Python OOP (Object Oriented Programming)

•••

Eko Kurniawan Khannedy

Eko Kurniawan Khannedy

- Technical architect at one of the biggest ecommerce company in Indonesia
- 15+ years experiences
- www.programmerzamannow.com
- youtube.com/c/ProgrammerZamanNow

Eko Kurniawan Khannedy

- Telegram : @khannedy
- Linkedin : https://www.linkedin.com/company/programmer-zaman-now/
- Facebook : fb.com/ProgrammerZamanNow
- Instagram : instagram.com/programmerzamannow
- Youtube : youtube.com/c/ProgrammerZamanNow
- Telegram Channel : t.me/ProgrammerZamanNow
- Tiktok : https://tiktok.com/@programmerzamannow
- Email : echo.khannedy@gmail.com

Sebelum Belajar

- Python Dasar
- Python Modules

Pengenalan OOP

Apa itu Object-Oriented Programming?

- Object-Oriented Programming (OOP) adalah paradigma programming yang mengorganisir code berdasarkan objects dan classes.
- Bayangkan OOP seperti cara kita berpikir dalam kehidupan nyata - setiap benda memiliki karakteristik (properties) dan kemampuan (behaviors).
- Contoh Sederhana:
  - Mobil memiliki karakteristik: warna, merk, tahun
  - Mobil memiliki kemampuan: maju, mundur, belok, klakson
- Dalam OOP:
  - Karakteristik = Attributes (variables dalam object)
  - Kemampuan = Methods (functions dalam object)

Keuntungan Object-Oriented Programming

- Data dan behavior tergabung dalam satu unit
- Lebih mudah dipahami dan maintain
- Reusable - bisa buat multiple instances
- Data protection - tidak mudah corrupt
- Scalable untuk sistem yang kompleks

4 Pilar Object-Oriented Programming

- Encapsulation (Pembungkusan), menggabungkan data dan methods dalam satu unit (class), dan menyembunyikan detail implementasi.
- Inheritance (Pewarisan), child class mewarisi properties dan methods dari parent class.
- Polymorphism (Banyak Bentuk), objects yang berbeda bisa merespons method call yang sama dengan cara berbeda.
- Abstraction (Abstraksi), menyembunyikan complexity dan hanya menampilkan interface yang penting.

Real-World Analogies

Pabrik Mobil

- Class = Blueprint/Cetakan mobil
- Object = Mobil yang diproduksi dari blueprint
- Attributes = Warna, mesin, roda
- Methods = Maju, mundur, klakson

Membuat Project

Membuat Project

- Buat folder belajar-python-oop
- Buat virtual environment dengan perintah :
  python -m venv .venv
- Aktivasi virtual environment dengan perintah :
  (Mac / Linux) :
  source .venv/bin/activate
  (Windows) :
  .venv\Scripts\activate

Class dan Object

Apa itu Class dan Object?

- Class adalah blueprint atau template untuk membuat objects. Bayangkan class seperti cetakan kue - dengan satu cetakan, kita bisa membuat banyak kue dengan bentuk yang sama.
- Object adalah instance dari class - hasil "produksi" dari blueprint class. Setiap object memiliki data dan behavior yang sama strukturnya, tapi valuenya bisa berbeda.

class_dan_object.py

class_dan_object.py

```python
class Kampus:
    pass

class Mahasiswa:
    pass
```

class_dan_object.py

```
kampus1 = Kampus()
kampus2 = Kampus()

print(type(kampus1))
print(type(kampus2))

mahasiswa1 = Mahasiswa()
mahasiswa2 = Mahasiswa()

print(type(mahasiswa1))
print(type(mahasiswa2))
```

Class dengan Attributes

- Attributes adalah variables yang menyimpan data dalam object.

class_dan_object.py

```python
# instance attribute

mahasiswa1 = Mahasiswa()
mahasiswa1.nim = 123456
mahasiswa1.nama = "Arya"

print(mahasiswa1.nim)
print(mahasiswa1.nama)

mahasiswa2 = Mahasiswa()
mahasiswa2.nim = 123457
mahasiswa2.nama = "Budi"

print(mahasiswa2.nim)
print(mahasiswa2.nama)
```

