

Bab 7. Object Oriented Programming

OBJEKTIF:

1. Mahasiswa mampu memahami Prosedural dan *Object Oriented Programming*.
2. Mahasiswa mampu memahami *Class* pada Python.
3. Mahasiswa mampu memahami *Inheritance* pada Python.
4. Mahasiswa mampu memahami *Polymorphism* pada Python.

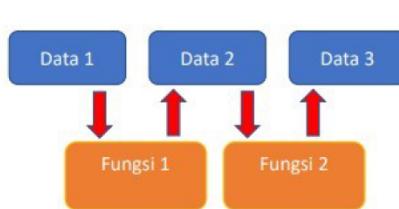
7.1 Prosedural dan *Object Oriented Programming*

Terdapat dua metode pemrograman yang digunakan saat ini yaitu:

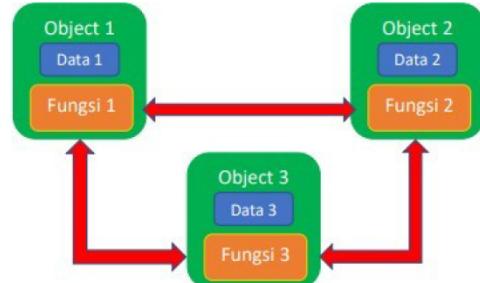
1. Prosedural: Memecah program menjadi prosedur-prosedur (fungsi-fungsi)
2. *Object oriented*: Memecah program menjadi *object-object*

Pada prosedural *programming* data dan fungsi terpisah. Sedangkan pada *object oriented programming* data dan fungsi adalah satu kesatuan.

Ilustrasi pengolahan data pada prosedural programming dan OOP:



Pada **procedural programming**, pengolahan data dilakukan dengan memroses data ke fungsi-fungsi.



Pada **OOP**, pengolahan data dilakukan dengan interaksi antar *object-object*.

Pada prosedural *programming* kita membuat program dengan pendekatan *top down* yaitu kita merancang terlebih dahulu logika utama (fungsi `main`) dari program lalu memecah program menjadi fungsi-fungsi. Sedangkan pada OOP, kita membuat program dengan pendekatan *bottom up* yaitu kita merancang program dengan membuat *object-object* dasar terlebih dahulu dan mengembangkan *object-object* tersebut lalu mengintegrasikan *object-object* dalam fungsi `main`. Memecah program menjadi fungsi-fungsi yang kita lakukan pada topik-topik sebelumnya adalah prosedural *programming*. Python selain mendukung prosedural *programming* juga mendukung *object oriented programming*. Pada bagian ini kita akan membahas *object oriented programming*.

7.2 Class

Object adalah entitas program yang terdiri dari data dan fungsi. Data yang berada di dalam *object* disebut sebagai *attribute*. Fungsi yang dilakukan oleh *object* disebut sebagai *method*. Sebelum sebuah *object* dapat dibuat, kita harus mendesain *object* tersebut dengan mendefinisikan *class*. Kita dapat membayangkan *class* sebagai "cetakan" yang digunakan untuk mencetak *object-object*.



Syntax umum penulisan definisi *class*:

```
class <Nama_Class>:  
    <body_dari_class>
```

Nama class umumnya dimulai dengan huruf besar.
 Kita menuliskan definisi class dengan keyword `class` yang diikuti dengan nama class lalu titik dua. Body dari class ditulis dengan indentasi di baris setelah header definisi class.

Dalam *body class* kita menuliskan:

1. Method `__init__` (dua underscore sebelum dan setelah kata `__init__`) untuk mendefinisikan *attribute*-*attribute* apa saja yang terdapat dalam *class* dan untuk menginisialisasi nilai-nilai *attribute* saat instansiasi.
2. Method-method lainnya.

Contoh definisi *class*:

```
class Dog:  
  
    def __init__(self, jenis, warna):  
        self.jenis = jenis  
        self.warna = warna  
  
    def menggonggong(self):  
        print('Guk! Guk!')
```

Penulisan method:

- Penulisan definisi method mirip dengan penulisan definisi fungsi
- Perbedaan dengan penulisan fungsi: setiap method harus mempunyai satu parameter khusus bernama `self` dan harus menjadi parameter pertama dalam definisi fungsi. Parameter `self` mereferensikan object. (sebenarnya parameter khusus ini tidak harus dinamakan dengan `self`, namun konvensi Python menggunakan nama `self`)

7.2.1 Method `__init__`

Method `__init__` adalah method khusus yang digunakan untuk menginisialisasi (memberikan nilai awal) ke *attribute*-*attribute* dari *object*. Method `__init__` dipanggil otomatis ketika kita membuat *object* (menginstansiasi) dari *class*. Untuk membuat *object* (menginstansiasi) dari *class* kita memanggil nama *class* dan memberikan nilai argumen ke parameter-parameter dari *method __init__*.

```
class Dog:  
  
    def __init__(self, jenis, warna):  
        self.jenis = jenis  
        self.warna = warna  
  
    def menggonggong(self):  
        print('Guk! Guk!')
```

Statement untuk membuat object dari class Dog:

```
heli = Dog('terrier', 'coklat')
```

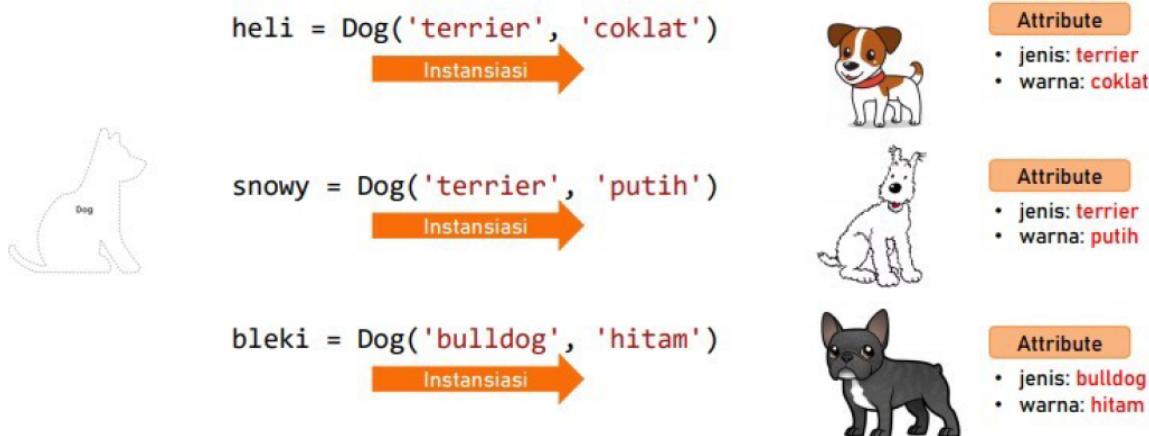
Kita memberikan nilai argument hanya untuk parameter-parameter setelah parameter `self`. Nilai untuk parameter `self` diberikan secara otomatis oleh interpreter.

7.2.2 Menginstansiasi Class

Kita dapat menginstansiasi lebih dari satu *object* dari suatu *class*. Sebagai contoh, terdapat *object* bernama `heli`, `bleki`, dan `snowy` yang merupakan instansiasi dari suatu *class* `Dog`. Lalu pada masing-masing *object* akan didefinisikan *attribute* nya masing-masing dari *class* `Dog`.

```
heli = Dog('terrier', 'coklat')
bleki = Dog('bulldog', 'hitam')
snowy = Dog('terrier', 'putih')
```

Setiap *object* akan mempunyai nilai-nilai *attribute* masing-masing.



Misalkan, kita membuat sebuah program yang mensimulasikan pelemparan koin. Dalam program ini, kita mensimulasikan pelemparan koin berulang kali dan setiap lemparan program menentukan sisi koin ("Heads" atau "Tails") yang menghadap atas ketika jatuh. Untuk merepresentasikan koin, kita membuat sebuah *class* dengan nama `coin`.



