



SOLID Principle

Silver- Chapter 2 - Topic 1

**Selamat datang di Chapter 2 Topic 1 online course
Android Developer dari Binar Academy!**



Selamat Datang di Chapter 2 ✨

Pada Chapter 1 kita sudah melewati dasar-dasar pemrograman Kotlin. Sekarang kita akan fokuskan pembahasan kita pada konsep dasar pemrograman **SOLID**.

Pertama bin utama, kita akan bahas apa yang dimaksud dengan **SOLID**. Yuk, lanjut!

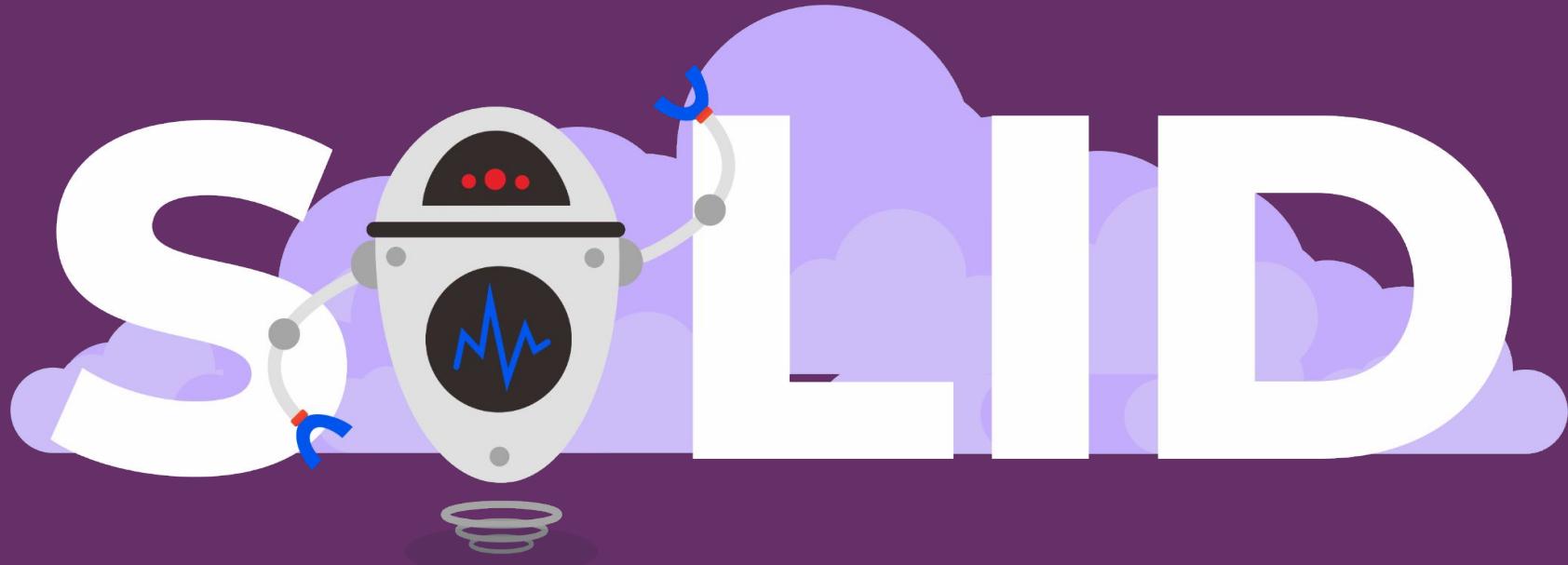




Detailnya, kita bakal bahas hal-hal berikut ini:

- Pengenalan Prinsip SOLID
- S - Single Responsibility Principle
- O - Open-Close Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle





Apa sih SOLID itu?

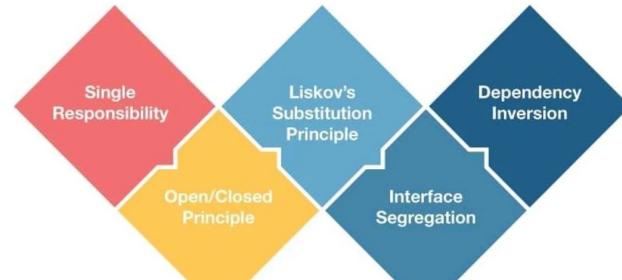


Apa sih SOLID Principle itu?

Di chapter 1, kita udah eksplor jauh banget tentang pemrograman Kotlin kan?

Banyak banget kode-kode yang mesti kita susun untuk ngebuaat satu fitur aplikasi. Saking banyaknya, bahkan kadang kita sendiri agak lama untuk nge-baca sendiri kodingan yang sudah kita buat.

S.O.L.I.D.



Nah, coba temen-temen bayangin, seandainya kalian kerja bareng tim developer lain, dan meneruskan pekerjaan mereka. Pasti bisa pusing sendiri 😅

Karenanya, butuh beberapa prinsip agar kerjaan developer nggak jadi terlalu pusing. Prinsip-prinsip ini lah yang disebut dengan SOLID Principle.





Sedikit ungkap sejarah tentang prinsip ini, kita bisa kenalan dengan tokoh disamping.

Bapak berambut putih dengan senyum istimewa ini merupakan ilmuwan komputer yang pertama kali memperkenalkan Prinsip SOLID, **Robert C. Martin** alias Paman Bob.

Walaupun paman Bob memperkenalkan prinsip ini melalui jurnalnya pada tahun 2000, namun beliau belum menggunakan akronim **SOLID**. Hingga kemudian akronim SOLID diperkenalkan oleh **Michael Feathers**.



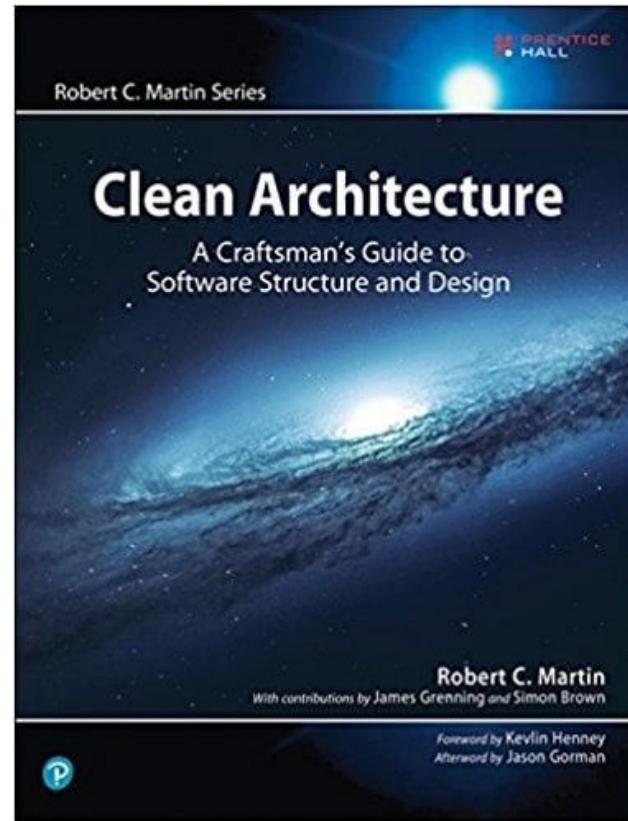


Gak cuma jurnal loh, ternyata ada buku nya juga~

Paman Bob juga menulis buku terlaris yaitu **“Clean Code dan Clean Architecture”**.

Itulah kenapa semua konsep clean code, object oriented arsitektur, dan design pattern ini entah bagaimana terhubung dan saling melengkapi satu sama lain.

Kayak kamu dan dia.

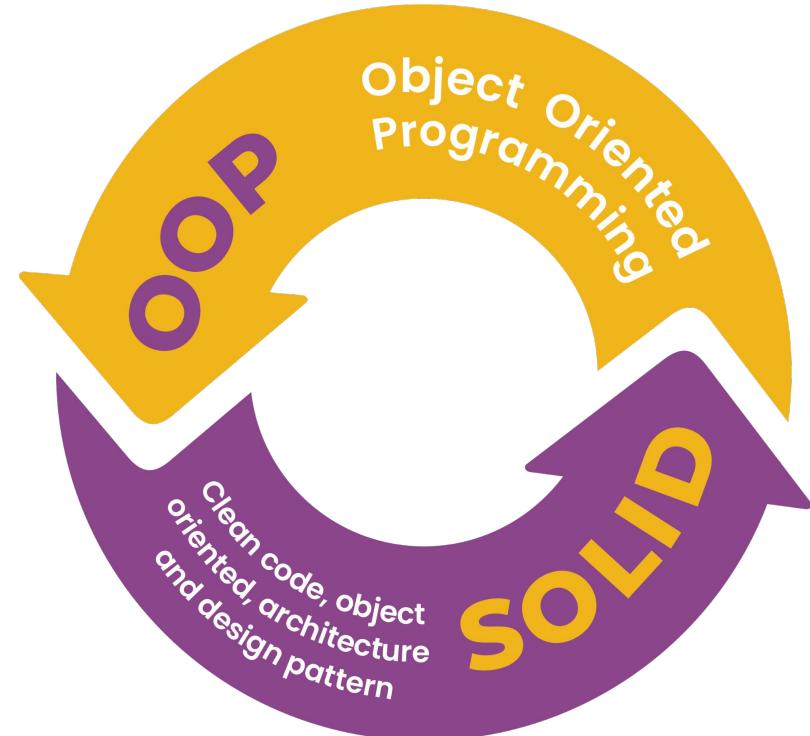




SOLID sangat terkait dengan OOP

Object Oriented Programming (OOP) dan SOLID akan saling melengkapi satu sama lain dan membuat sebuah syntax yang lebih rapi serta efisien sesuai dengan prinsip SOLID.

Kalau kamu penasaran sama OOP, kamu bisa cek di chapter sebelumnya ya. Kita udah sempet ngebahas itu, kok!





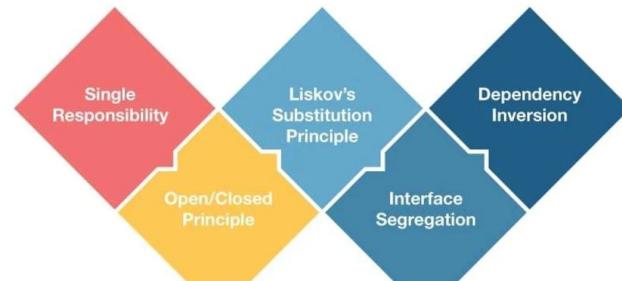
Kenapa SOLID?

