itu.
3. Instance – Instance adalah istilah lain dari objek suatu kelas. Sebuah objek yang dibuat dari prototipe kelas Lingkaran misalnya disebut sebagai instance dari kelas tersebut. Instansiasi adalah pembuatan instance/objek dari suatu kelas
4. Methods adalah fungsi yang didefinisikan di dalam kelas serta menggambarkan perilaku suatu objek.

1. Enkapsulasi ⇒ Data Hiding ⇒ Access Modifier
2. Enkapsulasi (Encapsulation) adalah sebuah teknik dalam OOP yang mengizinkan untuk menyembunyikan detil dari sebuah atribut dalam sebuah class.
3. Enkapsulasi konsep bundling (membungkus) data dan metode dalam satu unit. Jadi, misalnya, ketika membuat kelas, berarti telah menerapkan enkapsulasi. (Data Hiding)
4. Enkapsulasi memungkinkan untuk membatasi akses variabel dan metode secara langsung dan mencegah modifikasi data yang tidak disengaja dengan membuat anggota dan metode data terbuka, terbatas atau private di dalam kelas. (Access Modifiers)

Access Modifiers adalah sebuah konsep di dalam pemrograman berorientasi objek di mana kita bisa mengatur hak akses suatu atribut pada sebuah class.

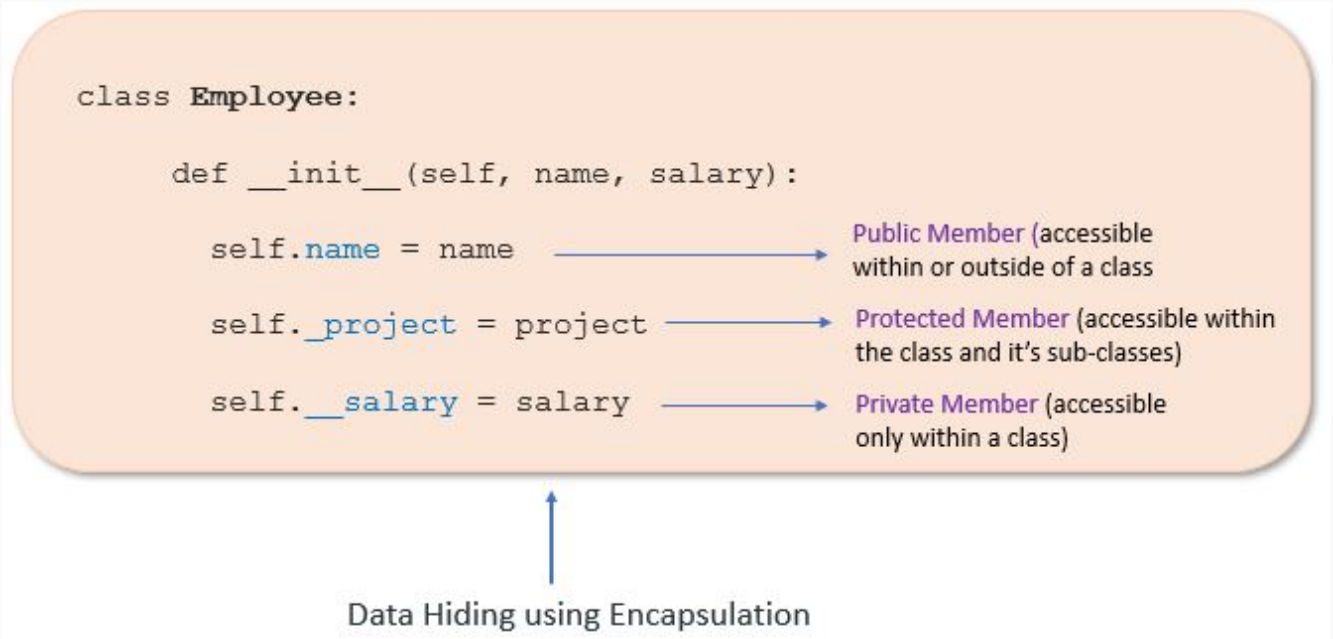
Konsep ini juga biasa disebut sebagai enkapsulasi, di mana kita akan mendefinisikan mana atribut yang boleh diakses secara terbuka, mana yang bisa diakses secara terbatas, atau mana yang hanya bisa diakses oleh internal kelas alias privat.


class Employee:
def __init__(self, name, project):
self.name = name
self.project = project
Data Members
Method
def work(self):
print(self.name, 'is working on', self.project)
Wrapping data and the methods that work on data within one unit
Class (Encapsulation)

```python
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()
```

