



# OOP in Kotlin

**Silver** - Chapter 1 - Topic 6

---

**Selamat datang di Chapter 1 Topic 6 online course  
Android Developer dari Binar Academy!**





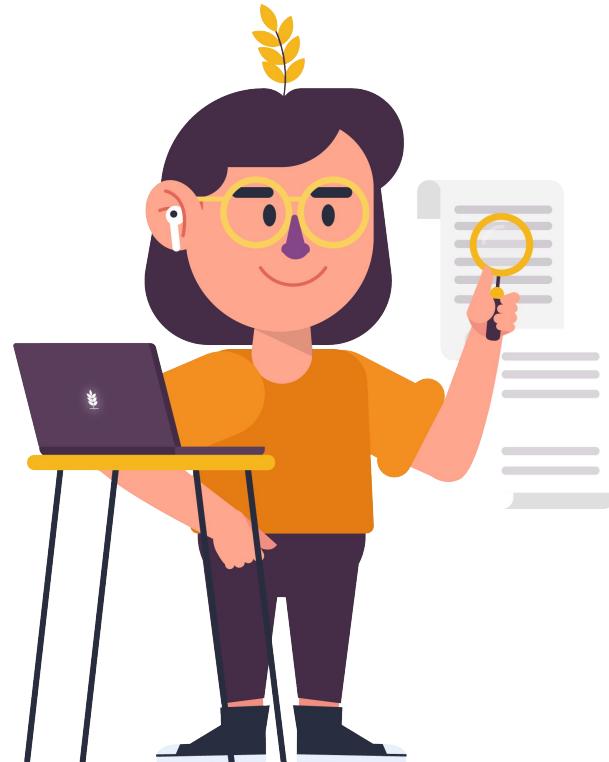
# Haaii Binarian 🙌

Masih di Chapter 1 niih~

Di Topik sebelumnya, kita sudah belajar tentang **Function** dan **Method**.

Pada **Topik 6** ini, kita bakal belajar tentang **konsep OOP (Object Oriented Programming)** pada bahasa pemrograman Kotlin.

Konsep ini akan semakin menambah wawasan kamu dalam membuat program dengan bahasa Kotlin. Gimana guys, udah siaapp? 😊



**Detailnya, kita bakal bahas hal-hal berikut ini:**

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction





Untuk Pengantar Topik ini, kita akan kenalan dengan Konsep **OOP** dulu

Apa sih **OOP** itu?

Untuk bahan bacaan tambahan, bisa cek [disini~](#)

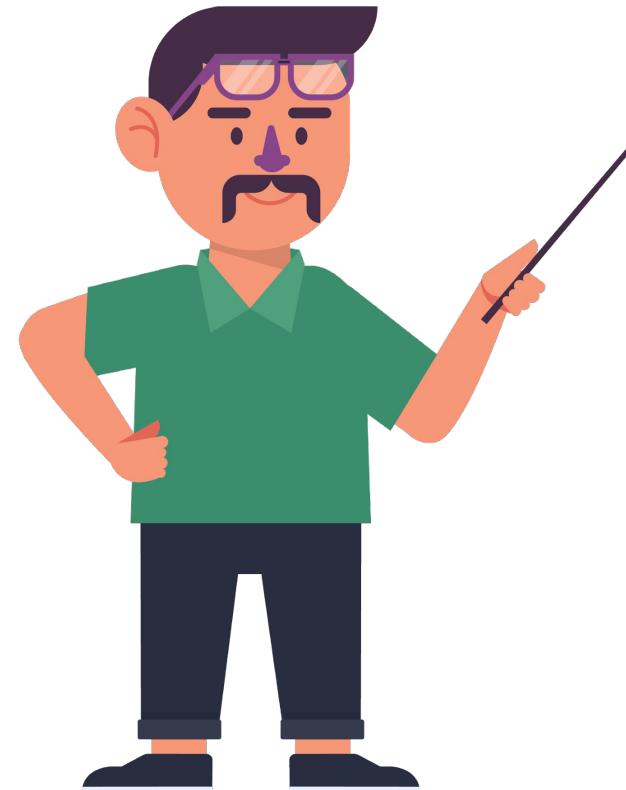


## Apa itu OOP?

Sederhananya, **Object Oriented Programming** (OOP) adalah metode merancang program menggunakan **Class** dan **Object**.

