# Object Oriented

# OOP?

- **Konsep Dasar OOP**
- **Paradigma Pemrograman**: Cara berpikir tentang masalah dengan memodelkannya sebagai **objek** di dunia nyata.
- **Objek**: Gabungan antara data (**atribut**) dan perilaku (**metode**).
- **Mengapa OOP?**:
  - **Terstruktur**: Kode lebih rapi dan mudah dikelola.
  - **Reusable**: Komponen (class) dapat digunakan kembali.
  - **Mudah Di-maintain**: Perubahan di satu bagian tidak merusak bagian lain.

# Class dan Object

**Blueprint dan Hasilnya** 🏠

- **Class**: Sebuah *blueprint* atau cetak biru. Mendefinisikan properti dan perilaku umum.
  - Contoh: Blueprint Mobil.
- **Object (Instance)**: Hasil nyata yang dibuat dari *class*. Setiap objek memiliki identitasnya sendiri.
  - Contoh: Mobil merah Budi, mobil biru Ani.

```
# CLASS (Blueprint)
class Mobil:
    # Untuk saat ini, kita kosongkan dulu
    pass

# OBJECT (Hasil nyata)
mobil_avanza = Mobil()
mobil_pajero = Mobil()

print(mobil_avanza)
print(mobil_pajero)

# Output:
# <__main__.Mobil object at 0x...>
# <__main__.Mobil object at 0x...>
```

# __init__ dan Atribut

**Konstruktor dan Properti Objek**

- **__init__(self, ...)**: Metode spesial (konstruktor) yang otomatis dijalankan saat objek dibuat.
- **Tujuan**: Untuk memberikan nilai awal pada **atribut** (data) objek.
- **self**: Keyword yang merujuk pada objek spesifik yang sedang dibuat.

## Contoh Kode: __init__

```
class Character:
    # __init__ adalah konstruktor
    def __init__(self, name, health):
        # self.name dan self.health adalah atribut
        self.name = name
        self.health = health
        print(f"{self.name} telah diciptakan!")

# Membuat objek sambil memberikan nilai awal
aragorn = Character("Aragorn", 100)
gandalf = Character("Gandalf", 150)

# Mengakses atributnya
print(f"{aragorn.name} memiliki {aragorn.health} HP.")
# Output: Aragorn memiliki 100 HP.
```

# Methods (Metode)

**Perilaku atau Kemampuan Objek** ⚔️

- **Methods**: Fungsi yang didefinisikan di dalam sebuah *class*.
- **Tujuan**: Mendefinisikan aksi atau perilaku yang bisa dilakukan oleh objek.
- Parameter pertama *method* selalu **self**, agar bisa mengakses atribut objek itu sendiri.

# Contoh Kode: Methods

```
class Character:
    def __init__(self, name, attack_power):
        self.name = name
        self.attack_power = attack_power

    # Ini adalah sebuah method
    def attack(self, target_name):
        # Method bisa mengakses atribut sendiri (self.name)
        print(f"{self.name} menyerang {target_name}!")
        print(f"Damage: {self.attack_power}")

# Membuat objek
aragorn = Character("Aragorn", 15)

# Memanggil method dari objek
aragorn.attack("Orc")
# Output:
# Aragorn menyerang Orc!
# Damage: 15
```

# Pilar OOP #1: Inheritance

**Mewariskan Sifat** 👪

- **Inheritance (Pewarisan)**: Mekanisme di mana sebuah *class* baru (**child class**) dapat mewarisi semua atribut dan *method* dari *class* yang sudah ada (**parent class**).
- **Tujuan**: *Code reuse*. Hindari duplikasi kode.
- Relasi "is-a" (adalah seorang/sebuah). Contoh: Mage **adalah seorang** Character.

# Contoh Kode: Inheritance

```
# Parent Class
class Character:
    def __init__(self, name):
        self.name = name

    def move(self):
        print(f"{self.name} bergerak.")

# Child Class (mewarisi dari Character)
class Mage(Character):
    def cast_spell(self):
        print(f"{self.name} mengeluarkan sihir!")

gandalf = Mage("Gandalf")
gandalf.move()       # Method warisan dari Character
gandalf.cast_spell()   # Method miliknya sendiri
# Output:
# Gandalf bergerak.
# Gandalf mengeluarkan sihir!
```

# Pilar OOP #2: Polymorphism

- **Satu Nama, Banyak Bentuk** 🎭
- **Polymorphism**: Kemampuan objek dari *class* yang berbeda untuk merespons *method* dengan nama yang sama, tetapi dengan cara yang berbeda.
- Bentuk paling umum adalah **Method Overriding**, di mana *child class* menyediakan implementasi baru untuk *method* yang sudah ada di *parent class*.

# Contoh Kode: Polymorphism

```
class Animal:
    def speak(self):
        print("Suara hewan...")

class Cat(Animal):
    # Method overriding
    def speak(self):
        print("Meow!")

class Dog(Animal):
    # Method overriding
    def speak(self):
        print("Woof!")

cat = Cat()
dog = Dog()

cat.speak() # Output: Meow!
dog.speak() # Output: Woof!
```

# Pilar OOP #3: Encapsulation

**Membungkus Data** 🛡️
•**Encapsulation (Pembungkusan)**: Menyatukan data (atribut) dan *method* yang mengoperasikannya ke dalam satu unit (*class*).
•Juga termasuk **menyembunyikan detail internal** objek dari akses luar.
Di Python, ini dilakukan dengan konvensi awalan underscore (_ atau __).
•**Tujuan**: Melindungi data agar tidak diubah secara sembarangan.

