LAPORAN RESMI

MODUL VI

POLYMORPHISM

PEMROGRAMAN BERBASIS OBJEK

NAMA	**: ADYTTA PUTRA TARIGAN**
N.R.P	**: 240441100139**
DOSEN	**: YUDHA DWI PUTRA NEGARA, S.KOM., M.KOM.**
ASISTEN	**: AHMAD RIKHAN ARBA'I**
TGL PRAKTIKUM	**: 24 MEI 2025**

Disetujui : 3 JUNI 2025
Asisten

AHMAD RIKHAN ARBA'I
23.04.411.00192

LABORATORIUM TEKNOLOGI INFORMASI
PRODI SISTEM INFORMASI
JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNIK
UNIVERSITAS TRUNOJOYO MADURA

BAB I
PENDAHULUAN

1.1 Latar Belakang

Dalam dunia pemrograman berorientasi objek (OOP), *polymorphism* merupakan salah satu konsep fundamental yang memungkinkan objek-objek dari kelas yang berbeda untuk merespons metode yang sama dengan cara yang berbeda. Istilah "*polymorphism*" berasal dari bahasa Yunani yang berarti "banyak bentuk", dan dalam konteks pemrograman, ini berarti bahwa satu *interface* dapat memiliki berbagai implementasi. Dengan adanya *polymorphism*, pengembang dapat menulis kode yang lebih *fleksibel* dan mudah diperluas karena tidak perlu mengetahui secara pasti jenis objek yang digunakan saat menjalankan metode tertentu.

Polymorphism juga membantu dalam meningkatkan keterbacaan dan pemeliharaan kode program, karena memungkinkan penggunaan metode yang sama untuk objek-objek dari berbagai kelas turunan. Misalnya, dalam konteks program penghitungan luas bangun datar, metode luas() dapat digunakan pada objek persegi, lingkaran, dan segitiga meskipun cara perhitungannya berbeda untuk masing-masing bentuk. Hal ini mempermudah pengembang dalam mengelola berbagai bentuk data yang memiliki perilaku serupa, sekaligus menjaga struktur kode tetap sederhana dan elegan. Dalam pengembangan perangkat lunak skala besar, *polymorphism* menjadi kunci untuk mengimplementasikan prinsip desain seperti *Open/Closed Principle*, di mana kode terbuka untuk ekstensi namun tertutup untuk modifikasi.

1.2 Tujuan

- Mahasiswa mampu memahami konsep *polymorphism* dalam pemrograman berbasis objek.
- Mahasiswa mampu mengimplementasikan *polymorphism* menggunakan Python.
- Mahasiswa dapat membedakan antara *method overriding* dan *duck typing*.

BAB II
DASAR TEORI

2.1 Pengertian *Polymorphism*

Polymorphism pada Python adalah konsep dalam pemrograman berorientasi objek (OOP) yang memungkinkan suatu entitas seperti objek, fungsi, atau metode untuk memiliki banyak bentuk atau perilaku yang berbeda, tergantung pada konteks penggunaannya. Secara harfiah, "*polymorphism*" berasal dari bahasa Yunani, yaitu poly berarti banyak dan morph berarti bentuk. Dalam Python, *polymorphism* memungkinkan objek-objek dari kelas yang berbeda untuk merespons pemanggilan metode yang sama dengan cara yang berbeda sesuai dengan implementasi pada masing-masing kelas. Contohnya, dua kelas berbeda dapat memiliki metode dengan nama yang sama, namun perilakunya bisa berbeda tergantung pada kelas mana objek tersebut berasal. *Polymorphism* juga memungkinkan penggunaan satu antarmuka (*interface*) untuk berbagai tipe objek, sehingga kode menjadi lebih fleksibel dan mudah dikembangkan.

