**PEMROGRAMAN BERORIENTASI OBJEK**

**JOB SHEET 5 : POLIMORFISME DAN MEMAHAMI OVERLOADING/OVERRIDING**



|  |  |
| --- | --- |
| **Mata Kuliah** | **: Pemrograman Berorientasi Objek** |
| **Dosen** | **: Prayitno, S.ST., M.T., Ph.D** |
|  |  |
| **Disusun Oleh** | |
| **Nama** | **: Vian Maulana Ramadhan** |
| **NIM** | **: 4.33.24.0.26** |
| **Kelas** | **: TI-1A** |

**PROGRAM STUDI D4 – TEKNOLOGI REKAYASA KOMPUTER**

**JURUSAN TEKNIK ELEKTRO**

**POLITEKNIK NEGERI SEMARANG**

**2024/2025**

**A. Tujuan Instruksional Khusus**

Setelah menyelesaikan praktikum ini, mahasiswa diharapkan mampu:

1. Menjelaskan konsep polimorfisme (polymorphism) dalam Pemrograman Berorientasi Objek.
2. Mengimplementasikan polimorfisme melalui pewarisan (inheritance) dan penggantian metode (method overriding).
3. Memahami konsep "duck typing" sebagai bentuk polimorfisme dalam Python.
4. Membedakan antara method overriding (yang didukung Python) dan method overloading (yang tidak didukung secara langsung seperti di bahasa lain).
5. Mengimplementasikan perilaku yang menyerupai overloading menggunakan default arguments, \*args, dan \*\*kwargs di Python.

**B. Alat dan Bahan**

**Perangkat Lunak**:

* Sistem operasi Windows/Linux/macOS (sesuai ketersediaan).
* Python 3.x (versi terbaru yang stabil).
* IDE atau Code Editor (misalnya, PyCharm, VS Code, atau IDLE). **Perangkat Keras**:
* Komputer/Laptop dengan spesifikasi memadai untuk menjalankan Python. **Bahan Pendukung**:
* Modul/slide perkuliahan yang menjelaskan dasar pemrograman Python.
* Koneksi internet (opsional, untuk referensi tambahan).

**C. Dasar Teori**

**1. Polimorfisme (Polymorphism)**

Polimorfisme berasal dari bahasa Yunani yang berarti "banyak bentuk" (poly = banyak, morph = bentuk). Dalam konteks OOP, polimorfisme merujuk pada kemampuan objek dari kelas yang berbeda untuk merespons terhadap pemanggilan metode yang sama (dengan nama yang sama) dengan cara yang spesifik sesuai dengan tipe atau kelas objek tersebut. Ini memungkinkan kita menulis kode yang lebih generik dan fleksibel, yang dapat beroperasi pada berbagai jenis objek tanpa perlu mengetahui tipe persisnya di muka, asalkan objek-objek tersebut mendukung antarmuka (metode) yang diharapkan.

* **Polimorfisme via Inheritance dan Overriding:** Cara paling umum untuk mencapai polimorfisme dalam bahasa berbasis kelas seperti Python adalah melalui pewarisan. Kelas induk mendefinisikan metode umum, dan kelas anak meng-override metode tersebut untuk menyediakan implementasi spesifik. Kode yang memanggil metode ini pada objek induk dapat secara otomatis menjalankan versi metode yang sesuai dari objek anak yang sebenarnya direferensikan. Contoh: Fungsi yang memanggil metode terbang() pada objek Burung akan menjalankan implementasi terbang() yang spesifik untuk Elang atau Pipit jika objek tersebut adalah instance dari kelas anak tersebut.
* **Polimorfisme via Duck Typing:** Python adalah bahasa yang diketik secara dinamis (*dynamically typed*), yang seringkali mengandalkan konsep "Duck Typing". Istilah ini berasal dari ungkapan "Jika berjalan seperti bebek dan bersuara seperti bebek, maka itu mungkin bebek." Artinya, Python tidak terlalu peduli dengan *tipe* objek secara eksplisit, tetapi lebih pada apakah objek tersebut memiliki *metode atau atribut yang diperlukan* untuk melakukan suatu operasi. Sebuah fungsi dapat bekerja dengan objek apa pun yang menyediakan metode yang dibutuhkan, terlepas dari kelas atau hierarki pewarisannya.

**2. Method Overriding vs. Method Overloading**

* **Method Overriding:** Seperti yang dibahas dalam materi Inheritance, overriding terjadi ketika kelas anak menyediakan implementasi ulang (definisi baru) untuk metode yang sudah ada di kelas induknya dengan nama dan signatur (parameter) yang sama. Ini adalah mekanisme kunci untuk polimorfisme dalam hierarki pewarisan. Python sepenuhnya mendukung method overriding.
* **Method Overloading:** Overloading merujuk pada kemampuan untuk mendefinisikan beberapa fungsi atau metode dengan *nama yang sama* dalam lingkup yang sama, tetapi dengan *daftar parameter yang berbeda* (berbeda dalam jumlah atau tipe parameter). Bahasa seperti Java atau C++ menggunakan ini untuk memungkinkan, misalnya, fungsi tambah(int a, int b) dan tambah(float a, float b) ada bersamaan. **Python tidak mendukung method overloading tradisional seperti ini.** Jika Anda mendefinisikan dua fungsi atau metode dengan nama yang sama, definisi terakhir akan menimpa (overwrite) definisi sebelumnya.

**3. Menyimulasikan Overloading di Python**

Meskipun tidak ada overloading bawaan berdasarkan signatur, Python menyediakan cara lain untuk mencapai fleksibilitas serupa dalam menerima argumen:

* **Nilai Parameter Default:** Memberikan nilai default pada parameter memungkinkan fungsi dipanggil dengan jumlah argumen yang lebih sedikit.

Python def sapa(nama, pesan="Halo"): print(f"{pesan}, {nama}!") sapa("Andi") # Output: Halo, Andi!

sapa("Budi", "Selamat Pagi") # Output: Selamat Pagi, Budi!

* **Argumen Posisi Variabel (\*args):** Mengizinkan fungsi menerima sejumlah argumen posisi yang tidak ditentukan. Argumen-argumen ini dikumpulkan ke dalam sebuah *tuple*.

|  |
| --- |
| Python def jumlahkan(\*angka):  total = 0 for n in angka: total += n return total  print(jumlahkan(1, 2, 3)) # Output: 6 |

print(jumlahkan(5, 10)) # Output: 15

* **Argumen Kata Kunci Variabel (\*\*kwargs):** Mengizinkan fungsi menerima sejumlah argumen kata kunci (keyword arguments) yang tidak ditentukan. Argumen-argumen ini dikumpulkan ke dalam sebuah *dictionary*.

|  |
| --- |
| Python def cetak\_info(\*\*data):  for kunci, nilai in data.items(): print(f"{kunci}: {nilai}")  cetak\_info(nama="Citra", usia=25, kota="Semarang") |

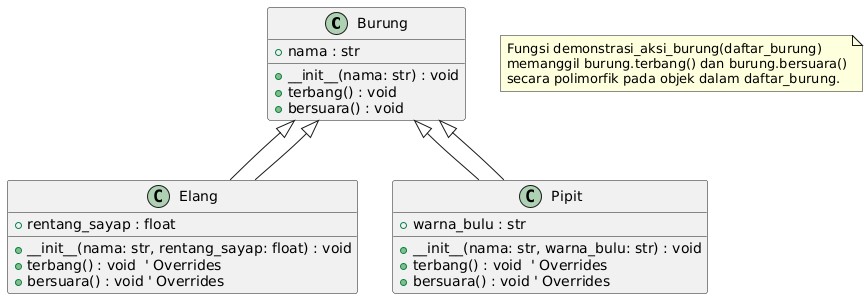
* **Pemeriksaan Tipe/Jumlah di Dalam Fungsi:** Anda dapat memeriksa tipe atau jumlah argumen yang diterima (menggunakan isinstance(), len(args), dll.) di dalam fungsi dan menjalankan logika yang berbeda sesuai kondisi.

