

MODUL PRAKTIKUM
PEMROGRAMAN BERORIENTASI OBJEK
Minggu 4



PEWARISAN DAN POLIMORFISME
(OVERLOADING, OVERRIDING, DYNAMIC CAST)

Disusun Oleh :

Andhika Putra Pratama	119140224
Andhika Wibawa B.	119140218
Nurul Aulia Larasati	119140088
Ihza Fajrur Rachman H.	119140130
Enrico Johanes S.	119140021
M. Ammar Fadhila R.	119140029

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK ELEKTRO, INFORMATIKA DAN SISTEM FISIS
INSTITUT TEKNOLOGI SUMATERA
2022

1. Inheritance (Pewarisan)

Inheritance adalah salah satu konsep dasar dari Pemrograman Berbasis Objek (OOP). Pada inheritance, kita dapat menurunkan kelas dari kelas lain untuk hirarki kelas yang saling berbagi atribut dan metode. Contohnya terdapat *base class* “Animal” dan terdapat *class* “Horse” yang merupakan turunan dari class “Animal”. ini berarti class “Horse” memiliki atribut dan method yang sama dengan class “Animal”. Dan objek “Horse” dapat menggantikan objek “Animal” dalam aplikasi.

Berikut merupakan sintaks dasar *inheritance*:

```
1 v class Parent:
2     pass
3
4 v class Child(Parent):
5     pass
```

Contoh :

```
1 v class Manusia:
2 v     def __init__(self, nama, umur):
3         self.nama = nama
4         self.umur = umur
5
6 v     def cetakProfile(self):
7         print(f"{self.nama} berumur {self.umur}")
8
9 v class Mahasiswa(Manusia):
10     pass
11
12 mhs1 = Mahasiswa("Joko", 20)
13 mhs1.cetakProfile()
```

Output :

```
Console  Shell
Joko berumur 20
> 
```

- Inheritance Identik

Inheritance identik merupakan pewarisan yang menambahkan constructor pada class child sehingga class child memiliki constructornya sendiri tanpa menghilangkan constructor pada class parentnya. Metode ini ditandai dengan adanya class child yang menggunakan constructor dan menggunakan kata kunci **super()**.

Contoh :

```
1 class Manusia:
2     def __init__(self, namadepan, namabelakang):
3         self.namadepan = namadepan
4         self.namabelakang = namabelakang
5
6     def biodata(self):
7         print("Nama saya " + self.firstname + " " + self.lastname)
8
9 class Pekerja(Manusia):
10     def __init__(self, namadepan, namabelakang, pekerjaan):
11         super().__init__(namadepan, namabelakang)
12         self.pekerjaan = pekerjaan
13
14     def biodata(self):
15         print("Nama saya " + self.namadepan + " " + self.namabelakang)
16         print("Pekerjaan : " + self.pekerjaan)
17
18 pelajar = Pekerja('Lukas', 'Sandy', 'Mahasiswa')
19 pelajar.biodata()
```

- Menambah karakteristik pada *child class*

Pada *child class* dapat ditambahkan beberapa fitur tambahan baik atribut maupun method sehingga *child class* tidak identik dengan *parent class*.

Contoh :

```
1 class Lingkaran():
2
3     phi = float(3.14)
4
5     def __init__(self, r):
6         self.r = r
7
8     def luasLingkaran(self):
9         self.luas = Lingkaran.phi * self.r * self.r
10        return self.luas
11
12 class Tabung(Lingkaran):
13     def __init__(self, r, t):
14         super().__init__(r)
15         self.tinggi = t
16
17     def luasPermukaan(self):
18         self.luas = 2 * self.phi * self.r * (self.r + self.tinggi)
19         return self.luas
20
21 tb1 = Tabung(3, 2)
22 l1 = Lingkaran(3)
23 print("luas permukaan tabung" , tb1.luasPermukaan())
24 print("Luas lingkaran", l1.luasLingkaran())
```

Output :

```
➔ ~ /usr/bin/python3.7 /home/klmn/class.py  
luas permukaan tabung 94.2  
Luas lingkaran 28.259999999999998
```

Dari contoh program terlihat bahwa terdapat penambahan karakteristik pada *child class* yaitu berupa atribut tinggi. Serta penambahan method `luasPermukaan()`

2. Polymorphism

Polymorphism berarti banyak (poly) dan bentuk (morphism), dalam Pemrograman Berbasis Objek konsep ini memungkinkan digunakannya suatu interface yang sama untuk memerintah objek agar melakukan aksi atau tindakan yang mungkin secara prinsip sama namun secara proses berbeda.

