RESUME PRAKTIKUM PBO
MATERI PERTEMUAN 3 DAN 4
(ABSTRAKSI, ENKAPSULASI, INHERITANCE, POLYMORPISM)

Oleh :

Nama : Novia Eka Putri

NIM : 121140030

Kelas : RB

INSTITUT TEKNOLOGI SUMATERA

LAMPUNG SELATAN

2023

- **Prinsip Utama Pemrograman Beroritentasi Objek dalam Python**

  Ada empat prinsip utama untuk pemrograman berorientasi objek.

  1. Abstraksi
  2. Enkapsulasi
  3. Inheritance
  4. Polimorphysm

1. Abstraksi

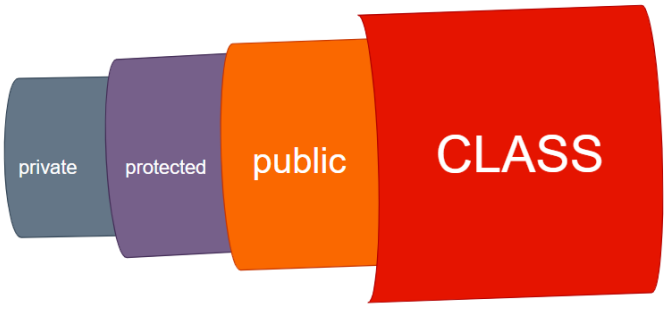
Dalam bahasa pemrograman Python, abstraksi dapat dicapai melalui penggunaan kelas abstrak atau penggunaan metode abstrak di kelas. Untuk membuat kelas abstrak, kita bisa menggunakan modul abc (Abstract Base Class) bawaan Python. Modul abc menyediakan kelas Basis Abstrak (ABC) yang dapat digunakan sebagai kelas abstrak. Untuk membuat kelas abstrak, kita perlu mewarisi dari kelas ABC dan mendeklarasikan beberapa metode abstrak yang akan diterapkan oleh kelas turunan.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Gug!!"

class Cat(Animal):
    def speak(self):
        return "Meow^^"

d = Dog()
print("Dog says:", d.speak())

c = Cat()
print("Cat says:", c.speak())
```

Sehingga code diatas akan menghasilkan output sebagai berikut :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] python -u "d:\python\tempCodeRunnerFile.python"
Dog says: Gug!!
Cat says: Meow^^
```

2. Enkapsulasi



Enkapsulasi adalah proses melindungi data dan fungsionalitas kelas ke dalam satu unit yang disebut objek. Mekanisme ini sering digunakan untuk melindungi data suatu objek dari objek lain. Ini adalah salah satu prinsip dasar dalam semua bahasa pemrograman yang mendukung pemrograman berorientasi objek.

a. Public Access Modifier

Variabel atau atribut yang memiliki hak akses publik bisa diakses dari mana saja baik dari luar kelas mau pun dari dalam kelas.

Perhatikan contoh berikut:

```python
class Animal:
    def __init__(self, tom, jerry):
        self.tom = tom
        self.jerry = jerry
```

Pada contoh di atas, kita membuat sebuah kelas dengan nama Animal. Kelas tersebut menerima dua buah parameter: yaitu tom dan jerry. Kemudian ia juga memiliki 2 buah atribut (tom dan jerry) yang mana **semuanya memiliki hak akses publik**.

Untuk membuktikannya, mari kita coba akses ke-2 atribut di atas dari luar kelas:

```
colour = Animal("Grey", "Brown")

# akses variabel tom dan jerry dari luar kelas
print(f'Tom   : {colour.tom}')
print(f'Jerry : {colour.jerry}')
```

Jika dijalankan, kita akan mendapatkan output seperti berikut:

```
Tom   : Grey
Jerry : Brown
```

b. Protected Access Modifier

Variabel atau atribut yang memiliki hak akses protected hanya bisa diakses secara terbatas oleh dirinya sendiri (yaitu di dalam internal kelas), dan juga dari kelas turunannya. Untuk mendefinisikan atribut dengan hak akses *protected*, kita **harus** menggunakan prefix underscore (_) sebelum nama variabel.

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

Pada kode program di atas, kita membuat sebuah kelas bernama Mobil. Dan di dalam kelas tersebut, kita mendefinisikan satu buah atribut bernama _merk, yang mana hak akses dari atribut tersebut adalah *protected* karena nama variabelnya diawali oleh *underscore* (_).

Mari kita coba akses dari luar kelas:

```
sedan = Mobil('Toyota')

# tampilkan _merk dari luar kelas
print(f'Merk: {sedan._merk}')
```

Kita masih bisa mengakses atribut _merk 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.