Dalam 20 tahun terakhir, prinsip SOLID ini telah merevolusi dunia pemrograman, dan mengubah cara kita menulis kode. Sebenarnya apa sih tujuan dari SOLID itu sendiri?

1. Membuat kode yang mudah dipelihara
2. Mudah dipahami
3. Flexible dalam perubahan

Masih belum kebayang? Yuk kita main analogi lagi~

S.O.L.I.D.





Jadi, Gampangnya Gini ...

Bayangin, kamu adalah seorang pecinta buku. Ketika kamu membeli rumah baru, kamu menemukan ada satu ruangan yang cocok dijadikan ruang baca.

Supaya kamu bisa dengan mudah mencari buku, punya ruangan yang enak untuk baca, dan punya ruang teratur dan gampang untuk menambah koleksi buku kamu dengan mudah, apa yang kamu lakukan?

Tentunya **bikin ‘personal library’ dengan standar yang kamu mau kan?** Kurang lebih, itu lah fungsi nya **SOLID**





SOLID itu singkatan, looh~

Dengan menerapkan prinsip SOLID, kita bisa mengurangi kerumitan dan lebih adaptif dalam memahami perubahan aplikasi yang semakin variatif juga.

Istilahnya supaya nggak membagongkan, SOLID ini membantu kita :

- S - Single Responsibility Principle
- O - Open-Close Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle



Kamu udah tahu nih akronim dari kata SOLID. Sekarang kita bakal bahas singkatannya satu per satu.

Mulai dari "S" for SOLID, yaitu [Single Responsibility Principle](#).



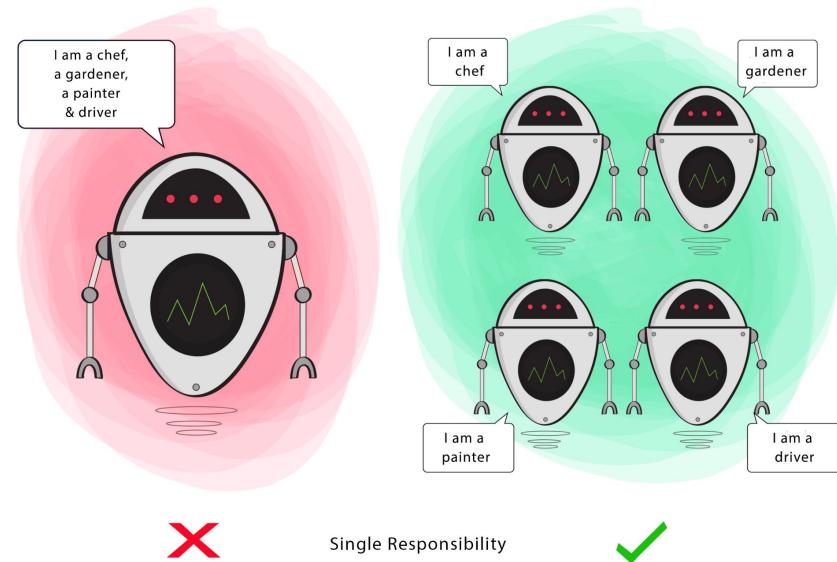


Single Responsibility Principle (SRP)

Kita mulai dari Single Responsibility Principle (SRP). SRP merupakan sebuah prinsip yang mudah diterapkan dalam pengembangan perangkat lunak.

Robert Martin menyimpulkan bahwa **“sebuah class harus memiliki satu dan hanya satu tanggung jawab”**.

Coba deh kamu lihat gambar disamping. Class yang ideal terdiri dari satu chef, satu gardener, satu driver dan satu painter. Namun bukan berarti semuanya punya fungsi yang terpisah.



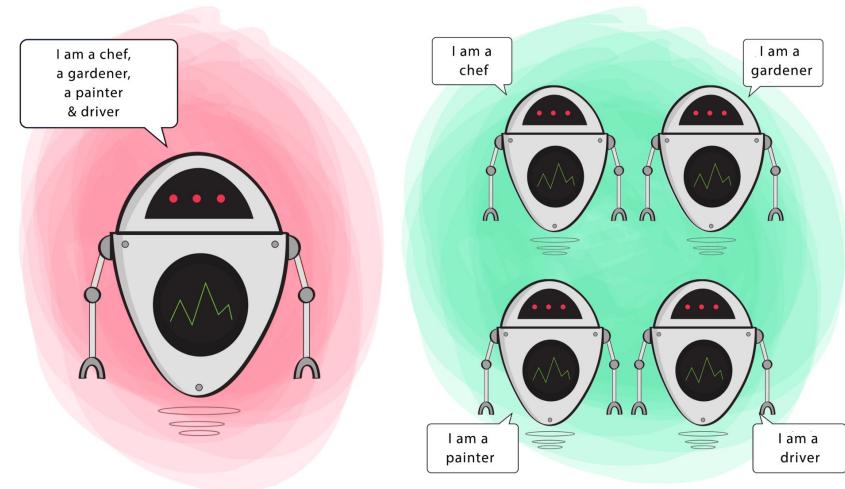
Single Responsibility



Walaupun satu class memiliki empat tanggung jawab/function berbeda, namun keempatnya itu harus saling berkaitan satu sama lain.

Semua function pada suatu class harus bekerja sama menuju satu tujuan.

SRP merupakan prinsip yang sederhana dan intuitif, tetapi dalam prakteknya terkadang sulit untuk menerapkannya dan banyak rintangan.



Single Responsibility





Contoh kasus!

Misalnya, kita membuat sebuah class untuk handle sebuah fungsi untuk **aktivitas login dan pendaftaran user** yang digambarkan seperti berikut :

```
● ● ●  
  
class Service {  
    fun login() {}  
    fun logout() {}  
    fun addToken() {}  
    fun getToken() {}  
    fun clearToken() {}  
}
```

Service

- + login()
- + logout()
- + addToken()
- + getToken()
- + clearToken()

Pada code di atas, class Service memiliki banyak tanggung jawab. Seperti, menambahkan token, mendapatkan token, login user, menghapus token, logout user.



Dengan menerapkan prinsip **SRP**, kita dapat memisahkan tanggung jawabnya dari masing-masing function ke dalam class yang berbeda. Hasilnya akan menjadi seperti berikut:

```
● ● ●  
class AuthService() {  
    fun login(token: AuthTokenStorage) {}  
    fun logout(token: AuthTokenStorage) {}  
}  
  
class AuthTokenStorage() {  
    fun addToken() {}  
    fun getToken() {}  
    fun clearToken() {}  
}
```

AuthService

+ login()
+ logout()

AuthTokenStorage

+ addToken()
+ getToken()
+ clearToken()

Pada kode di atas, kita memisahkannya menjadi dua class dengan tanggung jawab class yang berbeda.

Ketika ingin menyimpan token saat login, kita bisa memanggil class **AuthTokenStorage**. Sehingga, kita mendapatkan function menambahkan token, mendapatkan token, dan menghapus token.



Itu dia S for Single Responsibility Principle.
Selanjutnya kita menuju ke “O” for
[Open-Close Principle \(OCP\)](#).

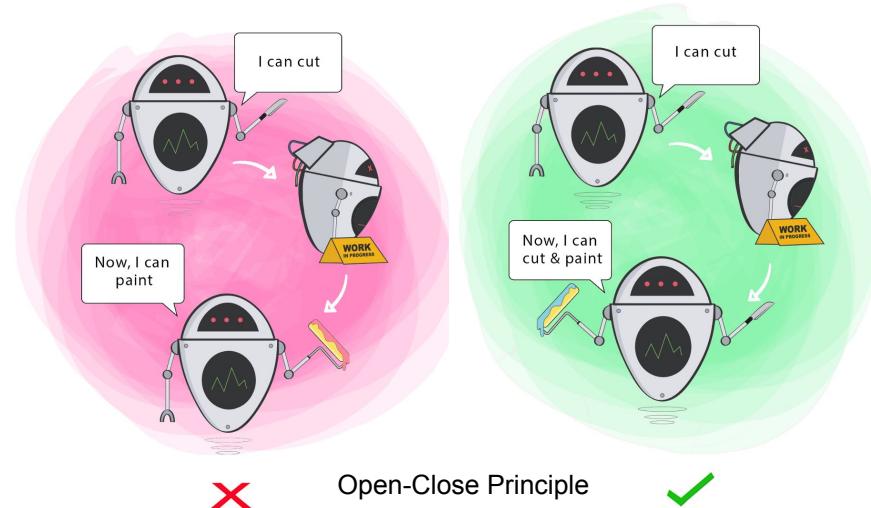
Apa sih OCP itu? Yuk kita intip~



Open-Close Principle (OCP)

Open-close principle adalah class yang terbuka dan sudah teruji dengan baik, namun sangat tidak direkomendasikan untuk modifikasi (mengubah kode pada class yang ada)

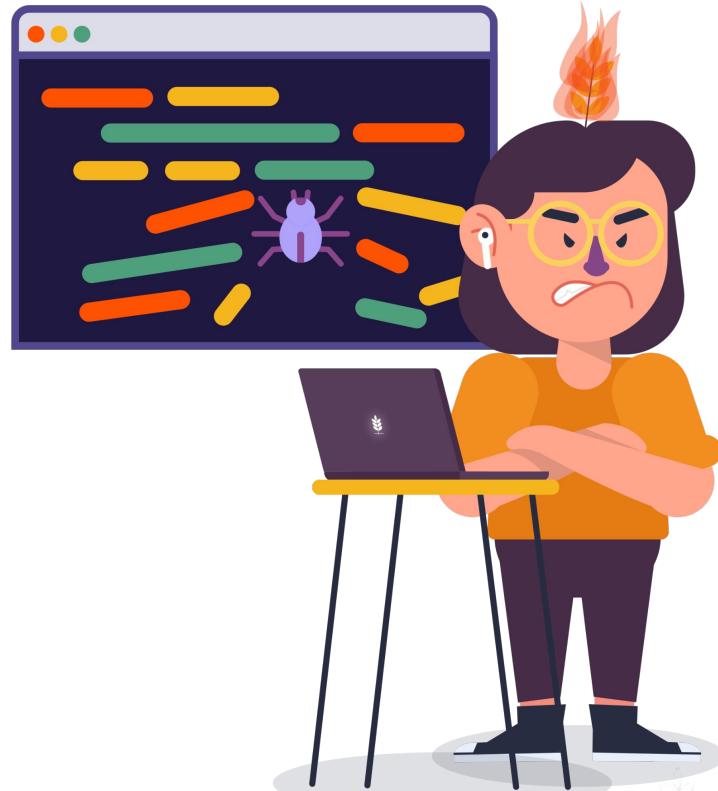
Sehingga **OCP terbuka untuk ditambahkan tetapi tertutup untuk modifikasi.** Maksudnya, "Kita harus menambahkan fungsionalitas baru tanpa menyentuh atau memodifikasi kode yang sudah ada pada class tersebut".