Perhatikan kode program berikut.

```
import random

# class Coin mensimulasikan sebuah coin # yang dapat dilempar
# yang dapat dilempar

class Coin:

    # Method __init__ menginisialisasi
    # attribute data sisiatas dengan 'Heads'.
    def __init__(self):
        self.sisiatas = 'Heads'

    # Method Tempar menggenerasi sebuah angka random
```

```

# di antara 0 s.d 1. Jika angka = 0, maka sisiatas
# ditetapkan sebagai 'Heads'
# Jika = 1, maka sisiatas ditetapkan sebagai 'Tails'

def lempar(self):
    if random.randint(0, 1) == 0:
        self.sisiatas = 'Heads'
    else:
        self.sisiatas = 'Tails'

# Method get_sideup mengembalikan nilai
# yang direferensikan oleh sisiatas

def get_sisiatas(self):
    return self.sisiatas

# Fungsi main
def main():
    # Buat object coin
    my_coin = Coin()

    # Tampilkan sisi coin yang menghadap atas
    print('Sisi koin yang menghadap atas:', my_coin.get_sisiatas())

    # Lempar koin
    print('Saya melempar koin...')
    my_coin.lempar()

    # Tampilkan sisi dari koin yang menghadap atas
    print('Sisi koin yang menghadap atas:', my_coin.get_sisiatas())

# Panggil fungsi main
main()

```

Output:

```

Sisi koin yang menghadap atas: Heads
Saya melempar koin...
Sisi koin yang menghadap atas: Tails

```

```

import random

# class Coin mensimulasikan sebuah coin
# yang dapat dilempar

class Coin:

    # Method __init__ menginisialisasi
    # attribute data sisiatas dengan 'Heads'.

    def __init__(self):
        self.sisiatas = 'Heads'

    # Method lempar menggenerasi sebuah angka random
    # di antara 0 s.d 1. Jika angka = 0, maka sisiatas
    # ditetapkan sebagai 'Heads'

```

```

# Jika = 1, maka sisiatas ditetapkan sebagai 'Tails'

def lempar(self):
    if random.randint(0, 1) == 0:
        self.sisiatas = 'Heads'
    else:
        self.sisiatas = 'Tails'

# Method get_sideup mengembalikan nilai
# yang direferensikan oleh sisiatas

def get_sisiatas(self):
    return self.sisiatas

```

Catatan:

Class `Coin` mempunyai:

1. Satu *Attribute*: `sisiatas`.
2. Tiga *Method*: `__init__`, `lempar`, `get_sisiatas`.

```

import random

# class Coin mensimulasikan sebuah coin
# yang dapat dilempar
class Coin:

    # Method __init__ menginisialisasi
    # attribute data sisiatas dengan 'Heads'.
    def __init__(self):
        self.sisiatas = 'Heads'

    # Method lempar menggenerasi sebuah angka random
    # di antara 0 s.d 1. Jika angka = 0, maka sisiatas
    # ditetapkan sebagai 'Heads'
    # Jika = 1, maka sisiatas ditetapkan sebagai 'Tails'
    def lempar(self):
        if random.randint(0, 1) == 0:
            self.sisiatas = 'Heads'
        else:
            self.sisiatas = 'Tails'

    # Method get_sisiatas mengembalikan nilai yang
    # direferensikan oleh sisiatas
    def get_sisiatas(self):
        return self.sisiatas

```

Fungsi `main` yang menggunakan *object* `coin`:

```

# Fungsi main
def main():
    # Buat object Coin
    my_coin = Coin()
    # Tampilkan sisi coin yang menghadap atas
    print('Sisi koin yang menghadap atas:', my_coin.get_sisiatas())

    # Lempar koin
    print('Saya melempar koin...')
    my_coin.lempar()
    # Tampilkan sisi dari koin yang menghadap atas
    print('Sisi koin yang menghadap atas:', my_coin.get_sisiatas())

# Panggil fungsi main
main()

```

Program berikut mendemonstrasikan pembuatan lebih dari satu *object* `coin`.

```

# coin_multiple_instance.py
# Mendemokan tiga object coin

```

```
import random

# class Coin mensimulasikan sebuah coin
# yang dapat dilempar
class Coin:
    # Method __init__ menginisialisasi
    # attribute data sisiatas dengan 'Heads'.
    def __init__(self):
        self.__sisiatas = 'Heads'

    # Method lempar menggenerasi sebuah angka random
    # di antara 0 s.d 1. Jika angka = 0, maka sisiatas
    # ditetapkan sebagai 'Heads'
    # Jika = 1, maka sisiatas ditetapkan sebagai 'Tails'
    def lempar(self):
        if random.randint(0, 1) == 0:
            self.__sisiatas = 'Heads'
        else:
            self.__sisiatas = 'Tails'

    # Method get_sisiatas mengembalikan nilai
    # yang direferensikan oleh sisiatas
    def get_sisiatas(self):
        return self.__sisiatas

# Fungsi main
def main():
    # Buat tiga object (instance) dari class Coin
    coin1 = Coin() # Instansiasi Coin dan namakan dengan coin1
    coin2 = Coin() # Instansiasi Coin dan namakan dengan coin2
    coin3 = Coin() # Instansiasi Coin dan namakan dengan coin3

    # Tampilkan sisi atas dari setiap koin
    print('Saya mempunyai tiga koin dengan sisi menghadap atas:')
    print(coin1.get_sisiatas())
    print(coin2.get_sisiatas())
    print(coin3.get_sisiatas())
    print()

    # Lempar koin
    print('Saya melempar tiga koin...')
    print()
    coin1.lempar()
    coin2.lempar()
    coin3.lempar()

    # Tampilkan sisi atas dari setiap koin
    print('Sekarang sisi-sisi berikut yang menghadap atas:')
    print(coin1.get_sisiatas())
    print(coin2.get_sisiatas())
    print(coin3.get_sisiatas())
    print()

    # Panggil fungsi main
main()
```

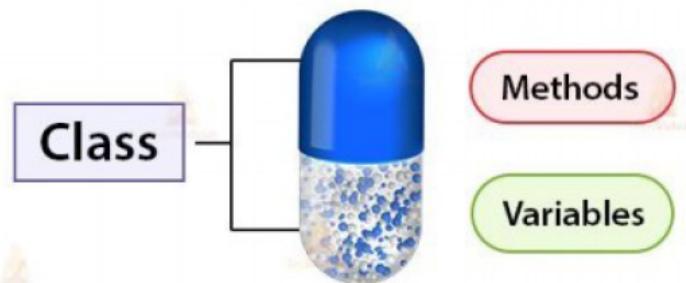
Output:

```
Saya mempunyai tiga koin dengan sisi menghadap atas:  
Heads  
Heads  
Heads
```

```
Saya melempar tiga koin...  
Sekarang sisi-sisi berikut yang menghadap atas:  
Heads  
Heads  
Tails
```

7.2.3 Encapsulation

Prinsip utama OOP adalah *encapsulation* (pengkapsulan) yang berarti *attribute* dan *method* merupakan satu kesatuan dan *attribute* haruslah *private*. *Attribute* haruslah *private* berarti *attribute*-*attribute object* tersembunyi dari kode di luar *object* dan hanya *method* dalam *object* tersebut yang dapat mengakses dan membuat perubahan ke data *attribute*. Dengan menyembunyikan *attribute* dari kode di luar *object*, kita memproteksi *attribute*-*attribute* dari perubahan yang membuat *object* kita tidak konsisten dengan desain *class* kita.