Dengan menggunakan teknik-teknik ini, kita dapat membuat fungsi dan metode Python yang fleksibel dalam menangani berbagai jenis atau jumlah input, meskipun tidak melalui mekanisme overloading berbasis signatur seperti pada bahasa lain.

**D. Langkah Praktikum**

**Praktikum 01: Polimorfisme dengan Inheritance dan Overriding**

Tujuan: Mengamati bagaimana objek dari kelas anak yang berbeda merespons pemanggilan metode yang sama (yang di-override) dengan cara yang berbeda, menunjukkan konsep polimorfisme.



**Penjelasan Kelas Diagram:**

* **Kelas Burung:**
  + **Atribut Publik (+):** nama.
  + **Metode Publik (+):** \_\_init\_\_(nama), terbang() (versi umum), bersuara() (versi umum).
* **Kelas Elang (Mewarisi dari Burung):**
  + **Atribut Publik (+):** rentang\_sayap (Atribut nama diwarisi). o **Metode Publik (+):** \_\_init\_\_(nama, rentang\_sayap) (memanggil super), terbang() (override), bersuara() (override).
* **Kelas Pipit (Mewarisi dari Burung):**
  + **Atribut Publik (+):** warna\_bulu (Atribut nama diwarisi). o **Metode Publik (+):** \_\_init\_\_(nama, warna\_bulu) (memanggil super), terbang() (override), bersuara() (override).

Diagram ini menunjukkan hierarki kelas dimana Elang dan Pipit mewarisi dari Burung. Kedua kelas anak meng-override metode terbang() dan bersuara() dari kelas induknya. Fungsi demonstrasi\_aksi\_burung (tidak digambarkan dalam diagram kelas tapi dijelaskan dalam catatan) dapat menerima daftar objek Burung (termasuk instance Elang dan Pipit). Ketika metode terbang() atau bersuara() dipanggil pada elemen dalam daftar tersebut, Python secara otomatis akan menjalankan versi metode yang sesuai dengan tipe objek aktual (versi Elang jika objeknya Elang, versi Pipit jika objeknya Pipit, atau versi Burung jika objeknya Burung), inilah inti dari polimorfisme.

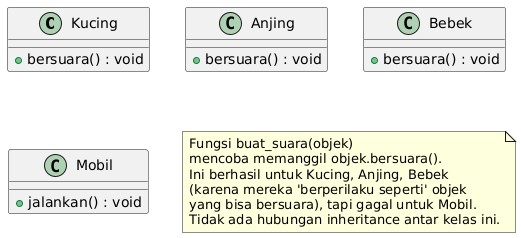
Kemudian untuk kode praktikum dalam python dapat dilihat sebagai berikut:

|  |
| --- |
| 01: # Kelas Induk 02: class Burung:  03: def \_\_init\_\_(self, nama): 04: self.nama = nama  05:  06: def terbang(self):  07: print(f"{self.nama} terbang dengan cara umum.")  08:  09: def bersuara(self):  10: print(f"{self.nama} mengeluarkan suara burung.") |
| 11:  12: # Kelas Anak 1  13: class Elang(Burung):  14: def \_\_init\_\_(self, nama, rentang\_sayap):  15: super().\_\_init\_\_(nama)  16: self.rentang\_sayap = rentang\_sayap  17:  18: # Override  19: def terbang(self):  20: print(f"{self.nama} terbang tinggi melayang di angkasa.")  21:  22: # Override  23: def bersuara(self):  24: print(f"{self.nama} berteriak nyaring!")  25:  26: # Kelas Anak 2  27: class Pipit(Burung):  28: def \_\_init\_\_(self, nama, warna\_bulu):  29: super().\_\_init\_\_(nama)  30: self.warna\_bulu = warna\_bulu  31:  32: # Override  33: def terbang(self):  34: print(f"{self.nama} terbang cepat di antara pepohonan.")  35:  36: # Override  37: def bersuara(self):  38: print(f"{self.nama} berkicau merdu: Cit cit!")  39:  40: # Fungsi yang memanfaatkan polimorfisme 41: def demonstrasi\_aksi\_burung(daftar\_burung):  42: print("\nAksi Burung:") 43: for burung in daftar\_burung:  44: print(f"-- Aksi untuk {burung.nama} --")  45: # Panggilan metode yang sama, tapi perilaku berbeda tergantung objek  46: burung.terbang()  47: burung.bersuara()  48: print("-" \* 15)  49:  50: # --- Kode Utama --- 51: if \_\_name\_\_ == "\_\_main\_\_":  52: elang\_jawa = Elang("Elang Jawa", 1.5)  53: pipit\_gereja = Pipit("Pipit Gereja", "Coklat")  54: burung\_aneh = Burung("Burung Misterius") # Objek dari kelas induk  55:  56: koleksi\_burung = [elang\_jawa, pipit\_gereja, burung\_aneh]  57:  58: demonstrasi\_aksi\_burung(koleksi\_burung) |

**Observasi:** Perhatikan bagaimana pemanggilan burung.terbang() dan burung.bersuara() di dalam fungsi demonstrasi\_aksi\_burung menghasilkan output yang berbeda-beda tergantung pada tipe objek aktual (Elang, Pipit, atau Burung) yang sedang diproses dalam loop. Ini adalah contoh polimorfisme melalui overriding metode.

**Praktikum 02: Polimorfisme dengan Duck Typing**

**Tujuan:** Menunjukkan bagaimana Python dapat menerapkan polimorfisme pada objek yang tidak terikat oleh hierarki pewarisan yang sama, selama objek tersebut memiliki metode yang dibutuhkan ("duck typing").



**Penjelasan Kelas Diagram:**

* **Kelas Kucing, Anjing, Bebek:** Masing-masing adalah kelas independen yang kebetulan memiliki metode publik bersuara() dengan nama yang sama.
* **Kelas Mobil:** Kelas independen yang memiliki metode jalankan() tetapi tidak memiliki metode bersuara().
* **Catatan (Note):** Menjelaskan inti dari praktikum ini. Fungsi eksternal buat\_suara tidak memeriksa tipe kelas secara eksplisit. Ia hanya mencoba memanggil metode bersuara() pada objek apa pun yang diterimanya. Jika objek tersebut "berperilaku seperti" sesuatu yang bisa bersuara (yaitu, memiliki metode bersuara()), pemanggilan berhasil. Jika tidak (seperti pada objek Mobil), terjadi error (yang ditangani try-except dalam kode).