Konsep ini penting implementasinya dalam pemrograman karena ...

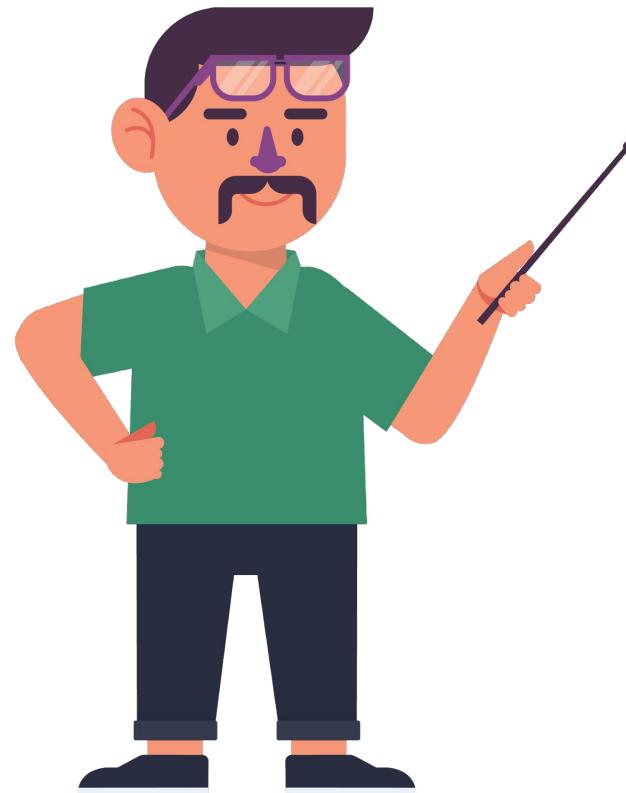
- OOP lebih cepat dan gampang buat dieksekusi.
- OOP punya struktur yang jelas.





[Continued...]

- OOP ngebantu kamu supaya nggak perlu terus mengulang, jadi kodennya lebih gampang dikelola, dimodifikasi, dan di-debug.
- OOP juga memungkinkan kamu buat bikin aplikasi yang bisa dipakai lagi dengan kode yang lebih sedikit dan waktu pengembangan yang lebih singkat.





## Biar gampangnya memahami konsep OOP, kita main analogi ini nih ...

Bayangan, kamu diterima kerja sebagai manajer di sebuah hotel bintang lima di dekat sebuah situs wisata.

Setiap hari, hotel ini dikunjungi ratusan tamu dari loka maupun mancanegara. Kamu sebagai manajer diharuskan mengatur karyawan-karyawan kamu agar mampu keep up dan melayani tamu dengan pelayanan terbaik.

Kira-kira, cara nya bagaimana ya?





Agar tamu terlayani dengan baik, kamu akan membagi tugas karyawan-karyawan mu ke berbagai peran. Seperti Bell Boy, Pelayan, Resepsiionis, dan lainnya.

