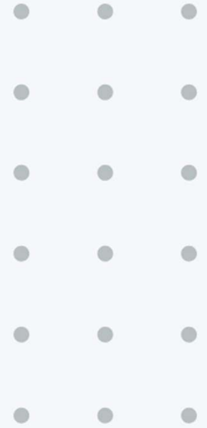
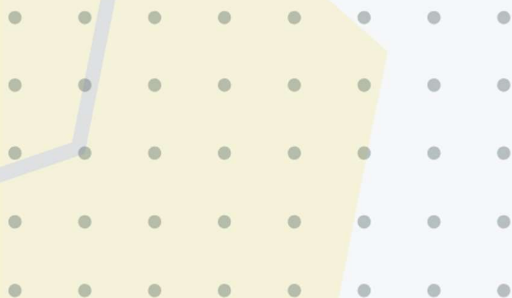
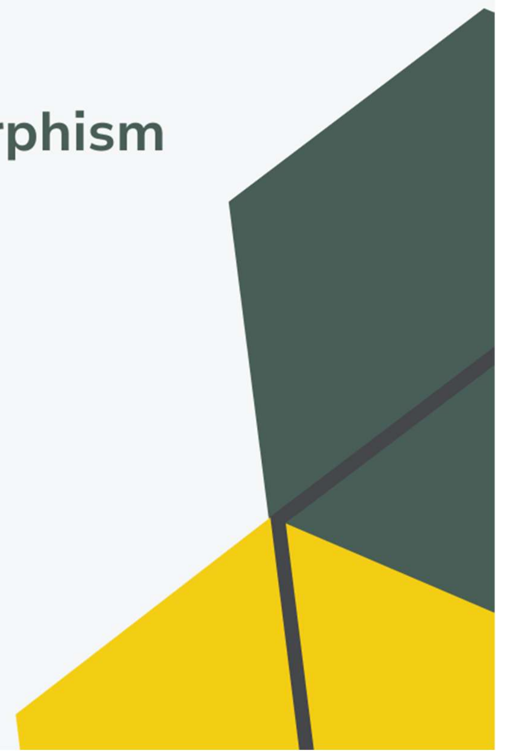


MODUL PRAKTIKUM

PEMROGRAMAN BERBASIS OBJEK

Pertemuan 3
Abstraction and Polymorphism





Kemarin kita sudah membahas tentang konsep dari Enkapsulasi dan Inheritance, pada minggu ini kita akan membahas tentang Abstraksi dan Polimorfisme untuk menyelesaikan fundamental pemrograman berorientasi objek / OOP

A. Abstraksi

- **Abstraksi** dalam Python adalah proses menyembunyikan implementasi sebenarnya dan hanya menekankan bagaimana cara menggunakannya.
- Kalian sudah melihat sedikit implementasinya dalam bagian **Pewarisan (Inheritance)**.
- Sederhananya, kita bisa memvisualisasikan ini dalam sebuah video game. Dalam video game, setiap karakter itu **unik**, baik dari segi **kemampuan (skills)**, **kesehatan (health)**, atau **peran (role)**, serta banyak atribut lainnya. Akan menjadi redundan jika kita harus mendefinisikan atribut karakter **satu per satu**.
- Salah satu cara mengatasinya adalah dengan membuat **Kelas Dasar (Base Class) / Kelas Abstrak (Abstract Class)** yang berisi **metode (method())** atau **properti (property)** yang bersifat **abstrak**, sehingga dapat diimplementasikan lebih lanjut oleh **Kelas Spesifik / Kelas Turunan (Specific / Derived Class)**.
- Konsep ini sangat erat kaitannya dengan **Pewarisan (Inheritance)**. Jadi, jika kalian masih bingung, tidak masalah, kita akan memahami konsep ini bersama-sama!

