

MODUL PRAKTIKUM
PEMROGRAMAN BERORIENTASI OBJEK
Minggu 6



KELAS ABSTRAK DAN INTERFACE
(KELAS ABC, METACLASS)

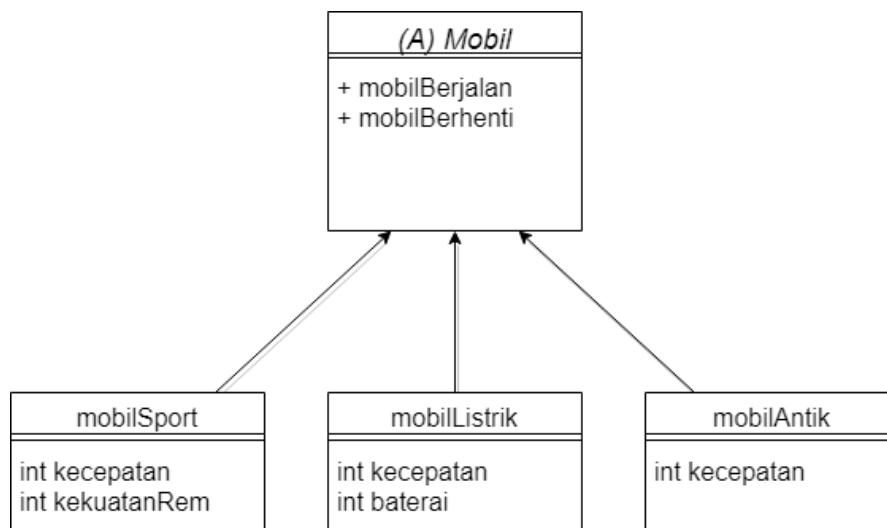
Disusun Oleh :

| | |
|------------------------|-----------|
| 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. Konsep Abstraksi

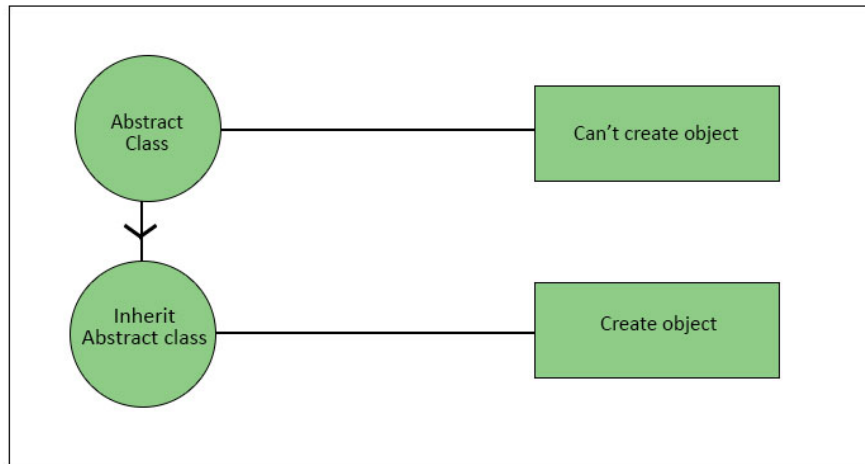
Sejatinya tidak semua yang kita lihat dan buat adalah sebuah objek konkrit tetapi ada juga yang bersifat abstrak. Saat ini kita telah sering kali menggunakan objek konkrit seperti apel, pisang, lingkaran, dan lainnya. Abstraksi dalam konsep OOP adalah sebuah yang dibuat hanya memperlihatkan atribut yang esensial dan menyembunyikan detail-detail yang tidak penting dari user. Gunanya adalah untuk mengurangi kompleksitas. User dapat mengetahui apa yang objek lakukan, tapi tidak tau mekanisme yang terjadi di belakang layar bagaimana. Contohnya ketika mengendarai mobil, user mengetahui bagaimana menyalakan mobil, menjalankan, menghentikan, dll, tetapi tidak mengetahui mekanisme apa yang terjadi pada mobil ketika mendapat perintah di atas.



