BAB VI

OBJECT-ORIENTED PROGRAMMING (OOP)

B. EMPAT PILAR OOP

1 . Pewarisan (Inheritance)

Pewarisan (*Inheritance*) adalah cara untuk membuat kelas baru dengan menggunakan detail (atribut dan *method*) dari kelas yang sudah ada tanpa perlu memodifikasinya. Kelas yang baru terbentuk disebut **kelas turunan** (atau *child class*), sedangkan kelas yang ada disebut **kelas dasar** (atau *parent class*).

```python
# class parent (Kelas Dasar)
class Mobil:
    def __init__(self, merek):
        self.merek = merek
        self.kecepatan = 0  # Atribut dasar: kecepatan awal 0

    def akselerasi(self, tambahan_kecepatan):
        self.kecepatan += tambahan_kecepatan
        print(f"{self.merek} berakselerasi. Kecepatan saat ini: {self.kecepatan} km/jam")

# class child (Kelas Turunan)
class MobilSport(Mobil): # Inheritance ke class Mobil
    def __init__(self, merek, turbo_aktif=False):
        super().__init__(merek) # Memanggil __init__ parent untuk inisialisasi merek dan kecepatan
        self.turbo_aktif = turbo_aktif # Atribut spesifik MobilSport

    def aktifkan_turbo(self):
        self.turbo_aktif = True
        print(f"Turbo pada {self.merek} diaktifkan! 🚀")

# objek
ferrari = MobilSport("Ferrari")
ferrari.akselerasi(50) # Memanggil method 'akselerasi' dari class Mobil (Parent)
ferrari.aktifkan_turbo() # Memanggil method spesifik MobilSport

print(f"Merek: {ferrari.merek}")

```

Output:

```
Ferrari berakselerasi. Kecepatan saat ini: 50 km/jam
Turbo pada Ferrari diaktifkan! 🚀
Merek: Ferrari
```

Berdasarkan contoh di atas, class MobilSport dapat memanggil *method* dan atribut yang telah didefinisikan pada class Mobil meskipun pada class MobilSport atribut dan *method* tersebut tidak didefinisikan ulang. Meskipun *method* diwariskan secara otomatis, **konstruktor** (__init__) tidak. Namun, konstruktor kelas *parent* masih dapat dipanggil dari kelas *child* menggunakan *keyword* super().__init__().

Contoh lain penggunaan Inheritance (Pewarisan)

1. **Parent Class**

   Kelas ini menyediakan konstruktor dan *method* yang akan diwarisi oleh kelas turunan

```python
class Bentuk:
    def __init__(self, nama):
        self.nama = nama
        print(f"Bentuk '{self.nama}' telah dibuat.")

    # Method yang diwarisi
    def deskripsi(self):
        return f"Ini adalah objek dari bentuk {self.nama}."

    # Method yang HARUS di-override oleh Child Class
    def hitung_luas(self):
        # Menyebabkan error jika Child tidak meng-override
        raise NotImplementedError("Subkelas harus mengimplementasikan method 'hitung_luas'")


```

2. **Chill Class**

   Kelas ini mewarisi dari Bentuk dan menambahkan logika spesifiknya

```python
class Persegi(Bentuk):
    def __init__(self, sisi):
        # Memanggil konstruktor Parent (Bentuk)
        super().__init__("Persegi")
        self.sisi = sisi

    # Method Overriding: Implementasi spesifik untuk Persegi
    def hitung_luas(self):
        return self.sisi * self.sisi

# Membuat objek
kotak = Persegi(sisi=4)

# Menggunakan method yang diwarisi dari Bentuk (Parent)
print(kotak.deskripsi())

# Menggunakan method yang di-override (Child-specific)
print(f"Luas {kotak.nama}: {kotak.hitung_luas()}")

```

Output:

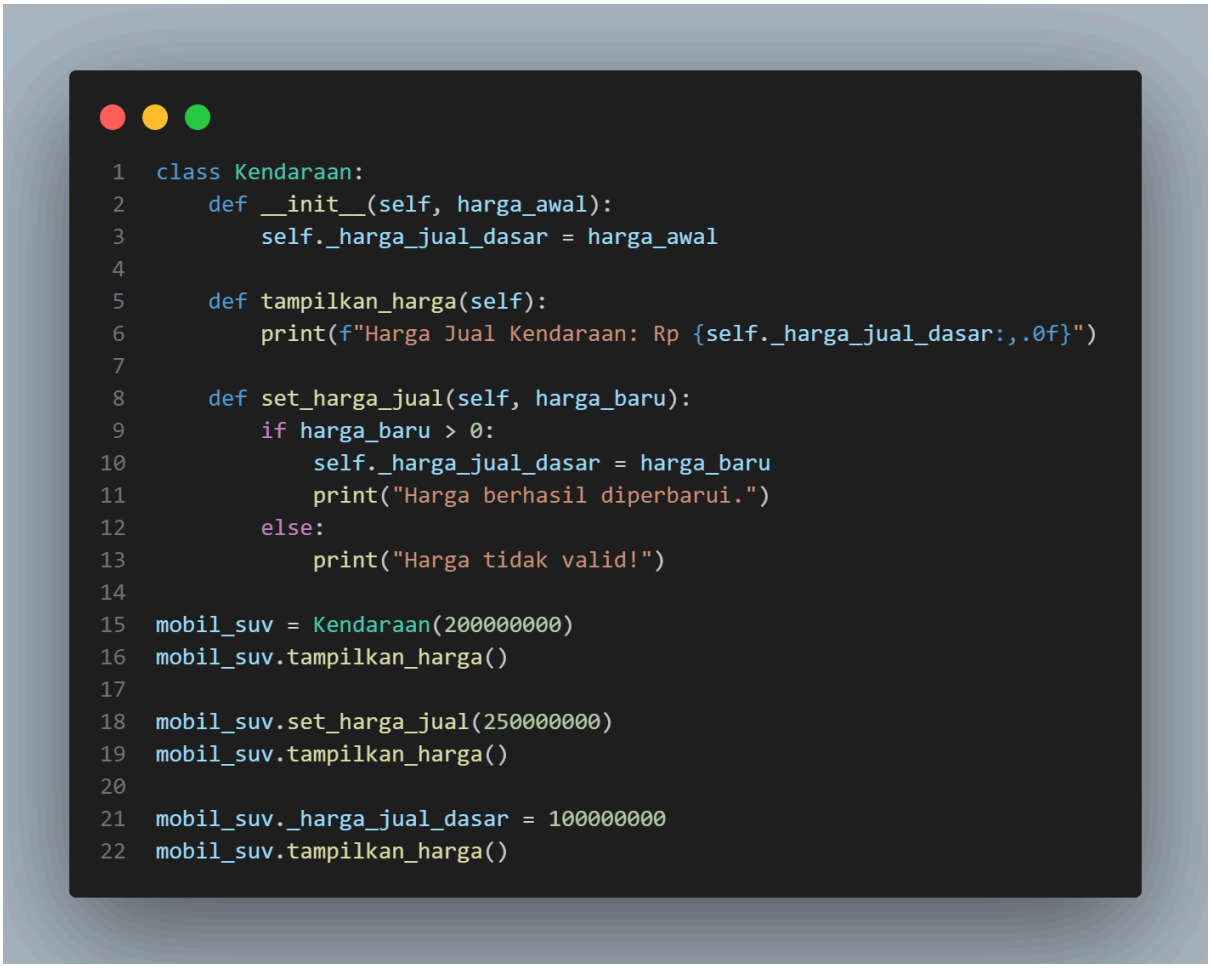
```
Bentuk 'Persegi' telah dibuat.
Ini adalah objek dari bentuk Persegi.
Luas Persegi: 16
```

2. Enkapsulasi (Encapsulation)

Enkapsulasi adalah konsep dalam OOP yang membatasi akses langsung ke atribut dan *method* sebuah objek. Tujuannya adalah untuk melindungi data internal objek agar tidak dimodifikasi secara tidak terduga dari luar. Dalam Python, kita dapat menandai atribut sebagai 'privat' (walaupun tidak sepenuhnya ketat seperti bahasa lain) dengan menggunakan awalan garis bawah:

- **Satu garis bawah tunggal (_)**: Menunjukkan bahwa atribut/method tersebut seharusnya diperlakukan sebagai **internal/dilindungi** (*protected*).