Class Attribute

- Instance attribute hanya akan ada di object hasil dari instansiasi dari class, dan tidak akan ada di object lain walaupun dari class yang sama
- Jika kita ingin membuat atribut yang secara default (bawaan) ada di semua object hasil instansiasi, maka kita perlu membuat Class Attribute, yaitu atribut yang di definisikan di dalam class nya

class_and_object.py

```python
class Kampus:
    nama = ""
    alamat = ""

kampus = Kampus()
print(kampus.nama)
print(kampus.alamat)

class Mahasiswa:
    nim = 0
    nama = ""

mahasiswa = Mahasiswa()
print(mahasiswa.nim)
print(mahasiswa.nama)
```

Class dengan Methods

- Methods adalah functions yang berada dalam class dan bisa dipanggil oleh objects.

class_dan_object.py

```python
class Mahasiswa:
    nim = 0
    nama = ""

    def perkenalan(self):
        print(f"Halo nama saya {self.nama}")


mahasiswa = Mahasiswa()
mahasiswa.nama = "Arya"
mahasiswa.perkenalan()
```

Parameter self

- Parameter self - Sangat Penting!
- self adalah parameter khusus yang merujuk pada instance/object yang sedang memanggil method.

Method dengan Parameters

- Parameter di method tidak hanya self, kita bisa menambahkan parameter lain seperti function biasanya
- Semua yang bisa kita lakukan di function bisa kita lakukan di method

class_dan_object.py

```python
class Mahasiswa:
    nim = 0
    nama = ""

    def perkenalan(self):
        print(f"Halo nama saya {self.nama}")

    def hello(self, nama):
        print(f"Halo {nama}, nama saya {self.nama}")


mahasiswa = Mahasiswa()
mahasiswa.nama = "Arya"
mahasiswa.perkenalan()
mahasiswa.hello("Budi")
```

Naming Conventions

- Class Names: PascalCase, misal Mahasiswa, MataPelajaran, Mobil, KategoriBarang
- Method Names: snake_case, misal perkenalan(), lari_ke_depan(), jalankan(), dan lain-lain
- Attribute Names: snake_case, misal nama, nama_depan, nama_belakang, dan lain-lain

Constructor dan Special Methods

Apa itu Constructor?

- Constructor adalah method special yang otomatis dipanggil saat object dibuat.
- Dalam Python, constructor adalah method __init__().
- Constructor digunakan untuk initialize (setup awal) object dengan data yang diperlukan.

constructor.py

constructor.py

```python
# Tanpa Constructor

class Mahasiswa:
    nim = 0
    nama = ""

    def setup(self, nama, nim):  # Manual setup
        self.nama = nama
        self.nim = nim


# Harus panggil setup manual
mhs = Mahasiswa()
mhs.setup("Alice", "123456")  # Extra step

```

constructor.py

```python
class Mahasiswa:
    nim = 0
    nama = ""

    def __init__(self, nama, nim):  # Manual setup
        self.nama = nama
        self.nim = nim


# Harus panggil setup manual
mhs = Mahasiswa("Alice", "123456")
print(mhs.nama)
print(mhs.nim)
```

Constructor dengan Validation

- Salah satu keuntungan menggunakan constructor adalah, kita bisa menambahkan validasi saat object pertama kali dibuat secara otomatis
- Contohnya misal kita bisa cek nilai dari parameter, jika tidak valid, kita bisa raise error
- Materi detail tentang raise error akan dibahas di materi khusus di kelas ini

constructor.py

```
16  class BankAccount:
17      no = ""
18      saldo = 0
19
20      def __init__(self, no, saldo=0):
21          # validasi saldo
22          if saldo < 0:
23              raise ValueError("Saldo tidak boleh negatif")
24
25  budi = BankAccount("1234567890", 1000) # sukses
26  eko = BankAccount("1234567890", -1000) # error
27
```

Method __str__() - String Representation

- Method __str__() menentukan bagaimana object ditampilkan sebagai string.