Yang harusnya kita lakukan adalah: mengakses atribut *protected* hanya dari kelas internal atau dari kelas turunan, perhatikan contoh berikut:

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

class MobilBalap(Mobil):
  def __init__(self, merk, total_gear):
    super().__init__(merk)
    self._total_gear = total_gear

  def pamer(self):
    # akses _merk dari subclass
    print(f'Ini mobil {self._merk} dengan total gear {self._total_gear}')
```

Bikin objek dari kelas MobilBalap:

```
ferrari = MobilBalap('Ferrari', 8)
ferrari.pamer()
```

Outputnya:

```
Ini mobil Ferrari dengan total gear 8
```

c. Private Access Modifier

Modifier selanjutnya adalah *private*. Setiap variabel di dalam suatu kelas yang memiliki hak akses *private* maka ia **hanya bisa diakses** di dalam kelas tersebut. Tidak bisa diakses dari luar bahkan dari kelas yang mewarisinya.

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

Perhatikan contoh berikut:

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

Pad kode di atas, kita telah membuat sebuah atribut dengan nama __nopel, __nama, __saldo. Dan karena nama tersebut diawali dua buah underscore, maka ia tidak bisa diakses kecuali dari dalam kelas AkunBank saja.

Mari kita buktikan:

```
laux = SepedaLipat('Laux')
print(f'Merk: {laux.__merk}')
```

Kita akan mendapatkan sebuah error sebagai berikut:

```
AttributeError: 'SepedaLipat' object has no attribute '__merk'
```

Tapi jika kita ubah kodenya menjadi seperti ini:

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

    def tampilkan_merk(self):
        print(f'Merk: {self.__merk}')

laux = SepedaLipat("Laux")
laux.tampilkan_merk()
```

Kita akan mendapatkan output seperti ini:

```
Merk: Laux
```

Kenapa? Karena kita mengakses variabel __merk dari dalam internal kelas SepedaLipat, bukan dari luar.

d. Getter dan Setter

Selanjutnya kita bisa membuat accessor dan mutator atau getter dan setter pada suatu kelas di python. Untuk apa accessor dan mutator? Ia adalah sebuah fungsi yang akan dieksekusi ketika kita mengakses (aksesor) suatu atribut pada suatu kelas, atau fungsi yang dieksekusi ketika hendak mengatur (mutator) suatu atribut pada suatu kelas. Untuk mendefinisikan accessor (getter), kita perlu mendefinisikan decorator @property sebelum nama fungsi. Sedangkan untuk mengatur mutator (setter), kita perlu mendefinisikan descriptor @<nama-atribut>.setter.

Perhatikan contoh berikut:

```python
class Mobil:
  def __init__(self, tahun):
    self.tahun = tahun

  @property
  def tahun(self):
    return self.__tahun

  @tahun.setter
  def tahun(self, tahun):
    if tahun > 2021:
      self.__tahun = 2021
    elif tahun < 1990:
      self.__tahun = 1990
    else:
      self.__tahun = tahun
```

Ketika kita nanti mengakses atribut tahun (tanpa underscore), maka fungsi tahun() yang pertama akan dieksekusi. Sedangkan jika kita mengisi / mengubah / memberi nilai pada atribut tahun (tanpa underscore), maka yang dieksekusi adalah fungsi tahun() yang kedua.

Mari kita coba kode program berikut:

```python
sedan = Mobil(2200)
print(f'Mobil ini dibuat tahun {sedan.tahun}')
```

Kode program di atas akan menampilkan tahun dari sedan yang sudah melalui if-else sebelumnya. Sehingga jika kita memberikan nilai tahun yang lebih dari tahun 2021, yang tersimpan tetaplah tahun 2021.

Berikut ini outputnya:

```
Mobil ini dibuat tahun 2021
```

Sedangkan jika kita mengakses atribut aslinya yaitu __tahun, kita akan mendapatkan error karena atribut tersebut bersifat *private*.

```
sedan = Mobil(2200)
print(f'Mobil ini dibuat tahun {sedan.__tahun}')
```

Error:

```
AttributeError: 'Mobil' object has no attribute '__tahun'. Did you mean: 'tahun'?
```

Selanjutnya mari kita coba ubah atribut tahun seperti berikut:

```
sedan = Mobil(2000)
sedan.tahun = 1800
print(f'Mobil ini keluaran {sedan.tahun}')
```

Output:

```
Mobil ini keluaran 1990
```

Lihat, kita mengubah tahun menjadi 1800 akan tetapi yang tersimpan adalah 1990. Itu terjadi karena ketika kita mengubah atribut tahun, sistem masih memanggil terlebih dahulu fungsi setter tahun().