Access Modifier

1. Public
   - Variabel atau atribut yang memiliki hak akses publik bisa diakses dari mana saja baik dari luar kelas mau pun dari dalam kelas.
2. Protected
   - hak akses protected hanya bisa diakses secara terbatas oleh dirinya sendiri (yaitu di dalam internal kelas), dan juga dari kelas turunannya.
3. Private
   - hak akses private maka ia hanya bisa diakses di dalam kelas tersebut. Tidak bisa diakses dari luar bahkan dari kelas yang mewarisinya.


class Employee:
def __init__(self, name, salary):
self.name = name
Public Member (accessible within or outside of a class
self._project = project
Protected Member (accessible within the class and it's sub-classes)
self.__salary = salary
Private Member (accessible only within a class)
Data Hiding using Encapsulation

```
class Segitiga:
  def __init__(self, alas, tinggi):
    self.alas = alas
    self.tinggi = tinggi
    self.luas = 0.5 * alas * tinggi
```

- Kelas dengan nama Segitiga. Kelas tersebut menerima dua buah parameter: yaitu alas dan tinggi.
- Kelas juga memiliki 3 buah atribut (alas, tinggi, dan luas) yang mana semuanya memiliki hak akses publik.

Akses Publik

```
class Segitiga:
  def __init__(self, alas, tinggi):
    self.alas = alas
    self.tinggi = tinggi
    self.luas = 0.5 * alas * tinggi

segitiga1 = Segitiga(100,20)

# Akses atribut dari luar kelas
print('Panjang alas segitiga: {}'.format(segitiga1.alas))
```

Akses Protected

Untuk mendefinisikan atribut dengan hak akses protected, kita harus menggunakan prefix underscore "_" sebelum nama variabel.

```
class Mobil:
  def __init__(self, merk):
    self._merk = merk

sedan = Mobil('Toyota')

# tampilkan _merk dari luar kelas
print('Merek mobil adalah: {}'.format(sedan._merk))
```

Atribut _merk masih dapat diases dari luar kelas, karena hal ini hanya bersifat convention alias adat atau kebiasaan saja yang harus disepakati oleh programmer. Di mana jika suatu atribut diawali oleh "_", maka ia harusnya tidak boleh diakses kecuali dari internal kelas tersebut atau dari kelas yang mewarisinya."

Akses Private

Untuk membuat sebuah atribut menjadi private, kita harus menambahkan dua buah underscore sebagai prefix nama atribut.

```
class Mobil:
  def __init__(self, merk):
    self.__merk = merk

sedan = Mobil('Toyota')

# tampilkan _merk dari luar kelas
print('Merek mobil adalah: {}'.format(sedan.__merk))
```

Contoh Access Modifier

```
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()
```

```python
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)
```

Akses private member dari luar kelas dapat menggunakan dua pendekatan:

1. Membuat metode publik untuk mengakses anggota pribadi atau
2. Gunakan name mangling

Name Mangling dibuat dengan menambahkan dua garis bawah utama dan satu garis bawah tambahan pada sebuah identifier, seperti _class-name__dataMember,

di mana classname adalah kelas saat ini, dan anggota data adalah nama private variabel.

```
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)
```

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

Protected Member

1. Protected Member dapat diakses di dalam kelas dan juga tersedia untuk sub-kelasnya. Untuk menentukan Protected Member, diawali dengan tanda "_".
2. Protected Member digunakan saat akan menerapkan pewarisan dan ingin mengizinkan data member hanya mengakses sub-kelas (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)
```

Getter dan Setter

1. Enkapsulasi melibatkan kemasan yang berhubungan dengan variabel dan fungsi-fungsi ke dalam sebuah objek tunggal yang mudah digunakan.
2. Konsep Enkapsulasi berhubungan dengan penyembunyian data (data hiding), dimana kondisi detail dari implementasi sebuah kelas harus disembunyikan, dan sebuah interface standar yang bersih dapat dipresentasikan untuk yang ingin menggunakan kelas tersebut.
3. Tujuan utama menggunakan getter dan setter dalam program berorientasi objek adalah untuk memastikan enkapsulasi data. Gunakan getter method untuk mengakses anggota data dansetter methods untuk memodifikasi anggota data.

4. Pada Python, variabel Private bukanlah field tersembunyi seperti dalam bahasa pemrograman lain. Metode getter dan setter sering digunakan untuk:
    - mencegah akses langsung ke variabel private
    - Validasi data

```
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())
```

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

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

    # getter methods
    def get_roll_no(self):
        return self.__roll_no
```

```
    # setter method to modify data member
    # condition to allow data modification with rules
    def set_roll_no(self, number):
        if number > 50:
            print('Invalid roll no. Please set correct \
            roll number')
        else:
            self.__roll_no = number

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

# before Modify
jessa.show()
# changing roll number using setter
jessa.set_roll_no(120)


jessa.set_roll_no(25)
jessa.show()
```

Keuntungan Enkapsulasi