Ini karena setiap kali kita memodifikasi kode yang sudah ada, kita mengambil resiko menciptakan potensi bug.

Jadi kita perlu menghindari untuk menyentuh kode yang sudah ada.





OCP ini memang unik, karena bertentangan satu sama lain.

Tapi nggak perlu khawatir, karena saat kita bisa mengatur kebutuhan sistem dengan baik dan benar, dengan mudahnya aturan dari **OCP** bisa kita implementasikan.





Kayak ketika kamu pake Kotlin. Kita bisa menerapkan interface dan abstraksi class yang bertujuan untuk mempermudah perbaikan setelah pengembangan.

Tanpa harus mengganggu kelas yang mewarisi.

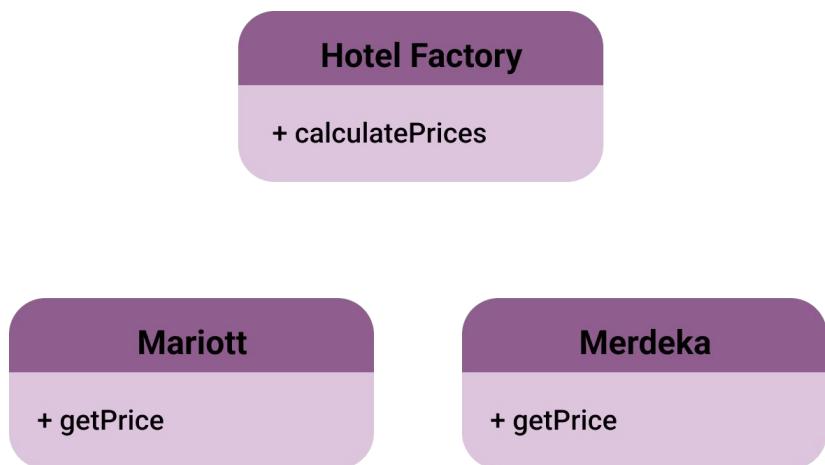
Adapun ketika ingin membuat fungsionalitas baru, cukup dengan membuat class baru dan mewarisi interface tersebut.





Contoh kasus!

Kita asumsikan bahwa kita akan membuat perhitungan biaya hotel dari beberapa Hotel yang ada di Indonesia.





```
● ● ●

class Mariott {
    private val basePrice = 2000
    private val tax = 500
    fun getPrice(): Int {
        return basePrice + tax
    }
}

class Merdeka {
    private val basePrice = 3000
    fun getPrice(): Int {
        return basePrice
    }
}

class HotelFactory {
    fun calculatePrices(hotels: List<Any>): Int {
        var price = 0
        hotels.forEach {
            price += when (it) {
                is Mariott -> it.getPrice()
                is Merdeka -> it.getPrice()
                else -> throw RuntimeException("Hotel Not Listed.")
            }
        }
        return price
    }
}

fun main() {
    print(HotelFactory().calculatePrices(listOf(Mariott(), Merdeka())))
}
```

Di sini kelas **HotelFactory** sedang memeriksa apakah Hotel itu Mariott atau Merdeka. Jika tidak, itu hanya mengembalikan info "**Hotel Tidak Terdaftar**".

Oleh karena itu setiap kali kita menambahkan Hotel baru, kita perlu menyentuh Kelas **PriceFactory** untuk mengubah kontennya yang melanggar Prinsip Open-Close.

Untuk menaati aturan Open-Close, kita harus mengubahnya seperti code selanjutnya



```
● ● ●

interface Hotel {
    fun getPrice(): Int
}

class Mariott : Hotel {
    private val basePrice = 2000
    private val tax = 500
    override fun getPrice(): Int {
        return basePrice + tax
    }
}

class Merdeka : Hotel {
    private val basePrice = 3000
    override fun getPrice(): Int {
        return basePrice
    }
}

class HotelFactory {
    fun calculatePrices(hotels: List<Hotel>): Int {
        var totalPrice = 0
        hotels.forEach {
            totalPrice += it.getPrice()
        }
        return totalPrice
    }
}

fun main() {
    print(PriceFactory().calculatePrices(listOf(Mariott(),Merdeka())))
}
```

Pada kode di samping, kita memanfaatkan interface sebagai penampung sebuah function yang mana function tersebut dapat dipanggil oleh class lain dengan mudahnya.

Dan untuk class yang memanggil interface tersebut, akan mewarisi semua function yang ada pada interface tersebut.

Nah, dengan begini, kita bisa meminimalisir error jika kita perlu memperbaiki kode~



Kita udah melewati huruf S dan O pada SOLID.

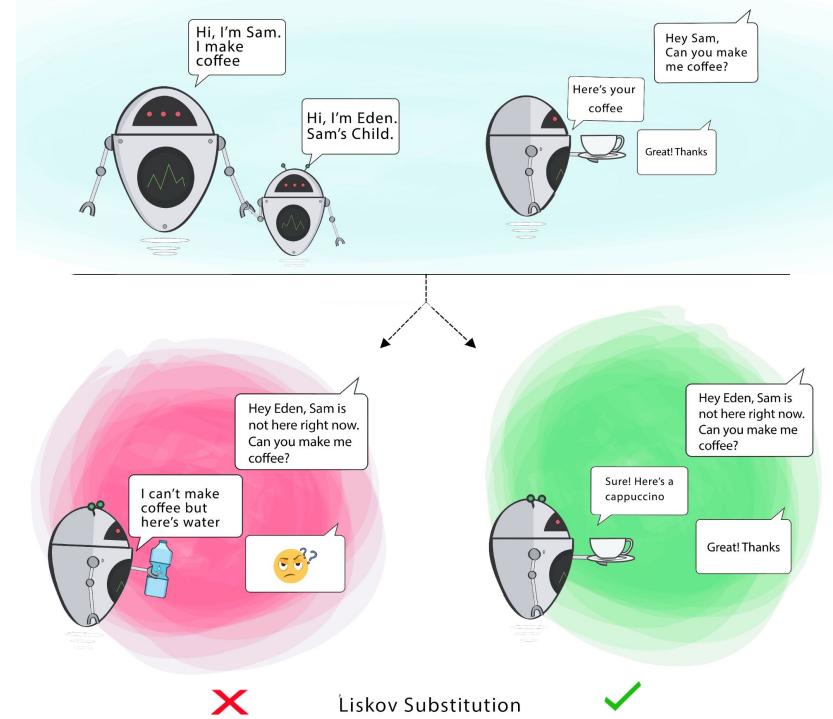
Sekarang kita menuju ke “L” for [Liskov Substitution Principle \(LSP\)](#).



Liskov Substitution Principle (LSP)

Prinsip ini dinamai oleh **Barbara Liskov** - seorang ilmuwan komputer yang cukup cerdas.

Ide dari prinsip ini adalah sebuah object harus dapat diganti dengan turunan subClass-nya **tanpa mengubah perilaku program**.

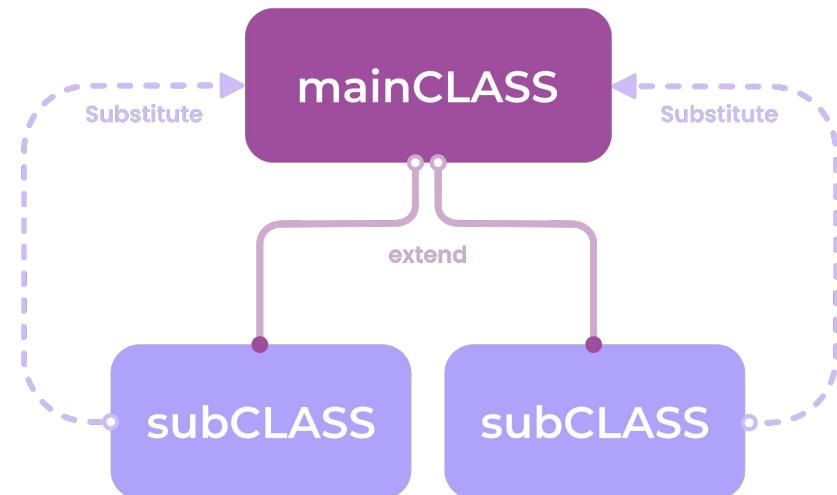




Konsep Dasarnya, gimana nih?

Kita harus dapat menggunakan subClass daripada mainClass yang telah di extend, tanpa perlu membuat sebuah perubahan apapun pada kode kita.

Singkatnya, **subClass harus dapat mensubstitusikan mainClass.**





Bagi kalian yang masih belum memahami apa itu substitusi, substitusi menurut KBBI adalah pengganti.

Jadi pada prinsip ini, subClass diharapkan dapat menggantikan mainClass. Ketika subClass di extend dari mainClass, mereka dapat **mewarisi perilaku dari mainClass.**





Kalau **subClass** belum bisa menirukan perilaku milik **mainClass**, mungkin kita belum menuliskan kode dengan benar.

Untuk menjadikan sebuah **class** benar menjadi **subClass**, **class** tersebut tidak hanya wajib untuk menerapkan fungsi dan properti dari **mainClass**.