Pada contoh *class* `Coin` sebelumnya, *attribute* `sisiatas` belum tersembunyi. Karena belum tersembunyi, kita bisa mengubah *attribute* `sisiatas` tanpa melalui *method*.

```
my_coin = Coin()  
my_coin.sisiatas = 'Head'
```

Pada contoh di atas kita mengubah `sisiatas` menjadi nilai '*Head*' dan ini tidak sesuai dengan desain *class* kita, karena kita mendesain *attribute* `sisiatas` hanya dapat bernilai '*Heads*' atau '*Tails*'.

Untuk membuat *attribute* menjadi *private* kita menambahkan dua karakter *underscore* sebelum nama *attribute*:

```
self.__sisiatas = 'Heads'
```

Contoh program *class* `Coin` yang menyembunyikan *attribute*-nya:

```
import random  
  
# class Coin mensimulasikan sebuah coin yang dapat dilempar  
class Coin:  
  
    # Method __init__ menginisialisasi attribute data sisiatas dengan 'Heads'.
```

```

def __init__(self):
    self.__sisiatas = 'Heads'

    # Method Lempar menggenerasi sebuah angka random
    # di antara 0 s.d 1. Jika angka = 0, maka sisiatas ditetapkan sebagai
    'Heads'
    # Jika = 1, maka sisiatas ditetapkan sebagai 'Tails'
def Lempar(self):
    if random.randint(0, 1) == 0:
        self.__sisiatas = 'Heads'
    else:
        self.__sisiatas = 'Tails'

    # Method get_sisiatas mengembalikan nilai yang direferensikan oleh sisiatas
def get_sisiatas(self):
    return self.__sisiatas

```

Kita menyembunyikan *attribute* dengan menambahkan dua *underscore* sebelum nama *attribute*.

```
self.__sisiatas = 'Heads'
```

Ketika kita mencoba mengakses *attribute* secara langsung atau mencoba mengubah nilainya tanpa melalui *method* maka *interpreter* akan memberikan *error*.

```

my_coin = Coin()
print(my_coin.__sisiatas)

```

```

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    my_coin.__sisiatas
AttributeError: 'Coin' object has no attribute '__sisiatas'

```

Pada contoh program yang mensimulasikan lemparan koin, kita menuliskan definisi `class Coin` dan fungsi `main` yang menggunakan `class Coin` dalam satu *file*. Ini tidak masalah jika kita menuliskan program kecil yang hanya menggunakan satu atau dua `class`, namun ketika kita menuliskan program dengan banyak `class` kita perlu mengorganisasi `class-class` tersebut. *Programmer* umumnya mengorganisasi `class` dalam sebuah *module*.

Contoh `class Coin` pada *module*:

File: coin.py

```

import random

# class Coin mensimulasikan sebuah coin
# yang dapat dilempar
class Coin:

    # Method __init__ menginisialisasi
    # attribute data sisiatas dengan 'Heads'.
    def __init__(self):
        self.__sisiatas = 'Heads'

    # Method Lempar menggenerasi sebuah angka random
    # di antara 0 s.d 1. Jika angka = 0, maka sisiatas

```

```

# ditetapkan sebagai 'Heads'
# Jika = 1, maka sisiatas ditetapkan sebagai 'Tails'
def lempar(self):
    if random.randint(0, 1) == 0:
        self.__sisiatas = 'Heads'
    else:
        self.__sisiatas = 'Tails'

# Method get_sisiatas mengembalikan nilai yang
# direferensikan oleh sisiatas
def get_sisiatas(self):
    return self.__sisiatas

```

File: test_coin.py

```

import coin

# Fungsi main
def main():
    # Buat sebuah object dari class Coin
    my_coin = coin.Coin()

    # Tampilkan sisi coin yang menghadap atas
    print('Sisi koin yang menghadap atas:', my_coin.get_sisiatas())

    # Lempar koin
    print('Saya melempar koin...')
    my_coin.lempar()

    # Tampilkan sisi dari koin yang menghadap atas
    print('Sisi koin yang menghadap atas: ', my_coin.get_sisiatas())

# Panggil fungsi main
main()

```

File: test_coin.py

```

import coin
# Fungsi main
def main():
    # Buat sebuah object dari class Coin
    my_coin = coin.Coin()

```

Kita mengimport module coin yang berisi class Coin.

Menggunakan `<module>. <nama_object>` untuk membuat object dari class yang disimpan dalam module.

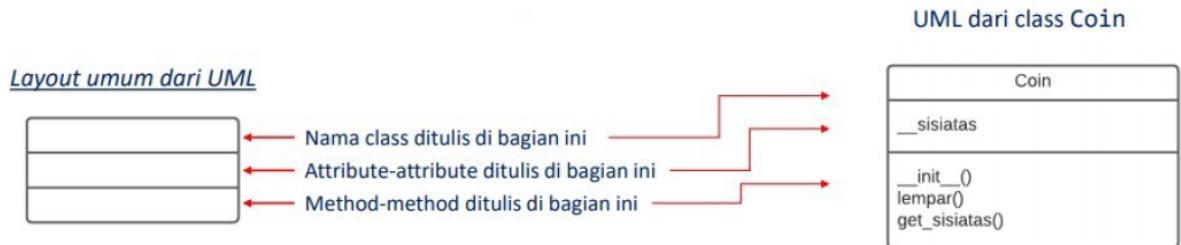
7.2.4 Unified Modeling Language

Ketika mendesain *class*, programmer umumnya menggunakan diagram *Unified Modeling Language* (atau disingkat dengan UML) untuk merepresentasikan desain dari *class*.

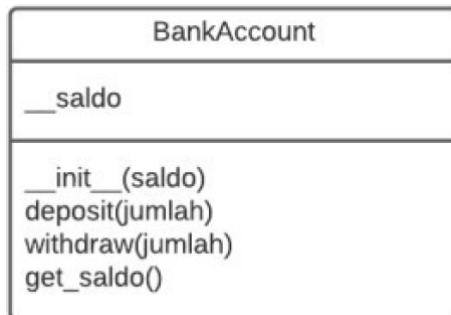
Diagram UML adalah sebuah kotak yang dibagi menjadi tiga bagian:

1. Bagian atas: nama *class*.
2. Bagian tengah: *attribute-attribute* dari *class*.

3. Bagian bawah: *method-method* dari *class*.



Misalkan kita membuat sebuah *class* `BankAccount` yang mensimulasikan sebuah rekening tabungan bank:



Class `BankAccount` merepresentasikan rekening tabungan bank:

```
# bankaccount.py
# class BankAccount mensimulasikan sebuah rekening bank
class BankAccount:

    # Method __init__ menginisialisasi attribute __saldo
    def __init__(self, saldo):
        self.__saldo = saldo

    # Method deposit (menabung) melakukan setoran
    # sejumlah uang ke rekening sehingga saldo bertambah
    def deposit(self, jumlah):
        self.__saldo += jumlah

    # Method withdraw (menarik) melakukan penarikan
    # sejumlah uang dari rekening bank
    def withdraw(self, jumlah):
        if self.__saldo >= jumlah:
            self.__saldo -= jumlah
        else:
            print('Gagal: Dana tidak mencukupi')

    # Method get_saldo mengembalikan saldo
    def get_saldo(self):
        return self.__saldo
```

Program berikut mendemonstrasikan *class* `BankAccount`:

```
# test_bankaccount.py
# Program ini mendemonstrasikan object dari class BankAccount
import bankaccount
```

```

def main():

    # Buat saldo awal
    saldo_awal = float(input('Masukkan saldo awal: '))

    # Buat object BankAccount
    tabungan = bankaccount.BankAccount(saldo_awal)

    # Tabung gaji pengguna
    gaji = float(input('Masukkan gaji Anda minggu ini: '))
    print('Saya akan depositokan gaji Anda ke rekening Anda.')
    tabungan.deposit(gaji)

    # Tampilkan saldo
    print(f'Saldo Anda adalah Rp.{tabungan.get_saldo():,.2f}')

    # Mengambil sejumlah uang
    cash = float(input(
        'Berapa banyak uang yang ingin Anda ambil? '))
    print('Saya akan ambil uang tersebut dari rekening Anda.')
    tabungan.withdraw(cash)

    # Tampilkan saldo
    print(f'Saldo Anda adalah Rp.{tabungan.get_saldo():,.2f}')

# Panggil fungsi main
main()