Gambar 1. Contoh Abstraksi

A. Kelas Abstrak dalam Python

Kelas Abstrak adalah sebuah *class* yang tidak bisa diinstansiasi (tidak bisa dibuat menjadi objek) langsung dan berperan sebagai ‘kerangka dasar’ bagi class turunannya. Di dalam kelas abstrak umumnya akan memiliki sebuah *abstract method*. *Abstract method* adalah sebuah method yang harus diimplementasikan ulang di dalam class anak (*child class*). *Abstract method* ditulis tanpa isi dari method, melainkan hanya ‘signature’-nya saja. *Signature* dari sebuah method adalah bagian method yang terdiri dari nama method dan parameternya (jika ada).



Kelas Abstrak digunakan di dalam inheritance (pewarisan class) untuk ‘memaksakan’ atau OVERRIDE method yang sama bagi seluruh class yang diturunkan dari abstract class. Kelas Abstrak digunakan untuk membuat struktur logika penurunan di dalam pemrograman objek. Contoh pemakaian kelas abstrak misalnya dapat dipakai pada konsep mobil, dimana tiap-tiap mobil tentunya memiliki suatu kesamaan namun implementasi yang berbeda (misal dari segi kecepatan, bahan bakar, dll).

B. Implementasi Kelas Abstrak dengan Modul ABC

Python memiliki modul untuk menggunakan Abstract Base Classes (ABC). Fungsi abstrak pada kelas abstrak ditandai dengan memberikan decorator `@abstractmethod` pada fungsi yang dibuat.

```
1  from abc import ABC, abstractmethod
2
3  # kelas Mobil merupakan kelas abstrak dan tidak bisa diinstansiasi
4  class Mobil(ABC):
5      @abstractmethod
6      def berjalan(self):
7          # fungsi ini wajib di implementasikan pada child-class
8          pass
9
10     @abstractmethod
11     def berhenti(self):
12         # fungsi ini wajib di implementasikan pada child-class
13         pass
```

Gambar 2. Implementasi kelas abstrak

Dari potongan kode diatas akan menghasilkan error jika kelas tersebut langsung dijadikan sebuah objek. Untuk mengatasi hal tersebut dan melakukan implementasi terhadap kelas abstrak diperlukan sebuah kelas 'child' dari kelas abstrak tersebut.

```
Traceback (most recent call last):
  File "f:\Andhika P P\Code\Asprak-PBO\MG5\ABC.py", line 15, in <module>
    car = Mobil()
TypeError: Can't instantiate abstract class Mobil
with abstract methods berhenti, berjalan
PS F:\Andhika P P\Code\Asprak-PBO\MG5>
```

Gambar 3. Implementasi kelas abstrak Error

Membuat kelas 'child' untuk kelas abstrak sendiri ada ketentuannya, salah satunya adalah ketika membuat kelas 'child' dari kelas abstrak kita harus membuat kembali seluruh metode/fungsi yang ada pada *parent class* (kelas abstrak). Contohnya sebagai berikut:

```
1 class MobilSport(Mobil):
2     def berjalan(self):
3         print("Mobil Sport berjalan dengan cepat")
4
5     def berhenti(self):
6         print("Mobil Sport berhenti")
```

```
PS F:\Andhika P P\Code\Asprak-PBO\MG5>
/ASUS/AppData/Local/Programs/Python/Py
hon.exe "f:/Andhika P P/Code/Asprak-PB
O/MG5/ABC.py"
Mobil Sport berjalan dengan cepat
PS F:\Andhika P P\Code\Asprak-PBO\MG5>
```

Gambar 4. Implementasi kelas child

```
1 class MobilSport(Mobil):
2     def berjalan(self):
3         print("Mobil Sport berjalan dengan cepat")
```