Contoh *Polymorphism*:

Misalkan ada kelas induk Binatang dengan metode bersuara(), lalu ada dua kelas turunan Anjing dan Kucing yang masing-masing mengimplementasikan metode bersuara() dengan perilaku berbeda:

```
class Binatang:
    def bersuara(self):
        pass

class Anjing(Binatang):
    def bersuara(self):
        return "Woof!"

class Kucing(Binatang):
    def bersuara(self):
        return "Meow!"

def suara_hewan(hewan):
    return hewan.bersuara()

anjing = Anjing()
kucing = Kucing()

print(suara_hewan(anjing))  # Output: Woof!
print(suara_hewan(kucing))  # Output: Meow!
```

Pada contoh di atas, fungsi suara_hewan dapat menerima objek dari kelas apa pun yang memiliki metode bersuara(), dan hasil yang diberikan akan berbeda tergantung

pada objek yang diberikan.

2.2 Jenis-jenis *polymorphism*

Python mendukung beberapa jenis *polymorphism* yang umum dalam pemrograman berorientasi objek. Berikut penjelasan jenis-jenis *polymorphism* yang relevan di Python:

2.2.1 *Duck Typing*

Duck Typing adalah konsep *polymorphism* di Python yang berdasarkan pada perilaku objek, bukan tipe objeknya. Jika sebuah objek memiliki metode atau atribut yang diperlukan, maka objek tersebut dapat digunakan, tanpa mempedulikan kelas atau tipe aslinya. Prinsipnya: "If it walks like a *duck* and quacks like a *duck*, it is a *duck*."

Contoh Kode :

```
class Bebek:
    def suara(self):
        return "Kwek kwek"

class Angsa:
    def suara(self):
        return "Kriik kriik"

def buat_suara(hewan):
    print(hewan.suara())

bebek = Bebek()
angsa = Angsa()

buat_suara(bebek)  # Output: Kwek kwek
buat_suara(angsa)  # Output: Kriik kriik
```

Di sini, fungsi buat_suara dapat menerima objek dari kelas apa pun selama objek tersebut memiliki metode suara(), tanpa perlu inheritance

2.2.2 Operator Overloading

Operator overloading adalah kemampuan untuk mendefinisikan ulang perilaku operator (seperti +, -, , dll) untuk objek dari kelas tertentu. Dengan ini, objek dapat merespons operator dengan cara yang sesuai dengan konteksnya.

Contoh Kode:

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

    def __add__(self, other):
        return Titik(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

t1 = Titik(1, 2)
t2 = Titik(3, 4)
t3 = t1 + t2

print(t3)  # Output: (4, 6)
```

Metode __add__ di kelas Titik meng-overload operator + sehingga objek Titik dapat dijumlahkan secara khusus.

2.2.3 *Method Overloading*

Method overloading adalah kemampuan membuat beberapa metode dengan nama yang sama tetapi berbeda parameter (jumlah atau tipe). Python tidak mendukung *method* overloading secara eksplisit seperti bahasa lain (misal Java). Namun, efek serupa dapat dicapai dengan menggunakan parameter *default, args,* atau *kwargs*.

Contoh Kode:

```python
class Kalkulator:
    def tambah(self, a, b, c=0):
        return a + b + c

calc = Kalkulator()
print(calc.tambah(2, 3))      # Output: 5
print(calc.tambah(2, 3, 4))   # Output: 9
```

Metode tambah dapat dipanggil dengan 2 atau 3 argumen, sehingga memberikan efek *overloading*.

2.2.4 *Method* Overriding

Method overriding terjadi ketika *subclass* mengubah implementasi metode yang sudah ada di superclass. Ini adalah bentuk *polymorphism* dinamis (*runtime polymorphism*), di mana metode yang dipanggil bergantung pada tipe objek actual saat runtime.

```python
class Binatang:
    def suara(self):
        return "Suara binatang"

class Anjing(Binatang):
    def suara(self):
        return "Guk guk"

class Kucing(Binatang):
    def suara(self):
        return "Meong"

def buat_suara(hewan):
    print(hewan.suara())

anjing = Anjing()
kucing = Kucing()