```

Output:

```

Masukkan saldo awal: 1000000
Masukkan gaji Anda minggu ini: 250000
Saya akan depositokan gaji Anda ke rekening Anda.
Saldo Anda adalah Rp.1,250,000.00
Berapa banyak uang yang ingin Anda ambil? 100000
Saya akan ambil sebanyak uang tersebut dari rekening Anda.
Saldo Anda adalah Rp.1,150,000.00

```

7.2.5 Method `__str__`

Seringkali kita perlu menampilkan sebuah pesan yang mengindikasikan *state* dari *object*. *State* dari *object* adalah nilai-nilai *attribute* dalam satu waktu. Menampilkan *state* adalah hal yang umum dilakukan, sehingga *programmer* biasanya mempunyai sebuah *method* mengembalikan sebuah `string` yang menampilkan *state* dari *object*. Dalam python, kita memberikan nama *method* yang mengembalikan *state* dari *object* dengan nama spesial: `__str__`.

Menambahkan *method* `__str__` pada *class* `BankAccount`:

```

def __str__(self):
    return f'Saldo Anda adalah Rp.{self.__saldo:,.2f}'

```

Kita tidak memanggil langsung *method* `__str__`, *method* ini dipanggil otomatis ketika kita memberikan argument nama *object* ke fungsi `print`:

```
tabungan = BankAccount(100000)
print(tabungan)
```

Catatan:

Ketika kita memberikan nama *object* sebagai argument ke fungsi `print`, *interpreter* otomatis memanggil *method* `__str__`.

Class `BankAccount` dengan *method* `__str__`

```
# bankaccount2.py
# class BankAccount mensimulasikan sebuah rekening bank
class BankAccount:

    # Method __init__ menginisiasi attribute __saldo
    def __init__(self, saldo):
        self.__saldo = saldo

    # Method deposit (menabung) melakukan setoran
    # sejumlah uang ke rekening sehingga saldo bertambah
    def deposit(self, jumlah):
        self.__saldo += jumlah

    # Method withdraw (menarik) melakukan penarikan
    # sejumlah uang dari rekening bank
    def withdraw(self, jumlah):
        if self.__saldo >= jumlah:
            self.__saldo -= jumlah
        else:
            print('Gagal: Dana tidak mencukupi')

    # Method get_saldo mengembalikan saldo
    def get_saldo(self):
        return self.__saldo

    # Method __str__ mengembalikan sebuah string yang menunjukkan state dari
    # object
    def __str__(self):
        return f'Saldo Anda adalah Rp.{self.__saldo:.2f}'
```

Program yang mendemonstrasikan `BankAccount` dengan *method* `__str__`:

```
# test_bankaccount2.py
# Program ini mendemonstrasikan object dari class BankAccount
import bankaccount2

def main():

    # Buat saldo awal
    saldo_awal = float(input('Masukkan saldo awal: '))

    # Buat object BankAccount
    tabungan = bankaccount2.BankAccount(saldo_awal)

    # Tabung gaji pengguna
    gaji = float(input('Masukkan gaji Anda minggu ini: '))
```

```

print('Saya akan depositokan gaji Anda ke rekening Anda.')
tabungan.deposit(gaji)

# Tampilkan saldo
print(tabungan)

# Mengambil sejumlah uang
cash = float(input('Berapa banyak uang yang ingin Anda ambil? '))
print('Saya akan ambil uang tersebut dari rekening Anda.')
tabungan.withdraw(cash)

# Tampilkan saldo
print(tabungan)

# Panggil fungsi main
main()

```

Output:

```

Masukkan saldo awal: 1000000
Masukkan gaji Anda minggu ini: 250000
Saya akan depositokan gaji Anda ke rekening Anda.
Saldo Anda adalah Rp.1,250,000.00
Berapa banyak uang yang ingin Anda ambil? 100000
Saya akan ambil sebanyak uang tersebut dari rekening Anda.
Saldo Anda adalah Rp.1,150,000.00

```

7.2.6 Method Accessor dan Mutator

Encapsulation pada OOP berarti kita harus mendesain *class* kita dengan menyembunyikan *attribute*. Ini juga berarti kita harus menyediakan *method-method* yang dapat digunakan untuk mengubah nilai *attribute* dan yang dapat digunakan untuk mengakses nilai *attribute*. *Method-method* untuk mengakses nilai *attribute* tanpa mengubah nilainya disebut sebagai *method accessor* dan *method-method* untuk mengubah nilai *attribute* disebut sebagai *method mutator*. Pada contoh *class* `BankAccount` kita mempunyai *method accessor* yang kita gunakan untuk mengakses nilai *attribute* `saldo`: *method* `get_saldo`. Namun pada *class* `BankAccount` kita tidak mempunyai *method mutator* karena kita mendesain *class* kita dengan tidak memperbolehkan *attribute* `saldo` diubah langsung namun melalui *method-method* tidak langsung: *method* `deposit` dan *method* `withdraw`.

Misalkan kita menambahkan sebuah *attribute* `nomor_rekening` pada *class* `BankAccount` dan kita membolehkan *attribute* ini ditetapkan dan diakses oleh kode di luar *object* melalui sebuah *method mutator* dan sebuah *method accessor*.

```

class BankAccount:

    # Method __init__ menginisiasi attribute __saldo
    def __init__(self, saldo, nomor_rekening):
        self.__saldo = saldo
        self.__nomor_rekening = nomor_rekening

    ...
    # Method accessor untuk nomor_rekening
    def get_nomor_rekening(self):
        return self.__nomor_rekening ] Method accessor untuk attribute nomor_rekening

    # Method mutator untuk nomor_rekening
    def set_nomor_rekening(self, nomor_rekening):
        self.__nomor_rekening = nomor_rekening ] Method mutator untuk attribute nomor_rekening

```

Method-method accessor umumnya dinamakan dengan diawali kata `get` yang diikuti dengan nama *attribute*, sehingga *method accessor* seringkali disebut juga sebagai *getters*. Sedangkan *method-method mutator* umumnya dinamakan dengan diawali kata `set` yang diikuti dengan nama *attribute*, sehingga *method mutator* seringkali disebut juga sebagai *setters*.

Class `BankAccount` dengan *attribute* `nomor_rekening` dan *method accessor* dan *mutator*nya:

```

# bankaccount3.py
# class BankAccount mensimulasikan sebuah rekening bank
class BankAccount:

    # Method __init__ menginisiasi attribute __saldo
    def __init__(self, saldo, nomor_rekening):
        self.__saldo = saldo
        self.__nomor_rekening = nomor_rekening

    # Method deposit (menabung) melakukan setoran
    # sejumlah uang ke rekening sehingga saldo bertambah
    def deposit(self, jumlah):
        self.__saldo += jumlah

    # Method withdraw (menarik) melakukan penarikan
    # sejumlah uang dari rekening bank
    def withdraw(self, jumlah):
        if self.__saldo >= jumlah:
            self.__saldo -= jumlah
        else:
            print('Gagal: Dana tidak mencukupi')

    # Method get_saldo mengembalikan saldo
    def get_saldo(self):
        return self.__saldo

    # Method accessor untuk nomor_rekening
    def get_nomor_rekening(self):
        return self.__nomor_rekening

    # Method mutator untuk nomor_rekening
    def set_nomor_rekening(self, nomor_rekening):
        self.__nomor_rekening = nomor_rekening

    # Method __str__ mengembalikan sebuah string
    # yang menunjukkan state dari object

```

```
def __str__(self):
    return f'Nomor rekening: {self.__nomor_rekening}' + \
        f'\nSaldo: Rp.{self.__saldo:,.2f}'
```