constructor.py

```python
class Mahasiswa:
    nim = 0
    nama = ""

    def __init__(self, nama, nim):  # Manual setup
        self.nama = nama
        self.nim = nim

    def __str__(self):
        return f"Mahasiswa nim = {self.nim}, nama = {self.nama}"


# Harus panggil setup manual
mhs = Mahasiswa("Alice", "123456")
print(f"Info {mhs}")
```

Methods untuk Object Comparison

- Method __eq__() - Equality, bisa digunakan untuk membandingkan object dengan object lainnya
- Saat kita menggunakan operator perbandingan == , maka method ini yang akan dipanggil

constructor.py

```python
    # class Mahasiswa
    def __eq__(self, other):
        return self.nim == other.nim and self.nama == other.nama

mhs1 = Mahasiswa("Alice", "123456")
mhs2 = Mahasiswa("Alice", "123456")
print(mhs1 == mhs2)
```

Methods Comparison Lainnya

- Selain __eq__, kita juga bisa membuat method untuk operator perbandingan yang lain
- __lt__(self, other) untuk < (kurang dari)
- __gt__(self, other) untuk > (lebih dari)
- __le__(self, other) untuk <= (kurang atau sama dengan)
- __ge__(self, other) untuk >= (lebih dari atau sama dengan)

Decorator

Apa itu Decorator?

- Decorator adalah cara untuk "membungkus" atau memodifikasi behavior dari function atau method.
- Decorator ditandai dengan simbol @ diikuti nama decorator, dan ditulis di atas function/method.
- Python menyediakan beberapa built-in decorators untuk OOP:
- @staticmethod - membuat method yang tidak perlu self atau cls
- @classmethod - membuat method yang menerima cls sebagai parameter pertama
- @property - membuat method yang bisa diakses seperti attribute

Static Methods - Independent Functions

- Static method adalah method yang berdiri sendiri secara independen
- Untuk mengakses static method, kita tidak perlu menggunakan object, kita bisa langsung menggunakan class nya
- Static method tidak bisa mengakses class attribute ataupun instance attribute
- Untuk membuat static method, kita bisa menambahkan decorator @staticmethod pada method di class

decorator.py

decorator.py

```python
class Matematika:

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

result = Matematika.tambah(10, 20)
print(result)

```

Class Method

- Class method adalah method yang menerima class sebagai parameter pertama (biasanya dinamakan cls), bukan instance.
- Berguna untuk factory methods atau operasi yang berhubungan dengan class itu sendiri.
- Class method bisa mengakses class attribute, namun tidak bisa mengakses instance attribute

decorator.py

```python
class BankAccount:
    no = ""
    balance = 0
    active = True

    def __init__(self, no, balance=0):
        self.no = no
        self.balance = balance

    @classmethod
    def disabled(cls, no, balance=0):
        result = cls(no, balance)
        result.active = False
        return result

bank_account1 = BankAccount("1", 10000)
bank_account2 = BankAccount.disabled("2", 20000)
print(f"Bank account {bank_account1.no} has balance {bank_account1.balance} and status {bank_account1.active}")
print(f"Bank account {bank_account2.no} has balance {bank_account2.balance} and status {bank_account2.active}")
```

Kapan Menggunakan Masing-masing?

- Instance Method: Ketika butuh akses ke instance (object) data
- Class Method: Untuk factory methods (method untuk pembuatan instance object), atau operasi pada class level
- Static Method: Untuk utility functions yang berhubungan dengan class tapi tidak butuh instance/class data

Property Methods untuk Mengontrol Akses

- Salah satu kekurangan menggunakan attribute adalah, kita tidak bisa mengontrol nilai yang dimasukkan atau didapatkan
- Jika menggunakan function, maka hal ini bisa dilakukan, karena kita bisa memvalidasi nilai terlebih dahulu di function
- Banyak yang menggunakan method setter dan getter untuk mengontrol akses ke attribute

decorator.py

```python
class Category:
    _name = ""

    def set_name(self, name):
        if name == "":
            raise ValueError("Nama tidak boleh kosong")
        self._name = name

    def get_name(self):
        return self._name

category1 = Category()
category1.set_name("Laptop")
print(category1.get_name())
```

Property Decorator

- Di Python baru, terdapat fitur bernama property decorator, dimana kita bisa menandai function sebagai setter dan getter menggunakan decorator @property
- Selanjutnya, kita bisa menggunakan method tersebut layaknya seperti menggunakan attribute

decorator.py

```python
class Category:
    _name = ""

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        if name == "":
            raise ValueError("Nama tidak boleh kosong")
        self._name = name


category1 = Category()
category1.name = "Laptop"
print(category1.name)
```

Encapsulation

Apa itu Encapsulation?

