BAB VI

OBJECT-ORIENTED PROGRAMMING (OOP)

Sebagai bahasa pemrograman multi-paradigma, Python mendukung berbagai pendekatan pemrograman. Salah satu cara populer untuk memecahkan masalah pemrograman adalah melalui penciptaan **objek**, yang dikenal sebagai **Pemrograman Berorientasi Objek** (Object-Oriented Programming, disingkat **OOP**).

Sebuah objek ditandai dengan dua karakteristik utama:

- **Atribut**: Variabel yang mendefinisikan karakteristik atau sifat suatu objek.
- **Perilaku** (**Method**): Fungsi yang menunjukkan tindakan atau kemampuan suatu objek.
- **Class Variable:** Variabel statis yang dimiliki oleh kelas itu sendiri dan tidak diwariskan ke setiap objeknya.

Sebagai contoh, sebuah **mobil** adalah objek karena memiliki sifat-sifat berikut:

- **merek**, **warna**, **kecepatan_maksimal** sebagai atribut.
- **menyalakan_mesin**, **mengerem** sebagai perilaku (method).

OOP dalam Python bertujuan untuk memodelkan entitas dunia nyata, mengimplementasikan konsep-konsep seperti pewarisan (**inheritance**), polimorfisme (**polymorphism**), enkapsulasi (**encapsulation**), dan lain-lain.

Dalam Python, konsep OOP dibangun di atas beberapa prinsip dasar:

A. Class dan Objek

Kelas adalah struktur inti dalam OOP yang berfungsi sebagai **cetak biru** (*blueprint*) untuk menciptakan objek. Kelas mendefinisikan atribut dan *method* yang akan dimiliki oleh objek-objek yang dibuat darinya. Kita bisa membayangkan kelas sebagai **desain mobil** dengan label. Desain ini mencakup semua detail tentang merek, warna, ukuran mesin, dan sebagainya, tetapi belum menjadi mobil yang sebenarnya.

Membuat Kelas

Contoh sintaks untuk mendefinisikan kelas **Mobil**:

kunci **class** digunakan untuk mendefinisikan kelas. **pass** menandakan bahwa ini adalah kelas kosong untuk saat ini.

Membuat Objek (Instance)

Dari sebuah kelas, kita dapat membuat **instance**. **Instance** adalah objek spesifik yang dibuat dari kelas tertentu. Setiap ***instance*** akan memiliki semua atribut dan perilaku dari kelas asalnya. Namun, nilai atau karakteristik atribut dari masing-masing objek dari kelas yang sama bisa saja berbeda.

Contoh sintaks untuk membuat objek (***instance***) dari kelas **Mobil**:

Operator *assignment* = digunakan untuk menyimpan objek dengan *identifier* **(suv_hitam, sedan_merah)** di sisi kiri dan nama kelas **(Mobil())** di sisi kanan operator.

1. Konstruktor

Konstruktor adalah fungsi khusus yang secara otomatis dipanggil tepat setelah suatu objek dibuat. Fungsi ini didefinisikan di dalam blok utama kelas. Umumnya, konstruktor digunakan untuk **menginisialisasi nilai atribut** dari entitas objek.

Contoh sintaks untuk membuat konstruktor:

```
# class
class Mobil:
    def __init__(self):
        print("Objek mobil berhasil dibuat dan siap diinisialisasi!")

# objek
suv_hitam = Mobil()
```

Output:

```
Objek mobil berhasil dibuat dan siap diinisialisasi!
```

Selain **__init__**, terdapat dua konstruktor/method khusus lainnya yang memiliki fungsi tertentu dalam siklus hidup objek:

Method Khusus	Deskripsi
__new__(cls)	Dipanggil otomatis untuk membuat *instance* baru.
__init__(self)	Dipanggil otomatis untuk **menginisialisasi** *instance* baru.
__del__(self)	Dipanggil otomatis ketika objek akan dihapus/dihancurkan.

2. Menambahkan Atribut

Atribut adalah variabel Python yang secara khusus terkait dengan satu objek (*instance*). Atribut dapat diakses dalam lingkup objek dan harus diinisialisasi di dalam fungsi konstruktor kelas menggunakan *keyword* **self.[nama atribut]**.