Program yang mendemonstrasikan `class BankAccount` dengan `attribute nomor_rekening` beserta `accessor` dan `mutator`nya:

```
# test_bankaccount3.py
# Program ini mendemonstrasikan object dari class BankAccount
import bankaccount3

def main():
    # Buat saldo awal
    saldo_awal = float(input('Masukkan saldo awal: '))

    # Buat object BankAccount
    tabungan = bankaccount3.BankAccount(saldo_awal, '123456789')

    # Tabung gaji pengguna
    gaji = float(input('Masukkan gaji Anda minggu ini: '))
    print('Saya akan depositkan gaji Anda ke rekening Anda.')
    tabungan.deposit(gaji)

    # Tampilkan state
    print(tabungan)

    # Mengambil sejumlah uang
    cash = float(input('Berapa banyak uang yang ingin Anda ambil? '))
    print('Saya akan ambil uang tersebut dari rekening Anda.')
    tabungan.withdraw(cash)

    # Tampilkan state
    print(tabungan)

    # Ubah nomor rekening
    nomor_rekening = input('Masukkan nomor rekening baru: ')
    tabungan.set_nomor_rekening(nomor_rekening)

    # Tampilkan nomor rekening
    print(f'Nomor rekening baru {tabungan.get_nomor_rekening()}')

# Panggil fungsi main
main()
```

Output program `test_bankaccount3.py`:

```
Masukkan saldo awal: 1000000
Masukkan gaji Anda minggu ini: 250000
Saya akan depositokan gaji Anda ke rekening Anda.
Nomor rekening: 123456789
Saldo: Rp.1,250,000.00
Berapa banyak uang yang ingin Anda ambil? 100000
Saya akan ambil uang tersebut dari rekening Anda.
Nomor rekening: 123456789
Saldo: Rp.1,150,000.00
Masukkan nomor rekening baru: 234567890
Nomor rekening baru 234567890
```

7.2.7 Object Sebagai Argumen Fungsi

Ketika kita membuat program yang bekerja dengan *object*, seringkali kita perlu membuat sebuah fungsi yang menerima *object* sebagai argumen. Sebagai contoh, fungsi berikut menerima *object coin* sebagai argumen:

```
def status_koin(obj_coin):
    print('Sisi koin yang menghadap atas:', obj_coin.get_sisiatas())
```

Kode berikut menunjukkan bagaimana kita membuat *object coin* lalu memberikannya sebagai argumen ke fungsi `status_koin`:

```
my_coin = coin.Coin()
status_koin(my_coin)
```

Ketika kita memberikan argumen berupa *object* ke suatu fungsi, variabel parameter dari fungsi tersebut akan menerima referensi ke *object* tersebut, sehingga kita dapat memanggil *method* dari *object* tersebut.

Program yang mendemonstrasikan pemberian *object* ke fungsi:

```
# Program ini mendemonstrasikan pemberian argumen
# berupa object ke fungsi
import coin

# Fungsi Tempar_koin melempar koin
def Tempar_koin(obj_coin):
    obj_coin.Tempar()

# Fungsi main
def main():
    # Buat object Coin
    my_coin = coin.Coin()

    # Ini akan menampilkan 'Heads'
    print(my_coin.get_sisiatas())

    # Berikan object my_coin ke fungsi Tempar_koin
    Tempar_koin(my_coin)
```

```
# Ini akan menampilkan 'Heads' atau 'Tails'
print(my_coin.get_sisiatas())

# Panggil fungsi main
main()
```

Output:

```
Heads
Tails
```

Output:

```
Heads
Heads
```

7.3 Inheritance

Konsep penting OOP setelah *encapsulation* adalah *inheritance* (pewarisan). Konsep *inheritance* didasarkan pada pengamatan *object-object* di dunia nyata. Banyak *object* merupakan versi spesialisasi dari *object* lainnya yang lebih generik. Misalkan, serangga merupakan jenis hewan dengan karakteristik-karakteristik tertentu sedangkan lebah dan belalang adalah versi spesialisasi dari serangga yang mempunyai (mewarisi) karakteristik-karakteristik dari serangga namun masing-masing juga mempunyai karakteristik khusus tertentu.



7.3.1 Superclass dan Subclass

Class yang merupakan versi spesialisasi dari suatu *class* disebut dengan *subclass* (atau *class turunan*). Sedangkan *class* yang merupakan versi generik dari suatu *class* disebut dengan *superclass* (atau *class dasar*).

Syntax umum penulisan *subclass*:

```
class SuperClass:
    <kode-kode>
```

```
class SubClass(SuperClass):
    <kode-kode>
```

Syntax penulisan class yang merupakan subclass dari superclass

Nama superclass ditulis di dalam tanda kurung setelah nama subclass

Subclass akan mewarisi semua *attribute* dan *method* dari *superclass*. Pada *subclass* juga dapat ditambahkan *attribute-attribute* dan *method-method* baru yang membuatnya versi spesialisasi dari *superclass*.

```
class Orang:
    def __init__(self, nama, umur):
        self.__nama = nama
        self.__umur = umur

    def tampilan_nama(self):
        print('Nama saya adalah', self.__nama)

    def tampilan_umur(self):
        print('Umur saya adalah', self.__umur)
```

Class *Orang* adalah *superclass*.
Attribute: *__nama* dan *__umur*.
Method:

- *tampilan_nama()*
- *tampilan_umur()*

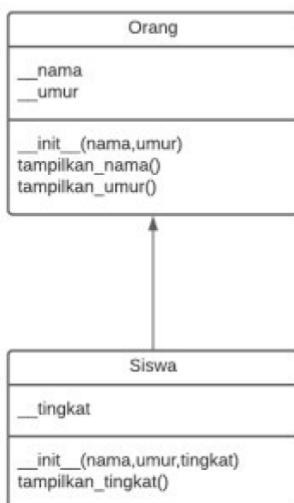
Superclass dituliskan dalam tanda kurung.

```
class Siswa(Orang):
    def __init__(self, nama, umur, tingkat):
        Orang.__init__(self, nama, umur)
        self.__tingkat = tingkat

    def tampilan_tingkat(self):
        print('Saya tingkat', self.__tingkat)
```

Class *Siswa* merupakan *subclass* dari class *Orang*.
Semua *attribute* dan *method* dari *superclass* diwarisi ke *class Siswa*.
Class Siswa juga mempunyai *attribute* khusus dan *method* khusus.
Attribute khusus: *__tingkat*
Method khusus: *tampilan_tingkat()*

UML yang merepresentasikan Superclass Orang dan subclass Siswa



Panah ke atas berarti cabang ini adalah subclass dari class di atas.

Kode untuk menguji class *Siswa*:

```
siswa_1 = Siswa('Budi Susilo', 17, 12)
siswa_1.tampilan_nama()
siswa_1.tampilan_umur()
siswa_1.tampilan_tingkat()
```

Method-method pada superclass diwarisi ke subclass, sehingga bisa kita panggil pada instance subclass.

Kode untuk menguji class *Siswa*:

```
siswa_1 = Siswa('Budi Susilo', 17, 12)
siswa_1.tampilan_nama()
siswa_1.tampilan_umur()
siswa_1.tampilan_tingkat()
```

Method-method pada superclass diwarisi ke subclass, sehingga bisa kita panggil pada instance subclass.