  Contoh Penggunaan Satu garis bawah tunggal



```python
class Kendaraan:
    def __init__(self, harga_awal):
        self._harga_jual_dasar = harga_awal

    def tampilkan_harga(self):
        print(f"Harga Jual Kendaraan: Rp {self._harga_jual_dasar:,.0f}")

    def set_harga_jual(self, harga_baru):
        if harga_baru > 0:
            self._harga_jual_dasar = harga_baru
            print("Harga berhasil diperbarui.")
        else:
            print("Harga tidak valid!")

mobil_suv = Kendaraan(200000000)
mobil_suv.tampilkan_harga()

mobil_suv.set_harga_jual(250000000)
mobil_suv.tampilkan_harga()

mobil_suv._harga_jual_dasar = 100000000
mobil_suv.tampilkan_harga()
```

  Output

```
Harga Jual Kendaraan: Rp 200,000,000
Harga berhasil diperbarui.
Harga Jual Kendaraan: Rp 250,000,000
Harga Jual Kendaraan: Rp 100,000,000
```

- **Garis bawah ganda (__)**: Memicu **Name Mangling** oleh Python, menjadikannya sangat sulit (tetapi tidak mustahil) untuk diakses dari luar kelas, yang secara efektif menunjukkan status **privat** (*private*).

  Contoh Penggunaan Garis bawah ganda

```python
class Kendaraan:
    def __init__(self):
        # Atribut privat (menggunakan __): tidak dapat diakses langsung dari luar
        self.__harga_jual_dasar = 200000000

    def tampilkan_harga(self):
        print("Harga jual kendaraan: Rp {}".format(self.__harga_jual_dasar))

    # Method publik untuk memodifikasi atribut privat (Setter)
    def set_harga_jual(self, harga_baru):
        if harga_baru > 0:
            self.__harga_jual_dasar = harga_baru
            print("Harga berhasil diperbarui.")
        else:
            print("Harga tidak valid!")

mobil_suv = Kendaraan()
mobil_suv.tampilkan_harga()

# Mencoba mengubah atribut privat secara langsung (Python tidak mengizinkannya)
mobil_suv.__harga_jual_dasar = 100000000
mobil_suv.tampilkan_harga() # Output tetap 200.000.000

# Menggunakan method set (Setter) yang disediakan untuk mengubah atribut privat
mobil_suv.set_harga_jual(250000000)
mobil_suv.tampilkan_harga()
```

  Output:

```
Harga jual kendaraan: Rp 200000000
Harga jual kendaraan: Rp 200000000
Harga berhasil diperbarui.
Harga jual kendaraan: Rp 250000000
```

3. Polimorfisme (Polymorphism)

Polimorfisme adalah kemampuan (dalam OOP) untuk menggunakan **satu *interface* publik (nama *method* yang sama)** untuk berbagai bentuk atau tipe data. Ini berarti objek yang berbeda dapat merespons panggilan *method* yang sama dengan **perilaku yang berbeda** sesuai dengan tipe objeknya sendiri.

Jenis Polimorfisme (pada konteks OOP):

1. **Polimorfisme Compile-Time (atau Static Polymorphism):**
   - Dicapai melalui **Overloading** (method atau operator).
   - Sistem menentukan method mana yang akan dipanggil saat program dikompilasi.

    - Contoh: Mendefinisikan beberapa method dengan nama yang sama tetapi dengan **jumlah atau tipe parameter yang berbeda**.
2. **Polimorfisme Run-Time (atau Dynamic Polymorphism):**
    - Dicapai melalui **Overriding** (method).
    - Sistem menentukan method mana yang akan dipanggil saat program berjalan.
    - Ini adalah inti dari bagaimana **pewarisan** dan interface bekerja dalam mewujudkan polimorfisme, memungkinkan method kelas anak (turunan) menggantikan method kelas induk (dasar).

Manfaat Polimorfisme:

- **Fleksibilitas dan Ekstensibilitas:** Memungkinkan penambahan kelas baru tanpa mengubah kode yang memanfaatkan interface polimorfik (seperti fungsi uji_klakson() di bawah).
- **Keterbacaan dan Kemudahan Perawatan (Maintainability):** Kode menjadi lebih general dan abstract, karena dapat berinteraksi dengan berbagai objek melalui satu interface yang seragam.