- Encapsulation adalah prinsip OOP yang menggabungkan data dan methods dalam satu unit (class) dan mengontrol akses ke data tersebut.
- Bayangkan encapsulation seperti kotak aman - data penting disimpan di dalam, dan hanya bisa diakses melalui cara yang terkontrol.

Mengapa Encapsulation Penting?

- Data Protection - Mencegah data diubah sembarangan
- Data Validation - Memastikan data selalu dalam kondisi valid
- Interface Control - Menentukan bagaimana data boleh diakses
- Maintenance - Mudah mengubah internal implementation
- Security - Menyembunyikan detail sensitive

Access Modifiers dalam Python

- Python menggunakan naming conventions untuk menandai level akses:
- Public Attributes (Default), bisa diakses dari mana saja - tidak ada protection.
- Protected Attributes - Single Underscore _, convention bahwa attribute ini untuk internal use class dan subclasses.
- Private Attributes - Double Underscore __, Python akan melakukan name mangling (pengubahan nama attribute dari luar) sehingga attribute sulit diakses dari luar.

encapsulation.py

```python
class BankAccount:
    __no = ""
    __balance = 0

    def __init__(self, no):
        self.__no = no

    def get_balance(self):
        return self.__balance

    def topup(self, amount):
        self.__balance += amount

    def cashout(self, amount):
        if amount > self.__balance:
            raise ValueError("Saldo tidak mencukupi")
        self.__balance -= amount


eko_account = BankAccount("eko")
eko_account.topup(1000000)
print(eko_account.get_balance())
eko_account.cashout(500000)
print(eko_account.get_balance())
```

Getter dan Setter Methods

- Di materi sebelumnya kita sudah kenalan dengan decorator dan @property.
- @property decorator untuk membuat getter/setter yang lebih elegan dibanding manual menggunakan getter dan setter method.
- Property decorator memungkinkan kita menggunakan syntax seperti attribute biasa, tapi dengan control seperti method.

encapsulation.py

```python
class BankAccount:
    __no = ""
    __balance = 0

    def __init__(self, no):
        self.__no = no

    @property
    def balance(self):
        return self.__balance

    def topup(self, amount):
        self.__balance += amount

    def cashout(self, amount):
        if amount > self.__balance:
            raise ValueError("Saldo tidak mencukupi")
        self.__balance -= amount


eko_account = BankAccount("eko")
eko_account.topup(1000000)
print(eko_account.balance)
eko_account.cashout(500000)
print(eko_account.balance)
```

Best Practices untuk Encapsulation

- Gunakan private attributes untuk sensitive data
- Pastikan mengontrol akses melalui methods atau properties
- Validasi input dalam setters method
- Gunakan getter method untuk kalkulasi property
- Tetap buat internal method atau attribute private dengan prefix _ atau __
- Jangan expose internal data tanpa validation
- Jangan buat semua attributes private tanpa alasan
- Jangan lupa validation dalam constructor dan setters
- Jangan akses private attributes dari luar class

Inheritance

Apa itu Inheritance (Pewarisan)?

- Inheritance adalah kemampuan untuk membuat class baru berdasarkan class yang sudah ada.
- Class baru (child/subclass) mewarisi properties dan methods dari class lama (parent/superclass), dan bisa menambahkan atau mengubah functionality.