buat_suara(anjing)  # Output: Guk guk
buat_suara(kucing)  # Output: Meong
```

Metode suara pada Anjing dan Kucing menimpa metode suara di kelas Binatang, sehingga perilaku berbeda muncul saat dipanggil

Kesimpulan dalam bentuk tabel:

Jenis Polimorphism	Penjelasan Singkat	Contoh Implementasi
Duck Typing	Polimorfisme berdasarkan perilaku objek	Fungsi menerima objek dengan metode tertentu tanpa perlu inheritance
Operator Overloading	Mendefinisikan ulang operator untuk objek tertentu	Override __add__, __str__
Method Overloading	Metode dengan nama sama tapi parameter berbeda (dengan trik Python)	Parameter default atau args/kwargs
Method Overriding	Subclass mengganti metode superclass	Override metode di subclass

Keempat jenis ini adalah bentuk polimorfisme yang paling sering digunakan di Python untuk membuat kode fleksibel dan mudah dikembangkan.

2.3 *Polymorphism* dengan *Duck Typing*

Polimorfisme dengan *duck typing* di Python adalah pendekatan *polymorphism* yang memungkinkan penggunaan objek dari berbagai tipe berbeda secara bergantian, selama objek-objek tersebut memiliki metode atau atribut tertentu yang dibutuhkan, tanpa harus berbagi kelas induk atau tipe yang sama. Konsep ini diambil dari pepatah: "*If it walks like a duck and quacks like a duck, it must be a duck.*" Artinya, selama objek memiliki perilaku yang diharapkan (misal metode tertentu), objek tersebut dapat digunakan di tempat yang sama. *Duck typing* memungkinkan fungsi atau metode untuk bekerja dengan objek apa pun yang

memiliki metode atau atribut yang diperlukan, tanpa memeriksa tipe objek secara eksplisit menggunakan *type*() atau *isinstance*(). Ini membuat kode lebih fleksibel dan *loosely coupled* karena tidak tergantung pada hirarki kelas. Polimorfisme tercapai karena objek-objek berbeda dapat diperlakukan sama selama mereka "berperilaku" sama.

Contoh *Duck Typing*:

```
class Duck:
    def swim(self):
        print("The duck is swimming")
    def fly(self):
        print("The duck is flying")

class Swan:
    def swim(self):
        print("The swan is swimming")
    def fly(self):
        print("The swan is flying")

class Airplane:
    def fly(self):
        print("The airplane is flying")

def perform_actions(entity):
    entity.fly()
    if hasattr(entity, 'swim'):
        entity.swim()

duck = Duck()
swan = Swan()
plane = Airplane()

for obj in (duck, swan, plane):
    perform_actions(obj)
```

Output:

```
The duck is flying
The duck is swimming
The swan is flying
The swan is swimming
The airplane is flying
```

Penjelasan:

a. Fungsi perform_actions memanggil metode *fly*() pada objek apa pun yang diberikan.
b. Jika objek memiliki metode *swim*(), maka juga dipanggil.
c. Tidak ada pemeriksaan tipe *eksplisit*, cukup mengandalkan keberadaan metode.
d. Objek *Duck* dan *Swan* memiliki kedua metode, sedangkan *Airplane* hanya punya *fly*().

e. Ini menunjukkan *duck typing* memungkinkan polimorfisme tanpa inheritance

Duck typing adalah bentuk polimorfisme dinamis di Python yang mengandalkan kesamaan antarmuka (metode/atribut) daripada tipe objek. Ini membuat kode lebih fleksibel dan mudah diperluas tanpa harus membuat hierarki kelas yang ketat.

2.4 *Polymorphism* dengan *Abstract Base Class* (ABC)

Polimorfisme dengan *Abstract Base Class* (ABC) adalah cara formal dan terstruktur untuk menerapkan *polymorphism* di Python. ABC memungkinkan kita mendefinisikan kelas dasar abstrak yang berisi metode-metode abstrak (tanpa implementasi) yang harus diimplementasikan oleh semua subclass konkrit. Dengan begitu, ABC memaksa subclass untuk menyediakan implementasi spesifik dari metode tersebut, sehingga menjamin konsistensi antarmuka sekaligus memungkinkan perilaku yang berbeda sesuai kelas turunan.

Cara Kerja ABC untuk Polimorfisme:

a. ABC didefinisikan dengan mewarisi kelas ABC dari modul abc.
b. Metode abstrak didekorasi dengan *@abstractmethod*, yang berarti metode ini tidak memiliki implementasi di kelas dasar dan wajib diimplementasikan di *subclass*.
c. Kelas abstrak tidak bisa diinstansiasi secara langsung.
d. Subclass yang tidak mengimplementasikan semua metode abstrak akan menghasilkan error saat instansiasi.
e. Polimorfisme tercapai karena fungsi atau kode lain dapat menggunakan objek dari kelas turunan yang berbeda melalui antarmuka yang sama (metode abstrak).

Contoh Implementasi Polimorfisme dengan ABC:

```python
from abc import ABC, abstractmethod

# Kelas abstrak dengan metode abstrak
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

# Subclass konkrit yang mengimplementasikan metode abstrak
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Fungsi yang menggunakan polimorfisme
def animal_sound(animal: Animal):
    print(animal.make_sound())

dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!
```

Penjelasan:

a. *Animal* adalah kelas abstrak dengan metode abstrak *make_sound*().
b. *Dog* dan *Cat* adalah *subclass* yang mengimplementasikan make_sound() secara berbeda.
c. Fungsi *animal_sound* dapat menerima objek dari tipe *Animal* mana pun dan memanggil metode *make_sound*() tanpa peduli tipe spesifiknya.
d. Ini adalah contoh polimorfisme yang terstruktur dan aman secara tipe.

Abstract Base Class (ABC) menyediakan cara formal untuk mendefinisikan antarmuka kelas dengan metode abstrak. ABC memastikan *subclass* mengimplementasikan metode yang diperlukan, sehingga memaksa konsistensi. Polimorfisme tercapai karena objek dari berbagai subclass dapat diperlakukan sama melalui antarmuka ABC. ABC cocok digunakan saat Anda ingin mendefinisikan kontrak yang jelas untuk subclass dalam aplikasi besar dan terstruktur.

.

BAB III

TUGAS PENDAHULUAN

3.1 Soal

1. Apa yang dimaksud dengan polymorphism dalam oop python? Jelaskan dengan bahasa sendiri.
2. Jelaskan perbedaan antara method overriding dan duck typing dalam python.
3. Bagaimana cara mengimplementasikan polymorphism menggunakan pewarisan (inheritance)? Buat contoh sederhanaya.
4. Apa itu modul abc di python? Bagaimana kaitannya dengan konsep inheritance dalam oop? Buat contoh sederhanaya.
5. Buatlah dua buah class yang memiliki method dengan nama yang sama, lalu buat satu fungsi yang dapat menerima objek dari kedua class tersebut. Jelaskan kenapa ini termasuk polymorphism.

3.2 Jawaban

1. Polymorphism artinya satu nama method yang bisa digunakan oleh berbagai objek dari class yang berbeda, tetapi dengan hasil yang sesuai karakter masing-masing objek.
2. Method overriding bergantung pada inheritance dan digunakan untuk mengganti method dari subclass. Sedang duck typing tidak bergantung pada inheritance, tapi pada objek punya method yang sesuai.
3. Dengan membuat superclass yang memiliki method tertentu, lalu membuat sub class yang mewarisi class induk dan mengoverride method tersebut sesuai kebutuhan masing-masing class.

contoh sederhanaya:

```
class Hewan:
    def bersuara(self):
        print("Hewan mengeluarkan suara")

class Kucing(Hewan):
    def bersuara(self):
        print("Kucing : meong")

# fungsi polymorphism
def buat_suara(hewan):
    hewan.bersuara()

hu = Kucing()

buat_suara(hu)
```

4. modul abc (Abstract Base Class) di Python digunakan untuk membuat class abstrak, yaitu class yang tidak bisa langsung dibuat objeknya dan biasanya berisi metod abstrak yang harus diimplementasikan oleh class turunannya.

Contoh sederhananya

```
from abc import ABC, abstractmethod

class Kendaraan(ABC):
    @abstracmethod
    def bergerak(self):
        pass

class Mobil(Kendaraan):
    def bergerak(self):
        Print("Mobil berjalan di Jalan")

def jalankan_kendaraan(kendaraan):
    kendaraan.bergerak()

m = mobil()

Jalankan_kendaraan(m)
```

5.
```
Class Piano:
    def mainkan(self):
        Print("Piano memainkan melodi lembut")