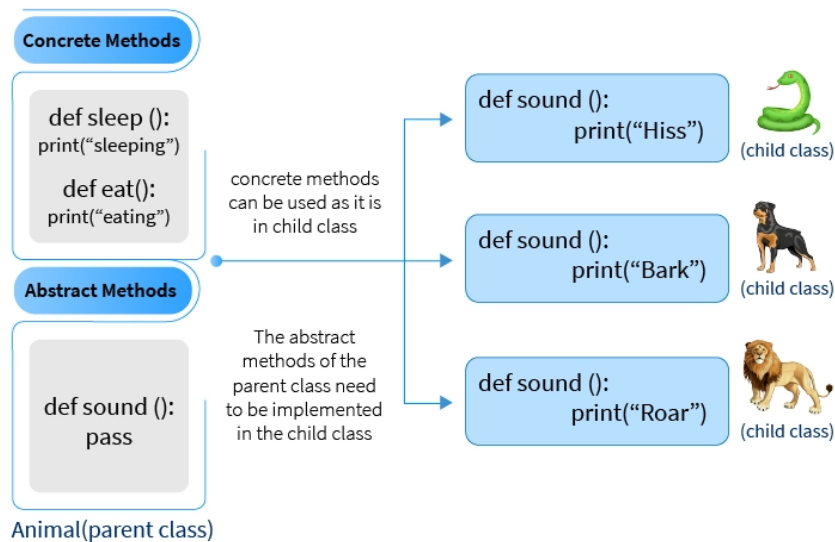
```

PS F:\Andhika P P\Code\Asprak-PBO\MG5> C:/Users/ASU
S/AppData/Local/Programs/Python/Python310/python.exe
"f:/Andhika P P/Code/Asprak-PBO/MG5/ABC.py"
Traceback (most recent call last):
  File "f:/Andhika P P\Code\Asprak-PBO\MG5\ABC.py",
line 19, in <module>
    car = MobilSport()
TypeError: Can't instantiate abstract class MobilSpo
rt with abstract method berhenti
PS F:\Andhika P P\Code\Asprak-PBO\MG5>

```

Gambar 4. Implementasi kelas child [Error]

Selain fungsi sederhana seperti gambar diatas, sebuah **kelas abstrak juga bisa memiliki konstruktor dan fungsi biasa** (yang tidak sepenuhnya kosong).



Misal, sama halnya ketika membuat konstruktor kelas pada umumnya, untuk membuat konstruktor pada kelas abstrak kita dapat menggunakan `def __init__` seperti pada contoh berikut berikut:

```

1  from abc import ABC, abstractmethod
2  # kelas Mobil merupakan kelas abstrak dan tidak bisa diinstansiasi
3  class Mobil(ABC):
4      def __init__(self, warna):
5          self.warna = warna
6
7      @abstractmethod
8      def berjalan(self):
9          # fungsi ini wajib di implementasikan pada child-class
10         pass
11
12     @abstractmethod
13     def berhenti(self):
14         # fungsi ini wajib di implementasikan pada child-class
15         pass
16
17 class MobilSport(Mobil):
18     def __init__(self, warna):
19         super().__init__(warna)
20
21     def berjalan(self):
22         print(f"Mobil Sport berjalan dengan warna {self.warna}")
23
24     def berhenti(self):
25         print("Mobil Sport berhenti")
26
27 car = MobilSport("Merah")
28 car.berjalan()

```

```

PS F:\Andhika P P\Code\Asprak-PBO\MG5> & C:/Users/AS
US/AppData/Local/Programs/Python/Python310/python.exe
e "f:/Andhika P P/Code/Asprak-PBO/MG5/ABC.py"
Mobil Sport berjalan dengan warna Merah
PS F:\Andhika P P\Code\Asprak-PBO\MG5>