Analogi Dunia Nyata

- Kendaraan (parent) → Mobil, Motor, Pesawat (children)
- Hewan (parent) → Anjing, Kucing, Burung (children)
- Karyawan (parent) → Manager, Developer, Designer (children)

Keuntungan Inheritance

- Code Reuse - Tidak menulis ulang kode yang sama
- Consistency - Interface/blueprint yang sama antar class yang berelasi
- Maintainability - Mengubah parent class berefek ke semua children class
- Extensibility - Mudah menambah tipe baru tanpa ubah terlalu banyak kode
- Polymorphism - Memperlakukan object yang berbeda dengan interface/blueprint yang sama

inheritance.py

```python
# class Parent
class Kendaraan:
    def __init__(self, merek, tahun):
        self.merek = merek
        self.tahun = tahun

    def info(self):
        return f"Merek: {self.merek}, Tahun: {self.tahun}"

    def nyalakan(self):
        print(f"{self.info()} dinyalakan")
```

inheritance.py

```python
# class Child Mobil
class Mobil(Kendaraan):

    def klakson(self):
        print(f"Mobil {self.info()} memiliki klakson")

avanza = Mobil("Avanza", 2019)
avanza.nyalakan()
avanza.klakson()

# class Child Motor
class Motor(Kendaraan):

    def klakson(self):
        print(f"Motor {self.info()} tidak memiliki klakson")

scoopy = Motor("Scoopy", 2018)
scoopy.nyalakan()
scoopy.klakson()
```

Function super() - Akses Parent Class

- super() digunakan untuk mengakses methods dan attributes dari parent class.
- super() biasanya sering digunakan untuk memanggil constructor (__init__) di parent class

inheritance.py

```python
# class Child Mobil
class Mobil(Kendaraan):

    def __init__(self, merek, tahun, jumlah_roda):
        super().__init__(merek, tahun)
        self.jumlah_roda = jumlah_roda

    def klakson(self):
        print(f"Mobil {self.info()} memiliki klakson")

avanza = Mobil("Avanza", 2019, 4)
avanza.nyalakan()
avanza.klakson()
print(avanza.jumlah_roda)
```

Method Overriding

- Method Overriding adalah mendefinisikan ulang method parent di child class dengan implementasi berbeda.
- Saat kita melakukan method overriding, jika kita ingin memanggil method parent class yang sama, kita bisa memanfaatkan super().

inheritance.py

```python
# class Child Motor
class Motor(Kendaraan):

    def klakson(self):
        print(f"Motor {self.info()} tidak memiliki klakson")

    def nyalakan(self):
        # jika ingin menggunakan method nyalakan dari parent class
        # super().nyalakan()
        print(f"Motor {self.merek} dinyalakan secara otomatis")


scoopy = Motor("Scoopy", 2018)
scoopy.nyalakan()
scoopy.klakson()
```

Multilevel Inheritance

- Class bisa inherit dari class yang juga sudah inherit (inheritance chain).
- Tidak ada batasan untuk inheritance
- Namun jangan terlalu banyak level pewarisannya karena akan sulit untuk di maintain dan sulit untuk dibaca kodenya

inheritance.py

```python
class Karyawan:
    def __init__(self, nama, gaji):
        self.nama = nama
        self.gaji = gaji

class KaryawanTetap(Karyawan):
    pass

class Manager(KaryawanTetap):
    pass

class VicePresident(Manager):
    pass
```

Multiple Inheritance

- Di beberapa bahasa pemrograman, class tidak bisa melakukan pewarisan dari lebih dari satu class
- Namun di Python itu bisa dilakukan
- Namun perlu diingat, pewarisan lebih dari satu kadang bisa berbahaya, apalagi jika terjadi diamond problem
- Jadi sebisa mungkin jika memang tidak butuh melakukan pewarisan lebih dari satu class, jangan lakukan hal ini