Polymorphism merupakan kemampuan suatu method untuk bekerja dengan lebih dari satu tipe argumen. Pada bahasa lain (khususnya C++), konsep ini sering disebut dengan method overloading. Pada dasarnya, Python tidak menangani hal ini secara khusus. Hal ini disebabkan karena Python merupakan suatu bahasa pemrograman yang bersifat duck typing(dynamic typing).

Contoh:

```
1 class Tomato():
2     def type(self):
3         print("Vegetable")
4     def color(self):
5         print("Red")
6 class Apple():
7     def type(self):
8         print("Fruit")
9     def color(self):
10        print("Red")
11
12 def func(obj):
13     obj.type()
14     obj.color()
15
16 obj_tomato = Tomato()
17 obj_apple = Apple()
18 func(obj_tomato)
19 func(obj_apple)
```

3. Override/Overriding

Pada konsep OOP di python kita dapat menimpa suatu metode yang ada pada *parent class* dengan mendefinisikan kembali method dengan nama yang sama pada *child class*. Dengan begitu maka method yang ada *parent class* tidak berlaku dan yang akan dijalankan adalah method yang terdapat di *child class*.

Contoh :

Output:

```
1 class bangunDatar():
2     def tampil(self):
3         print("Ini bangun datar")
4
5 class Persegi(bangunDatar):
6     def tampil(self):
7         print("Ini persegi")
8
9 P1 = Persegi()
10 P1.tampil()
```

```
→ ~ /usr/bin/python3.7 /home/klmn/class.py
Ini persegi
```

Pada contoh terlihat bahwa terdapat *parent class* yaitu class bangunDatar dengan method tampil. Kemudian terdapat *child class* yaitu class Persegi yang memiliki method yang sama dengan method pada *parent class*. Dengan begitu ketika melakukan instansiasi objek *child class* dan memanggil method tampil() maka yang akan dipanggil ialah method pada *child class*, method pada *parent class* tidak berlaku.

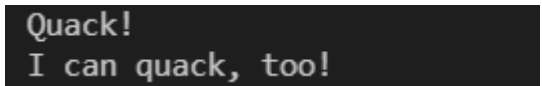
4. Overloading

Metode overloading mengizinkan sebuah class untuk memiliki sekumpulan fungsi dengan nama yang sama dan argumen yang berbeda. Akan tetapi, Python tidak mengizinkan pendeklarasian fungsi (baik pada class ataupun tidak) dengan nama yang sama. Hal itu menyebabkan implementasi *overloading* pada python menjadi “tricky”.

Secara umum *overloading* memiliki beberapa *signature*, yaitu jumlah argumen, tipe argumen, tipe keluaran dan urutan argumen. Akan tetapi, seperti yang telah dijelaskan python tidak menangani *overloading* secara khusus. Karena python adalah bahasa pemrograman yang bersifat *duck typing* (*dynamic typing*), overloading secara tidak langsung dapat diterapkan.

Contoh :

```
1 class Duck:
2     def __init__(self, name):
3         self.name = name
4     def quack(self):
5         print('Quack!')
6 class Car:
7     def __init__(self, model):
8         self.model = model
9     def quack(self):
10        print('I can quack, too!')
11
12 def quacks(obj):
13     obj.quack()
14
15 donald = Duck('Donald Duck')
16 car = Car('Tesla')
17 quacks(donald)
18 quacks(car)
```