Diagram ini secara visual menunjukkan bahwa kelas-kelas tersebut tidak terhubung melalui pewarisan. Polimorfisme di sini terjadi bukan karena hierarki kelas, melainkan karena kesamaan *interface* (adanya metode bersuara() yang dapat dipanggil), yang merupakan ciri khas dari duck typing di Python.

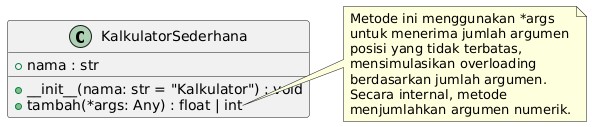
Kode Python:

|  |
| --- |
| 01: class Kucing:  02: def bersuara(self): 03: print("Kucing: Meow!")  04:  05: class Anjing:  06: def bersuara(self):  07: print("Anjing: Guk guk!")  08:  09: class Bebek:  10: def bersuara(self):  11: print("Bebek: Kwek kwek!")  12:  13: class Mobil: # Kelas ini TIDAK punya metode bersuara()  14: def jalankan(self): 15: print("Mobil: Brummm!")  16:  17: # Fungsi ini tidak peduli tipe objeknya,  18: # asal punya metode .bersuara()  19: def buat\_suara(objek\_yang\_bisa\_bersuara): |
| 20: try:  21: # Memanggil metode 'bersuara' jika ada 22: objek\_yang\_bisa\_bersuara.bersuara()  23: except AttributeError:  24: # Menangani jika objek tidak punya metode 'bersuara'  25: print(f"Objek {type(objek\_yang\_bisa\_bersuara).\_\_name\_\_} tidak bisa bersuara.")  26:  27: # --- Kode Utama --- 28: if \_\_name\_\_ == "\_\_main\_\_":  29: kucing1 = Kucing()  30: anjing1 = Anjing()  31: bebek1 = Bebek()  32: mobil1 = Mobil()  33:  34: daftar\_objek = [kucing1, anjing1, bebek1, mobil1]  35:  36: print("Demonstrasi Duck Typing:")  37: for item in daftar\_objek:  38: buat\_suara(item) # Fungsi polimorfik via duck typing |

**Observasi:** Fungsi buat\_suara berhasil memanggil metode bersuara() pada objek Kucing, Anjing, dan Bebek meskipun mereka tidak mewarisi dari kelas induk yang sama. Fungsi ini hanya peduli apakah objek *memiliki* metode bersuara(). Ketika objek Mobil (yang tidak memiliki metode bersuara()) dimasukkan, fungsi menangani AttributeError dengan baik. Ini menunjukkan fleksibilitas duck typing.

**Praktikum 03: Simulasi Overloading dalam Metode Kelas dengan \*args**

**Tujuan:** Mendemonstrasikan bagaimana sebuah metode di dalam kelas dapat dirancang untuk menerima jumlah argumen yang bervariasi menggunakan \*args, sehingga meniru perilaku overloading.



**Penjelasan Kelas Diagram:**

* **Kelas KalkulatorSederhana:** o **Atribut Publik (+):** nama (string) o **Metode Publik (+):**
  + \_\_init\_\_(nama: str = "Kalkulator") : void: Konstruktor dengan parameter opsional untuk nama kalkulator.
  + tambah(\*args: Any) : float | int: Metode tambah yang menerima sejumlah argumen (\*args). Tipe Any menunjukkan fleksibilitas tipe input, dan tipe return bisa int atau float tergantung hasil penjumlahan.
* **Catatan (Note):** Menjelaskan peran \*args dalam metode tambah untuk menerima jumlah argumen yang bervariasi, yang merupakan cara Python untuk meniru perilaku overloading berdasarkan jumlah parameter.

Diagram ini menunjukkan kelas KalkulatorSederhana dengan metode tambah yang dirancang secara fleksibel menggunakan \*args.

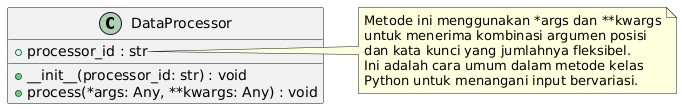
Kode Program

|  |
| --- |
| 01: class KalkulatorSederhana:  02: def \_\_init\_\_(self, nama="Kalkulator"):  03: self.nama = nama  04: print(f"{self.nama} siap digunakan.")  05:  06: def tambah(self, \*args):  07: """  08: Metode ini menjumlahkan semua argumen angka yang diberikan.  09: Menerima sejumlah argumen posisi menggunakan \*args.  10: """  11: print(f"\nMemanggil metode tambah dengan argumen: {args}")  12: if not args:  13: print("Tidak ada angka untuk dijumlahkan.")  14: return 0 # Atau bisa juga None atau raise error  15:  16: total = 0  17: valid\_input = True 18: for angka in args:  19: # Melakukan pengecekan tipe sederhana 20: if isinstance(angka, (int, float)): 21: total += angka 22: else:  23: print(f"Peringatan: Argumen '{angka}' bukan angka dan akan diabaikan.")  24: valid\_input = False  25:  26: if valid\_input:  27: print(f"Hasil penjumlahan: {total}") 28: else:  29: print(f"Hasil penjumlahan (dengan beberapa input diabaikan):  {total}")  30: return total  31:  32: # --- Kode Utama --- 33: if \_\_name\_\_ == "\_\_main\_\_":  34: calc = KalkulatorSederhana("Calc-01")  35:  36: # Memanggil metode 'tambah' dengan jumlah argumen berbeda  37: print("\n--- Percobaan Penjumlahan ---")  38: calc.tambah(5, 10)  39: calc.tambah(2, 3, 5, 10)  40: calc.tambah(100)  41: calc.tambah() # Tanpa argumen  42: calc.tambah(1, 2, "tiga", 4, 5.5) # Dengan input non-angka |

**Observasi:** Perhatikan bagaimana metode tambah dari objek calc dapat dipanggil dengan dua, empat, satu, atau bahkan nol argumen angka. Penggunaan \*args mengumpulkan semua argumen posisi ke dalam sebuah tuple args, yang kemudian diiterasi di dalam metode untuk melakukan penjumlahan. Metode ini juga menunjukkan penanganan sederhana jika input bukan angka atau jika tidak ada argumen yang diberikan. Ini mensimulasikan kemampuan fungsi/metode untuk bekerja dengan jumlah input yang berbeda.

**Praktikum 04: Simulasi Overloading dalam Metode Kelas dengan \*args dan \*\*kwargs**

**Tujuan:** Menggunakan \*args dan \*\*kwargs di dalam sebuah metode kelas untuk menerima jumlah argumen posisi dan kata kunci yang fleksibel, meniru perilaku metode yang bisa dipanggil dengan cara berbeda.



**Penjelasan Kelas Diagram:**

* **Kelas DataProcessor:** o **Atribut Publik (+):** processor\_id (string).

o **Metode Publik (+):**

* + \_\_init\_\_(processor\_id: str) : void: Konstruktor untuk inisialisasi ID prosesor.
  + process(\*args: Any, \*\*kwargs: Any) : void: Metode process yang menerima self (implisit dalam diagram), sejumlah argumen posisi (\*args), dan sejumlah argumen kata kunci (\*\*kwargs).
* **Catatan (Note):** Menjelaskan fungsi \*args dan \*\*kwargs dalam konteks metode process untuk memberikan fleksibilitas dalam menerima argumen, sebuah teknik yang digunakan untuk tujuan serupa dengan overloading di bahasa lain.