Class Drum:
    def mainkan(self):
        Print("Drum menghasilkan irama keras")

def tampilkan_suara(instrumen):
    instrumen.mainkan()

alat1 = Piano()
alat2 = Drum()
tampilkan_suara(alat1)
tampilkan_suara(alat2)
```

Termasuk polymorphism karena satu fungsi bisa menangani banyak objek berbeda yang punya method dengan nama yang sama mainkan() tanpa perlu tahu classnya.

BAB IV
IMPLEMENTASI

4.1 Tugas Praktikum

4.1.1 Tugas Praktikum No. 1

Buatlah sebuah program dengan konsep *inheritance* dan *method overriding*. Buat class induk bernama BangunDatar yang memiliki *method* luas(). Buat *class* turunan Persegi, Lingkaran, dan Segitiga yang masing-masing mengimplementasikan *method* luas() secara berbeda. Buat sebuah fungsi yang dapat mencetak luas dari masing-masing bangun datar secara *polymorphic*.

4.1.2 Tugas Praktikum No. 2

Gunakan modul abc untuk membuat class abstrak Karyawan dengan *method* abstrak hitung_gaji(). Buat dua class turunan KaryawanTetap dan KaryawanKontrak yang masing-masing mengimplementasikan perhitungan gaji secara berbeda. Buat fungsi yang menerima objek dari class tersebut dan mencetak gaji masing-masing karyawan.

4.2 Source Code

4.2.1 Source Code Soal 1

```
class BangunDatar:
    def __init__(self):
        self.nama = "Bangun Datar"

    def luas(self):
        return 0

class Persegi(BangunDatar):
    def __init__(self, sisi):
        self.nama = "Persegi"
        self.sisi = sisi

    def luas(self):
```

```
        return self.sisi ** 2

class Lingkaran(BangunDatar):
    def __init__(self, jari_jari):
        self.nama = "Lingkaran"
        self.jari_jari = jari_jari

    def luas(self):
        return 3.14 * self.jari_jari ** 2

class Segitiga(BangunDatar):
    def __init__(self, alas, tinggi):
        self.nama = "Segitiga"
        self.alas = alas
        self.tinggi = tinggi

    def luas(self):
        return 0.5 * self.alas * self.tinggi

def cetak_luas(bangun_datar):
    print(f"Luas {bangun_datar.nama}: {bangun_datar.luas()}")

while True:
    print("\nPilih bangun datar:")
    print("1. Persegi")
    print("2. Lingkaran")
    print("3. Segitiga")
    print("4. Keluar")

    pilihan = input("Masukkan pilihan (1-4): ")

    if pilihan == "1":
```

```
        sisi = float(input("Masukkan sisi persegi: "))
        bangun = Persegi(sisi)
    elif pilihan == "2":
        jari = float(input("Masukkan jari-jari lingkaran: "))
        bangun = Lingkaran(jari)
    elif pilihan == "3":
        alas = float(input("Masukkan alas segitiga: "))
        tinggi = float(input("Masukkan tinggi segitiga: "))
        bangun = Segitiga(alas, tinggi)
    elif pilihan == "4":
        print("Program selesai.")
        break
    else:
        print("Pilihan tidak valid.")
        continue

    cetak_luas(bangun)
```

4.2.2 Source Code Soal 2

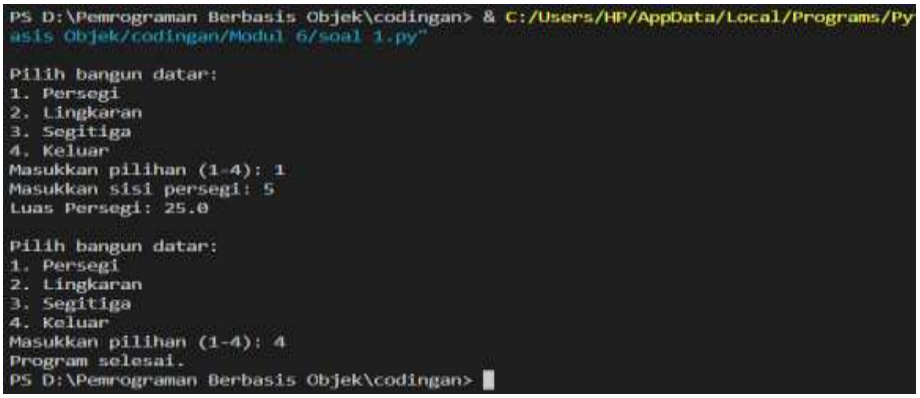
```
from abc import ABC, abstractmethod