Daan, agar semua pelayanan terstandar, kamu perlu membuat standar operational procedure (SOP) agar karyawan kamu bisa bekerja dengan baik. Meski tugas dan perannya beda-beda, standar pelayanan mereka akan tetap baik di mata tamu undangan

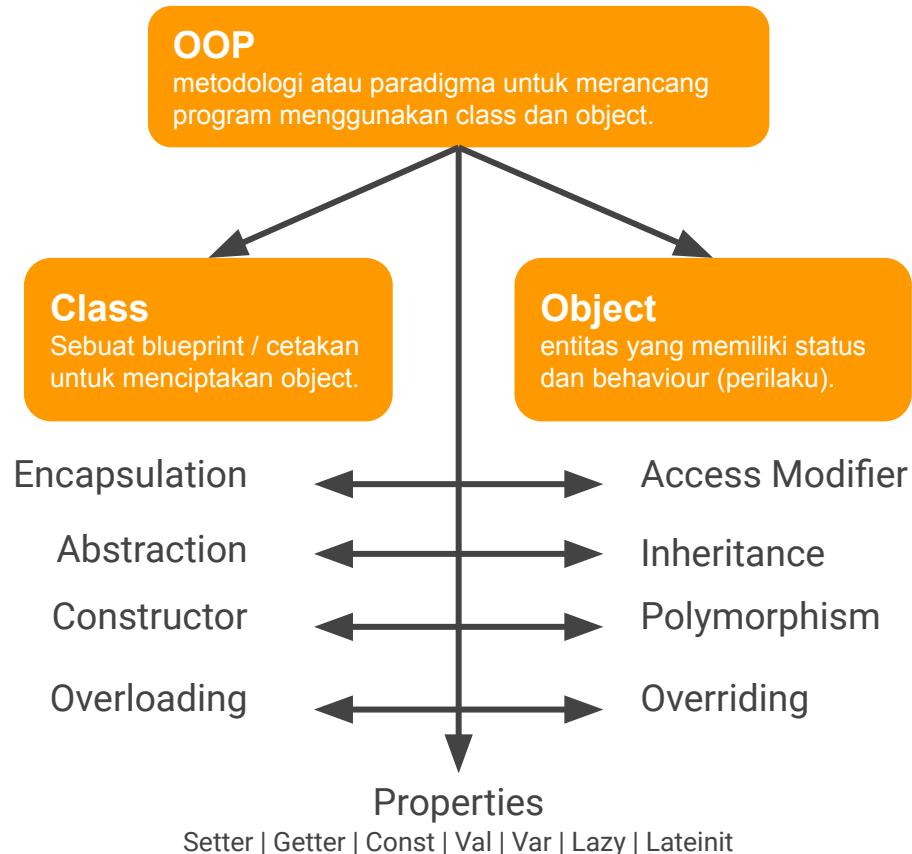
Naah, SOP ini mirip dengan apa yang akan kita pelajari berikutnya, yaitu OOP~





## Jadi, Konsep OOP itu meliputi apa aja?

Kurang lebih, bagan konsep **OOP** bisa dijelaskan seperti gambar disebelah ini :





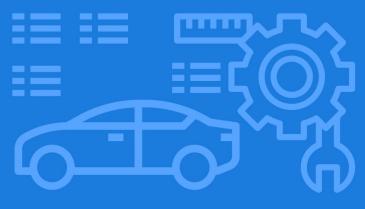
Pada dasarnya **konsep OOP di semua bahasa pemrograman sama aja, kok!**  
Kita hanya perlu penyesuaian aja~

Biar enak pemahamannya, kita review ulang materi **Class & Object** dulu, ya!



## Class

Blueprint



## Object



Van



Audi



Sports car

Biar gampang paham **Class** dan **Object**, kita pakai analogi lagi nih...

Coba kamu bayangan pabrik pembuatan mobil 🚗

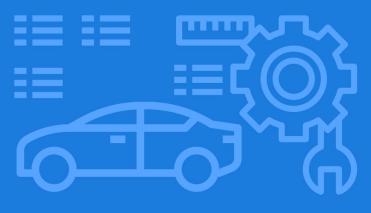
Ingat, **Class** adalah blueprint (cetakan) yang bisa membuat object atau instance. Sedangkan, **Object** adalah hasil dari class.

Dan, sebuah class bisa membuat banyak object ya!



## Class

Blueprint



## Object



Van



Audi



Sports car

Seperti pada gambar, kita punya blueprint mobil. Dari blueprint itu, kita tahu mobil itu ada 4 Roda, Ada pintunya, dll.