Diagram ini menunjukkan kelas DataProcessor dan metode process yang dirancang untuk menangani berbagai jenis pemanggilan argumen menggunakan \*args dan \*\*kwargs.

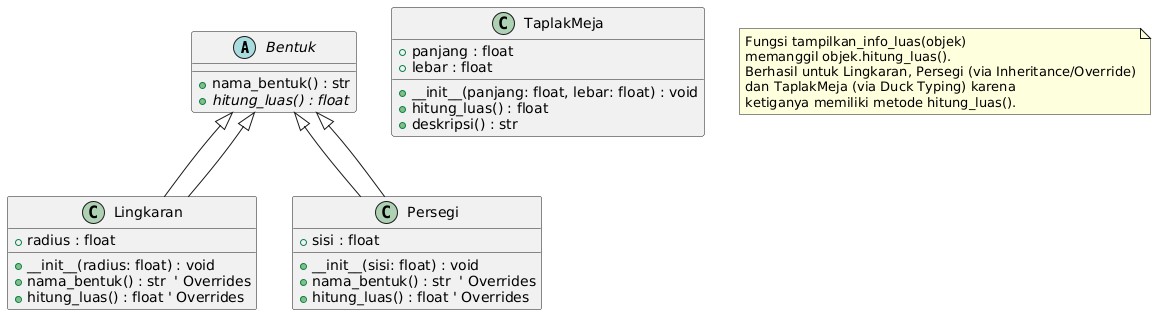
Kode Program:

|  |
| --- |
| 01: class DataProcessor:  02: def \_\_init\_\_(self, processor\_id):  03: self.processor\_id = processor\_id  04: print(f"Data Processor {self.processor\_id} siap.")  05:  06: def process(self, \*args, \*\*kwargs):  07: """  08: Metode ini memproses data dengan argumen posisi (\*args)  09: dan kata kunci (\*\*kwargs) yang jumlahnya variabel. 10: """  11: print(f"\n--- {self.processor\_id} Memproses Data ---")  12:  13: # Memproses argumen posisi (\*args -> tuple)  14: if args:  15: print("Argumen Posisi Diterima:") 16: for i, arg in enumerate(args):  17: print(f" args[{i}]: {arg} (tipe: {type(arg).\_\_name\_\_})") 18: else:  19: print("Tidak ada argumen posisi.")  20:  21: # Memproses argumen kata kunci (\*\*kwargs -> dict)  22: if kwargs:  23: print("Argumen Kata Kunci Diterima:") 24: for kunci, nilai in kwargs.items():  25: print(f" {kunci}: {nilai} (tipe: {type(nilai).\_\_name\_\_})") 26: else:  27: print("Tidak ada argumen kata kunci.") 28: print("---------------------------------")  29:  30: # --- Kode Utama --- |
| 31: if \_\_name\_\_ == "\_\_main\_\_":  32: # Membuat instance dari kelas  33: processor1 = DataProcessor("DP-001")  34:  35: # Memanggil metode 'process' dengan berbagai cara  36: print("\nPanggilan 1: Tanpa argumen tambahan") 37: processor1.process()  38:  39: print("\nPanggilan 2: Hanya argumen posisi")  40: processor1.process(100, "Status OK", 99.9, False)  41:  42: print("\nPanggilan 3: Hanya argumen kata kunci")  43: processor1.process(user="admin", level=5, mode="verbose")  44:  45: print("\nPanggilan 4: Kombinasi argumen posisi dan kata kunci")  46: processor1.process("Task-A", "Task-B", status="Running", priority="High", thread\_id=54321)  47:  48: # Panggilan dengan list dan dictionary sebagai argumen posisi  49: print("\nPanggilan 5: Argumen posisi kompleks")  50: list\_ids = ["A1", "B2", "C3"]  51: dict\_params = {"timeout": 30, "retry": 3}  52: processor1.process(list\_ids, dict\_params, owner="system", enabled=True) |

**Observasi:** Perhatikan bagaimana metode process milik objek processor1 dapat dipanggil dengan berbagai kombinasi argumen. \*args dan \*\*kwargs dalam definisi metode process(self, \*args, \*\*kwargs) memungkinkan fleksibilitas ini. self tetap menjadi parameter pertama yang merujuk pada instance objek, diikuti oleh \*args untuk argumen posisi dan \*\*kwargs untuk argumen kata kunci.

**Praktikum 05: Kombinasi Polimorfisme (Inheritance & Duck Typing)**

**Tujuan:** Menunjukkan bagaimana sebuah fungsi dapat bekerja secara polimorfik dengan objek-objek dari hierarki inheritance yang sama *dan* objek dari kelas lain yang tidak terkait, selama semuanya menyediakan metode yang dibutuhkan.



**Penjelasan Kelas Diagram:** • **Kelas Bentuk (Abstrak):**

o Kelas induk yang mendefinisikan interface umum. Metode hitung\_luas() ditandai {abstract} untuk menunjukkan bahwa ia harus diimplementasikan oleh subclass.

* **Kelas Lingkaran dan Persegi:**
  + Mewarisi dari Bentuk. o Masing-masing memiliki atribut spesifik (radius, sisi). o Meng-override metode nama\_bentuk() dan hitung\_luas() dari kelas induk.
* **Kelas TaplakMeja:**
  + Kelas yang **tidak** mewarisi dari Bentuk. o Memiliki atribut panjang, lebar. o Memiliki metode hitung\_luas() yang **kebetulan namanya sama** dengan metode di Bentuk. o Memiliki metode lain (deskripsi) yang tidak ada di Bentuk.

**Catatan (Note):** Menjelaskan bagaimana fungsi tampilkan\_info\_luas berinteraksi dengan objekobjek ini. Ia bekerja pada Lingkaran dan Persegi karena mereka adalah turunan Bentuk (polimorfisme via inheritance). Ia juga bekerja pada TaplakMeja karena TaplakMeja memiliki metode hitung\_luas() yang dibutuhkan (polimorfisme via duck typing), meskipun tidak ada hubungan inheritance formal.