4 Poin dalam **LSP**

Liskov Substitution
Principle

Beberapa aturan yang perlu diperhatikan dalam prinsip ini, ada di poin-poin berikut :

1. Contravariant dan Covariant
2. Preconditions dan Postconditions
3. Invariant
4. Constraint

Yuk kita bahas satu persatu~

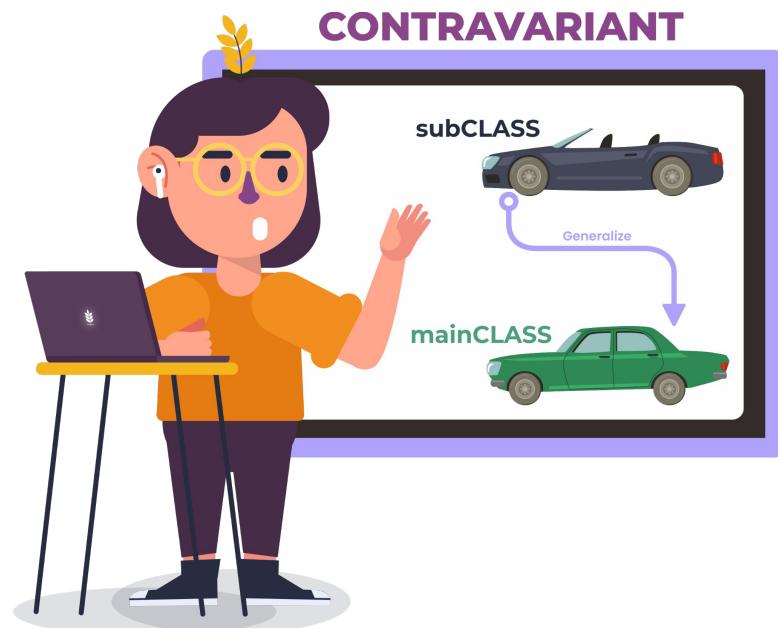




Pertama, Contravariant dan Covariant

Contravariance adalah **konversi class dari subclass ke class yang tingkat hirarkinya lebih tinggi**.

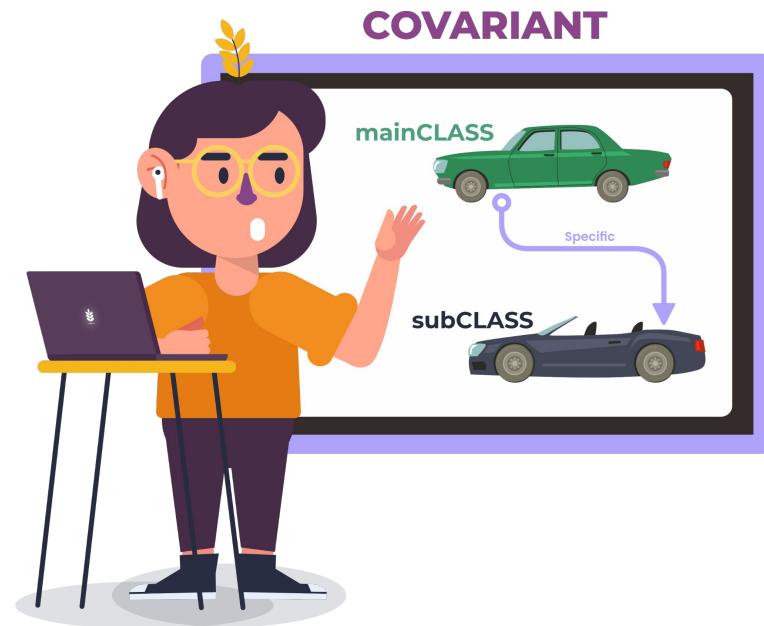
Gampangnya, seperti men-generalisir ‘Rolls Royce’ ke ‘Mobil’.





Covariance itu kebalikan dari Contravariance; yaitu **konversi class di hirarki tertinggi ke class yang lebih spesifik.**

Kalau ikut contoh kita sebelumnya, berarti saat kita ingin menspesifikasi permintaan mobil kita ke merk Rolls Royce.

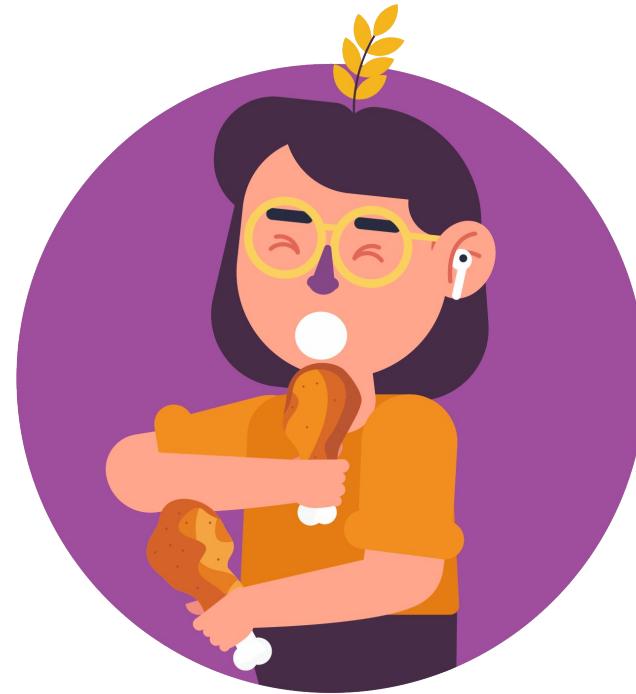




Kedua, Preconditions dan Postconditions

Merupakan sebuah tindakan yang perlu dilakukan **sebelum atau sesudah proses dijalankan.**

Contohnya, ketika kita memanggil sebuah function yang berfungsi untuk mendapatkan data dari server.



PRECONDITIONS



“Sebelumnya” adalah ketika kita harus memastikan data yang dibutuhkan oleh server sebelum ditampilkan, nah ini disebut dengan **preconditions**

“Setelahnya” adalah data yang diambil dari server dapat digunakan oleh kita untuk menampilkannya ke user, inilah yang disebut dengan **postconditions**.



POSTCONDITIONS



Ketiga, Invariant

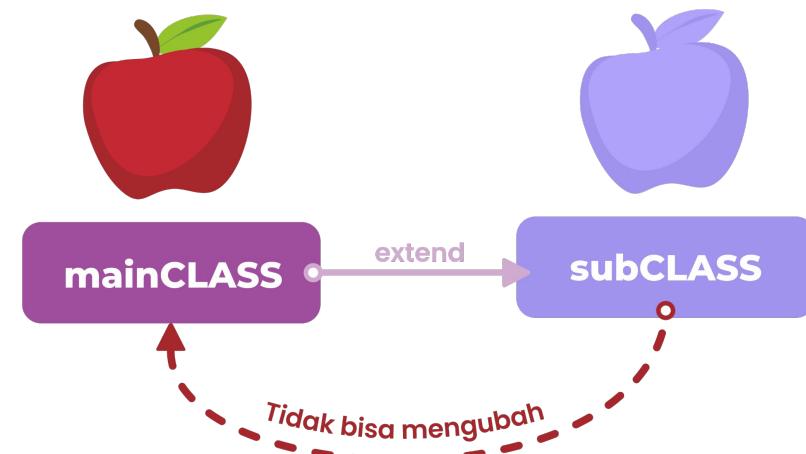
Invariant menggambarkan **kondisi proses yang benar, sebelum proses dimulai dan tetap benar setelahnya.**

Misalnya, class mungkin menyertakan function yang dapat membaca teks dari file.



Jika function tersebut menangani pembukaan dan penutupan file, invariant memastikan bahwa file tidak dibuka sebelum pemanggilan function atau sesudahnya.

Untuk mematuhi aturan LSP, invariant dari mainClass **tidak boleh diubah** oleh subClass.





Keempat, constraint.

Berdasarkan sifatnya, subClass mencakup semua function dan properti dari mainClass-nya. Mereka juga dapat menambahkan anggota selanjutnya.





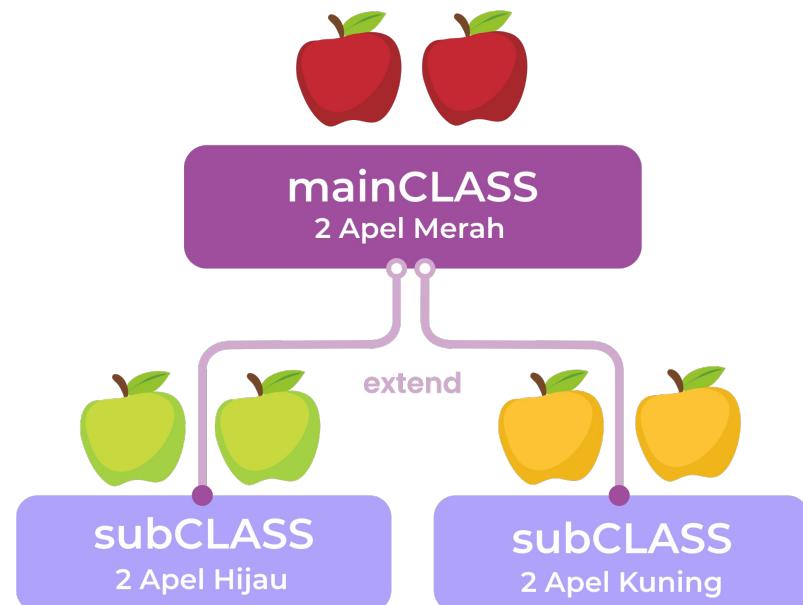
CONSTRAINT

Dalam Subclass, warna boleh berubah,
tetapi Jumlah harus sama

Constraint adalah **batasan yang ditentukan oleh mainClass**.

MainClass menentukan bahwa anggota baru atau yang dimodifikasi, tidak boleh mengubah status object dengan cara yang tidak diizinkan oleh mainClass.

Semisal class dasar mewakili object dengan ukuran yang tetap, subClass tidak diizinkan ukuran ini untuk diubah.



Contoh kasus!

Untuk lebih memahami LSP, mari kita simak contoh klasik yang mudah dipahami dengan pewarisan Persegi dan Persegi Panjang.

Kita akan membuat program yang membuat class bangun segi empat, dengan pemisahan class Persegi dan Persegi Panjang.

Yuk geser slide untuk cari tahu penerapan LSP dari kasus diatas!





Kode yang salah menerapkan Kaidah LSP

Kita ingin membuat sebuah program untuk menghitung luas persegi dan persegi panjang dengan menerapkan kaidah LSP. Di mana **subClass dapat menirukan behaviour yang ada pada mainClass**.