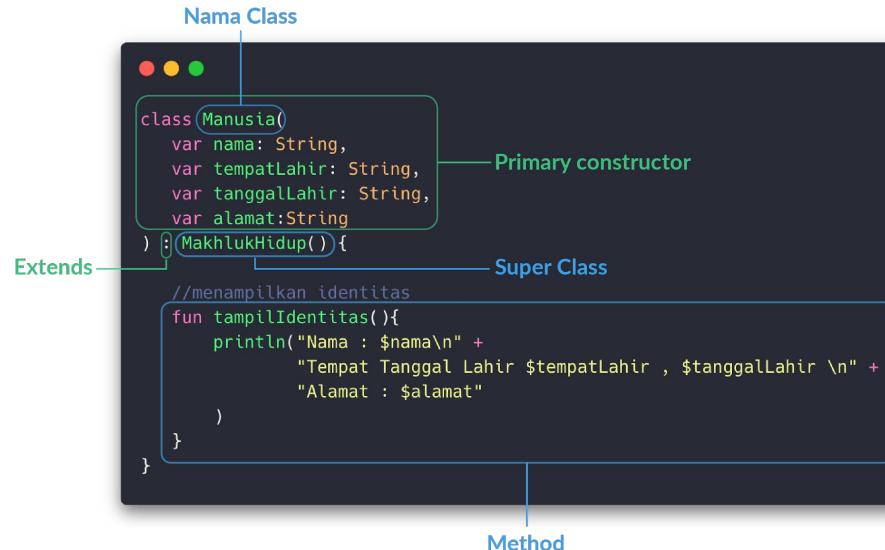
Nah perwujudan dari blueprint mobil, bisa banyak. Ada yang jadi mobil van, mobil Audi, ataupun mobil sport.



## Pembuatan Class di Kotlin

Nggak afdol rasanya kalau belum pakai contoh penerapan.

Gambar di bawah ini adalah contoh pembuatan **Class** di Kotlin.





## Ingat cara buat Object di Kotlin?

Kita tidak membutuhkan syntax **new** untuk menginstansiasi suatu Object di Kotlin. Ini contohnya:

```
//pembuatan object manusia
val manusia1 = Manusia("Sabrina","Tangerang","01 September 1996","BSD")
val manusia2 = Manusia("Mas Gun","Yogyakarta","06 Juni 1986","Serpong")
```

Nama Object

Constructor Class method



Nah, udah mulai inget kan cara bikin **Class** dan **Object** di Kotlin?

Sekarang, kita lanjut recall lagi tentang **Constructor** pada Kotlin!

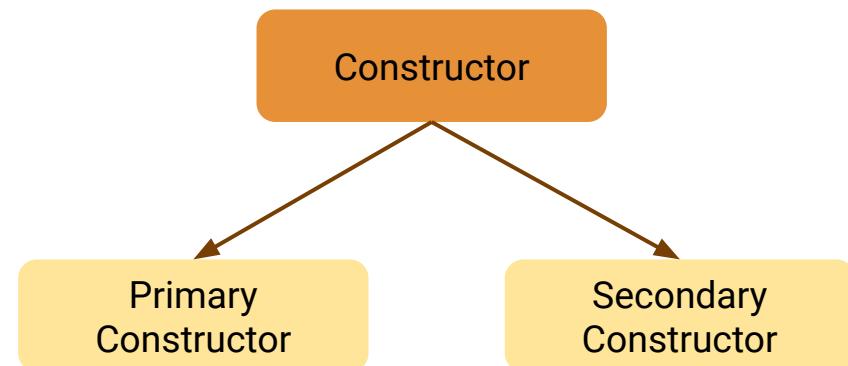


## Constructor itu apa hayoo?

Tujuan utama dari **Constructor** adalah menginisialisasi properti pada **Class**. Perintah ini bisa dipanggil ketika kita membuat **Object**.