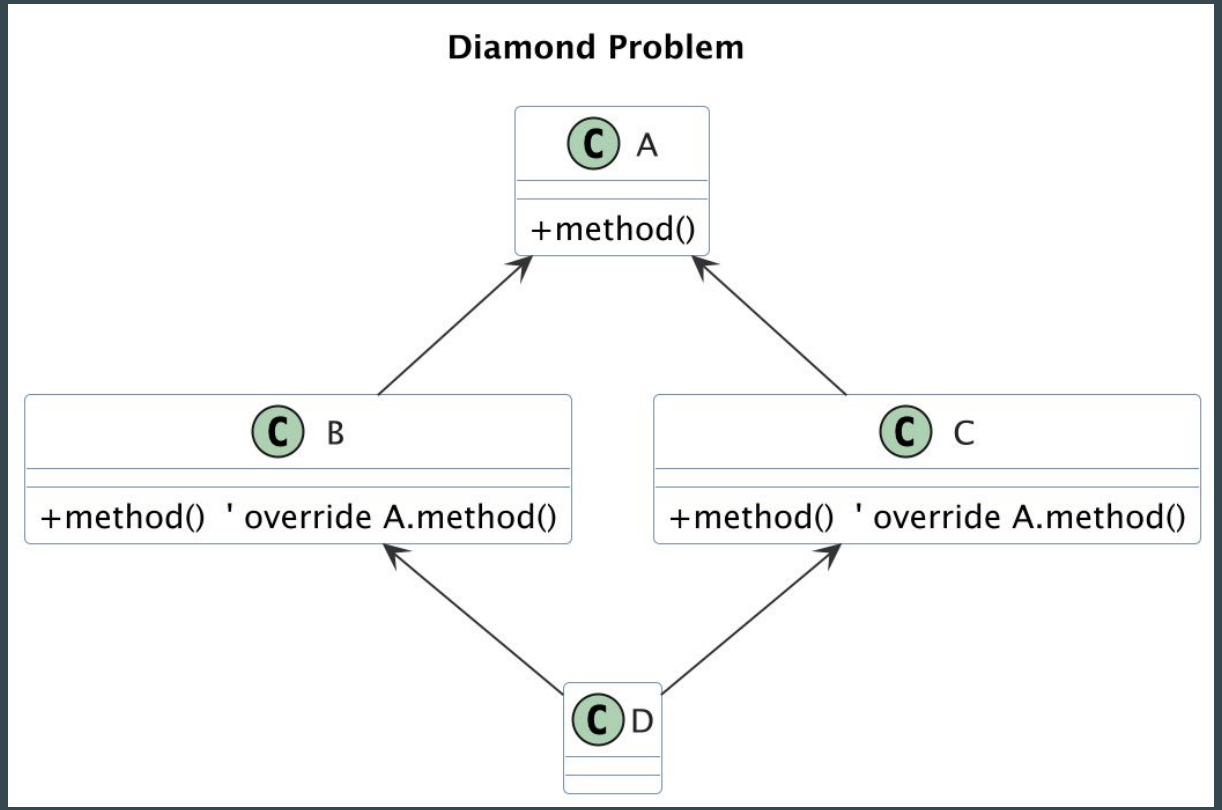
inheritance.py

```python
class BisaBerenang:
    def berenang(self):
        print("Bisa berenang")


class BisaBerlari:
    def berlari(self):
        print("Bisa berlari")


class Atlit(BisaBerenang, BisaBerlari):
    def __init__(self, nama):
        self.nama = nama

eko = Atlit("Eko")
eko.berenang()
eko.berlari()
```

Diamond Problem

- Seperti yang dijelaskan sebelumnya, multiple inheritance bisa menyebabkan diamond problem
- Misal kita punya class A
- Class B adalah child A
- Class C adalah child A
- Lalu ada class D child B dan C
- Maka bentuk relasi 4 class tersebut akan terjadi diamond problem

Diagram : Diamond Problem



inheritance.py

```python
class A:
    def method(self):
        return "Method from A"

class B(A):
    def method(self):
        return "Method from B"

class C(A):
    def method(self):
        return "Method from C"

class D(B, C):
    pass

# Python menggunakan Method Resolution Order (MRO)
d = D()
print(d.method())
```

Type Checking

- isinstance(value, Type) - Cek tipe object
- Akan menghasilkan True jika Class-nya adalah adalah Type tersebut atau Sub Class dari Type tersebut
- Jika bukan Class atau Sub Class dengan Type tersebut, maka hasilnya adalah False

inheritance.py

```python
eko = Karyawan("Eko", 1000000)
tono = KaryawanTetap("Tono", 1000000)
budi = Manager("Budi", 1000000)
joko = VicePresident("V", 1000000)

print(isinstance(eko, Karyawan)) # True
print(isinstance(tono, Karyawan)) # True
print(isinstance(budi, Karyawan)) # True
print(isinstance(joko, Karyawan)) # True
```