Pertama, dibuat sebuah **mainClass** dengan nama **Rectangle** yang menerapkan rumus luas persegi panjang.



```
open class Rectangle {  
    open var width: Int = 0  
    open var height: Int = 0  
    open fun area(): Int {  
        return width * height  
    }  
}  
  
class Square : Rectangle() {  
    override var width: Int  
        get() = super.width  
        set(width) {  
            super.width = width  
            super.height = width  
        }  
    override var height: Int  
        get() = super.height  
        set(height) {  
            super.width = height  
            super.height = height  
        }  
}  
  
fun main() {  
    val rectangleFirst: Rectangle = Rectangle()  
    rectangleFirst.width = 2  
    rectangleFirst.height = 3  
  
    val rectangleFirstText = rectangleFirst.area().toString()  
    println(rectangleFirstText) // output = 6  
  
    val rectangleSecond: Rectangle = Square()  
  
    rectangleSecond.width = 2  
    rectangleSecond.height = 3  
  
    val rectangleSecondText = rectangleSecond.area().toString()  
    println(rectangleSecondText) //Output = 9  
}
```



Setelahnya dibuat class baru dengan nama **Square**, yang menjadi subClass dan mensubstitusikan **mainClass Rectangle**, Rumusnya pun sama, sisi x sisi.

Disini, **class Square** didefinisikan bahwa width dan height menjadi sisi (height)



```
open class Rectangle {  
    open var width: Int = 0  
    open var height: Int = 0  
    open fun area(): Int {  
        return width * height  
    }  
}  
  
class Square : Rectangle() {  
    override var width: Int  
        get() = super.width  
        set(width) {  
            super.width = width  
            super.height = width  
        }  
  
    override var height: Int  
        get() = super.height  
        set(height) {  
            super.width = height  
            super.height = height  
        }  
}  
  
fun main() {  
    val rectangleFirst: Rectangle = Rectangle()  
    rectangleFirst.width = 2  
    rectangleFirst.height = 3  
  
    val rectangleFirstText = rectangleFirst.area().toString()  
    println(rectangleFirstText) // output = 6  
  
    val rectangleSecond: Rectangle = Square()  
  
    rectangleSecond.width = 2  
    rectangleSecond.height = 3  
  
    val rectangleSecondText = rectangleSecond.area().toString()  
    println(rectangleSecondText) //Output = 9  
}
```



Di sini, kita benar menirukan konsep LSP. Sayangnya ketika kita menjalankan programnya, hasilnya akan salah.

Hasil yang akan didapat ketika pada **Rectangle** akan benar menjadi 6.

Tetapi hasil **Square** akan salah, karena menjadi 9. Padahal yang kita inputkan adalah sama-sama 3 dan 2.

Oleh karena itu kita harus membenarkan program yang kita buat menjadi lebih baik lagi.



```
open class Rectangle {  
    open var width: Int = 0  
    open var height: Int = 0  
    open fun area(): Int {  
        return width * height  
    }  
  
    class Square : Rectangle() {  
        override var width: Int  
            get() = super.width  
            set(width) {  
                super.width = width  
                super.height = width  
            }  
        override var height: Int  
            get() = super.height  
            set(height) {  
                super.width = height  
                super.height = height  
            }  
    }  
  
    fun main() {  
        val rectangleFirst: Rectangle = Rectangle()  
        rectangleFirst.width = 2  
        rectangleFirst.height = 3  
  
        val rectangleFirstText = rectangleFirst.area().toString()  
        println(rectangleFirstText) // output = 6  
  
        val rectangleSecond: Rectangle = Square()  
  
        rectangleSecond.width = 2  
        rectangleSecond.height = 3  
  
        val rectangleSecondText = rectangleSecond.area().toString()  
        println(rectangleSecondText) //Output = 9  
    }  
}
```



Kode yang benar nya begini....

Pada kode yang diperbaiki, kita membuat sebuah **mainClass** baru dengan nama **Shape** dan membuat sebuah **abstract fun** dengan nama **area**.

```
abstract class Shape {  
    abstract fun area(): Int  
}  
  
class Rectangle(var width: Int, var height: Int) : Shape() {  
    override fun area(): Int {  
        return width * height  
    }  
}  
  
class Square(var edge: Int) : Shape() {  
    override fun area(): Int {  
        return edge * edge  
    }  
}  
  
fun main() {  
    val rectangleFirst: Shape1 = Rectangle1(2, 3)  
    val rectangleSecond: Shape1 = Square1(3)  
  
    val rectangleFirstText = rectangleFirst.area().toString()  
    println(rectangleFirstText)  
    val rectangleSecondText = rectangleSecond.area().toString()  
    println(rectangleSecondText)  
}
```



Selanjutnya pada **subClass Rectangle** dan **Square** akan mensubstitusikan **mainClass Shape**.

Dimana di dalamnya akan menerapkan rumusnya masing-masing dan juga mendefinisikan kebutuhan constructor-nya masing-masing.

Jika kita menjalankan programnya, maka hasil yang dikeluarkan akan benar sesuai dengan rumus luas Persegi dan Persegi Panjang sesuai yang kita semua ketahui.

```
abstract class Shape {  
    abstract fun area(): Int  
}  
  
class Rectangle(var width: Int, var height: Int) : Shape() {  
    override fun area(): Int {  
        return width * height  
    }  
}  
  
class Square(var edge: Int) : Shape() {  
    override fun area(): Int {  
        return edge * edge  
    }  
}  
  
fun main() {  
    val rectangleFirst: Shape1 = Rectangle1(2, 3)  
    val rectangleSecond: Shape1 = Square1(3)  
  
    val rectangleFirstText = rectangleFirst.area().toString()  
    println(rectangleFirstText)  
    val rectangleSecondText = rectangleSecond.area().toString()  
    println(rectangleSecondText)  
}
```

Pemahaman LSP akan kita telusuri lebih dalam pada hari selanjutnya.

To Be Continued

Next day~





Gengs, kita udah sampai sini! Wah.. sebelum lanjut lagi kita pantun dulu~

Stroberi mangga tomat. Mari kita semangat! Cakep.

Sekarang kita menuju “I” for Interface Segregation Principle (ISP).



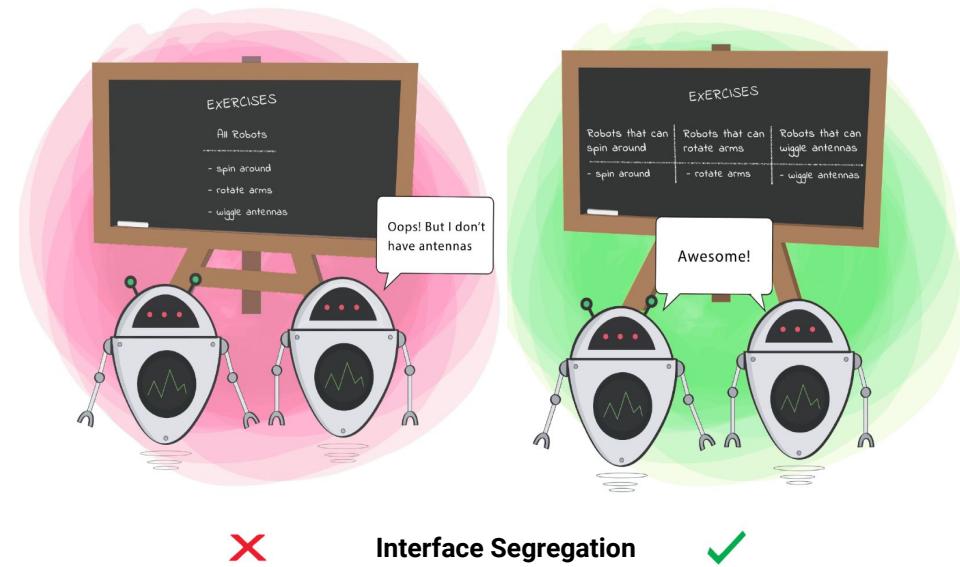


Interface Segregation Principle (ISP)

Prinsip ini dikemukakan oleh **Robert Cecil Martin** dalam bukunya yang berjudul Design Principle.

Om Robert memberi petuah bahwa :

“Clients should not be forced to depend on methods that they do not use.”