```

Gambar 5. Implementasi kelas abstrak dengan konstruktor

Lalu, untuk implementasi fungsi biasa (konkret) pada kelas abstrak juga dapat dilakukan seperti contoh di bawah ini (yaitu fungsi `get_warna`).

```

1  from abc import ABC, abstractmethod
2  # kelas Mobil merupakan kelas abstrak dan tidak bisa diinstansiasi
3  class Mobil(ABC):
4      def __init__(self, warna):
5          self.warna = warna
6
7      @abstractmethod
8      def berjalan(self):
9          # fungsi ini wajib di implementasikan pada child-class
10         pass
11
12     @abstractmethod
13     def berhenti(self):
14         # fungsi ini wajib di implementasikan pada child-class
15         pass
16
17     def get_warna(self):
18         return self.warna
19
20 class MobilSport(Mobil):
21     def __init__(self, warna):
22         super().__init__(warna)
23
24     def berjalan(self):
25         print(f"Mobil Sport berjalan dengan warna {self.warna}")
26
27     def berhenti(self):
28         print("Mobil Sport berhenti")
29
30 car = MobilSport("Merah")
31 print(car.get_warna())
32 car.berjalan()

```

```

PS F:\Andhika P P\Code\Asprak-PBO\MG5> & C:/Users/AS
US/AppData/Local/Programs/Python/Python310/python.exe
e "f:/Andhika P P/Code/Asprak-PBO/MG5/ABC.py"
Merah
Mobil Sport berjalan dengan warna Merah
PS F:\Andhika P P\Code\Asprak-PBO\MG5>

```

Gambar 6. Implementasi kelas abstrak dengan fungsi

2. Interface

A. Definisi Interface

- Interface adalah koleksi dari method atau fungsi-fungsi yang perlu disediakan oleh implementing class (child class).
- Interface mengandung metode yang bersifat abstract. Metode abstract akan memiliki satu-satunya deklarasi karena tidak adanya implementasi.
- Pengimplementasian sebuah interface adalah sebuah cara untuk menulis kode yang elegan dan terorganisir.

Perbedaan Interface dan Abstract class

| Python interface | Python abstract class |
|--|---|
| Semua metode dari sebuah (formal) interface adalah abstrak | Sebuah abstract class dapat mempunyai metode abstract begitu juga dengan metode konkrit |
| Interface digunakan jika semua fitur perlu untuk diimplementasikan secara berbeda untuk objek yang berbeda | Abstract class digunakan ketika ada beberapa fitur umum yang dimiliki oleh semua objek |
| Interface cenderung lebih lambat jika dibandingkan dengan abstract class | Abstract class lebih cepat (tidak semua implementasi perlu ditulis pada child class) |

B. Informal interface

Karena pengimplementasian interface di Python berbeda dengan bahasa lain seperti C++ dan Java, interface di Python kadang lebih diartikan sebagai sebuah konsep tanpa aturan yang terlalu ketat.

Informal interface adalah kelas yang mendefinisikan metode atau fungsi-fungsi yang bisa di override/implementasi namun tanpa adanya unsur paksaan (cukup diimplementasi hanya bila dibutuhkan). Untuk mengimplementasikannya kita harus membuat kelas konkrit. Kelas konkrit adalah subclass dari (abstrak) interface yang menyediakan implementasi dari method atau fungsi-fungsi di kelas interface. Untuk melakukan implementasi menggunakan inheritance.

Contoh kasus informal interface adalah sebagai berikut:


```

class Hewan:
    def __init__(self, hewan):
        self.__list_hewan = hewan

    def __len__(self):
        return len(self.__list_hewan)

    def __contains__(self, hewan):
        return hewan in self.__list_hewan

class KebunBinatang(Hewan):
    pass

ragunan = KebunBinatang(["Rusa", "Harimau", "Unta"])
# Cek banyaknya hewan
print(len(ragunan))
# Cek apakah terdapat hewan tertentu
print("Rusa" in ragunan)
print("Gajah" in ragunan)
print("Unta" not in ragunan)
# Iterasi list hewan yang ada
for hewan in ragunan:
    print(hewan)

```

Pada contoh di atas, kebun binatang Ragunan dapat mengecek banyaknya hewan (`__len__`) dan mengecek keberadaan hewan (`__contains__`) pada kebun binatang tersebut. Namun, ketika dibutuhkan list dari hewan-hewan (melalui iterasi) yang berada pada kebun binatang Ragunan, maka akan menghasilkan error.

```

/run/media/dhika/Multi/Project/Praktikum/PB0 /bin/pyth
3
True
False
False
Traceback (most recent call last):
  File "/run/media/dhika/Multi/Project/Praktikum/PB0/test1.py",
    for hewan in ragunan:
TypeError: 'KebunBinatang' object is not iterable