Best Practices untuk Inheritance

- Selalu panggil super().__init__() dalam child constructor
- Hanya override methods yang butuh behavior berbeda
- Gunakan isinstance() untuk type checking
- Level inheritance tidak terlalu dalam (max 3-4 levels)
- Jangan override tanpa alasan yang jelas
- Jangan buat inheritance hierarchy yang terlalu kompleks
- Jangan multiple inheritance kecuali benar-benar diperlukan

Polymorphism

Apa itu Polymorphism?

- Polymorphism berasal dari bahasa Yunani yang berarti "banyak bentuk" (poly = banyak, morph = bentuk).
- Dalam OOP, polymorphism adalah kemampuan objects yang berbeda untuk merespons method call yang sama dengan cara yang berbeda.

Analogi Sederhana Polymorphism

- Bayangkan kita punya remote control universal:
- Tombol "PLAY" yang sama bisa:
- TV: Mulai memutar channel
- DVD Player: Mulai memutar disc
- Speaker: Mulai memutar musik
- Game Console: Mulai game
- Satu interface yang sama (tombol PLAY), tapi behavior berbeda tergantung device-nya.

Konsep Dasar Polymorphism

- Method Overriding, subclass bisa override method dari parent class dengan implementasi berbeda.
- Duck Typing, "Jika berjalan seperti bebek dan bersuara seperti bebek, maka itu bebek"
- Python tidak peduli tipe object, yang penting object punya method yang dibutuhkan.

polymorphism.py - Polymorphism

```python
class Hewan:
    def __init__(self, nama):
        self.nama = nama

    def suara(self):
        return "Hewan bersuara"


class Anjing(Hewan):
    def suara(self):  # Override method
        return "Guk guk!"


class Kucing(Hewan):
    def suara(self):  # Override method
        return "Meow!"
```

```python
class Sapi(Hewan):
    def suara(self):  # Override method dari parent
        return "Moooo!"


# Polymorphism in action
hewan_list = [
    Anjing("Buddy"),
    Kucing("Whiskers"),
    Sapi("Bessie")
]

# Method yang sama, behavior berbeda
for hewan in hewan_list:
    print(hewan.suara())
```

polymorphism.py - Duck Typing

```python
class Mobil:
    def start(self):
        return "mesin mobil menyala"

class Motor:
    def start(self):
        return "mesin motor menyala"

class Perahu:
    def start(self):
        return "mesin perahu menyala"
```

```python
# Function yang polymorphic
def operasikan_kendaraan(kendaraan):
    print(kendaraan.start())


# Polymorphism dengan duck typing
kendaraan_list = [
    Mobil(),
    Motor(),
    Perahu()
]

for kendaraan in kendaraan_list:
    operasikan_kendaraan(kendaraan)
```

Operator Overloading

- Python memungkinkan kita override operators (+, -, *, ==, dll) dengan magic methods.
- __add__ untuk +
- __sub__ untuk -
- __mul__ untuk *
- __eq__ untuk ==
- __lt__ untuk <
- __le__ untuk <=
- __gt__ untuk >
- __ge__ untuk >=
- __ne__ untuk !=

polymorphism.py - Operator Overloading

```python
class Apple:
    def __init__(self, jumlah):
        self.jumlah = jumlah

    def __add__(self, other):
        return Apple(self.jumlah + other.jumlah)

    def __str__(self):
        return f"Apple: {self.jumlah}"

apple1 = Apple(5)
apple2 = Apple(3)
apple3 = apple1 + apple2
print(apple3)
```

Polymorphism dengan Interface (Abstract Base Class)

- Jika sebelumnya kita pernah belajar Java atau C#, terdapat jenis tipe Interface, yang sering dijadikan kontrak untuk Class
- Di Python, hal itu tidak ada, namun kita bisa menggunakan konsep ABC (Abstract Base Class), yaitu konsep class yang method nya abstract dan belum ada implementasinya
- Dengan ini Subclass dipaksa untuk implementasi dari method yang ada di Parent nya
- Untuk menandai method sebagai abstract, kita bisa gunakan decorator @abstractmethod

polymorphism.py - Abstract Base Class

```python
from abc import ABC, abstractmethod
import math

class Shape(ABC):

    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```