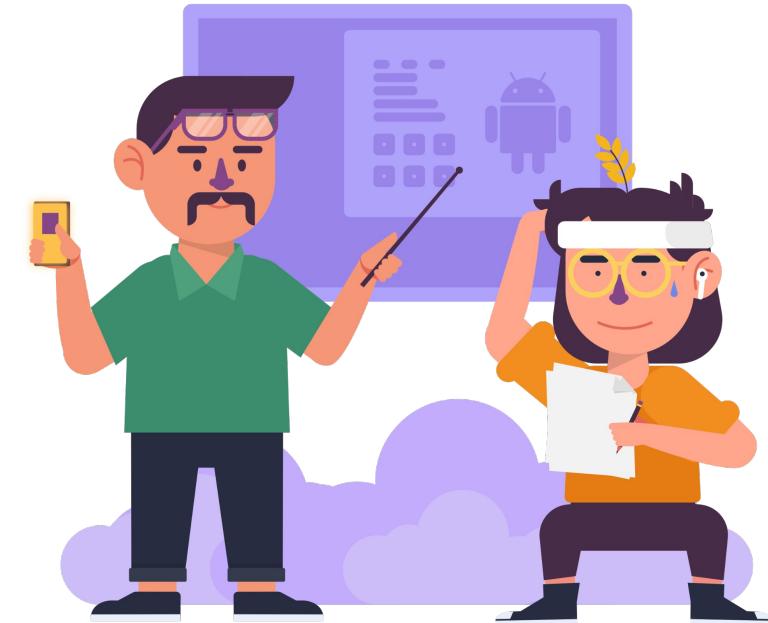




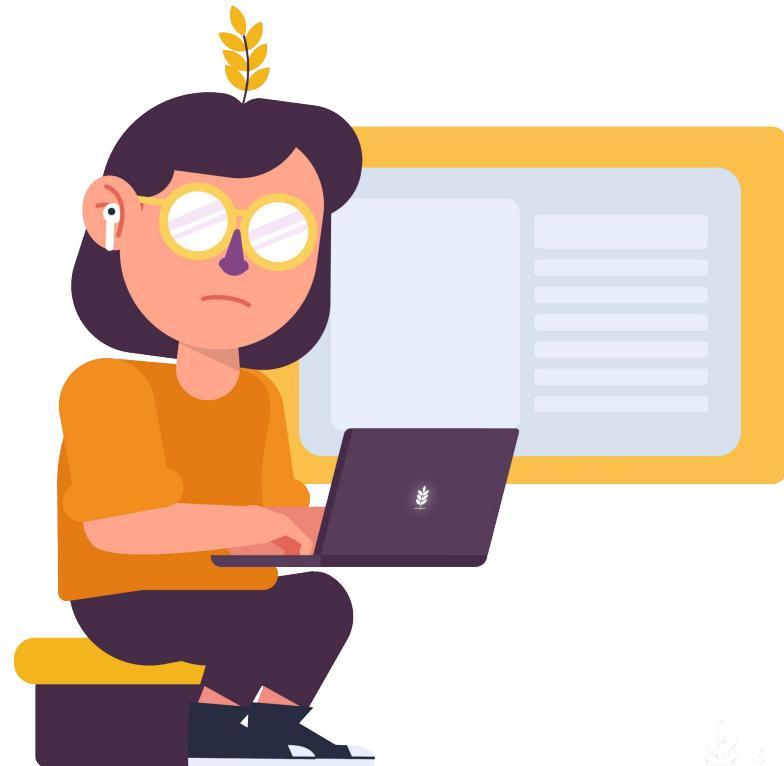
“Clients should not be forced to depend on methods that they do not use.”

Maksud dari kata-kata di atas adalah :

“Ketika kita membuat sebuah program dan memisahkannya ke dalam interface, maka kita harus membuat beberapa interface melalui beberapa function yang hanya sesuai dengan kebutuhan”.



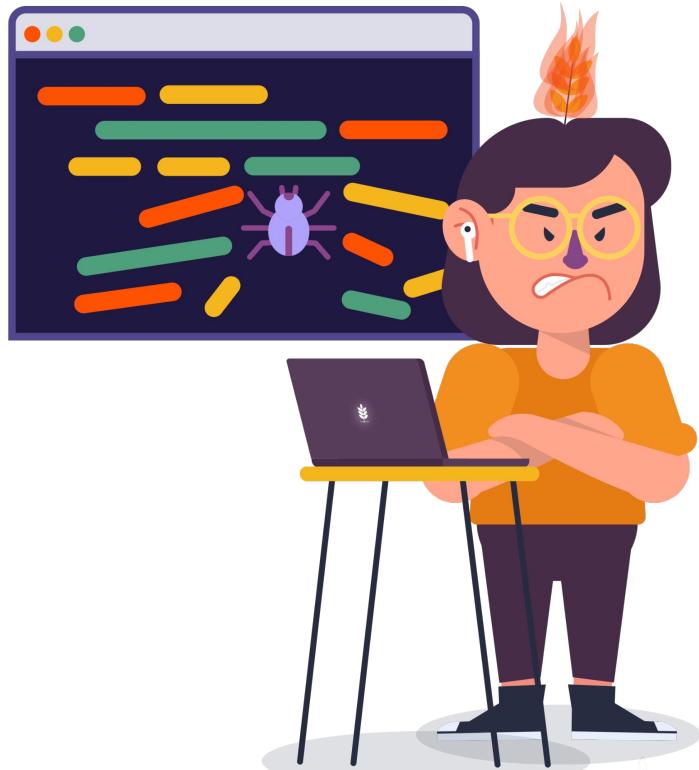
Terkadang ketika kita membuat sebuah aplikasi dengan jumlah function yang banyak, class lain yang bergantung pada class tersebut sebenarnya hanya membutuhkan satu atau dua function saja.





Hal ini mungkin memang efisien bagi kita karena tidak perlu mendeklarasikan class baru setiap membuat sebuah kumpulan function.

Tapi ternyata hal tersebut tidak disarankan oleh prinsip ini, karena nantinya **akan membuat sebuah interface semakin gemuk dan menyusahkan kita ketika melakukan maintenance dan bug fixing 😔**





Contoh Kodingnya?

Biar lebih paham, kita coba buat contoh penyusunan kodenya yah~

Bayangin, kita bikin sebuah game, dimana di dalamnya, kita akan membuat interface hewan dengan perilaku tertentu.

Class yang akan kita buat disini adalah 'kucing' dan 'burung'





Contoh Kode yang Keliru ..

Coba deh cek contoh disamping👉

Pada interface Animal Dibawahnya, dibuat 3 function yang menggambarkan perilaku hewan (eat, sleep, dan fly). Kemudian dibuat lagi dua class Cat and Bird.

Tapi, setelah dibuat, kita baru sadar bahwa kucing tidak bisa terbang. Menurut konsep ISP, cara tersebut kurang tepat.

Terus, gimana dong?

Karena kucing tidak bisa terbang, harusnya function ini tidak ada.

```
● ● ●\n\ninterface Animal {\n    fun eat()\n    fun sleep()\n    fun fly()\n}\n\nclass Cat : Animal {\n    override fun eat() {\n        println("Cat is eating fish")\n    }\n    override fun sleep() {\n        println("Cat is sleeping on its owner's bed")\n    }\n    override fun fly() {\n        TODO("Not yet implemented") // Cats can't fly\n    }\n}\n\nclass Bird : Animal {\n    override fun eat() {\n        println("Bird is eating forage")\n    }\n    override fun sleep() {\n        println("Bird is sleeping in the nest")\n    }\n    override fun fly() {\n        println("Bird is flying so high")\n    }\n}
```





Kita tidak perlu menerapkan function terbang untuk hewan yang sudah jelas tidak dapat terbang.

kita tinggal membuat interface baru untuk hewan yang terbang dan menghapus function terbang dari interface hewan.

Penambahan interface FlyingAnimal,
sehingga function dari tiap interface lebih
efisien



```
interface Animal{
    fun eat()
    fun sleep()
}

interface FlyingAnimal{
    fun fly()
}

class Cat: Animal{
    override fun eat() {
        println("Cat is eating fish")
    }
    override fun sleep() {
        println("Cat is sleeping on its owner's bed")
    }
}

class Bird: Animal, FlyingAnimal{
    override fun eat() {
        println("Bird is eating forage")
    }

    override fun sleep() {
        println("Bird is sleeping in the nest")
    }

    override fun fly() {
        println("Bird is flying so high")
    }
}
```



Itu dia I dari SOLID... setelah ini kita punya satu huruf terakhir!

Menuju ke “D” for [Dependency Inversion Principle \(DIP\)](#).

Apa sih DIP itu? Yuk kita intip~



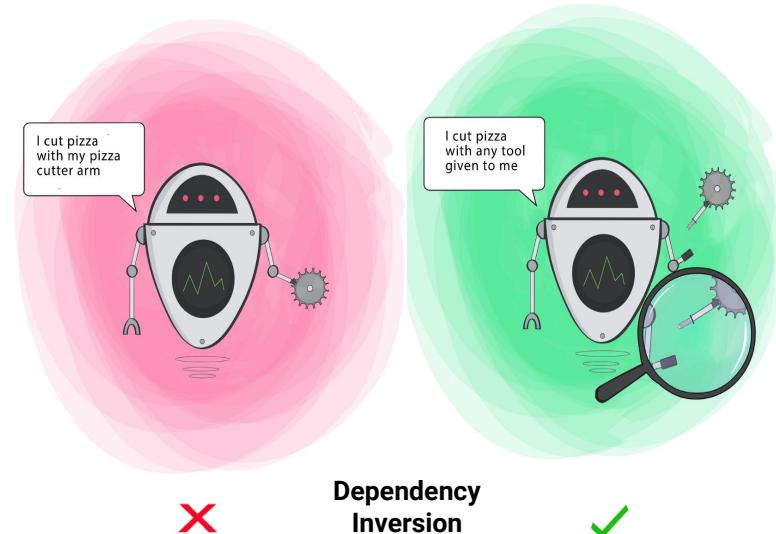


Dependency Inversion Principle (DIP)

Prinsip terakhir dalam SOLID ini juga merupakan prinsip yang diberikan oleh **Robert C. Martin**, beliau mengatakan:

- 1. High-level modules should not depend on low-level modules. Both should depend on abstractions.**
- 2. Abstractions should not depend on details. Details should depend on abstractions.**

Supaya lebih kebayang, kita bahas di slide selanjutnya~



Dependency
Inversion



Apa sih maksudnya? 🤔

Di atas Robert C. Martin menjelaskan tentang high-level module, low-level module, dan juga abstraction.

Lantas apa sih sebenarnya itu?

- **High-level Module** : Class yang menjalankan aksi dengan menggunakan alat.
- **Low-level Module** : Alat yang diperlukan untuk menjalankan sebuah tindakan.
- **Abstraction** : Sebuah interface yang menghubungkan dua class.

