





|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

Berikut adalah penjelasan detail mengenai konsep \*abstraction\*, \*encapsulation\*, dan \*inheritance\* dalam pemrograman berorientasi objek (OOP) beserta contohnya.

### 1. Abstraksi (\*Abstraction\*)

Abstraksi adalah konsep untuk menyembunyikan detail yang tidak penting dari pengguna dan hanya menampilkan informasi yang relevan. Abstraksi memungkinkan kita fokus pada apa yang dilakukan sebuah objek tanpa perlu mengetahui bagaimana cara kerja internalnya. Dengan abstraksi, kita dapat mendefinisikan objek-objek melalui interface atau kelas abstrak.

\*\*Contoh:\*\*

Misalkan kita memiliki kelas `Mobil`. Saat kita mengemudi, kita hanya perlu tahu bagaimana cara mengemudikan mobil tanpa mengetahui detail mesin dan cara kerjanya. Dalam OOP, kita bisa membuat kelas `Mobil` yang hanya menampilkan informasi penting seperti `mengemudi()` tanpa perlu menunjukkan detail fungsi mesin.

```python

from abc import ABC, abstractmethod

class Kendaraan(ABC):

@abstractmethod

def mengemudi(self):

pass

class Mobil(Kendaraan):

def mengemudi(self):

print("Mobil berjalan di jalan raya")

# Hanya perlu tahu metode mengemudi, detail lainnya tersembunyi

kendaraan = Mobil()

kendaraan.mengemudi()

```

### 2. Enkapsulasi (\*Encapsulation\*)

Enkapsulasi adalah konsep membungkus data (variabel) dan metode dalam satu kesatuan kelas, serta mengatur aksesibilitas data tersebut menggunakan kata kunci akses seperti `private`, `protected`, atau `public`. Tujuan enkapsulasi adalah untuk menjaga data dari modifikasi langsung oleh luar kelas, sehingga bisa lebih terkontrol.

\*\*Contoh:\*\*

Misalkan kita memiliki kelas `BankAccount` yang memiliki informasi saldo yang tidak boleh diakses secara langsung dari luar kelas. Kita bisa menggunakan enkapsulasi untuk membatasi akses ke variabel `saldo`.

```python

class BankAccount:

def \_\_init\_\_(self, initial\_balance):

self.\_\_saldo = initial\_balance # saldo hanya bisa diakses di dalam kelas

def deposit(self, amount):

self.\_\_saldo += amount

def get\_balance(self):

return self.\_\_saldo

# Penggunaan

akun = BankAccount(1000)

akun.deposit(500)

print(akun.get\_balance()) # Output: 1500

```

### 3. Pewarisan (\*Inheritance\*)

Pewarisan memungkinkan sebuah kelas untuk mewarisi sifat dan metode dari kelas lain. Dengan pewarisan, kita dapat membuat hubungan antara kelas induk (\*parent\*) dan kelas turunan (\*child\*). Kelas turunan dapat mewarisi sifat-sifat kelas induknya dan dapat menambahkan atau mengubah metode sesuai kebutuhan.

\*\*Contoh:\*\*

Misalkan kita memiliki kelas `Hewan` sebagai kelas induk dan `Kucing` sebagai kelas turunan. `Kucing` dapat mewarisi properti dan metode dari `Hewan` serta menambahkan fungsionalitas tambahan.

```python

class Hewan:

def bergerak(self):

print("Hewan bergerak")

class Kucing(Hewan):

def bersuara(self):

print("Meong!")

# Penggunaan

kucing = Kucing()

kucing.bergerak() # Output: Hewan bergerak

kucing.bersuara() # Output: Meong!

```

Dalam contoh ini, kelas `Kucing` mewarisi metode `bergerak()` dari kelas `Hewan` tetapi juga memiliki metode tambahan `bersuara()`.