```

Untuk mengatasi error tersebut, pada kelas `KebunBinatang` perlu kita implementasikan sebuah fungsi baru yaitu `__iter__` sehingga memungkinkan terjadinya iterasi untuk mendapatkan list hewan yang ada.

```

class Hewan:
    def __init__(self, hewan):
        self.__list_hewan = hewan

    def __len__(self):
        return len(self.__list_hewan)

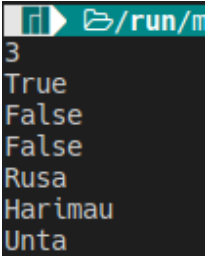
    def __contains__(self, hewan):
        return hewan in self.__list_hewan

class KebunBinatang(Hewan):
    def __iter__(self):
        return iter(self.__list_hewan)

ragunan = KebunBinatang(["Rusa", "Harimau", "Unta"])
# Cek banyaknya hewan
print(len(ragunan))
# Cek apakah terdapat hewan tertentu
print("Rusa" in ragunan)
print("Gajah" in ragunan)
print("Unta" not in ragunan)
# Iterasi list hewan yang ada
for hewan in ragunan:
    print(hewan)

```

Dalam kasus ini, fungsi `__iter__` pada kelas `KebunBinatang` perlu diimplementasikan karena dibutuhkan untuk mendapatkan/meng-iterasi list hewan yang ada. Namun, implementasi `__iter__` ini tidak dipaksakan oleh kelas parent dari `KebunBinatang` yaitu kelas `Hewan`. Oleh karena itu, konsep ini disebut dengan informal interface. Hasil outputnya sekarang adalah sebagai berikut:



```

3
True
False
False
Rusa
Harimau
Unta

```

C. Formal Interface

Selain informal interface, terdapat pula formal interface. Pada formal interface kelas parent (abstrak) dapat dibangun hanya dengan sedikit baris kode, untuk kemudian diimplementasikan pada kelas turunannya (konkret). Implementasi ini bersifat wajib/dipaksakan sehingga dinamakan formal interface. Salah satu cara

membuat formal interface adalah dengan abstract class method seperti pada contoh kelas abstrak sebelumnya.

```
1 from abc import ABC
2 from abc import abstractmethod
3
4 class GPS(ABC):
5
6     @abstractmethod
7     def aktifkan_gps(self):
8         pass
9
10    @abstractmethod
11    def matikan_gps(self):
12        pass
13
14    @abstractmethod
15    def get_location(self):
16        pass
17
```

Contoh kelas abstrak

```
class Smartphone(GPS):
    def aktifkan_gps(self):
        print("gps aktif")

    def matikan_gps(self):
        print("gps mati")

    def get_location(self):
        print("lokasi anda di...")

samsung=Smartphone()

# samsung.aktifkan_asisten()
samsung.get_location()
```

contoh kelas konkret

3. Metaclass

A. Apa itu Metaclass?

Tidak seperti bahasa pemrograman lain seperti C++ dan Java yang memiliki suatu tipe dasar (misal int, float, dll), pada Python semua tipe pada dasarnya adalah suatu objek/kelas juga (termasuk int, float, dll).

```
angka = 420
pi = 3.14
nama = "Rangga"

print("Tipe dari angka adalah", type(angka))
print("Tipe dari pi adalah", type(pi))
print("Tipe dari nama adalah", type(nama))
```

```
run /me/dhika/Multi/Project/P
Tipe dari angka adalah <class 'int'>
Tipe dari pi adalah <class 'float'>
Tipe dari nama adalah <class 'str'>
```