01

**HIGH LEVEL
MODULE**

02

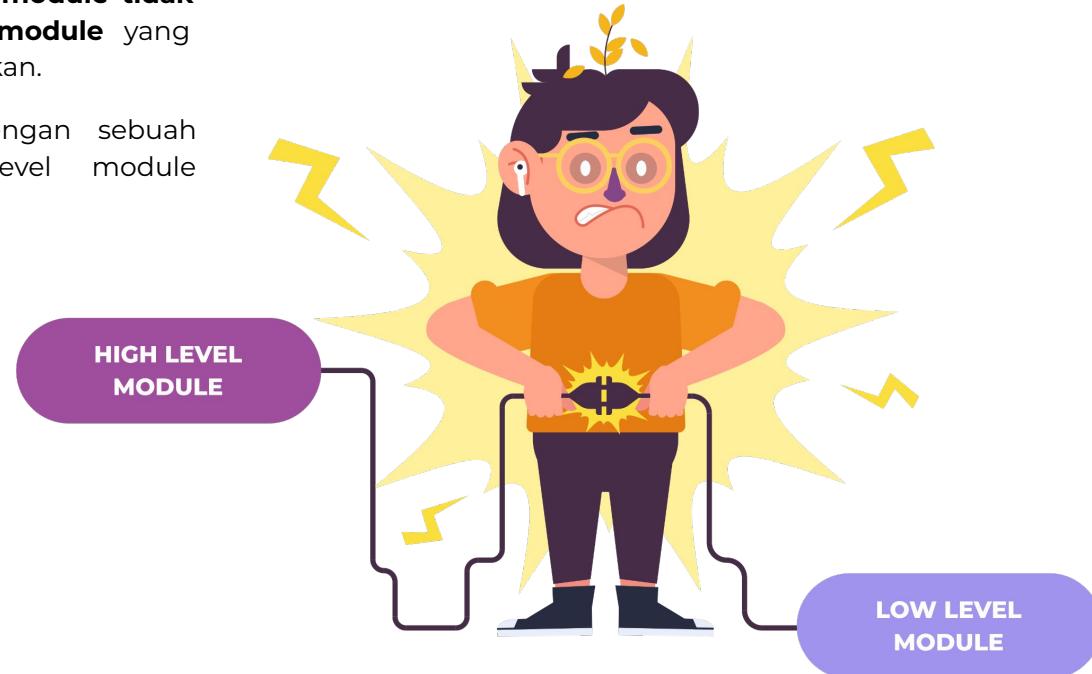
ABSTRACTION

03

**LOW LEVEL
MODULE**

Prinsip ini mengatakan bahwa **high-level module tidak boleh digabungkan dengan low-level module** yang digunakan untuk menjalankan suatu tindakan.

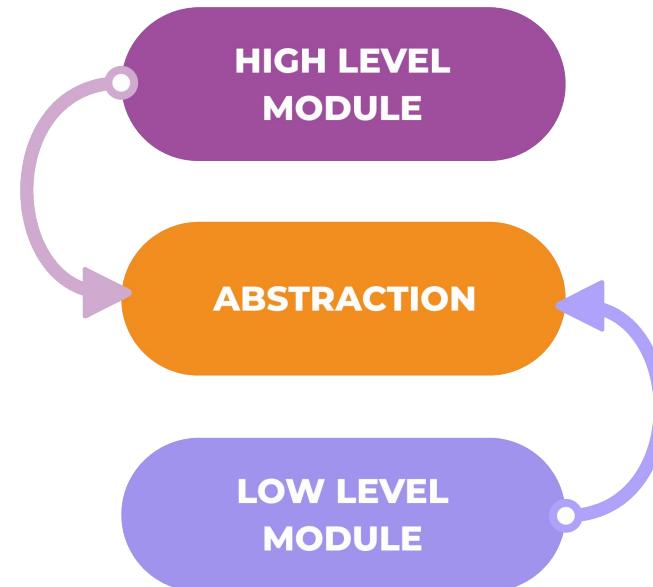
Sebaliknya, mereka harus menyatu dengan sebuah interface yang memungkinkan low-level module terhubung dengan high-level module.





Beliau juga mengatakan bahwa high-level module dan abstraction/interface seharusnya **tidak mengetahui cara kerja low-level module.**

Namun, low-level module tersebut harus memenuhi spesifikasi abstraction.





Kalau disimpulkan, DIP itu ...

Mengarahkan developer untuk mengatur ketergantungan antar module, dalam menggunakan abstraction/interface. Sehingga DIP bisa dibilang menawarkan cara untuk memisahkan module pada perangkat lunak.

Tujuan dari prinsip ini adalah untuk **mengurangi ketergantungan high-level module pada low-level module dengan memperkenalkan abstraction/interface**.



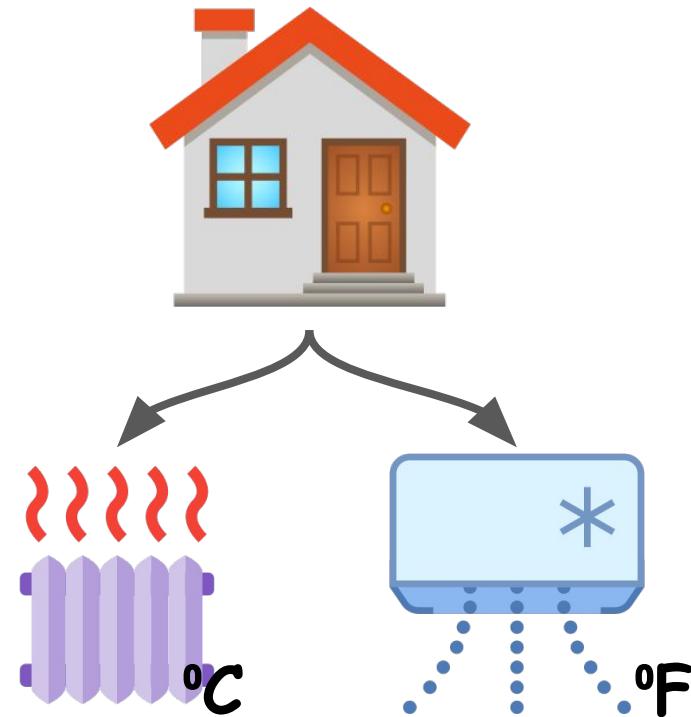


Contoh kasus!

Mari kita asumsikan bahwa kita ingin membuat function untuk menjalankan sebuah metode untuk menghangatkan ruangan.

Kita akan membuat sebuah kelas dengan nama **SmartHome** yang berfungsi untuk memanggil metode tersebut.

Lalu kita membutuhkan kelas lainnya yaitu **AC (AirConditioner)** dan **Radiator** yang digunakan untuk mengeksekusi sebuah fungsi untuk menyalaakan alat tersebut. AC menggunakan satuan suhu Fahrenheit ($^{\circ}\text{F}$) dan Radiator menggunakan satuan suhu Celcius ($^{\circ}\text{C}$).





Contoh yang Keliru...

Pada contoh di samping, kita mendeklarasikan 2 buah kelas dengan nama Radiator dan AirConditioner

Didalamnya terdapat sebuah function dengan nama turnOnTools. Function tersebut mengatur cara menyalakan tools, baik itu Radiator maupun AirConditioner



```
● ● ●

class Radiator {
    var temperatureCelsius: Int = 0
    fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureCelsius = newTemperatureCelsius
        TODO("not implemented")
    }
}

class AirConditioner {
    var temperatureFahrenheit: Int = 0
    fun turnOnTools(newTemperatureFahrenheit: Int) {
        temperatureFahrenheit = newTemperatureFahrenheit
        TODO("not implemented")
    }
}

class SmartHome {

    var radiator: Radiator = Radiator()
    var recommendedTemperatureCelsius: Int = 20

    fun warmUpRoom() {
        radiator.turnOnTools(recommendedTemperatureCelsius)
    }
}
```



Tapi, kebutuhan dari class tersebut berbeda, dimana Radiator menggunakan suhu Celcius dan AirConditioner menggunakan Fahrenheit.

Karenanya, function turnOnTools untuk Radiator dan AirCondiner perlu dibedakan.

Dua Class ini memiliki fungsi turnOnTools. Namun, prosedurnya berbeda, karena satuan suhunya berbeda

```
class Radiator {  
    var temperatureCelsius: Int = 0  
    fun turnOnTools(newTemperatureCelsius: Int) {  
        temperatureCelsius = newTemperatureCelsius  
        TODO("not implemented")  
    }  
}  
  
class AirConditioner {  
    var temperatureFahrenheit: Int = 0  
    fun turnOnTools(newTemperatureFahrenheit: Int) {  
        temperatureFahrenheit = newTemperatureFahrenheit  
        TODO("not implemented")  
    }  
}  
  
class SmartHome {  
  
    var radiator: Radiator = Radiator()  
  
    var recommendedTemperatureCelsius: Int = 20  
  
    fun warmUpRoom() {  
        radiator.turnOnTools(recommendedTemperatureCelsius)  
    }  
}
```



Pada class SmartHome, kita bisa memanggil class Radiator dan memasukkan nilai suhu yang ingin kita masukkan, dan sebuah function untuk melakukan pemanggilannya.

Tapi jika kita mau ubah radiator jadi AC, apa yang terjadi?



```
● ● ●

class Radiator {
    var temperatureCelsius: Int = 0
    fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureCelsius = newTemperatureCelsius
        TODO("not implemented")
    }
}

class AirConditioner {
    var temperatureFahrenheit: Int = 0
    fun turnOnTools(newTemperatureFahrenheit: Int) {
        temperatureFahrenheit = newTemperatureFahrenheit
        TODO("not implemented")
    }
}

class SmartHome {
    var radiator: Radiator = Radiator()
    var recommendedTemperatureCelsius: Int = 20
    fun warmUpRoom() {
        radiator.turnOnTools(recommendedTemperatureCelsius)
    }
}
```



Gambarannya pakai kode di bawah ini :

```
var airConditioner: AirConditioner = AirConditioner()
```

Kelas SmartHome ini bergantung pada kelas Radiator dan melanggar Prinsip Open-Close.

Fitur SmartHome kita menggunakan Radiator untuk pemanas ruangan.

Function yang dipanggil ada turnOnTools, yang memberi perintah memanaskan ruangan hingga 20 °C.

Nah, apa yang terjadi jika Radiator kita ganti dengan AirConditioner?



```
class Radiator {  
    var temperatureCelsius: Int = 0  
    fun turnOnTools(newTemperatureCelsius: Int) {  
        temperatureCelsius = newTemperatureCelsius  
        TODO("not implemented")  
    }  
  
    class AirConditioner {  
        var temperatureFahrenheit: Int = 0  
        fun turnOnTools(newTemperatureFahrenheit: Int) {  
            temperatureFahrenheit = newTemperatureFahrenheit  
            TODO("not implemented")  
        }  
  
        class SmartHome {  
            var radiator: Radiator = Radiator()  
            var recommendedTemperatureCelsius: Int = 20  
            fun warmUpRoom() {  
                radiator.turnOnTools(recommendedTemperatureCelsius)  
            }  
        }  
    }  
}
```



Jika kita memutuskan untuk mengabaikan prinsip DIP, mungkin **akan terjadi beberapa kesalahan penting**.

Kayak gini, di sini kita mengirimkan suhu yang disarankan dalam celsius tetapi metode AC yang kita buat mengharapkan untuk mendapatkannya dalam Fahrenheit.