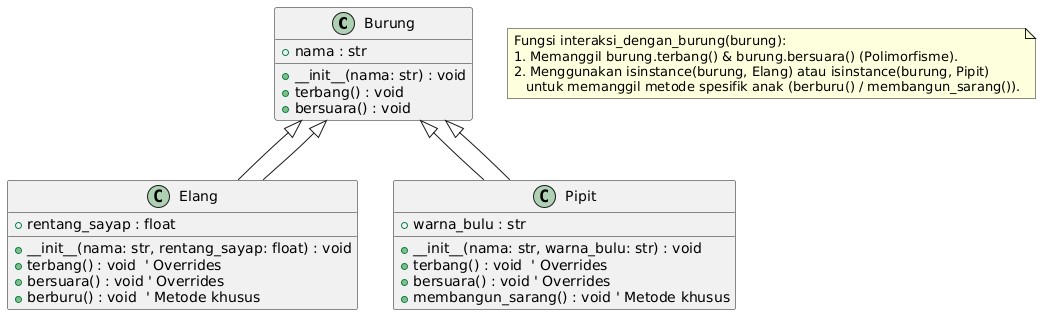
Kode Program:

|  |
| --- |
| 01: import math  02:  03: # --- Definisi Kelas dari Hierarki Inheritance ---  04: class Bentuk:  05: # Metode ini bisa dianggap sebagai 'template' atau interface dasar  06: def nama\_bentuk(self): 07: return "Bentuk Generik"  08:  09: # Metode ini wajib di-override oleh subclass 10: def hitung\_luas(self):  11: raise NotImplementedError("Subclass harus mengimplementasikan metode ini")  12:  13: class Lingkaran(Bentuk): 14: def \_\_init\_\_(self, radius):  15: self.radius = radius  16:  17: # Override  18: def nama\_bentuk(self):  19: return "Lingkaran"  20:  21: # Override  22: def hitung\_luas(self):  23: return math.pi \* (self.radius \*\* 2)  24:  25: class Persegi(Bentuk): 26: def \_\_init\_\_(self, sisi): 27: self.sisi = sisi  28:  29: # Override  30: def nama\_bentuk(self):  31: return "Persegi"  32:  33: # Override  34: def hitung\_luas(self):  35: return self.sisi \* self.sisi  36:  37: # --- Definisi Kelas Lain (Tidak Terkait Inheritance dengan Bentuk) ---  38: class TaplakMeja:  39: def \_\_init\_\_(self, panjang, lebar):  40: self.panjang = panjang  41: self.lebar = lebar  42:  43: # Metode dengan nama sama -> 'hitung\_luas' |
| 44: # Ini memungkinkan TaplakMeja bekerja dengan fungsi polimorfik di bawah  45: def hitung\_luas(self):  46: return self.panjang \* self.lebar  47:  48: # Metode yang berbeda, tidak ada di kelas Bentuk  49: def deskripsi(self):  50: return f"Taplak Meja {self.panjang}x{self.lebar}"  51:  52: # --- Fungsi Polimorfik ---  53: def tampilkan\_info\_luas(objek\_dengan\_luas): 54: print("-" \* 20)  55: try:  56: # Mencoba memanggil hitung\_luas() - inti polimorfisme  57: luas = objek\_dengan\_luas.hitung\_luas()  58:  59: # Mencoba mendapatkan nama jika ada (duck typing tambahan)  60: try:  61: nama = objek\_dengan\_luas.nama\_bentuk()  62: except AttributeError:  63: # Jika tidak ada nama\_bentuk(), gunakan nama kelasnya  64: nama = type(objek\_dengan\_luas).\_\_name\_\_  65:  66: print(f"Objek: {nama}")  67: print(f"Luas : {luas:.2f}") # Format luas 2 angka desimal  68:  69: except AttributeError:  70: # Menangani jika objek sama sekali tidak punya .hitung\_luas() 71: print(f"Objek {type(objek\_dengan\_luas).\_\_name\_\_} tidak dapat dihitung luasnya (metode tidak ditemukan).") 72: except NotImplementedError:  73: # Menangani jika metode ada tapi belum diimplementasi (dari kelas Bentuk)  74: print(f"Metode 'hitung\_luas' belum diimplementasi untuk  {type(objek\_dengan\_luas).\_\_name\_\_}.")  75:  76:  77: # --- Kode Utama --- 78: if \_\_name\_\_ == "\_\_main\_\_":  79: lingkaran1 = Lingkaran(7)  80: persegi1 = Persegi(5)  81: taplak1 = TaplakMeja(1.5, 0.8)  82: bentuk\_dasar = Bentuk() # Objek kelas induk (akan raise NotImplementedError)  83: string\_biasa = "Ini string" # Objek yang tidak relevan 84:  85: daftar\_item = [lingkaran1, persegi1, taplak1, bentuk\_dasar, string\_biasa]  86:  87: print("Menampilkan Info Luas (Polimorfisme Campuran):")  88: for item in daftar\_item:  89: tampilkan\_info\_luas(item) |

**Observasi:** Fungsi tampilkan\_info\_luas bekerja dengan Lingkaran dan Persegi karena mereka mewarisi dari Bentuk dan mengimplementasikan hitung\_luas(). Fungsi ini juga bekerja dengan TaplakMeja karena, meskipun tidak terkait inheritance, ia memiliki metode hitung\_luas() (duck typing). Fungsi ini juga dapat menangani objek Bentuk (yang metodenya belum diimplementasi) dan objek string\_biasa (yang tidak memiliki metode yang relevan) dengan try-except.

**Praktikum 06: Kontrol Perilaku Polimorfik dengan isinstance()**

Tujuan: Menggunakan isinstance() di dalam fungsi yang bekerja secara polimorfik untuk menambahkan perilaku khusus berdasarkan tipe objek, selain memanggil metode yang dioverride.



**Penjelasan Kelas Diagram:**

* **Kelas Burung, Elang, Pipit:** Struktur hierarki sama seperti Praktikum 1, namun kelas Elang dan Pipit sekarang memiliki metode tambahan yang spesifik untuk mereka (berburu() dan membangun\_sarang()).
* **Catatan (Note):** Menjelaskan bagaimana fungsi interaksi\_dengan\_burung (yang berada di luar kelas) menggunakan dua pendekatan:
  1. Memanfaatkan polimorfisme dengan memanggil metode terbang() dan bersuara() yang perilakunya bergantung pada tipe objek aktual.
  2. Menggunakan isinstance() untuk memeriksa tipe spesifik objek (Elang atau Pipit) agar dapat memanggil metode unik yang hanya dimiliki oleh subclass tersebut.

Diagram ini menunjukkan struktur kelas yang memungkinkan polimorfisme, dan catatan menjelaskan bagaimana logika eksternal dapat menggunakan isinstance() untuk berinteraksi secara lebih spesifik dengan objek dalam hierarki tersebut.

Kode Program:

|  |
| --- |
| 01: # --- Definisi Kelas Burung, Elang, Pipit ---  02: class Burung:  03: def \_\_init\_\_(self, nama):  04: self.nama = nama 05: def terbang(self):  06: print(f"{self.nama} terbang dengan cara umum.")  07: def bersuara(self):  08: print(f"{self.nama} mengeluarkan suara burung.")  09:  10: class Elang(Burung):  11: def \_\_init\_\_(self, nama, rentang\_sayap): 12: super().\_\_init\_\_(nama)  13: self.rentang\_sayap = rentang\_sayap  14: # Override  15: def terbang(self):  16: print(f"{self.nama} terbang tinggi melayang di angkasa.")  17: # Override  18: def bersuara(self):  19: print(f"{self.nama} berteriak nyaring!") 20: # Metode khusus Elang 21: def berburu(self):  22: print(f"{self.nama} sedang mencari mangsa dari ketinggian.")  23:  24: class Pipit(Burung): |
| 25: def \_\_init\_\_(self, nama, warna\_bulu):  26: super().\_\_init\_\_(nama)  27: self.warna\_bulu = warna\_bulu  28: # Override  29: def terbang(self):  30: print(f"{self.nama} terbang cepat di antara pepohonan.")  31: # Override  32: def bersuara(self):  33: print(f"{self.nama} berkicau merdu: Cit cit!")  34: # Metode khusus Pipit 35: def membangun\_sarang(self):  36: print(f"{self.nama} sedang mengumpulkan ranting untuk sarang.")  37:  38: # --- Fungsi yang memanfaatkan polimorfisme DAN isinstance() --- 39: def interaksi\_dengan\_burung(burung):  40: print(f"\n--- Berinteraksi dengan {type(burung).\_\_name\_\_}:  {getattr(burung, 'nama', 'Objek tidak dikenal')} ---")  41:  42: # Cek dulu apakah objek adalah instance dari Burung atau turunannya  43: if isinstance(burung, Burung):  44: # Perilaku polimorfik dasar (memanggil metode override)  45: burung.terbang()  46: burung.bersuara()  47:  48: # Menambahkan perilaku spesifik berdasarkan tipe turunan  49: if isinstance(burung, Elang):  50: print("-> Ini adalah Elang!")  51: burung.berburu() # Panggil metode khusus Elang  52: elif isinstance(burung, Pipit):  53: print("-> Ini adalah Pipit!")  54: burung.membangun\_sarang() # Panggil metode khusus Pipit 55: else:  56: # Hanya Burung generik, bukan turunan spesifik yang dikenali 57: print("-> Ini adalah burung jenis umum (bukan Elang/Pipit).") 58: else:  59: # Jika objek BUKAN instance Burung sama sekali  60: print("-> Objek ini bukan termasuk jenis Burung.")  61: print("-" \* 25)  62:  63: # --- Kode Utama --- 64: if \_\_name\_\_ == "\_\_main\_\_":  65: elang\_sumatra = Elang("Elang Sumatra", 1.8)  66: pipit\_rumah = Pipit("Pipit Rumah", "Abu-abu")  67: merak = Burung("Merak") # Contoh Burung lain (induk)  68: kucing\_tetangga = "Meong" # Objek non-burung 69:  70: koleksi\_makhluk = [elang\_sumatra, pipit\_rumah, merak, kucing\_tetangga]  71:  72: for makhluk in koleksi\_makhluk:  73: interaksi\_dengan\_burung(makhluk) |

**Observasi:** Fungsi interaksi\_dengan\_burung pertama-tama memeriksa apakah objek merupakan Burung. Jika ya, ia memanggil metode terbang() dan bersuara() secara polimorfik. Kemudian, ia menggunakan isinstance() lagi untuk pemeriksaan yang lebih spesifik (Elang atau Pipit) dan memicu perilaku tambahan (berburu() atau membangun\_sarang()) yang hanya dimiliki oleh kelas anak tersebut. Jika objek bukan Burung, pesan berbeda ditampilkan. Ini menunjukkan cara menggabungkan polimorfisme dinamis (memanggil metode yang mungkin di-override) dengan pemeriksaan tipe eksplisit untuk kontrol alur program yang lebih spesifik.

* 1. **Hasil Praktikum**

Lengkapi hasil tabel praktikum berikut:

|  |  |  |
| --- | --- | --- |
| **No.** | **Nama Praktikum** | **Hasil Praktikum** |
| 1 | Praktikum 1: Polimorfisme dengan Inheritance dan Overriding | 1    Kelas induk Burung memiliki metode terbang() dan bersuara(). Dua kelas turunan, Elang dan Pipit, melakukan **override** terhadap kedua metode tersebut untuk menampilkan perilaku yang berbeda sesuai jenis burung.  Melalui fungsi demonstrasi\_aksi\_burung(), ditunjukkan bahwa meskipun metode yang dipanggil sama, hasilnya berbeda tergantung objek burung yang digunakan. |
| 2 | Praktikum 2: Polimorfisme dengan Duck Typing | 2    konsep **duck typing**, yaitu sebuah objek dianggap valid jika memiliki metode yang dibutuhkan, tanpa mempedulikan tipe kelasnya. Fungsi buat\_suara() memanggil metode bersuara() dari berbagai objek (Kucing, Anjing, Bebek), dan berhasil karena mereka memiliki metode tersebut. Namun, saat objek Mobil digunakan, muncul pesan kesalahan karena tidak memiliki metode bersuara(). Ini menunjukkan bagaimana duck typing bekerja berdasarkan **perilaku**, bukan pewarisan kelas. |
| 3 | Praktikum 3: Simulasi Overloading dalam Metode Kelas dengan \*args | 3    metode tambah() pada kelas KalkulatorSederhana dapat menerima jumlah argumen yang bervariasi menggunakan \*args, yang mensimulasikan **method overloading**. Metode ini menjumlahkan semua argumen numerik, mengabaikan input non-angka seperti string. Jika tidak ada argumen, hasilnya adalah 0. |
| 4 | Praktikum 4: Simulasi Overloading dalam Metode Kelas dengan \*args dan \*\*kwargs | 4    metode process() pada kelas DataProcessor mensimulasikan **overloading** dengan menerima jumlah argumen yang fleksibel melalui \*args (posisi) dan \*\*kwargs (kata kunci). |
| 5 | Praktikum 5: Kombinasi Polimorfisme  (Inheritance & Duck Typing) | 5    Kelas *Bentuk* mendefinisikan metode hitung\_luas() yang di-override oleh subclass seperti *Lingkaran* dan *Persegi*. Kelas lain, seperti *TaplakMeja*, juga memiliki metode hitung\_luas() meskipun tidak terhubung dengan *Bentuk*. Fungsi tampilkan\_info\_luas() memanfaatkan polimorfisme untuk menangani berbagai objek dengan metode hitung\_luas(), menggunakan duck typing untuk mendeteksi metode lain seperti nama\_bentuk() jika ada. |
| 6 | Praktikum 6: Kontrol Perilaku Polimorfik dengan isinstance() | 6    Fungsi interaksi\_dengan\_burung() memeriksa tipe objek yang diterima, apakah merupakan instance dari kelas *Burung* atau turunannya (seperti *Elang* atau *Pipit*). Berdasarkan tipe objek, metode yang sesuai dipanggil, seperti terbang() dan bersuara() dari kelas induk, serta metode spesifik seperti berburu() untuk *Elang* dan membangun\_sarang() untuk *Pipit*. Dengan isinstance(), program bisa menentukan perilaku spesifik objek, sekaligus menangani objek yang tidak termasuk dalam hierarki kelas *Burung*. |

* 1. **Penugasan**

1. Kumpulkan Laporan Praktikum dari jobsheet ini dalam bentuk Microsoft word sesuai dengan format jobsheet praktikum dan dikumpulkan di web LMS. (JANGAN DALAM BENTUK PDF)
2. Kumpulkan luaran kode praktikum dalam bentuk ipynb yang sudah diunggah pada akun github masing-masing. Lampirkan tautan github yang sudah di unggah melalui laman LMS.

**Link Github :** <https://github.com/vian151004/SPBO--P-Prayitno->

1. **Buat Program dan Kelas diagram** untuk hierarki kelas Komputer dan mendemonstrasikan polimorfisme serta simulasi overloading.