# Contoh Kode: Encapsulation

```
class Player:
    def __init__(self, name):
        self.name = name
        self._health = 100 # Atribut "protected"

    def take_damage(self, amount):
        # Logika untuk mengubah health ada di sini
        if self._health > 0:
            self._health -= amount
        print(f"{self.name} HP: {self._health}")

player1 = Player("Budi")
# Jangan lakukan ini: player1._health = 999
# Gunakan method yang disediakan:
player1.take_damage(20) # Output: Budi HP: 80
```

# Pilar OOP #4: Abstraction

- **Menyembunyikan Kompleksitas** 🚙
- **Abstraction (Abstraksi)**: Menyembunyikan detail implementasi yang rumit dan hanya menunjukkan fungsionalitas yang esensial kepada pengguna.
- **Contoh**: Anda hanya perlu tahu cara menekan pedal gas mobil, bukan cara kerja mesin di dalamnya. *Method* adalah bentuk abstraksi.

# Contoh Kode: Abstraction

```
# Pengguna class ini tidak perlu tahu bagaimana
# detail koneksi atau query SQL bekerja.
class Database:
    def __init__(self, connection_string):
        self._connect(connection_string)

    def _connect(self, conn_str):
        # Logika rumit untuk koneksi ada di sini...
        print("Koneksi berhasil!")

    # Ini adalah interface yang simpel
    def get_user_data(self, user_id):
        # Detail query SQL yang rumit disembunyikan di sini
        print(f"Mengambil data untuk user {user_id}...")
        return {"id": user_id, "name": "Budi"}

# Pengguna hanya perlu tahu ini:
db = Database("server=xyz;user=abc;")
user = db.get_user_data(123)
```

# Static & Private Methods

**Metode Khusus**

- **Static Method (`@staticmethod`):**
  - Fungsi *utility* di dalam *class*.
  - Tidak terikat pada objek (`self`).
  - Dipanggil langsung dari *class*: `NamaClass.nama_method()`.
- **Private Method (`__nama_method`):**
  - *Method* internal yang hanya boleh digunakan di dalam *class* itu sendiri.
  - Diawali dua underscore `__`.

# Contoh Kode: Static & Private Methods

```
class Kalkulator:
    def __init__(self):
        self._history = [] # Private attribute

    def hitung(self, op, a, b):
        hasil = self._lakukan_perhitungan(op, a, b)
        return hasil

    def _lakukan_perhitungan(self, op, a, b): # Private
        if op == '+': return self.tambah(a, b)
        # ... logika lain

    @staticmethod
    def tambah(a, b): # Static
        return a + b

# Memanggil static method
hasil = Kalkulator.tambah(10, 5) # Output: 15
```

# @property Decorator

**Metode yang Berpura-pura Jadi Atribut** 🏷️

- **@property**: Mengubah *method* menjadi atribut yang *read-only* (hanya bisa dibaca).
- **@nama.setter**: Memberikan kontrol untuk memvalidasi data saat atribut diubah nilainya.
- **Tujuan**: Menjaga sintaks tetap simpel (objek.nama) sambil memiliki kontrol layaknya sebuah *method*.

# Contoh Kode: @property

```
class Produk:
    def __init__(self, harga):
        self._harga = harga

    @property
    def harga(self):
        """Getter: dipanggil saat 'produk.harga' dibaca"""
        return self._harga

    @harga.setter
    def harga(self, nilai_baru):
        """Setter: dipanggil saat 'produk.harga = nilai'"""
        if nilai_baru < 0:
            raise ValueError("Harga tidak boleh negatif!")
        self._harga = nilai_baru

baju = Produk(50000)
print(baju.harga) # Memanggil getter
baju.harga = 45000 # Memanggil setter
```

# Operator Overloading

**Memberi Arti Baru pada Operator +, -, ==**

•**Operator Overloading**: Mendefinisikan ulang perilaku operator standar untuk objek buatan kita.
•Dilakukan dengan mengimplementasikan *magic method*
seperti __add__ (untuk +), __eq__ (untuk ==), dan __lt__ (untuk <).
•**Tujuan**: Membuat kode lebih intuitif dan mudah dibaca.

# Contoh Kode: Operator Overloading

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

    # Overloading untuk operator +
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # Overloading untuk print()
    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 4)
v2 = Vector(5, 1)
v3 = v1 + v2 # Memanggil v1.__add__(v2)

print(v3) # Output: Vector(7, 5)
```

# Custom Iterator

**Membuat Objek yang Bisa Di-looping** 📜

- **Iterator**: Objek yang memungkinkan kita melintasinya satu per satu.
- Untuk membuat *custom iterator*, sebuah *class* harus memiliki:
  - **__iter__(self)**: Mengembalikan objek iterator (biasanya self).
  - **__next__(self)**: Mengembalikan item berikutnya. Jika habis, raise StopIteration.
- **Tujuan**: Efisiensi memori untuk data besar dan logika perulangan yang kustom.

# Contoh Kode: Custom Iterator

```
class Countdown:
  def __init__(self, start):
    self.current = start

  def __iter__(self):
    return self

  def __next__(self):
    if self.current < 0:
      raise StopIteration
    else:
      value = self.current
      self.current -= 1
      return value

# Objek kita sekarang bisa dipakai di for loop!
for i in Countdown(3):
  print(i)
# Output:
# 3
# 2
# 1
# 0
```