Menggunakan Konstruktor

Contoh sintaks untuk menambahkan atribut menggunakan konstruktor:

```python
# class
class Mobil:
    def __init__(self, merek, warna):
        self.merek = merek
        self.warna = warna

# objek
sedan_merah = Mobil("Toyota", "Merah")
print(sedan_merah.merek)
```

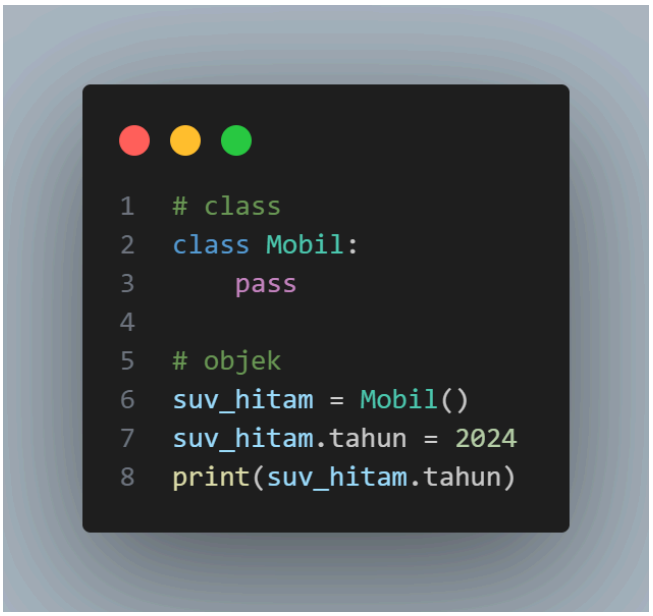
Output

```
Toyota
```

Menambahkan Atribut Secara Langsung

Atribut juga dapat ditambahkan langsung setelah objek dibuat menggunakan sintaks [nama **objek].[nama atribut]**. Namun, pendekatan ini umumnya **tidak dianjurkan** untuk atribut utama objek.

Contoh sintaks untuk menambahkan atribut secara langsung:



```python
# class
class Mobil:
    pass

# objek
suv_hitam = Mobil()
suv_hitam.tahun = 2024
print(suv_hitam.tahun)
```

Output:

```
2024
```

3. Menambahkan Perilaku (*Method*)

Method adalah tindakan atau kemampuan yang dapat dilakukan oleh suatu objek. *Method* pada dasarnya adalah fungsi yang didefinisikan di dalam kelas objek. Di tingkat pemrograman, *method* serupa dengan fungsi dalam pemrograman terstruktur, tetapi *method* memiliki akses langsung ke semua data (atribut) yang terkait dengan objeknya melalui parameter **self**. Seperti fungsi, *method* juga dapat menerima parameter tambahan dan mengembalikan nilai.

Contoh sintaks untuk menambahkan *method* (perilaku):

```python
# class
class Mobil:
    def __init__(self, merek):
        self.merek = merek
        self.bahan_bakar = 50 # dalam liter

    def isi_bensin (self, liter):
        self.bahan_bakar += liter
        print(f"Bensin ditambahkan {liter} liter.")

    def cek_bensin (self):
        return self.bahan_bakar

# objek
sedan_merah = Mobil("Honda")
sedan_merah.isi_bensin(10)
print("Bahan Bakar Tersisa : " + str(sedan_merah.cek_bensin()) + " liter")
```

Output:

```
Bensin ditambahkan 10 liter.
Bahan Bakar Tersisa : 60 liter
```

Contoh Program OOP: Class, Objek, Atribut, Constructor, dan Method :

```python
# Membuat class Mobil
class Mobil:
    # Constructor -> digunakan untuk menginisialisasi atribut saat objek dibuat
    def __init__(self, merek, warna, tahun, kecepatan_maksimal):
        self.merek = merek                  # Atribut 1
        self.warna = warna                  # Atribut 2
        self.tahun = tahun                  # Atribut 3
        self.kecepatan_maksimal = kecepatan_maksimal  # Atribut 4
        self.kecepatan_sekarang = 0         # Atribut tambahan (nilai awal = 0)

    # Method untuk menampilkan informasi mobil
    def tampilkan_info(self):
        print("=== INFORMASI MOBIL ===")
        print(f"Merek: {self.merek}")
        print(f"Warna: {self.warna}")
        print(f"Tahun Pembuatan: {self.tahun}")
        print(f"Kecepatan Maksimal: {self.kecepatan_maksimal} km/jam")
        print(f"Kecepatan Sekarang: {self.kecepatan_sekarang} km/jam")

    # Method untuk menyalakan mesin mobil
    def nyalakan_mesin(self):
        print(f"Mesin {self.merek} telah dinyalakan.")

    # Method untuk menambah kecepatan
    def tambah_kecepatan(self, nilai):
        if self.kecepatan_sekarang + nilai <= self.kecepatan_maksimal:
            self.kecepatan_sekarang += nilai
            print(f"Kecepatan ditambah {nilai} km/jam.")
        else:
            print(" Kecepatan sudah mencapai batas maksimal!")

    # Method untuk mengerem mobil
    def rem(self, nilai):
        if self.kecepatan_sekarang - nilai >= 0:
            self.kecepatan_sekarang -= nilai
            print(f"Mobil mengerem {nilai} km/jam.")
        else:
            self.kecepatan_sekarang = 0
            print("Mobil sudah berhenti.")
```

```python

# Membuat objek (instance) dari class Mobil
mobil1 = Mobil("Toyota Supra", "Merah", 2022, 250)
mobil2 = Mobil("Honda Civic", "Hitam", 2021, 220)

# Menggunakan method pada masing-masing objek
mobil1.nyalakan_mesin()
mobil1.tambah_kecepatan(100)
mobil1.tampilkan_info()

print("\n---\n")

mobil2.nyalakan_mesin()
mobil2.tambah_kecepatan(80)
mobil2.rem(30)
mobil2.tampilkan_info()