Oleh karena itu, kita bisa saja membuat sebuah kelas turunan (subclass) dari int atau tipe dasar lainnya yang memiliki karakteristik tambahan seperti contoh berikut.

```
class Positif(int):
    def __new__(cls, nilai, *args, **kwargs):
        if nilai < 0:
            raise ValueError("Angka positif tidak boleh kurang dari nol")
        return super().__new__(cls, nilai, args, kwargs)

    def __add__(self, other): # Pertambahan
        hasil = super().__add__(other)
        # Return nilai tertinggi antara "hasil" dan nol
        return self.__class__(max(hasil, 0))

    def __sub__(self, other): # Pengurangan
        hasil = super(Positif, self).__sub__(other)
        return self.__class__(max(hasil, 0))

    def __mul__(self, other): # Perkalian
        hasil = super(Positif, self).__mul__(other)
        return self.__class__(max(hasil, 0))

    def __div__(self, other): # Pembagian
        hasil = super(Positif, self).__div__(other)
        return self.__class__(max(hasil, 0))
```

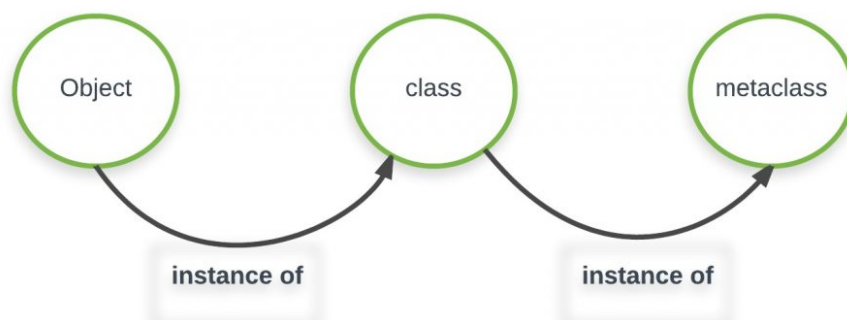
Konsep tipe ini bahkan juga berlaku pada objek dan kelas yang masih kosong seperti contoh di bawah.

```
class Kosong:
    pass

mahasiswa = Kosong()
print("Tipe dari mahasiswa adalah", type(mahasiswa))
print("Tipe dari Kosong adalah", type(Kosong))
```

```
/run/media/dhika/Multi/Project/Praktikum/PBO
Tipe dari mahasiswa adalah <class '__main__.Kosong'>
Tipe dari Kosong adalah <class 'type'>
```

Seperti yang terlihat pada output di atas, tipe dari objek mahasiswa adalah “Kosong” dan tipe dari kelas Kosong adalah “type”. Mengapa demikian? Di dalam Python, terdapat konsep hirarki dimana objek dibentuk dari kelas (class), sedangkan kelas dibentuk dari hirarki yang lebih tinggi yaitu metaclass (dalam hal ini berupa contoh metaclass tersebut adalah “**type**”).



Dengan kata lain, **metaclass** adalah tipe kelas spesial yang berfungsi untuk membuat kelas-kelas pada Python (atau biasa disebut juga dengan pabrik kelas/*class factory*).

B. Implementasi/Contoh Kasus Metaclass

1. Menggunakan **type**

Dibawah ini adalah contoh kasus pembuatan metaclass menggunakan **type**. Jika variabel “terpanggil” dipanggil, maka metaclass **type** tersebut akan membuat sebuah class bernama “dedikorbuzet”, dengan tuple berisi nama parent class (dikosongkan saja jika class yang dibuat tidak melakukan inheritance dari class manapun), dan atribut dari kelas tersebut yang berbentuk dictionary, dengan sintaks sebagai berikut:

```
type('NamaKelas', (Parent1,), {'nama_attr' :  
'nilai_attr',})
```

Contoh:

```
D: > =====ASPRAK SHITS===== > awiawok > Untitled-1.py > ...
1  terpanggil = type('dedikorbuzet',
2  (), {'gladiator': 'Vicky baku hantam sama Azka, tapi kalah',
3  'alasan': 'Vicky abis kerokan pas malem sebelum tanding'})
4
5  print ("Sebuah kelas telah dibuat, bernama : ",terpanggil)
6  print ("Apa yang terjadi hari Kamis kemarin? : ", terpanggil.gladiator)
7  print ("Alasan Vicky kalah : ", terpanggil.alasan)
8  print()
9  print(terpanggil())
10
```