class Karyawan(ABC):
    def __init__(self, nama):
        self.nama = nama

    @abstractmethod
    def hitung_gaji(self):
        pass

class KaryawanTetap(Karyawan):
    def __init__(self, nama, gaji_bulanan):
        super().__init__(nama)
```

```
        self.gaji_bulanan = gaji_bulanan

    def hitung_gaji(self):
        return self.gaji_bulanan

class KaryawanKontrak(Karyawan):
    def __init__(self, nama, jam_kerja, upah_per_jam):
        super().__init__(nama)
        self.jam_kerja = jam_kerja
        self.upah_per_jam = upah_per_jam

    def hitung_gaji(self):
        return self.jam_kerja * self.upah_per_jam

def cetak_gaji(karyawan):
    print(f"Gaji {karyawan.nama}: Rp {karyawan.hitung_gaji()}")

while True:
    print("\nPilih jenis karyawan:")
    print("1. Karyawan Tetap")
    print("2. Karyawan Kontrak")
    print("3. Keluar")

    pilihan = int(input("Masukkan pilihan (1/2/3): "))

    if pilihan == 1 :
        nama = input("Nama karyawan: ")
        gaji = float(input("Gaji bulanan: Rp "))
        karyawan = KaryawanTetap(nama, gaji)
        cetak_gaji(karyawan)

    elif pilihan == 2 :
```

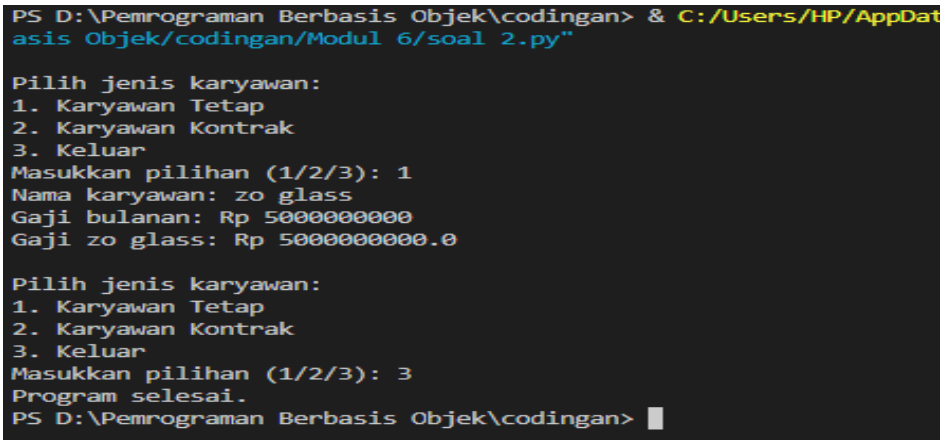
```
        nama = input("Nama karyawan: ")
        jam = float(input("Jumlah jam kerja: "))
        upah = float(input("Upah per jam: Rp "))
        karyawan = KaryawanKontrak(nama, jam, upah)
        cetak_gaji(karyawan)