Di Kotlin, ada dua jenis constructor yang bisa dimanfaatkan, yaitu **Primary Constructor** dan **Secondary Constructor**.

Kita akan mempelajari keduanya dengan contoh. Plus, kita juga bakal belajar tentang blok penginisialisasi.





## Ingat Primary Constructor?

**Primary Constructor** adalah cara termudah untuk menginisialisasi sebuah class. Hal tersebut dinyatakan sebagai bagian dari **header class**.

Pada contoh di samping, kita akan coba mendeklarasikan sebuah **constructor** sebagai bagian dari header class.

Ini adalah contoh dari **primary constructor** yang kita buat untuk menginisialisasi properti **nama** dan **umur** (anggota data) pada class **Murid**.



```
fun main(args: Array<String>) {  
    //membuat object class Murid  
    val murid = Murid("Abika", 23)  
  
    println("Nama Muridnya: ${murid.nama}")  
    println("Umur Muridnya: ${murid.umur}")  
}  
  
//Primary Constructor  
class Murid(val nama: String, var umur: Int) {  
    //code here  
}
```



## Primary Constructor - Default Value

Kita juga dapat menentukan nilai **default** pada **constructor**.

Pada contoh, kita telah menetapkan nilai default nama murid sebagai "Farid", dan nilai default usia adalah "24".

Kita telah membuat tiga object dari class tersebut. Yang satu dengan **nama dan usia**, kedua adalah **nama tanpa usia**, dan terakhir adalah **tanpa nama dan usia**.

Di bagian **output**, bisa dilihat bahwa nilai default akan ditimpa oleh nilai yang diteruskan saat membuat sebuah object dalam class.

```
fun main(args: Array<String>) {  
  
    //membuat object class Murid  
    val murid = Murid("Abika", 23)  
    val murid2 = Murid("Didik")  
    val murid3 = Murid()  
  
    println("Nama: ${murid.nama} dan Umur: ${murid.umur}")  
    println("Nama: ${murid2.nama} dan Umur: ${murid2.umur}")  
    println("Nama: ${murid3.nama} dan Umur: ${murid3.umur}")  
}  
  
class Murid(val nama: String = "Farid", var umur: Int = 24) {  
    //code here  
}
```



Output:

```
Nama: Abika dan Umur: 23  
Nama: Didik dan Umur: 24  
Nama: Farid dan Umur: 24
```

Default Value



## Primary Constructor - Init Block

Lanut nih, kita belajar mengenai **inisialisasi tambahan pada kode**.

Dalam contoh berikut, kita memiliki blok penginisialisasi yang telah kita nyatakan di dalam *constructor* menggunakan **init**.

Dalam blok ini, kita dapat memiliki logika inisialisasi tambahan sesuai yang kita mau.

```
fun main(args: Array<String>) {  
  
    val murid = Murid("Abika", 23)  
    val murid2 = Murid("Didik")  
    val murid3 = Murid()  
}  
  
class Murid(val nama: String = "Murid", var umur: Int = 24) {  
    val murNama: String  
    var murUsia: Int  
  
    init {  
        if (nama == "Murid") {  
            murNama = "Farid"  
            murUsia = 25  
        } else {  
            murNama = nama.toUpperCase()  
            murUsia = umur  
        }  
        println("Nama Muridnya: $murNama")  
        println("Umur Muridnya: $murUsia")  
    }  
}
```



## Ingin lagi Secondary Constructor

Secondary constructor pada Kotlin dapat dibuat dengan menggunakan keyword **constructor**.