```

Output

```
Mesin Toyota Supra telah dinyalakan.
Kecepatan ditambah 100 km/jam.
=== INFORMASI MOBIL ===
Merek: Toyota Supra
Warna: Merah
Tahun Pembuatan: 2022
Kecepatan Maksimal: 250 km/jam
Kecepatan Sekarang: 100 km/jam

---

Mesin Honda Civic telah dinyalakan.
Kecepatan ditambah 80 km/jam.
Mobil mengerem 30 km/jam.
=== INFORMASI MOBIL ===
Merek: Honda Civic
Warna: Hitam
Tahun Pembuatan: 2021
Kecepatan Maksimal: 220 km/jam
Kecepatan Sekarang: 50 km/jam
```

B. Empat Pilar OOP

1 . Pewarisan (Inheritance)

Pewarisan (*Inheritance*) adalah cara untuk membuat kelas baru dengan menggunakan detail (atribut dan *method*) dari kelas yang sudah ada tanpa perlu memodifikasinya. Kelas yang baru terbentuk disebut **kelas turunan** (atau *child class*), sedangkan kelas yang ada disebut **kelas dasar** (atau *parent class*).

```python
# class parent (Kelas Dasar)
class Mobil:
    def __init__(self, merek):
        self.merek = merek
        self.kecepatan = 0  # Atribut dasar: kecepatan awal 0

    def akselerasi(self, tambahan_kecepatan):
        self.kecepatan += tambahan_kecepatan
        print(f"{self.merek} berakselerasi. Kecepatan saat ini: {self.kecepatan} km/jam")

# class child (Kelas Turunan)
class MobilSport(Mobil): # Inheritance ke class Mobil
    def __init__(self, merek, turbo_aktif=False):
        super().__init__(merek) # Memanggil __init__ parent untuk inisialisasi merek dan kecepatan
        self.turbo_aktif = turbo_aktif # Atribut spesifik MobilSport

    def aktifkan_turbo(self):
        self.turbo_aktif = True
        print(f"Turbo pada {self.merek} diaktifkan! 🚀")

# objek
ferrari = MobilSport("Ferrari")
ferrari.akselerasi(50) # Memanggil method 'akselerasi' dari class Mobil (Parent)
ferrari.aktifkan_turbo() # Memanggil method spesifik MobilSport

print(f"Merek: {ferrari.merek}")

```

Output:

```
Ferrari berakselerasi. Kecepatan saat ini: 50 km/jam
Turbo pada Ferrari diaktifkan! 🚀
Merek: Ferrari
```

Berdasarkan contoh di atas, class MobilSport dapat memanggil *method* dan atribut yang telah didefinisikan pada class Mobil meskipun pada class MobilSport atribut dan *method* tersebut tidak didefinisikan ulang. Meskipun *method* diwariskan secara otomatis, **konstruktor** (__init__) tidak. Namun, konstruktor kelas *parent* masih dapat dipanggil dari kelas *child* menggunakan *keyword* super().__init__().