Kalau kita ganti dari Radiator jadi AC, maka kode ini berubah:

```
AirConditioner.turnOnTools(disarankanTemperatureFarenheit)
```



```
class Radiator {  
    var temperatureCelsius: Int = 0  
    fun turnOnTools(newTemperatureCelsius: Int) {  
        temperatureCelsius = newTemperatureCelsius  
        TODO("not implemented")  
    }  
}  
  
class AirConditioner {  
    var temperatureFahrenheit: Int = 0  
    fun turnOnTools(newTemperatureFahrenheit: Int) {  
        temperatureFahrenheit = newTemperatureFahrenheit  
        TODO("not implemented")  
    }  
}  
  
class SmartHome {  
    var radiator: Radiator = Radiator()  
  
    var recommendedTemperatureCelsius: Int = 20  
  
    fun warmUpRoom() {  
        radiator.turnOnTools(recommendedTemperatureCelsius)  
    }  
}
```





Contoh yang tepat

Disini, kita membuat tambahan dalam function turnOnTools yang mengkonversikan nilai Farenheit ke Celcius

```
interface Heating {
    fun turnOnTools(newTemperatureCelsius: Int)
}

class Radiator : Heating {
    var temperatureCelsius: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureCelsius = newTemperatureCelsius
        TODO("not implemented")
    }
}

class AirConditioner : Heating {
    var temperatureFahrenheit: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureFahrenheit = newTemperatureCelsius * 9 / 5 + 32
        TODO("not implemented")
    }
}

class SmartHome {
    var radiator: Heating = Radiator()
    var recommendedTemperatureCelsius: Int = 20

    fun warmUpRoom() {
        radiator.turnOnTools(recommendedTemperatureCelsius)
    }
}
```

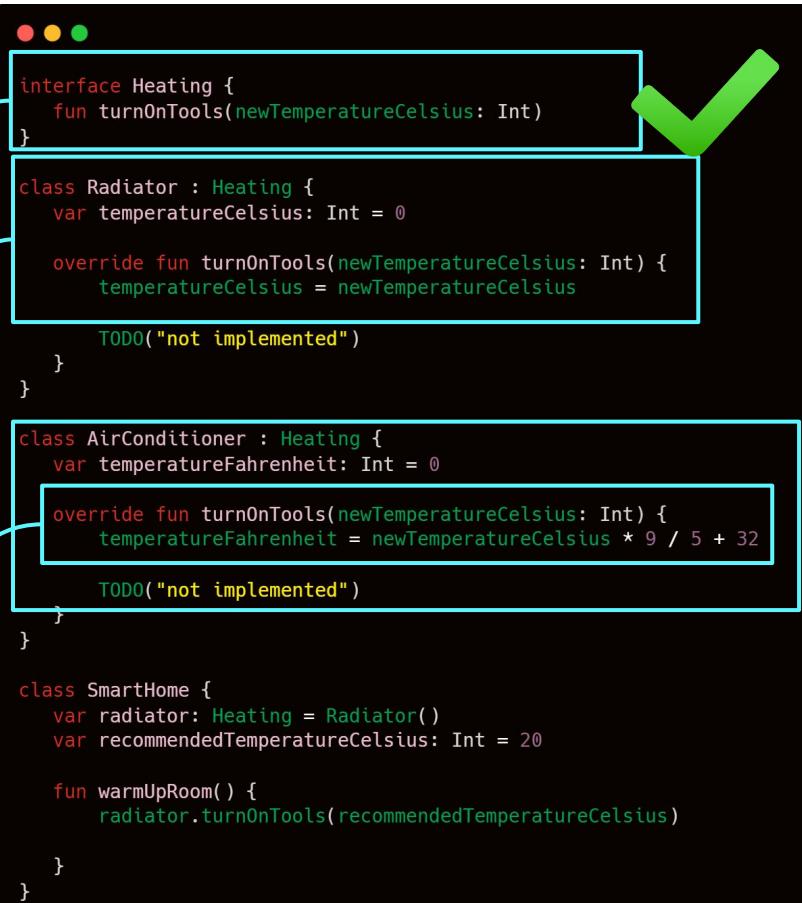




Di sini dibuat interface terlebih dahulu untuk menghandle function turnOnTools yang nantinya akan dipanggil di masing2 class Radiator dan AC.

Di class Radiator akan menjalankan function sesuai dengan prosedur penggunaannya.

Di class AC, terdapat konversi nilai celcius ke fahrenheit terlebih dahulu.



```
interface Heating {
    fun turnOnTools(newTemperatureCelsius: Int)
}

class Radiator : Heating {
    var temperatureCelsius: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureCelsius = newTemperatureCelsius
    }

    TODO("not implemented")
}

class AirConditioner : Heating {
    var temperatureFahrenheit: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureFahrenheit = newTemperatureCelsius * 9 / 5 + 32
    }

    TODO("not implemented")
}

class SmartHome {
    var radiator: Heating = Radiator()
    var recommendedTemperatureCelsius: Int = 20

    fun warmUpRoom() {
        radiator.turnOnTools(recommendedTemperatureCelsius)
    }
}
```



Dengan cara ini, ketika kita input satu value ke dalam dua class tersebut, value tersebut akan memberikan output yang benar karena kita sudah me-handlenya di dalam interface



```
interface Heating {
    fun turnOnTools(newTemperatureCelsius: Int)
}

class Radiator : Heating {
    var temperatureCelsius: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureCelsius = newTemperatureCelsius
        TODO("not implemented")
    }
}

class AirConditioner : Heating {
    var temperatureFahrenheit: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureFahrenheit = newTemperatureCelsius * 9 / 5 + 32
        TODO("not implemented")
    }
}

class SmartHome {
    var radiator: Heating = Radiator()
    var recommendedTemperatureCelsius: Int = 20

    fun warmUpRoom() {
        radiator.turnOnTools(recommendedTemperatureCelsius)
    }
}
```





Untuk sistem kontrol rumah pintar, kita akan menambahkan kontrol untuk radiator.

Sekarang kita punya jawaban untuk sebuah pertanyaan "**apa jadinya jika nanti kita memutuskan untuk mengganti radiator ke AC**".

Kelas kita sudah bergantung pada interface daripada kelas lain yang di extend.

```
var airConditioner: Heating = AirConditioner()
```

Karena kita bergantung pada antarmuka yang umum, tidak ada lagi peluang untuk salah.



```
interface Heating {
    fun turnOnTools(newTemperatureCelsius: Int)
}

class Radiator : Heating {
    var temperatureCelsius: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureCelsius = newTemperatureCelsius
        TODO("not implemented")
    }
}

class AirConditioner : Heating {
    var temperatureFahrenheit: Int = 0

    override fun turnOnTools(newTemperatureCelsius: Int) {
        temperatureFahrenheit = newTemperatureCelsius * 9 / 5 + 32
        TODO("not implemented")
    }
}

class SmartHome {
    var radiator: Heating = Radiator()
    var recommendedTemperatureCelsius: Int = 20

    fun warmUpRoom() {
        radiator.turnOnTools(recommendedTemperatureCelsius)
    }
}
```

Saatnya kita Quiz!





1. Di bawah ini, adalah beberapa pernyataan yang tepat terkait SOLID Principle, kecuali...

- A. Membuat kode yang mudah dipelihara
- B. Membuat kode yang mudah dipahami
- C. Membuat Kode yang Solid satu sama lain



1. Di bawah ini, adalah beberapa pernyataan yang tepat terkait SOLID Principle, kecuali...

- A. Membuat kode yang mudah dipelihara
- B. Membuat kode yang mudah dipahami
- C. Membuat Kode yang Solid satu sama lain

Membuat Kode yang Solid satu sama lain bukanlah tujuan dari prinsip SOLID. Karena SOLID yang dimaksud bukanlah solid dalam artian kuat/padat. Tetapi dalam arti yang lain



2. Manakah dari hal di bawah ini yang bukan termasuk tujuan dari SOLID principle?

- A. Kode yang terstruktur
- B. Mudah dipahami
- C. Toleran terhadap perubahan



2. Manakah dari hal di bawah ini yang bukan termasuk tujuan dari SOLID principle?

- A. Kode yang terstruktur
- B. Mudah dipahami
- C. Toleran terhadap perubahan

Kode yang terstruktur bukanlah salah satu tujuan dari penerapan SOLID principle.



4. Di bawah ini merupakan beberapa point yang perlu diperhatikan dalam prinsip Liskov substitution principle, kecuali?

- A. Contravariant dan Covariant.
- B. Inavariant
- C. Relative



4. Di bawah ini merupakan beberapa point yang perlu diperhatikan dalam prinsip Liskov substitution principle, kecuali?

- A. Contravariant dan Covariant.
- B. Inavariant
- C. Relative

Relative, bukanlah salah satu point yang perlu diperhatikan dalam penggunaan LSP.

4 Point yang perlu diperhatikan dalam penggunaan LSP adalah:

- Contravariant dan Covariant
- Preconditions dan Postconditions
- Invariant
- Constraint



5. Pada Dependency Inversion Principle, apakah yang dimaksud high-level module?

- A. Alat yang diperlukan untuk menjalankan sebuah tindakan.
- B. Class yang menjalankan aksi dengan menggunakan alat.
- C. Sebuah interface yang menghubungkan dua class.



5. Pada Dependency Inversion Principle, apakah yang dimaksud high-level module?

- A. Alat yang diperlukan untuk menjalankan sebuah tindakan.
- B. Class yang menjalankan aksi dengan menggunakan alat.
- C. Sebuah interface yang menghubungkan dua class.

Pada Dependency Inversion Principle, High-Level module diartikan sebagai sebuah class yang menjalankan aksi dengan menggunakan alat

Referensi dan bacaan lebih lanjut~

1. [S.O.L.I.D Principles: The Kotlin Way](#)
2. [An introduction to the SOLID principles](#)
3. [The S.O.L.I.D Principles in Pictures](#)
4. [The SOLID Principles of Object-Oriented Programming Explained in Plain English](#)



Nah, selesai sudah pembahasan kita di
Chapter 2 Topic 1 ini.

Selanjutnya, kita akan jump-in ke salah satu
tool andalan Android Developer, yaitu GIT

Penasaran kayak gimana? Cus langsung ke
topik selanjutnya~



Terima Kasih!



Next Topic

loading...