* **Buat Kelas Induk (Komputer):** o Atribut: merk (string), processor (string), ram\_gb (integer). o Konstruktor (\_\_init\_\_): Menerima dan menginisialisasi ketiga atribut di atas. o Metode info\_spesifikasi(self): Mencetak informasi dasar komputer (merk, processor, RAM). Contoh:
  + - Merk: [Merk Komputer]
    - Processor: [Nama Processor] o RAM: [Jumlah RAM] GB
    - Metode jalankan\_aplikasi(self, nama\_aplikasi): Mencetak pesan bahwa komputer sedang menjalankan aplikasi. Contoh: "[Merk Komputer] menjalankan aplikasi: [nama\_aplikasi]..."
* **Buat Kelas Anak 1 (Laptop):**
  + - Mewarisi dari Komputer.
    - Atribut Tambahan: ukuran\_layar\_inch (float), berat\_kg (float).
    - Konstruktor (\_\_init\_\_): Menerima merk, processor, ram\_gb, ukuran\_layar\_inch, dan berat\_kg. Gunakan super().\_\_init\_\_() untuk inisialisasi bagian Komputer.
    - **Override** metode info\_spesifikasi(self): Panggil super().info\_spesifikasi(), lalu tambahkan informasi ukuran layar dan berat laptop. Contoh output tambahan:
    - Ukuran Layar: [Ukuran Layar] inch o Berat: [Berat Laptop] Kg
* **Buat Kelas Anak 2 (Desktop):**
  + - Mewarisi dari Komputer.
    - Atribut Tambahan: jenis\_casing (string, misal: "Tower", "Mini-PC"), monitor\_external (boolean, True jika ada monitor terpisah).
    - Konstruktor (\_\_init\_\_): Menerima merk, processor, ram\_gb, jenis\_casing, dan monitor\_external.

Gunakan super().\_\_init\_\_() untuk inisialisasi bagian Komputer.

* + - **Override** metode info\_spesifikasi(self): Panggil super().info\_spesifikasi(), lalu tambahkan informasi jenis casing dan status monitor eksternal. Contoh output tambahan:
    - Jenis Casing: [Jenis Casing] o Monitor External: [Ya / Tidak]
* **Demonstrasi Polimorfisme:** o Buat sebuah fungsi terpisah, misalnya cetak\_semua\_spesifikasi(daftar\_komputer). o Fungsi ini menerima sebuah list daftar\_komputer yang berisi objek-objek dari Laptop dan Desktop.

o Di dalam fungsi, lakukan loop pada daftar\_komputer dan untuk setiap komputer dalam list, panggil metode komputer.info\_spesifikasi(). o Amati bagaimana pemanggilan metode yang sama menghasilkan output yang berbeda

(sesuai override di Laptop dan Desktop).

* **Simulasi Overloading (Pilih salah satu metode berikut):**
  + - **Opsi A (Default Argument):** Tambahkan metode upgrade\_ram(self, tambahan\_gb) pada kelas Komputer. Modifikasi metode ini agar bisa menerima argumen opsional tipe\_ram dengan nilai default (misal, "DDR4"). Metode ini harus mencetak informasi upgrade. Contoh: RAM [Merk Komputer] diupgrade sebesar [tambahan\_gb] GB (Tipe: [tipe\_ram]). Total RAM sekarang: [RAM baru] GB. Panggil metode ini dengan dan tanpa argumen tipe\_ram.
    - \**Opsi B (args):* Tambahkan metode install\_software(self, \*args) pada kelas Komputer. Metode ini harus bisa menerima satu atau lebih nama software (string) sebagai argumen posisi. Di dalam metode, cetak nama setiap software yang diinstal. Contoh: Menginstall [nama\_software] di [Merk Komputer]... Panggil metode ini dengan satu nama software dan dengan beberapa nama software.
* **Buat Kode Utama (Bagian if \_\_name\_\_ == "\_\_main\_\_":)** o Buat minimal satu objek Laptop dan satu objek Desktop. o Masukkan objek-objek tersebut ke dalam sebuah list.

o Panggil fungsi cetak\_semua\_spesifikasi() yang Anda buat di langkah 4 dengan list tersebut. o Demonstrasikan pemanggilan metode simulasi overloading yang Anda pilih di langkah 5 pada salah satu objek komputer Anda.

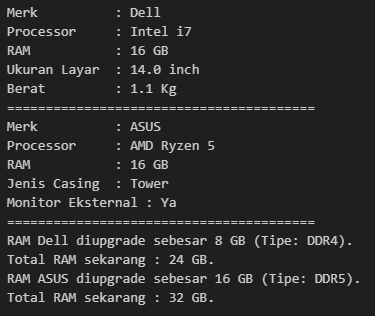
**HASIL PENUGASAN :**

**Code Program yang telah dibuat :**

|  |
| --- |
| 01: class Komputer:  02: def \_\_init\_\_(self, merk, processor, ram\_gb):  03: self.merk = merk  04: self.processor = processor  05: self.ram\_gb = ram\_gb  06:  07: def info\_spesifikasi(self):  08: print(f"Merk : {self.merk}")  09: print(f"Processor : {self.processor}")  10: print(f"RAM : {self.ram\_gb} GB")  11:  12: def jalankan\_aplikasi(self, nama\_aplikasi):  13: print(f"{self.merk} menjalankan aplikasi : {nama\_aplikasi}.")  14:  15: #Simulasi overloading (opsi A - default argumen)  16: def upgrade\_ram(self, tambahan\_gb, tipe\_ram="DDR4"):  17: self.ram\_gb += tambahan\_gb  18: print(f"RAM {self.merk} diupgrade sebesar {tambahan\_gb} GB (Tipe: {tipe\_ram}). Total RAM sekarang : {self.ram\_gb} GB.")  19:  20: class Laptop(Komputer):  21: def \_\_init\_\_(self, merk, processor, ram\_gb, ukuran\_layar\_inch, berat\_kg):  22: super().\_\_init\_\_(merk, processor, ram\_gb)  23: self.ukuran\_layar\_inch = ukuran\_layar\_inch  24: self.berat\_kg = berat\_kg  25:  26: def info\_spesifikasi(self):  27: super().info\_spesifikasi()  28: print(f"Ukuran layar : {self.ukuran\_layar\_inch} inch")  29: print(f"Berat : {self.berat\_kg} Kg")  30:  31: class Desktop(Komputer):  32: def \_\_init\_\_(self, merk, processor, ram\_gb, jenis\_casing, monitor\_external):  33: super().\_\_init\_\_(merk, processor, ram\_gb)  34: self.jenis\_casing = jenis\_casing  35: self.monitor\_external = monitor\_external  36:  37: def info\_spesifikasi(self):  38: super().info\_spesifikasi()  39: print(f"Jenis Casing : {self.jenis\_casing}")  40: print(f"Monitor External : {'Ya' if self.monitor\_external else 'Tidak'}")  41:  42: #Polimorfisme  43: def cetak\_semua\_spesifikasi(daftar\_komputer):  44: for komputer in daftar\_komputer:  45: komputer.info\_spesifikasi()  46: print("=" \* 40)  47:  48: #Kode Utama  49: if \_\_name\_\_ == "\_\_main\_\_":  50: laptop1 = Laptop("Dell", "Intel i7", 8, 14.0, 1.1)  51: desktop1 = Desktop("ASUS", "AMD Ryzen 5", 16, "Tower", True)  52:  53: daftar = [laptop1, desktop1]  54: cetak\_semua\_spesifikasi(daftar)  55:  56: #Demonstrasi Overloading  57: laptop1.upgrade\_ram(4)  58: desktop1.upgrade\_ram(8, "DDR5") |