Contoh Implementasi Polimorfisme (Menggunakan Overloading)

Polimorfisme jenis ini disebut juga **Compile-Time Polymorphism (Static Polymorphism)**, karena pemilihan method yang dijalankan sudah ditentukan saat proses kompilasi.
Konsep ini dicapai melalui **Overloading**, yaitu ketika sebuah kelas memiliki **beberapa method dengan nama yang sama**, namun memiliki **jumlah atau tipe parameter yang berbeda**.

Dengan overloading, kita dapat menggunakan satu nama method untuk berbagai tujuan tergantung pada **argumen yang dikirimkan** saat pemanggilan.
Hal ini membuat kode menjadi lebih **fleksibel**, **ringkas**, dan **mudah dibaca**.

Dalam contoh berikut, kita akan membuat kelas BangunDatar sebagai **kelas induk** dan kelas Segitiga sebagai **kelas anak**.
Keduanya memiliki method hitung_luas() yang sama namanya, tetapi berbeda dalam cara menghitung luas tergantung parameter yang diberikan.

```python
class BangunDatar:
    def hitung_luas(self, a=None, b=None):
        if a is not None and b is not None:
            return a * b   # Luas persegi panjang
        elif a is not None:
            return a * a   # Luas persegi
        else:
            return 0       # Tidak ada parameter

# Kelas turunan (anak) yang mewarisi BangunDatar
class Segitiga(BangunDatar):
    # Overloading dengan menambahkan parameter berbeda
    def hitung_luas(self, a=None, b=None, tinggi=None):
        if a is not None and tinggi is not None:
            return 0.5 * a * tinggi  # Luas segitiga
        elif a is not None and b is not None:
            return a * b             # Masih bisa hitung persegi panjang
        elif a is not None:
            return a * a             # Bisa hitung persegi
        else:
            return 0

# Membuat objek dari kelas induk dan anak
bd = BangunDatar()
sg = Segitiga()

# Pemanggilan method yang sama dengan bentuk berbeda
print("=== Dari Kelas BangunDatar ===")
print("Luas persegi         :", bd.hitung_luas(4))
print("Luas persegi panjang :", bd.hitung_luas(4, 6))

print("\n=== Dari Kelas Segitiga ===")
print("Luas segitiga        :", sg.hitung_luas(6, tinggi=4))
print("Luas persegi panjang :", sg.hitung_luas(4, 8))
print("Luas persegi         :", sg.hitung_luas(5))

```

Output:

```
=== Dari Kelas BangunDatar ===
Luas persegi         : 16
Luas persegi panjang : 24

=== Dari Kelas Segitiga ===
Luas segitiga        : 12.0
Luas persegi panjang : 32
Luas persegi         : 25
```

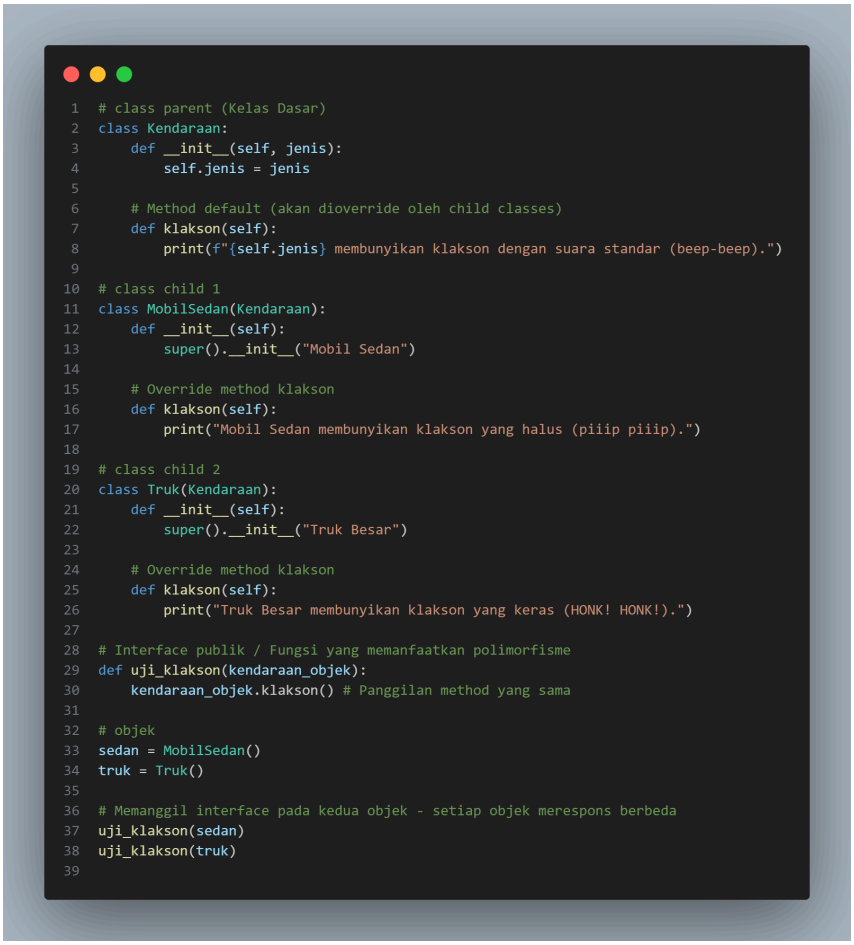
Method hitung_luas() pada contoh di atas memiliki nama yang sama, tetapi dapat menerima parameter yang berbeda-beda sesuai kebutuhan. Pada kelas induk yaitu BangunDatar, method tersebut digunakan untuk menghitung luas persegi dan persegi panjang. Sementara itu, pada kelas anak yaitu Segitiga, method yang sama diperluas dengan menambahkan parameter tinggi sehingga dapat menghitung luas segitiga. Meskipun nama method yang digunakan sama, hasil yang diperoleh berbeda tergantung pada parameter yang diberikan saat pemanggilan. Hal ini menunjukkan bahwa satu nama method dapat memiliki berbagai bentuk perilaku sesuai konteks penggunaannya, yang merupakan penerapan dari konsep Polimorfisme melalui Overloading.