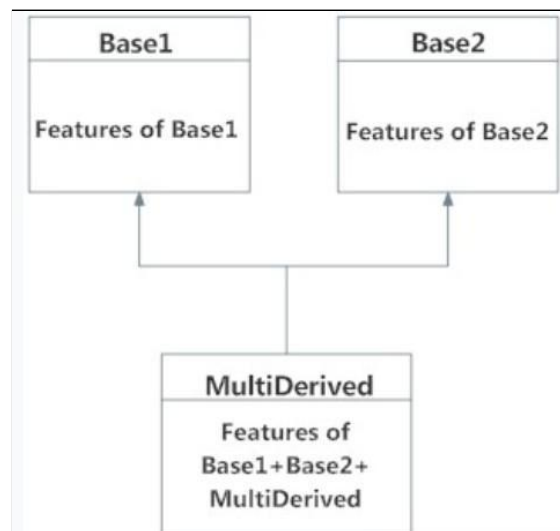


5. Multiple Inheritance

Python mendukung pewarisan ke banyak kelas. Kelas dapat mewarisi dari banyak orang tua. Bentuk syntax multiple inheritance adalah sebagai berikut

```
class Base1:  
    pass  
  
class Base2:  
    pass  
  
class MultiDerived(Base1, Base2):  
    pass
```

Dimana class MultiDerived merupakan kelas yang berasal dari class base1 dan class base2. Adapun bentuk diagramnya adalah sebagai berikut :

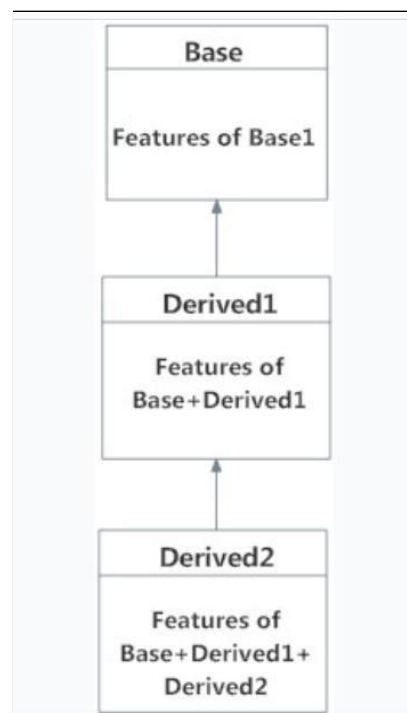


Kita juga dapat mewarisi dari kelas turunan atau disebut warisan bertingkat. Pewarisan ini dapat dilakukan hingga ke dalaman berapa pun. Dalam kelas turunan dari kelas dasar dan turunannya dapat diwarisi ke dalam kelas turunan yang baru

Contoh :

```
class Base:  
    pass  
  
class Derived1(Base):  
    pass  
  
class Derived2(Derived1):  
    pass
```

Visualisasi warisan bertingkat dalam bentuk diagram :



6. Method Resolution Order di Python

MRO adalah urutan pencarian metode dalam hirarki class. Hal ini terutama berguna dalam multiple inheritance. Urutan MRO dalam python yaitu bawah-atas dan kiri-kanan. Artinya, method dicari pertama kali di kelas objek. jika tidak ditemukan, pencarian berlanjut ke super class. Jika terdapat banyak superclass (multiple inheritance), pencarian dilakukan di kelas yang paling kiri dan dilanjutkan ke kelas sebelah kanan.

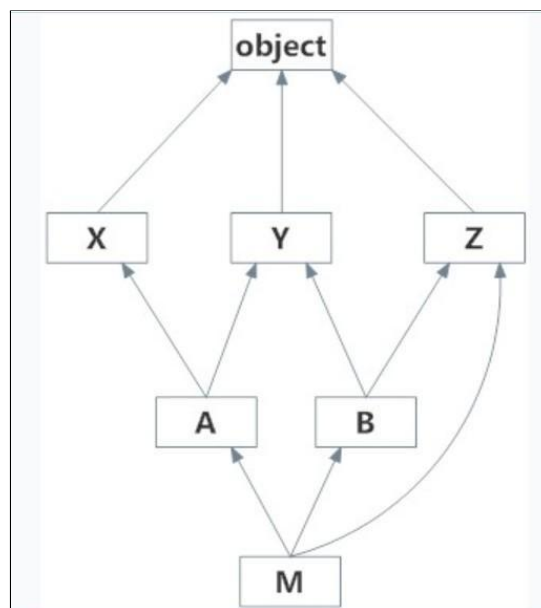
Contoh:

```
1 class A:
2     def method(self):
3         print("A.method() dipanggil")
4
5 class B:
6     def method(self):
7         print("B.method() dipanggil")
8
9 class C( A, B):
10    pass
11
12 class D(B,A):
13    pass
14
15 c=C()
16 c.method()
17
18 d=D()
19 d.method()
20
```