**Hasil dalam VSC :**

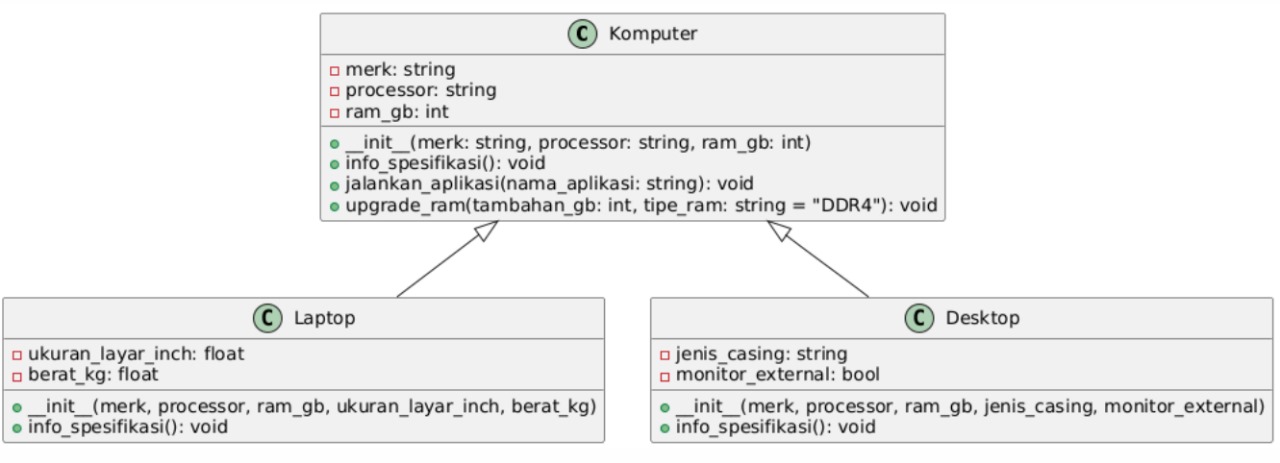


**Kelas Diagram :**

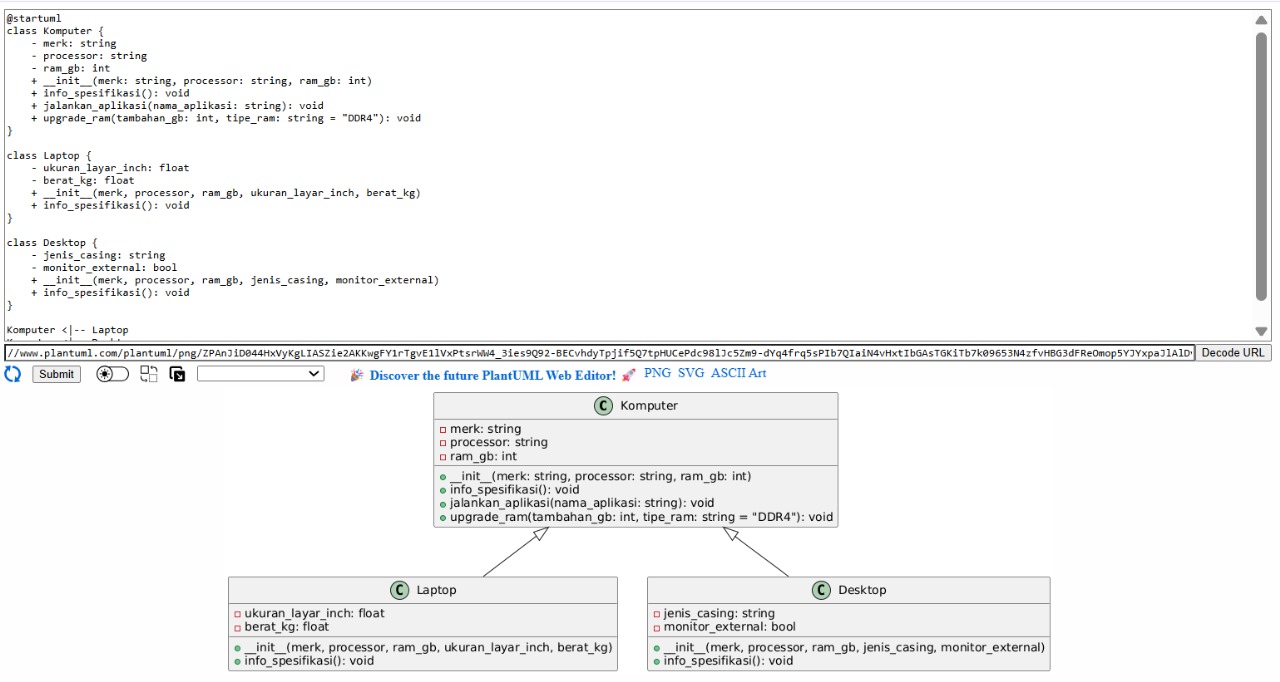
**Code kelas diagram dalam uml :**

|  |
| --- |
| @startuml  class Komputer {  - merk: string  - processor: string  - ram\_gb: int  + \_\_init\_\_(merk: string, processor: string, ram\_gb: int)  + info\_spesifikasi(): void  + jalankan\_aplikasi(nama\_aplikasi: string): void  + upgrade\_ram(tambahan\_gb: int, tipe\_ram: string = "DDR4"): void  }  class Laptop {  - ukuran\_layar\_inch: float  - berat\_kg: float  + \_\_init\_\_(merk: string, processor: string, ram\_gb: int, ukuran\_layar\_inch: float, berat\_kg: float)  + info\_spesifikasi(): void  }  class Desktop {  - jenis\_casing: string  - monitor\_external: bool  + \_\_init\_\_(merk: string, processor: string, ram\_gb: int, jenis\_casing: string, monitor\_external: bool)  + info\_spesifikasi(): void  }  Komputer <|-- Laptop  Komputer <|-- Desktop  @enduml |

**Hasil Diagram Kode Kelas :**



**Percobaan dalam plantuml :**



* 1. **Kesimpulan**

Melalui praktikum ini, mahasiswa telah memahami konsep polimorfisme sebagai kemampuan objek berbeda untuk merespons metode yang sama dengan cara unik, baik melalui mekanisme pewarisan dan method overriding maupun melalui pendekatan fleksibel *duck typing* yang umum di Python. Selain itu, telah dipelajari perbedaan antara method overriding yang didukung penuh dan keterbatasan method overloading tradisional di Python, serta bagaimana menyimulasikan fleksibilitas overloading menggunakan teknik seperti nilai parameter default, \*args, dan \*\*kwargs. Penguasaan konsep-konsep ini memungkinkan penulisan kode yang lebih adaptif, dapat diperluas, dan mampu berinteraksi dengan beragam tipe objek secara elegan dalam paradigma Pemrograman Berorientasi Objek.

* 1. **Daftar Pustaka**

1. Lutz, M. (2013). *Learning Python*. O'Reilly Media.
2. Guttag, J. V. (2016). *Introduction to Computation and Programming Using Python*. MIT Press.
3. Python Software Foundation. *Python 3 Documentation*. <https://docs.python.org/3/>