```python
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

shape = [
    Rectangle(5, 3),
    Circle(2)
]

for s in shape:
    print(f"Area is {s.area()}")
```

Kapan Menggunakan Polymorphism?

- Multiple classes dengan interface yang sama tapi behavior berbeda
- Butuh flexibilitas untuk menangani tipe yang berbeda dengan cara yang sama

Contoh Kasus Polymorphism

- Sistem pembayaran (Credit Card, Bank Transfer, E-wallet)
- Format file (PDF, Word, Excel readers)
- Jembatan ke sistem basis data (MySQL, PostgreSQL, SQLite)
- Karakter game dengan kemampuan bisa berbeda-beda
- Sistem notifikasi (Email, SMS, Push notification)

Exception Handling

Mengapa Exception Handling Penting dalam OOP?

- Dalam Object-Oriented Programming, kita sering bekerja dengan objects yang kompleks yang bisa mengalami berbagai error conditions.
- Exception handling membantu kita:
- Protect object state - Mencegah object rusak karena error
- Provide meaningful feedback - User mendapat informasi error yang jelas
- Graceful degradation - Program tetap berjalan meski ada error
- Debugging assistance - Lebih mudah untuk melacak dan memperbaiki masalah

Membuat Exception

- Sebelumnya pernah beberapa kali kita menggunakan kata kunci raise
- Kata kunci raise, digunakan untuk membuat exception terjadi
- Hal ini kadang diperlukan agar kode berhenti dan ditangkap oleh try except (yang pernah kita bahas di materi Python Dasar)

exception.py - Membuat Exception

```python
class BankAccount:
    def __init__(self, no, balance=0):
        self.no = no
        self.balance = balance

    def transter(self, amount):
        if amount > self.balance:
            raise ValueError("Saldo tidak mencukupi")
        self.balance -= amount


try:
    bank_account = BankAccount("1234567890", 100)
    bank_account.transter(1000000)
except ValueError as e:
    print(f"Error: {e}")
```

Custom Exceptions

- Kita bisa membuat custom exception classes untuk error conditions yang spesifik.
- Caranya kita perlu membuat Subclass dari class Exception

exception.py - Custom Exception Class

```python
class BalanceNotEnough(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message

class BankAccount:
    def __init__(self, no, balance=0):
        self.no = no
        self.balance = balance

    def transter(self, amount):
        if amount > self.balance:
            raise BalanceNotEnough("Saldo tidak mencukupi")
        self.balance -= amount
```

```python
try:
    bank_account = BankAccount("1234567890", 100)
    bank_account.transter(1000000)
except BalanceNotEnough as e:
    print(f"Error: {e}")
```

Best Practices Exception Handling

- Specific exceptions - Buat custom exception classes untuk jenis error berbeda
- Fail fast - Validasi input di awal method
- Protect object state - Jangan biarkan object corrupt karena error
- Meaningful messages - Pesan error yang informatif
- Log exceptions - Track errors untuk debugging
- Don't catch all - Hanya tangkap exceptions yang bisa ditangani

Proyek Nyata

Proyek Nyata

- Mengubah Aplikasi Ujian Sekolah menjadi OOP
- Membuat Aplikasi Manajemen Aset (Barang di Perusahaan)

Aplikasi Ujian Sekolah

Aplikasi Ujian Sekolah

- Ubah Aplikasi Ujian Sekolah menjadi OOP
- https://github.com/ProgrammerZamanNow/belajar-python-dasar/blob/main/app_ujian_sekolah.py

Aplikasi Manajemen Aset

Aplikasi Manajemen Aset

- Membuat Aplikasi Manajemen Aset
- Setiap Aset memiliki jenis yang berbeda, misal ada Kendaraan, Furniture, Tanah, Bangunan dan Alat Elektronik
- Aplikasi bisa digunakan untuk menambah, mengubah, menghapus dan mencari aset

Penutup