output:

```
.exe d:/itera/sm 6/asj
A.method() dipanggil
B.method() dipanggil
PS D:\itera\sm 6>
```

Visualisasi diagram :



Demonstrasi MRO

```
# Demonstration of MRO
```

```
class X:  
    pass
```

```
class Y:  
    pass
```

```
class Z:  
    pass
```

```
class A(X, Y):  
    pass
```

```
class B(Y, Z):  
    pass
```

```
class M(B, A, Z):  
    pass
```

```
# Output:
```

```
# [<class '__main__.M'>, <class '__main__.B'>,  
#  <class '__main__.A'>, <class '__main__.X'>,  
#  <class '__main__.Y'>, <class '__main__.Z'>]
```

7. Dynamic Cast

Dynamic cast atau type conversion adalah proses mengubah nilai dari satu tipe data ke tipe data lainnya seperti dari string ke int atau sebaliknya. Ada 2 tipe konversi yaitu:

a. Implisit

Python secara otomatis mengkonversikan tipe data ke tipe data lainnya tanpa ada campur tangan pengguna.

Contoh:

```
1 num_int = 123
2 num_flo = 1.23
3
4 num_new = num_int + num_flo
5
6 print("datatype of num_int:",type(num_int))
7 print("datatype of num_flo:",type(num_flo))
8
9 print("Value of num_new:",num_new)
10 print("datatype of num_new:",type(num_new))
```

Output:

```
datatype of num_int: <class 'int'>
datatype of num_flo: <class 'float'>
Value of num_new: 124.23
datatype of num_new: <class 'float'>
```

b. Eksplisit

Pengguna mengubah tipe data sebuah objek ke tipe data lainnya dengan fungsi yang sudah ada dalam python seperti int(), float(), dan str(). dapat berisiko terjadinya kehilangan data.

Contoh:

```
num_int = 123
num_str = "456"

print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))

num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

Output:

```
Data type of num_int: <class 'int'>
Data type of num_str before Type Casting: <class 'str'>
Data type of num_str after Type Casting: <class 'int'>
Sum of num_int and num_str: 579
Data type of the sum: <class 'int'>
```

8. Casting

a) **Downcasting:** Parent class mengakses atribut yang ada pada kelas bawah (child class)

```
class Manusia:
    def __init__(self, namadepan, namabelakang):
        self.namadepan = namadepan
        self.namabelakang = namabelakang

    def biodata(self):
        print(f"{self.namadepan} {self.namabelakang} ({self.pekerjaan})")

class Pekerja(Manusia):
    def __init__(self, namadepan, namabelakang, pekerjaan):
        super().__init__(namadepan, namabelakang)
        self.pekerjaan = pekerjaan

Andhika = Pekerja("Andhika", "Wibawa", "Mahasiswa")
Andhika.biodata()
```

Hasil:

```
Dhika@DESKTOP-R8DD3EM /e/P
$ D:/Apps/CommonFiles/Pytho
Andhika Wibawa (Mahasiswa)
```

b) **Upcasting:** Child class mengakses atribut yang ada pada kelas atas (parent class)

```
class Manusia:
    namabelakang = "Putra"

    def __init__(self, namadepan, namabelakang):
        self.namadepan = namadepan
        self.namabelakang = namabelakang

    def biodata(self):
        print(f"{self.namadepan} {self.namabelakang} ({self.pekerjaan})")

class Pekerja(Manusia):
    def __init__(self, namadepan, namabelakang, pekerjaan):
        # super() adalah alias untuk kelas parent (Manusia)
        super().__init__(namadepan, namabelakang)
        self.pekerjaan = pekerjaan

    def biodata(self):
        print(f"{self.namadepan} {super().namabelakang} ({self.pekerjaan})")