Output:

```
Nama saya adalah Budi Susilo  
Umur saya adalah 17  
Saya tingkat 12
```

7.3.2 Contoh Penerapan Inheritance

Misalkan kita membuat sebuah program manajemen inventaris untuk suatu diler mobil:

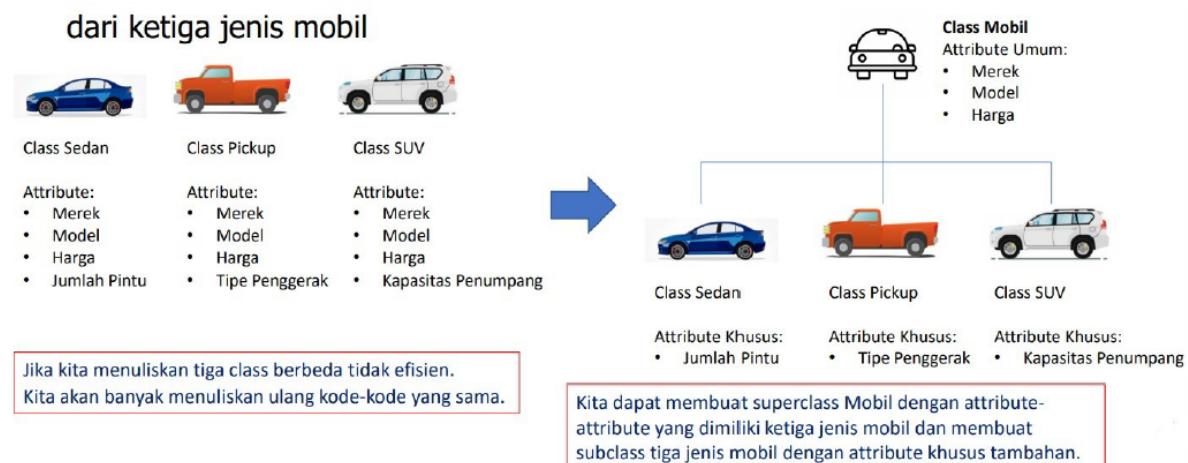
1. Diler mobil menjual tiga jenis mobil: sedan, pickup, SUV.
2. Kita mendesain program untuk menyimpan *attribute-attribute* data dari inventaris mobil berikut: merek, model, dan harga.

Selain tiga data yang sama untuk ketiga jenis mobil, setiap jenis mobil mempunyai data *attribute* tambahan sebagai berikut:

1. Sedan mempunyai *attribute* tambahan: banyak pintu (2 pintu atau 4 pintu).
2. Pickup mempunyai *attribute* tambahan: tipe penggerak (penggerak dua-roda atau penggerak empat-roda).
3. SUV mempunyai *attribute* tambahan: kapasitas penumpang (5 penumpang atau 7 penumpang).

Jika kita membuat sebuah *class* untuk masing-masing jenis mobil, program yang akan kita tulis tidaklah efisien karena ketiga jenis mobil mempunyai banyak *attribute-attribute* yang sama.

Daripada membuat tiga *class* berbeda, kita dapat membuat sebuah *class* generik (*superclass*) dengan *attribute-attribute* yang sama dari ketiga jenis mobil dan membuat tiga *class* spesialisasi (*subclass*) dengan *attribute* khusus dari ketiga jenis mobil.



Superclass Mobil:

```
# class Mobil menyimpan data-data umum mengenai inventaris mobil pada suatu diler mobil
class Mobil:
    # Method __init__ menerima argumen merek, model, dan harga dan menginisialisasi
    # attribute-attribute dari class Mobil dengan nilai-nilai argumen
    def __init__(self, merek, model, harga):
        self.__merek = merek
        self.__model = model
        self.__harga = harga

    # Method-method mutator
    def set_merek(self, merek):
        self.__merek = merek
    def set_model(self, model):
        self.__model = model
    def set_harga(self, harga):
        self.__harga = harga

    # Method-method accessor
    def get_merek(self):
        return self.__merek
    def get_model(self):
        return self.__model
    def get_harga(self):
        return self.__harga
```

Method-method Mutator

Pada class Mobil kita membuat method-method mutator yang digunakan untuk menetapkan nilai pada attribute-attribute.

Method-method Accessor

Pada class Mobil kita membuat method-method accessor yang digunakan untuk mengakses nilai attribute-attribute.

Subclass Sedan:

```
# class Sedan merupakan subclass (class turunan) dari class Mobil
class Sedan(Mobil):
    # Method __init__ menerima argumen merek, model,
    # harga dan banyak pintu
    def __init__(self, merek, model, harga, pintu):
        Mobil.__init__(self, merek, model, harga)
        self.__pintu = pintu

    # Method set_pintu adalah method khusus pada subclass Sedan
    # dan merupakan mutator untuk menetapkan attribute khusus __pintu
    def set_pintu(self, pintu):
        self.__pintu = pintu

    # Method get_pintu adalah method khusus pada subclass Sedan
    # dan merupakan accessor untuk mengakses attribute khusus __pintu
    def get_pintu(self):
        return self.__pintu
```

Pada method __init__ dari subclass Sedan:

- Kita menuliskan pemanggilan method __init__ dari superclass untuk menginisialisasi attribute-attribute generik.
- Kita menginisialisasi attribute khusus __pintu langsung dengan nilai dari argumen pintu.

Kita menambahkan method mutator untuk menetapkan attribute khusus __pintu dari subclass Mobil.

Kita menambahkan method accessor untuk mengakses attribute khusus __pintu dari subclass Mobil.

Subclass Pickup:

```
# class Pickup merupakan subclass (class turunan) dari class Mobil
class Pickup(Mobil):
    # Method __init__ menerima argumen merek, model,
    # harga dan tipe_penggerak
    def __init__(self, merek, model, harga, tipe_penggerak):
        Mobil.__init__(self, merek, model, harga)
        self.__tipe_penggerak = tipe_penggerak

    # Method set_tipe_penggerak adalah method khusus pada subclass Pickup
    # dan merupakan mutator untuk menetapkan attribute khusus __tipe_penggerak
    def set_tipe_penggerak(self, tipe_penggerak):
        self.__tipe_penggerak = tipe_penggerak

    # Method get_tipe_penggerak adalah method khusus pada subclass Pickup
    # dan merupakan accessor untuk mengakses attribute khusus __tipe_penggerak
    def get_tipe_penggerak(self):
        return self.__tipe_penggerak
```

Pada method __init__ dari subclass Pickup:

- Kita menuliskan pemanggilan method __init__ dari superclass untuk menginisialisasi attribute-attribute generik.
- Kita menginisialisasi attribute khusus __tipe_penggerak langsung dengan nilai dari argumen tipe_penggerak.

Kita menambahkan method mutator untuk menetapkan attribute khusus __tipe_penggerak dari subclass Pickup.

Kita menambahkan method accessor untuk mengakses attribute khusus __pintu dari subclass Mobil.

Subclass SUV:

```
# class SUV merupakan subclass (class turunan) dari class Mobil
class SUV(Mobil):
    # Method __init__ menerima argumen merek, model,
    # harga dan kap_penumpang (kapasitas penumpang)
    def __init__(self, merek, model, harga, kap_penumpang):
        Mobil.__init__(self, merek, model, harga)
        self.__kap_penumpang = kap_penumpang

    # Method set_kap_penumpang adalah method khusus pada subclass SUV
    # dan merupakan mutator untuk menetapkan attribute khusus __kap_penumpang
    def set_kap_penumpang(self, kap_penumpang):
        self.__kap_penumpang = kap_penumpang

    # Method get_kap_penumpang adalah method khusus pada subclass SUV
    # dan merupakan accessor untuk mengakses attribute khusus __kap_penumpang
    def get_kap_penumpang(self):
        return self.__kap_penumpang
```

Pada method `__init__` dari subclass SUV:

- Kita menuliskan pemanggilan method `__init__` dari superclass untuk menginisialisasi attribute-attribute generik.
- Kita menginisialisasi attribute khusus `__kap_penumpang` langsung dengan nilai dari argumen `kap_penumpang`.

Kita menambahkan method mutator untuk menetapkan attribute khusus `__kap_penumpang` dari subclass SUV.

Kita menambahkan method accessor untuk mengakses attribute khusus `__pintu` dari subclass Mobil.