    elif pilihan == 3 :
        print("Program selesai.")
        break

    else:
        print("Pilihan tidak valid. Coba lagi.")
```

4.3 Hasil

4.3.1 Hasil Soal 1



4.3.2 Hasil Soal 2



4.4 Penjelasan

4.4.1 Penjelasan Soal 1

Kode yang saya buat di atas merupakan implementasi dari konsep *abstraksi* dan *pewarisan* (*inheritance*) dalam pemrograman berorientasi objek (*OOP*) menggunakan bahasa Python. Kode ini dimulai dengan mendefinisikan kelas *abstrak* bernama `Manusia` yang diturunkan dari `ABC` (*Abstract Base Class*). Di dalamnya terdapat tiga metode *abstrak*, yaitu `berbicara()`, `bekerja()`, dan `makan()`, yang belum memiliki implementasi dan harus diimplementasikan oleh kelas turunannya. Selanjutnya, terdapat empat kelas turunan yaitu `Joko`, `Beni`, `Fani`, dan `Jani`, yang masing-masing mewarisi dari kelas `Manusia` dan memberikan implementasi *konkret* untuk ketiga metode tersebut sesuai dengan karakter masing-masing. Di bagian akhir kode, objek dari masing-masing kelas dibuat dan metode-metode yang telah diimplementasikan dipanggil untuk menampilkan aktivitas seperti perkenalan, pekerjaan, dan kegiatan makan.

.

4.4.2 Penjelasan Soal 2

Kode yang saya buat di atas mendefinisikan sebuah *class* abstrak bernama PerangkatElektronik yang memiliki atribut energi_tersisa dengan nilai awal 100 dan atribut tipe untuk menyimpan jenis perangkat, serta tiga *method* abstrak yaitu nyalakan(), matikan(), dan gunakan(jam: int) yang harus diimplementasikan oleh setiap *subclass*. *Method status*() digunakan untuk menampilkan tipe perangkat dan sisa energinya. Dua subclass yaitu Laptop dan Kulkas mewarisi *class* tersebut dan memberikan implementasi spesifik: Laptop mengurangi energi sebesar 10% dikali jumlah jam saat digunakan, dan akan menampilkan pesan baterai habis jika energinya turun di bawah 0, sedangkan Kulkas mengurangi energi sebesar 5% per jam dan akan memberi peringatan jika energi turun di bawah 20%. Nilai tipe diberikan secara manual melalui *constructor* masing-masing subclass saat memanggil super().__init__(), sehingga *method* status() dapat menampilkan nama perangkat dengan mudah. Di akhir kode, objek Laptop dan Kulkas dibuat dan seluruh *method*nya dipanggil dengan durasi penggunaan berbeda untuk menunjukkan bagaimana perubahan energi terjadi dan bagaimana pesan ditampilkan sesuai kondisi energinya.

BAB V
PENUTUP

5.1 Analisa

Polimorfisme dalam *object-oriented programming* (*OOP*) adalah konsep yang memberikan kemampuan pada sebuah *method* atau *object* untuk berperilaku berbeda tergantung pada konteks penggunaannya. Konsep ini mendukung prinsip *abstraction* dan *inheritance* dengan cara menggunakan satu *interface* atau *class* dasar untuk berbagai bentuk perilaku dalam turunan atau implementasinya. Terdapat dua jenis utama, yaitu *compile-time polymorphism* seperti *method overloading* dan *runtime polymorphism* seperti *method overriding*. Polimorfisme digunakan untuk menyederhanakan struktur kode, meningkatkan keterbacaan, dan memudahkan pengembangan sistem tanpa perlu mengubah bagian program yang sudah ada.

5.2 Kesimpulan

Secara keseluruhan, polimorfisme merupakan konsep kunci dalam *object-oriented programming* yang berperan penting dalam menciptakan kode yang fleksibel, efisien, dan mudah dikembangkan. Dengan adanya polimorfisme, proses pengembangan perangkat lunak menjadi lebih terstruktur karena objek-objek dari kelas yang berbeda dapat merespons terhadap pemanggilan fungsi yang sama dengan cara yang sesuai. Hal ini sangat membantu dalam membuat sistem yang kompleks tetap dapat dikelola dengan baik. Pemahaman mendalam terhadap konsep ini akan sangat berguna bagi programmer untuk menulis kode yang bersih, terorganisir, dan siap dikembangkan lebih lanjut di masa mendatang.

1. Mendukung *method overloading* dan *method overriding* sebagai bentuk *compile-time* dan *runtime polymorphism*.
2. Memperkuat prinsip dasar *OOP* seperti *abstraction* dan *inheritance*.
3. Mempermudah pengembangan dan pemeliharaan kode melalui struktur yang lebih rapi dan konsisten.