3. Inheritance

Ini adalah proses pembuatan kelas yang dapat memperoleh atau mewarisi properti dan metode dari kelas lain (induk / basis). Kita dapat merepresentasikan hubungan dunia nyata dengan mudah menggunakan fitur ini. Penggunaan kembali kode akan sederhana. Ini memiliki sifat transitif yang kuat - artinya ketika kelas B diwarisi dari kelas A, maka kelas yang diwarisi dari kelas B akan otomatis diwarisi dari kelas A.

```
class Komputer:
    def __init__(self, nama, jenis, harga, merk):
        self.nama = nama
        self.jenis = jenis
        self.harga = harga
        self.merk = merk

class Processor(Komputer):
    def __init__(self, nama, jenis, harga, merk, jumlah_core, kecepatan_processor):
        super().__init__(nama, jenis, harga, merk)
        self.jumlah_core = jumlah_core
        self.kecepatan_processor = kecepatan_processor

class RAM(Komputer):
    def __init__(self, nama, jenis, harga, merk, capacity):
        super().__init__(nama, jenis, harga, merk)
        self.capacity = capacity
```

Dalam contoh ini, Komputer adalah kelas induk yang dimiliki oleh nama, jenis, harga dan merk. Processor dan RAM adalah dua kelas yang diturunkan dari kelas induk.

```
def __init__(self, nama, jenis, harga, merk):
```

adalah fungsi di kelas induk. Karena Processor dan RAM merupakan dua kelas turunan Komputer, mereka dapat mengakses fungsi tersebut. Ini adalah cara penggunaan kembali kode melalui pewarisan.

4. Polymorphism

Polymorphism secara umum diartikan sebagai kondisi dimana terjadinya perbedaan bentuk. Dalam OOP, bentuk dapat dianggap sebagai perilaku. Dalam Python, suatu objek memiliki jenis perilaku yang berbeda berdasarkan data dan input melalui *polymorphism*. Penimpaan metode juga merupakan jenis *polymorphism*. Kita memiliki 4 cara yang berbeda untuk mendefinisikan Polymorphism pada pemrograman berorientasi objek Python.

a. Polymorphism dengan Operator (Overloading)

```
num1 = 1
num2 = 2
print(num1 + num2)
# OutPut: 3

str1 = "This is"
str2 = "Python"
print(str1+ " " +str2)
# OutPut : This is Python
```

Di sini kita dapat melihat bahwa satu operator + telah digunakan untuk melakukan operasi yang berbeda untuk tipe data yang berbeda. Ini adalah salah satu pembentukan Polymorphism paling sederhana di Python, dapat juga disebut dengan operator **Overloading**.

b. Polymorphism dengan Fungsi (Overloading)

Ada beberapa fungsi bawaan dalam Python yang kompatibel untuk dijalankan dengan beberapa tipe data. Mis. len(), dapat berjalan dengan banyak tipe data dengan Python.

```
107
108   print(len("PBO"))
109   print(len([10, 20]))
110
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
[Running] python -u "d:\python\tempCodeRunnerFile.python"
3
2

[Done] exited with code=0 in 0.177 seconds
```

Oleh karena itu, program di atas menghasilkan 3 dan 2.

```
111
112 def my_function(*kids):
113     return "The youngest child is " + kids[-1]
114 print(my_function("Emil", "Tobias", "Linus"))
115 print(my_function("Linus"))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] python -u "d:\python\tempCodeRunnerFile.python"
The youngest child is Linus
The youngest child is Linus

[Done] exited with code=0 in 0.145 seconds
```

Di sini, kita dapat melihat bahwa banyak penggunaan dapat bekerja dengan satu fungsi yang sama, ini bisa disebut juga dengan fungsi **Overloading**.

c. Polymorphism dengan Kelas (Overriding)

Kita dapat menggunakan konsep Polymorphism ketika membuat fungsi pada kelas. Python mendukung kalas yang berbeda untuk menggunakan fungsi dengan nama yang sama. Kemudian kita dapat memanggil fungsi tersebut dengan menggunakan objek yang kita buat.

```
117
118 class Hewan:
119     def __init__(self, nama, usia):
120         self.nama = nama
121         self.usia = usia
122     def bersuara(self):
123         pass
124 class Kucing(Hewan):
125     def bersuara(self):
126         return "Meow"
127 class Anjing(Hewan):
128     def bersuara(self):
129         return "Guk..guk..."
130 kucing = Kucing("Tom", 3)
131 anjing = Anjing("Spike", 4)
132
133 for hewan in (kucing, anjing):
134     print(hewan.bersuara())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] python -u "d:\python\tempCodeRunnerFile.python"
Meow
Guk..guk...

[Done] exited with code=0 in 0.878 seconds
```

Dalam Inheritance, kelas anak mewarisi fungsi dan atribut dari kelas induk. Kita dapat mendefinisikan ulang fungsi dan atribut tertentu agar sesuai dengan kelas anak, yang dikenal sebagai Overriding. Polymorphism memungkinkan kita untuk mendefinisikan fungsi dan atribut di kelas anak yang memiliki nama yang sama dengan kelas induk.

Referensi

https://ichi.pro/id/python-untuk-pemula-pemrograman-berorientasi-objek-65021974666569

https://jagongoding.com/python/menengah/oop/overriding/

https://jagongoding.com/python/menengah/oop/accessmodifiers/#:~:text=Untuk%20membuat%20sebuah%20atribut%20menjadi%20private%2C%20kita%20harus,kita%20telah%20membuat%20sebuah%20atribut%20dengan%20nama%20__merk.

https://www.programiz.com/python-programming/polymorphism

Modul praktikum PBO 3 dan 4

Modul perkuliahan minggu ke 6