Simpan definisi *superclass* Mobil dan *subclass* Sedan, Pickup, SUV dalam *module* kendaraan lalu uji dengan program berikut:

```
# Program ini mendemonstrasikan class Sedan, Pickup, dan SUV
import kendaraan

def main():
    # Buat object Sedan
    sedan = kendaraan.Sedan('BMW', '323', 500000000, 4)

    # Buat object Pickup
    pickup = kendaraan.Pickup('Toyota', 'Hilux', 350000000, '4WD')

    # Buat object SUV
    suv = kendaraan.SUV('Hyundai', 'Santa Fe', 550000000, 6)

    print('Data Kendaraan')
    print('=====')

    # Tampilkan data Sedan
    print('Merek:', sedan.get_merek())
    print('Model:', sedan.get_model())
    print(f'Harga: Rp.{sedan.get_harga():,.2f}')
    print('Jumlah Pintu:', sedan.get_pintu())
    print()

    # Tampilkan data Pickup
    print('Merek:', pickup.get_merek())
    print('Model:', pickup.get_model())
    print(f'Harga: Rp.{pickup.get_harga():,.2f}')
    print('Tipe Penggerak:', pickup.get_tipe_penggerak())
    print()

    # Tampilkan data SUV
    print('Merek:', suv.get_merek())
    print('Model:', suv.get_model())
    print(f'Harga: Rp.{suv.get_harga():,.2f}')
    print('Kapasitas Penumpang:', suv.get_kap_penumpang())
    print()

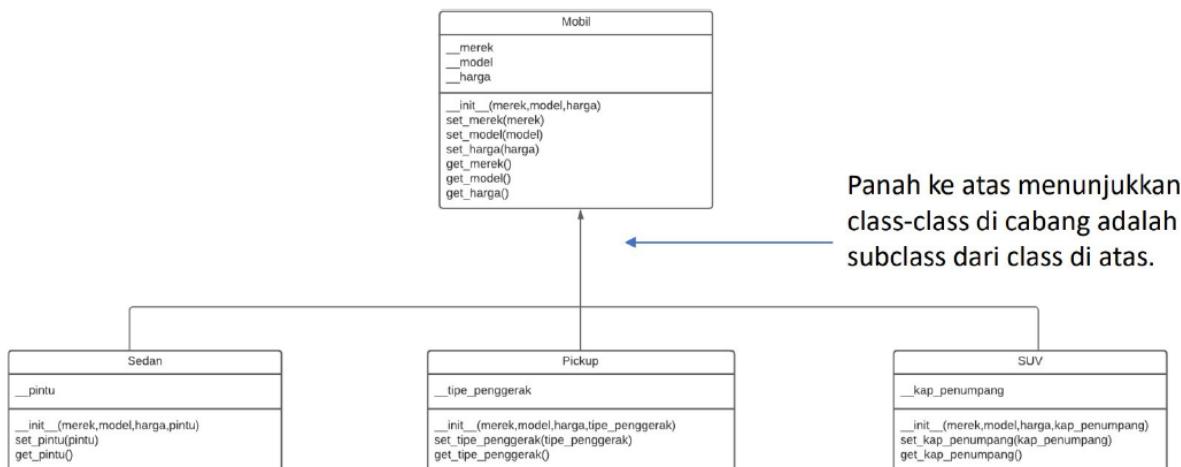
# Panggil fungsi main
main()
```

Output:

```
Data Kendaraan
=====
Merek: BMW
Model: 323
Harga: Rp. 500,000,000.00
Jumlah Pintu: 4

Merek: Toyota
Model: Hilux
Harga: Rp. 350,000,000.00
Tipe Penggerak: 4WD

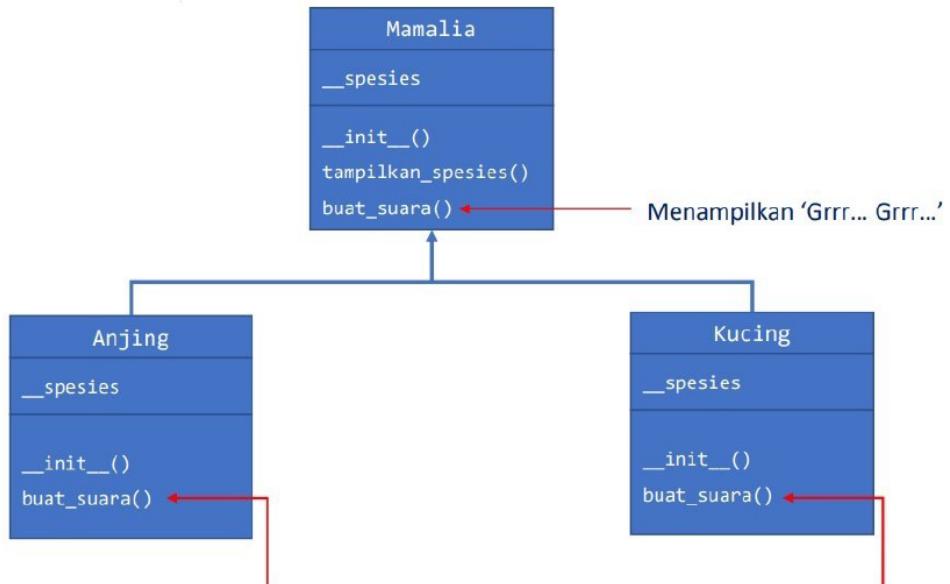
Merek: Hyundai
Model: Santa Fe
Harga: Rp. 550,000,000.00
Kapasitas Penumpang: 6
```



7.4 Polymorphism

Konsep penting OOP setelah *encapsulation* dan *inheritance* adalah *polymorphism*. "Poly" berarti banyak dan "Morphism" berarti bentuk. *Polymorphism* diartikan sebagai kemampuan object untuk mempunyai bentuk yang berbeda-beda. *Polymorphism* memungkinkan *method* dalam *subclass* memiliki nama yang sama seperti *method* pada *superclass* namun mempunyai implementasi berbeda. Proses untuk mendefinisikan *method* dengan nama sama namun dengan implementasi berbeda ini disebut dengan *method overriding*.

Contoh *Polymorphism*:



Meskipun superclass `Mamalia` telah mempunyai method `buat_suara()`, kita ingin method yang lebih spesifik untuk subclass `Anjing`. Kita ingin menampilkan ‘Guk! Guk!’ yang sesuai dengan suara anjing dibandingkan ‘Grrr... Grrr...’.

Meskipun superclass `Mamalia` telah mempunyai method `buat_suara()`, kita ingin method yang lebih spesifik untuk subclass `Kucing`. Kita ingin menampilkan ‘Meong!’ yang sesuai dengan suara kucing dibandingkan ‘Grrr... Grrr...’.

Contoh program:

```

# hewan.py

# Class mamalia merepresentasikan mamalia generik
class Mamalia:

    # Method __init__ menerima sebuah argumen untuk
    # spesies mamalia
    def __init__(self, spesies):
        self.__spesies = spesies

    # Method tampilan_spesies menampilkan sebuah pesan
    # yang menunjukkan spesies dari mamalia.
    def tampilan_spesies(self):
        print('saya adalah seekor', self.__spesies)

    # Method buat_suara menampilkan suara generik dari mamalia
    def buat_suara(self):
        print('Grrrr')

```

Catatan:

`Class Mamalia` mempunyai tiga method: `__init__()`, `tampilan_spesies()`, `buat_suara()`.