Kita bisa mulai dengan melihat kode dibawah ini. Kita akan membuat kelas Avatar yang nantinya akan menjadi kelas abstrak untuk kelas Warrior dan Mage

```
1  from abc import ABC, abstractmethod
2
3  class Avatar:
4      def __init__(self, name, attack, health):
5          self._name = name
6          self._attack = attack
7          self._health = health
8
9      @property
10     def name(self):
11         return self._name
12
13     @property
14     def attack(self):
15         return self._attack
16
17     @property
18     def health(self):
19         return self._health
20
21     @abstractmethod
22     def fight(self, opponent):
23         pass
24
25     def take_damage(self, damage):
26         self._health -= damage
27         if self._health <= 0:
28             print(f"{self.name} has been defeated!")
```

Lanjut, kita akan membuat dua kelas yang memakai / meng-inherit kelas abstrak diatas

```python
class Warrior(Avatar):
    def __init__(self, name, attack, health):
        super().__init__(name, attack, health)

    def fight(self, opponent):
        damage = self.attack - (opponent.attack // 2) # True Division
        print(f"{self.name} attacks {opponent.name} for {damage} damage.")
        opponent.take_damage(damage)

class Mage(Avatar):
    def __init__(self, name, attack, health):
        super().__init__(name, attack, health)

    def fight(self, opponent):
        damage = self.attack - (opponent.attack // 2) # True Division
        print(f"{self.name} casts a spell on {opponent.name} for {damage} damage.")
        opponent.take_damage(damage)
```

lalu kita akan membuat objek dari 2 kelas tersebut dan membuat keduanya bertarung

```python
# Init the Object
axe = Warrior("Axe", 80, 150)
aurora = Mage("Aurora", 100, 70)

while (axe.health >= 0 or aurora.health >= 0) :
    axe.fight(aurora)
    aurora.fight(axe)
    if (axe.health <= 0 or aurora.health <= 0) : # Implementation of decorator
        break
    print(f"Axe Health: {axe.health}")
    print(f"Aurora Health: {aurora.health}")
```

B. Polimorfisme dalam Pemrograman

- **Polimorfisme** berarti memiliki banyak bentuk. Dalam konteks pemrograman, ini berarti **nama fungsi yang sama** dapat digunakan untuk tipe data yang berbeda. Ada beberapa jenis polimorfisme yang akan kita bahas.

C. Method Overloading

Method Overloading biasanya digunakan dalam **constructor** pada bahasa seperti **Java, Kotlin, dan C++**. Namun, Python tidak mendukung method overloading secara langsung, tetapi kita tetap bisa memanfaatkannya dengan konsep lain.

Contoh method overloading dalam **C++**:

```cpp
class CountArea {
public:
    CountArea() {
        length = 2;
        width = 3;
    }

    CountArea(double l, double w) {
        length = l;
        width = w;
    }

private:
    double length;
    double width;
};
```

Meskipun Python tidak mendukung method overloading seperti di Java atau C++, kita bisa menggunakan trik seperti pengecekan jumlah argumen dalam satu metode untuk mencapai efek yang sama.

D. Method Overriding

Polimorfisme ini adalah konsep inti dari **Abstraksi** dan **Pewarisan (Inheritance)**, di mana metode dalam **kelas induk** dapat di-*override* oleh **kelas turunan**.

Misalnya, dalam Python:

```python
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"
```

Di sini, kelas **Dog** dan **Cat** meng-*override* metode make_sound() dari **Animal**, memberikan implementasi unik masing-masing.

E. Operator Overloading / Magic Method / Dunder Method

Di Python, kita bisa melakukan **operator overloading** menggunakan **Dunder Method**. Contoh operator overloading menggunakan + untuk menambahkan objek **Point**:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for +")

    def __str__(self):
        return f"Point({self.x}, {self.y})"

point1 = Point(2, 3)
point2 = Point(3, 4)
point3 = point1 + point2
print(point3)  # Output: Point(5, 7)
```

F. Duck Typing

Duck Typing adalah konsep dalam Python yang memungkinkan kita menggunakan metode/fungsi tanpa melihat tipe objeknya, selama metode tersebut tersedia.

Contoh:

```
def make_it_speak(animal):
    print(animal.make_sound())

class Duck:
    def make_sound(self):
        return "Quack!"

class Dog:
    def make_sound(self):
        return "Woof!"

make_it_speak(Duck())  # Output: Quack!
make_it_speak(Dog())  # Output: Woof!
```

Kesimpulan: Polimorfisme sangat berguna dalam pemrograman berbasis objek untuk meningkatkan fleksibilitas dan mempermudah pemeliharaan kode. Dengan memahami konsep ini, kita dapat menulis kode yang lebih modular dan dapat diperluas dengan lebih mudah!