Contoh Implementasi Polimorfisme (Menggunakan Overriding)

Dalam program di atas, kita mendefinisikan dua kelas berbeda (**MobilSedan** dan **Truk**), yang keduanya mewarisi dari **Kendaraan** dan memiliki *method* **klakson()** yang di-*override* dengan fungsi yang berbeda.

- MobilSedan meng-*override* klakson() untuk mencetak suara halus ("piiip piiip").
- Truk meng-*override* klakson() untuk mencetak suara keras ("HONK! HONK!").

Fungsi **uji_klakson(kendaraan_objek)** adalah *interface* polimorfik; ia memanggil *method* klakson() (**Panggilan *method* yang sama**) tanpa peduli apakah objek yang dilewatkan adalah MobilSedan atau Truk, dan mendapatkan perilaku yang benar dari masing-masing objek (**Perilaku yang berbeda**).



```python
# class parent (Kelas Dasar)
class Kendaraan:
    def __init__(self, jenis):
        self.jenis = jenis

    # Method default (akan dioverride oleh child classes)
    def klakson(self):
        print(f"{self.jenis} membunyikan klakson dengan suara standar (beep-beep).")

# class child 1
class MobilSedan(Kendaraan):
    def __init__(self):
        super().__init__("Mobil Sedan")

    # Override method klakson
    def klakson(self):
        print("Mobil Sedan membunyikan klakson yang halus (piiip piiip).")

# class child 2
class Truk(Kendaraan):
    def __init__(self):
        super().__init__("Truk Besar")

    # Override method klakson
    def klakson(self):
        print("Truk Besar membunyikan klakson yang keras (HONK! HONK!).")

# Interface publik / Fungsi yang memanfaatkan polimorfisme
def uji_klakson(kendaraan_objek):
    kendaraan_objek.klakson() # Panggilan method yang sama

# objek
sedan = MobilSedan()
truk = Truk()

# Memanggil interface pada kedua objek - setiap objek merespons berbeda
uji_klakson(sedan)
uji_klakson(truk)
```

Output:

```
Mobil Sedan membunyikan klakson yang halus (piiip piiip).
Truk Besar membunyikan klakson yang keras (HONK! HONK!).
```

Dalam program di atas, kita mendefinisikan dua kelas berbeda (MobilSedan dan Truk), yang keduanya mewarisi dari Kendaraan dan memiliki *method* klakson() yang di-override dengan fungsi yang berbeda. Fungsi uji_klakson() adalah *interface* polimorfik; ia memanggil *method* klakson() tanpa peduli apakah objek yang dilewatkan adalah MobilSedan atau Truk, dan mendapatkan perilaku yang benar dari masing-masing objek.