Hasil print di atas berupa :

```
Sebuah kelas telah dibuat, bernama : <class '__main__.dedikorbuzet'>
Apa yang terjadi hari Kamis kemarin? : Vicky baku hantam sama Azka, tapi kalah
Alasan Vicky kalah : Vicky abis kerokan pas malem sebelum tanding
<__main__.dedikorbuzet object at 0x00000214D27A1330>
```

Line terakhir merupakan alamat dari class “dedikorbuzet” yang telah dibuat. Dapat dilihat bahwa class “dedikorbuzet” merupakan sebuah “**object**” yang berada di alamat 0x00000214D27A1330.

Di bawah ini adalah contoh jika sebuah child class dibuat dan mewarisi parent class. Seperti yang terlihat, jika “anak_terpanggil” dipanggil, maka metaclass akan membuat sebuah child class “anak_dedikorbuzet” yang mewarisi parent class “terpanggil” pada contoh sebelumnya.

```
anak_terpanggil = type('anak_dedikorbuzet',
                        (terpanggil,),
                        {'akibat': 'Gempa 6,9 SR terdeteksi ketika Vicky tumbang di ring'})

print ("Sebuah kelas telah dibuat, bernama : ",anak_terpanggil)
print ("Apa yang terjadi hari Kamis kemarin? : ", anak_terpanggil.gladiator)
print ("Alasan Vicky kalah : ", anak_terpanggil.alasan)
print ("Akibat Vicky kalah : ", anak_terpanggil.akibat)
print()
print(anak_terpanggil())
```

Hasil print :

```
Sebuah kelas telah dibuat, bernama : <class '__main__.anak_dedikorbuzet'>
Apa yang terjadi hari Kamis kemarin? : Vicky baku hantam sama Azka, tapi kalah
Alasan Vicky kalah : Vicky abis kerokan pas malem sebelum tanding
Akibat Vicky kalah : Gempa 6,9 SR terdeteksi ketika Vicky tumbang di ring
<__main__.anak_dedikorbuzet object at 0x00000214D27A1330>
```

2. Menggunakan parameter metaclass

Selain dengan langsung menggunakan type untuk membuat sebuah kelas, kita juga dapat membuat sebuah kelas turunan hasil modifikasi terhadap type itu sendiri. Untuk memodifikasinya, kita perlu membuat sebuah kelas turunan dengan basis dari type

```
1 v class IniJugaMeta(type):
2 v     def __new__(cls, namakelas, basis, dictattr):
3         x = super().__new__(cls, namakelas, basis, dictattr)
4         x.jenis_metaclass = "IniJugaMeta metaclass"
5         return x
6
```

Seperti yang kita ketahui, bahwa sebuah type adalah metaclass. Maka, jika kita membuat sebuah kelas turunan dari type, maka kelas tersebut juga merupakan sebuah metaclass. Dari gambar diatas, kita membuat sebuah metaclass yang akan secara otomatis membuat atribut 'jenis_metaclass' dengan isi "IniJugaMeta metaclass"

```
6
7 v class Contoh(metaclass=IniJugaMeta):
8     pass
9
10 contoh = Contoh()
11 print(contoh.jenis_metaclass)
12
```

Secara default, metaclass yang digunakan saat membuat sebuah kelas adalah type. Setelah membuat sebuah metaclass hasil turunan dari type, maka kita dapat membuat sebuah kelas dengan metaclass hasil modifikasi kita sendiri. Output dari kode diatas adalah sebagai berikut

```
IniJugaMeta metaclass
> []
```

References

<https://auftechnique.com/4-pillar-pemrograman-berorientasi-objek/#abstraction>

<https://www.geeksforgeeks.org/abstract-classes-in-python/>

<https://docs.python.org/3/library/abc.html>

<https://realpython.com/python-interface/>

<https://pythonguides.com/python-interface/>

<https://realpython.com/python-metaclasses/>