Contoh kode yang membuat *instance* dari `class Mamalia` dan memanggil *method-method* dalam `class` tersebut:

```

import hewan
mamalia = hewan.Mamalia('Mamalia Generik')
mamalia.tampilan_spesies()
mamalia.buat_suara()

```

Output:

```
Saya adalah seekor Mamalia Generik Grrrrr
```

```
# Class Anjing adalah subclass dari Mamalia
class Anjing(Mamalia):

    # Method __init__ memanggil method __init__
    # dari superclass dan memberikan argumen spesies 'Anjing'
    def __init__(self):
        Mamalia.__init__(self, 'Anjing')

    # Method buat_suara meng-override method buat_suara
    # dari superclass Mamalia
    def buat_suara(self):
        print('Guk! Guk!')
```

Output:

```
Saya adalah seekor Anjing
Guk! Guk!
```

Catatan:

1. Class `Anjing` adalah *subclass* dari class `Mamalia`.
2. Method `__init__` yang kita definisikan pada class `Anjing` mengoverride method `__init__` yang diwariskan dari *superclass* `Mamalia`.
3. Class `Mamalia` yang merupakan *superclass* dari class `Anjing` mempunyai method `buat_suara()`. Pada *subclass* `Anjing` kita meng-override method `buat_suara()` dengan mendefinisikan method dengan nama yang sama namun dengan implementasi berbeda (disini kita menampilkan string 'Guk! Guk!' ke layar).

Contoh kode yang membuat *instance* dari class `Anjing` dan memanggil method-methodnya:

```
import hewan
anjing = hewan.Anjing()
anjing.tampilkan_spesies()
anjing.buat_suara()
```

Catatan:

1. Karena di *subclass* `Anjing` tidak ada definisi method `tampilkan_spesies()` maka method `tampilkan_spesies()` dari *superclass* `Mamalia` yang dipanggil.
2. Pada *subclass* `Anjing` kita meng-override method `buat_suara()` sehingga method yang kita definisikan pada *subclass* `Anjing` yang digunakan ketika kita memanggil method `buat_suara()` pada *object* `Anjing`.

```

# Class Kucing adalah subclass dari class Mamalia
class Kucing(Mamalia):

    # Method __init__ memanggil method __init__
    # dari superclass dan memberikan argumen spesies 'Kucing'
    def __init__(self):
        Mamalia.__init__(self, 'Kucing')

    # Method buat_suara meng-override method buat_suara
    # dari superclass Mamalia
    def buat_suara(self):
        print('Meong!')

```

Output:

```

Saya adalah seekor Kucing
Meong!

```

Contoh kode yang membuat *instance* dari *class* `Kucing` dan memanggil *method-methodnya*:

```

import hewan
kucing = hewan.Kucing()
kucing.tampilkan_spesies()
kucing.buat_suara()

```

Catatan:

1. Karena di *subclass* `Kucing` tidak ada definisi *method* `tampilkan_spesies()` maka *method* `tampilkan_spesies()` dari *superclass* `Mamalia` yang dipanggil.
2. Pada *subclass* `Kucing` kita meng-override *method* `buat_suara()` sehingga *method* yang kita definisikan pada *subclass* `Kucing` yang digunakan ketika kita memanggil *method* `buat_suara()` pada *object* `Kucing`.

7.4.1 Keuntungan *Polymorphism*

Polymorphism memberikan fleksibilitas dalam mendesain program. Sebagai contoh, kita dapat membuat sebuah fungsi seperti berikut:

```

def tampilkan_info_mamalia(makhluk):
    makhluk.tampilkan_spesies()
    makhluk.buat_suara()

```

Kita bisa memanggil fungsi `info_mamalia` dengan argumen *object* apapun selama *object* tersebut memiliki *method* `tampilkan_spesies()` dan `buat_suara()`. Tanpa *polymorphism* kita harus membuat sebuah fungsi untuk setiap jenis *object*.

```

# demo_polymorphism_fungsi.py
# Program ini mendemonstrasikan fungsi yang menerima object dengan polymorphism

import hewan

# Fungsi ini menerima object sebagai argumen
# dan memanggil method tampilkan_spesies dan buat_suara

```

```

def tampilkan_info_mamalia(makhluk):
    makhluk.tampilkan_spesies()
    makhluk.buat_suara()

# Fungsi main
def main():

    # Buat object Mamalia, object Anjing, dan object Kucing
    mamalia = hewan.Mamalia('Mamalia Generik')
    anjing = hewan.Anjing()
    kucing = hewan.Kucing()

    # Tampilkan informasi spesies satu per satu
    print('Berikut ini beberapa hewan dan')
    print('suara yang mereka buat.')
    tampilkan_info_mamalia(mamalia)
    print()
    tampilkan_info_mamalia(anjing)
    print()
    tampilkan_info_mamalia(kucing)
    print()

# Panggil fungsi main
main()

```

Output:

```

Berikut ini beberapa hewan dan
suara yang mereka buat.
Saya adalah seekor Mamalia Generik
Grrrr
Saya adalah seekor Anjing
Guk! Guk!
Saya adalah seekor Kucing
Meong!

```

7.4.2 Fungsi `Isinstance`

Ketika kita membuat fungsi yang menerima sebuah *object* sebagai argumen dan memanggil *method-methodnya* kita harus memastikan *object* tersebut memiliki *method-method* yang dipanggil.

Misalkan:

```

# salah_tipe.py

def tampilkan_info_mamalia(makhluk):
    makhluk.tampilkan_spesies()
    makhluk.buat_suara()

def main():
    # Berikan sebuah string ke tampilkan_info_mamalia
    tampilkan_info_mamalia('Saya adalah string')

main()

```

Output dari program akan menghasilkan *error*, karena *object* yang diberikan ke fungsi tidak mempunyai *method-method* yang dipanggil dalam fungsi:

```
Traceback (most recent call last):
...
AttributeError: 'str' object has no attribute 'tampilkan_spesies'
```

Kita bisa menggunakan fungsi `isinstance` untuk menentukan apakah suatu *object* adalah turunan dari *class* tertentu atau *subclass* dari suatu *class*.

Syntax pemanggilan fungsi `isinstance`:

```
isinstance(object, namaClass)
```

Jika *object* yang direferensikan dari *object* merupakan turunan dari nama *class* atau *subclass* dari nama *class* fungsi mengembalikan nilai *True* dan mengembalikan nilai *False* jika bukan. Kita dapat menggunakan fungsi `isinstance` sebagai *exception handler*.

```
def tampilkan_info_mamalia(makhluk):
    if isinstance(makhluk, hewan.Mamalia):
        makhluk.tampilkan_spesies()
        makhluk.buat_suara()
    else:
        print('Ini bukan mamalia.')
```

Contoh program dari fungsi `isinstance`:

```
# demo_polymorphism_fungsi2.py
# Program ini mendemonstrasikan fungsi yang menerima object dengan polymorphism
import hewan

# Fungsi ini menerima object sebagai argument dan memanggil method
# tampilkan_spesies dan buat_suara
def tampilkan_info_mamalia(makhluk):
    if isinstance(makhluk, hewan.Mamalia):
        makhluk.tampilkan_spesies()
        makhluk.buat_suara()
    else:
        print('Ini bukan mamalia.')

# Fungsi main
def main():
    # Buat object Mamalia, object Anjing, dan object Kucing
    mamalia = hewan.Mamalia('Mamalia Generik')
    anjing = hewan.Anjing()
    kucing = hewan.Kucing()
    # Tampilkan informasi spesies satu per satu
    print('Berikut ini beberapa hewan dan')
    print('suara yang mereka buat.')
    tampilkan_info_mamalia(mamalia)
    print()
    tampilkan_info_mamalia(anjing)
    print()
    tampilkan_info_mamalia(kucing)
    print()
```

```
tampilkan_info_mamalia('Saya adalah string')

# Panggil fungsi main
main()
```

Output:

```
Berikut ini beberapa hewan dan suara yang mereka buat. Saya adalah seekor Mamalia Generik
Grrrr

Saya adalah seekor Anjing
Guk! Guk!

Saya adalah seekor Kucing
Meong!

Ini bukan mamalia
```

REFERENSI

- [1] Gaddis, Tony. 2012. *Starting Out With Python Second Edition*. United States of America: Addison-Wesley.