4. Abstraksi (Abstraction)

Abstraksi adalah proses menyembunyikan detail implementasi dari pengguna dan hanya menampilkan fungsi atau fitur penting dari suatu objek.
Dengan abstraksi, pengguna cukup tahu *apa yang dilakukan* sebuah method, tanpa harus tahu *bagaimana cara kerjanya* di dalam.

Ciri Class Abstrak:

- Tidak bisa dibuat objek secara langsung.
- Digunakan sebagai dasar bagi class lain.
- Memiliki satu atau lebih method abstrak (belum diimplementasikan).

Tujuan utama abstraksi adalah untuk:

1. Menyederhanakan kompleksitas sistem.
   → Hanya menampilkan bagian penting yang relevan.

2. Meningkatkan keamanan kode.
   → Detail implementasi disembunyikan dari pengguna.

3. Meningkatkan efisiensi pengembangan.
   → Programmer cukup memahami *interface* tanpa harus tahu isi implementasi.

4. Mempermudah perawatan kode (maintenance).
   → Perubahan pada detail implementasi tidak mempengaruhi pengguna class.

Abstraksi dalam Python:

a. Menggunakan Kelas Abstrak (Abstract Class)

Python menyediakan modul **`abc`** (*Abstract Base Class*) untuk membuat kelas abstrak.
Kelas abstrak adalah kelas yang tidak dapat diinstansiasi (tidak bisa dibuat objek langsung) dan biasanya hanya berfungsi sebagai kerangka dasar (template) bagi class lain.

Kelas abstrak biasanya:

- Didefinisikan menggunakan **`from abc import ABC, abstractmethod`**
- Memiliki method yang diberi dekorator **`@abstractmethod`**
- Harus diturunkan (*inherited*) oleh class lain
- Class turunan wajib mengimplementasikan semua method abstrak

Contoh :

```
from abc import ABC, abstractmethod

# Membuat class abstrak
class Kendaraan(ABC):

    @abstractmethod
    def bergerak(self):
        pass  # Method abstrak (belum ada implementasi)

# Membuat class turunan
class Mobil(Kendaraan):
    def bergerak(self):
        return "Mobil berjalan di jalan raya"

class Pesawat(Kendaraan):
    def bergerak(self):
        return "Pesawat terbang di udara"

# Objek dari class turunan
mobil = Mobil()
pesawat = Pesawat()

print(mobil.bergerak())
print(pesawat.bergerak())
```

OUTPUT :

```
Mobil berjalan di jalan raya
Pesawat terbang di udara
```

Penjelasan:

- Kendaraan adalah **class abstrak**.
- Method bergerak() wajib diimplementasikan oleh class turunan.
- Jika class turunan tidak mengimplementasikan method tersebut, maka akan terjadi error.

Contoh Abstraksi dengan Lebih dari Satu Method :

```
from abc import ABC, abstractmethod
class Bentuk(ABC):
    @abstractmethod
    def luas(self):
        pass
    @abstractmethod
    def keliling(self):
        pass
class Persegi(Bentuk):
    def __init__(self, sisi):
        self.sisi = sisi
    def luas(self):
        return self.sisi * self.sisi
    def keliling(self):
        return 4 * self.sisi
# Membuat objek dari class turunan
p = Persegi(5)
print("Luas Persegi:", p.luas())
print("Keliling Persegi:", p.keliling())
```

Output :

```
Luas Persegi: 25
Keliling Persegi: 20
```

Penjelasan:

- Bentuk adalah kelas abstrak yang berisi dua *abstract method*: luas() dan keliling().
- Persegi mengimplementasikan kedua method tersebut sesuai dengan rumusnya.
- Kelas Bentuk tidak bisa dibuat objek langsung, tetapi Persegi bisa.

Kesimpulan

1. Abstraksi adalah proses menyembunyikan detail yang tidak perlu dan hanya menampilkan fungsi penting kepada pengguna.
2. Kelas abstrak digunakan sebagai kerangka dasar untuk membuat class turunan.
3. Abstraksi dapat dibuat di Python dengan modul abc dan dekorator @abstractmethod.

4. Class turunan wajib mengimplementasikan method abstrak agar bisa digunakan.
5. Dengan abstraksi, program menjadi lebih terstruktur, aman, dan mudah dikembangkan.