Contoh lain penggunaan Inheritance (Pewarisan)

1. **Parent Class**

   Kelas ini menyediakan konstruktor dan *method* yang akan diwarisi oleh kelas turunan

```python
class Bentuk:
    def __init__(self, nama):
        self.nama = nama
        print(f"Bentuk '{self.nama}' telah dibuat.")

    # Method yang diwarisi
    def deskripsi(self):
        return f"Ini adalah objek dari bentuk {self.nama}."

    # Method yang HARUS di-override oleh Child Class
    def hitung_luas(self):
        # Menyebabkan error jika Child tidak meng-override
        raise NotImplementedError("Subkelas harus mengimplementasikan method 'hitung_luas'")


```

2. Chill Class

Kelas ini mewarisi dari Bentuk dan menambahkan logika spesifiknya

```python
class Persegi(Bentuk):
    def __init__(self, sisi):
        # Memanggil konstruktor Parent (Bentuk)
        super().__init__("Persegi")
        self.sisi = sisi

    # Method Overriding: Implementasi spesifik untuk Persegi
    def hitung_luas(self):
        return self.sisi * self.sisi

# Membuat objek
kotak = Persegi(sisi=4)

# Menggunakan method yang diwarisi dari Bentuk (Parent)
print(kotak.deskripsi())

# Menggunakan method yang di-override (Child-specific)
print(f"Luas {kotak.nama}: {kotak.hitung_luas()}")

```

Output:

```
Bentuk 'Persegi' telah dibuat.
Ini adalah objek dari bentuk Persegi.
Luas Persegi: 16
```

2. Enkapsulasi (Encapsulation)

Enkapsulasi adalah konsep dalam OOP yang membatasi akses langsung ke atribut dan *method* sebuah objek. Tujuannya adalah untuk melindungi data internal objek agar tidak dimodifikasi secara tidak terduga dari luar. Dalam Python, kita dapat menandai atribut sebagai 'privat' (walaupun tidak sepenuhnya ketat seperti bahasa lain) dengan menggunakan awalan garis bawah:

- **Satu garis bawah tunggal (_)**: Menunjukkan bahwa atribut/method tersebut seharusnya diperlakukan sebagai **internal/dilindungi** (*protected*).

  Contoh Penggunaan Satu garis bawah tunggal

```python
class Kendaraan:
    def __init__(self, harga_awal):
        self._harga_jual_dasar = harga_awal

    def tampilkan_harga(self):
        print(f"Harga Jual Kendaraan: Rp {self._harga_jual_dasar:,.0f}")

    def set_harga_jual(self, harga_baru):
        if harga_baru > 0:
            self._harga_jual_dasar = harga_baru
            print("Harga berhasil diperbarui.")
        else:
            print("Harga tidak valid!")

mobil_suv = Kendaraan(200000000)
mobil_suv.tampilkan_harga()

mobil_suv.set_harga_jual(250000000)
mobil_suv.tampilkan_harga()

mobil_suv._harga_jual_dasar = 100000000
mobil_suv.tampilkan_harga()
```

  Output

```
Harga Jual Kendaraan: Rp 200,000,000
Harga berhasil diperbarui.
Harga Jual Kendaraan: Rp 250,000,000
Harga Jual Kendaraan: Rp 100,000,000
```

- **Garis bawah ganda (__)**: Memicu **Name Mangling** oleh Python, menjadikannya sangat sulit (tetapi tidak mustahil) untuk diakses dari luar kelas, yang secara efektif menunjukkan status **privat** (*private*).

  Contoh Penggunaan Garis bawah ganda

```python
class Kendaraan:
    def __init__(self):
        # Atribut privat (menggunakan __): tidak dapat diakses langsung dari luar
        self.__harga_jual_dasar = 200000000

    def tampilkan_harga(self):
        print("Harga jual kendaraan: Rp {}".format(self.__harga_jual_dasar))

    # Method publik untuk memodifikasi atribut privat (Setter)
    def set_harga_jual(self, harga_baru):
        if harga_baru > 0:
            self.__harga_jual_dasar = harga_baru
            print("Harga berhasil diperbarui.")
        else:
            print("Harga tidak valid!")

mobil_suv = Kendaraan()
mobil_suv.tampilkan_harga()

# Mencoba mengubah atribut privat secara langsung (Python tidak mengizinkannya)
mobil_suv.__harga_jual_dasar = 100000000
mobil_suv.tampilkan_harga() # Output tetap 200.000.000

# Menggunakan method set (Setter) yang disediakan untuk mengubah atribut privat
mobil_suv.set_harga_jual(250000000)
mobil_suv.tampilkan_harga()
```

  Output:

```
Harga jual kendaraan: Rp 200000000
Harga jual kendaraan: Rp 200000000
Harga berhasil diperbarui.
Harga jual kendaraan: Rp 250000000
```