Andhika = Pekerja("Andhika", "Wibawa", "Mahasiswa")
Andhika.biodata()
```

Hasil:

```
Dhika@DESKTOP-R8DD3EM /e/  
$ D:/Apps/CommonFiles/Pyth  
Andhika Putra (Mahasiswa)
```

c) **Type casting**: Konversi tipe kelas agar memiliki sifat/perilaku tertentu yang secara default tidak dimiliki kelas tersebut. Karena Python merupakan bahasa pemrograman berorientasi objek, maka semua variabel atau instansi di Python pada dasarnya merupakan objek (kelas) yang sifat/perilakunya dapat dimanipulasi jika dibutuhkan (umumnya melalui **magic method**)

Contoh 1 (kelas berperilaku seperti string dan integer)

```
class Mahasiswa:  
    def __init__(self, nama, nim, matkul):  
        self.__nama = nama  
        self.__nim = nim  
        self.__matkul = matkul  
  
    def __str__(self):  
        return f"{self.__nama} ({self.__nim}) merupakan mahasiswa kelas {self.__matkul}"  
  
    def __int__(self):  
        return self.__nim  
  
Rasyid = Mahasiswa("Rasyid", 1234, "PBO RA")  
print(Rasyid) # Rasyid (1234) merupakan mahasiswa kelas PBO RA  
print(int(Rasyid) == 1234) # True
```

Contoh 2 (kelas berperilaku seperti list/iterable)

```
class Mahasiswa:  
    list_mahasiswa = []  
  
    def __init__(self, *args):  
        for mhs in args:  
            Mahasiswa.list_mahasiswa.append(mhs)  
  
    def __iter__(self):  
        return iter(Mahasiswa.list_mahasiswa)  
  
    def __len__(self):  
        return len(Mahasiswa.list_mahasiswa)  
  
Asprak = Mahasiswa("Andhika", "Laras", "Ihza", "Ammar")  
  
for mahasiswa in Asprak:  
    print(mahasiswa)  
# Andhika Laras Ihza Ammar  
  
print(len(Asprak))  
# 4
```

Contoh 3 (pengalihan tipe kelas saat pendeklarasian objek baru)

```
class LuasBangun:
    def __new__(cls, *argumen):
        if len(argumen) == 1:
            return Persegi(argumen[0])
        elif len(argumen) == 2:
            return PersegiPanjang(argumen[0], argumen[1])

class Persegi:
    def __init__(self, panjang):
        self.panjang = panjang

    def __str__(self):
        return f"Luas persegi = {self.panjang * self.panjang}"

class PersegiPanjang:
    def __init__(self, panjang, lebar):
        self.panjang = panjang
        self.lebar = lebar

    def __str__(self):
        return f"Luas persegi panjang = {self.panjang * self.lebar}"

Bangun1 = LuasBangun(5) # Persegi
Bangun2 = LuasBangun(5, 10) # PersegiPanjang
print(Bangun1) # Luas persegi = 25
print(Bangun2) # Luas persegi panjang = 50
```

Magic method tidak hanya terbatas pada contoh di atas namun terdapat juga banyak lainnya, beberapa magic method juga digunakan untuk konsep **operator overloading** seperti `__add__` (pertambahan), `__sub__` (pengurangan), `__mul__` (perkalian), dll. Dokumentasi resmi mengenai magic method dapat dilihat pada link berikut:

<https://docs.python.org/3/reference/datamodel.html#special-method-names>

Referensi

[Inheritance and Composition: A Python OOP Guide – Real Python](#)

<https://towardsdatascience.com/python-class-inheritance-62fdb33ede47>

<https://www.codesdope.com/course/python-method-overriding-and-mro/>

<https://www.programiz.com/python-programming/multiple-inheritance>

<https://www.educative.io/edpresso/what-is-mro-in-python>

<https://data-flair.training/blogs/python-multiple-inheritance/>

<https://towardsdatascience.com/duck-typing-python-7aeac97e11f8>