Mereka yang memainkan peran utama dalam **pewarisan (inheritance)** sehingga child class bisa mengakses constructor yang ada pada parent class sesuai kebutuhannya.



```
class Murid {  
    constructor(nama: String) {  
        // code inside constructor  
    }  
    constructor(nama: String, umur: Int) {  
        // code inside constructor  
    }  
}
```



Berikut merupakan contoh sederhana dari deklarasi **secondary constructor** di dalam sebuah class:

```
fun main(args: Array<String>){  
    val obj = Murid ("Abika", 23)  
}  
  
class Murid{  
    constructor(nama: String, umur: Int){  
        println("Nama Muridnya: ${nama.toUpperCase( )}")  
        println("Umur Muridnya: $umur")  
    }  
}
```

Output:  
Nama Muridnya: ABIKA  
Umur Muridnya: 23



## Memanggil Satu Secondary Constructor dari Constructor Lain

Sebelumnya kita udah mempelajari **pemanggilan secondary constructor menggunakan satu parameter**.

Terus gimana nih cara **memanggil secondary constructor dari constructor lain?**

Contohnya bisa dilihat di samping. Perhatikan Outputnya juga yaa~

```
fun main(args: Array<String>){
    val obj = Murid ("Abika")
}
class Murid{
    constructor(nama: String): this(nama, 23){
        println("Secondary Constructor dengan satu parameter")
    }
    constructor(nama: String, umur: Int){
        println("Secondary Constructor dengan dua parameter")
        println("Nama Muridnya: ${nama.toUpperCase( )}")
        println("Umur Muridnya: $umur")
    }
}
```



Output:

```
Secondary Constructor dengan dua parameter
Nama Muridnya: ABIKA
Umur Muridnya: 24
Secondary Constructor dengan dua parameter
```

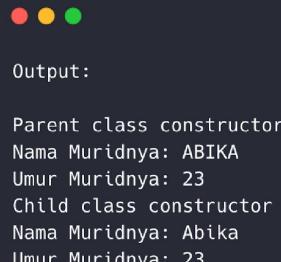


## Secondary Constructor dengan Parent dan Child Class

Dalam contoh berikut, kita akan membuat dua class **Sekolah** yang merupakan parent class dari child class **Murid**.

Nah, **child class** dari secondary constructor akan memanggil parent class secondary constructor jika kita menggunakan keyword **super**.

Contohnya bisa lihat kode di samping. Outputnya bisa dicek di bawah ini yaa~



Output:

```
Parent class constructor
Nama Muridnya: ABIKA
Umur Muridnya: 23
Child class constructor
Nama Muridnya: Abika
Umur Muridnya: 23
```

```
fun main(args: Array<String>){
    val stu = Murid("Abika", 23)
}

open class Sekolah{
    constructor(nama: String, umur: Int){
        println("Parent class constructor")
        println("Nama Muridnya: ${nama.toUpperCase()}")
        println("Umur Muridnya: $umur")
    }
}
class Murid: Sekolah{
    constructor(nama: String, umur: Int): super(nama, umur){
        println("Child class constructor")
        println("Nama Muridnya: $nama")
        println("Umur Muridnya: $umur")
    }
}
```



Dah lengkap banget deh, sesi review kita di Topik-topik sebelumnya~

Sekarang, baru nih kita masuk ke konsep OOP, yaitu **Encapsulation**.

Kalau kamu mau lihat dulu kilasan penjelasannya bisa [klik](#) disini~



# Prinsip Encapsulation dalam OOP, maksudnya apa sih?

Sama seperti konsep **OOP** pada umumnya, **encapsulation** adalah **cara membungkus data kita**.

Sebagai gambaran, Class memiliki **attribute** dan **method** yang disebut **member**.





Biasanya, akan ada tingkatan akses yang disebut **access modifier**. Nah, tingkatan inilah yang menentukan dari mana saja member tersebut dapat diakses.

Itulah yang disebut dengan konsep **encapsulation**. Singkatnya, **encapsulation** adalah **sebuah konsep yang memberikan akses terhadap member dan class**.





## Access Modifier

Seperti ini lah **access modifier** pada Kotlin

Secara default **modifier** pada member attribute & method adalah **public**.

Jika sebuah member tidak memiliki modifier, member tersebut secara otomatis dihitung ke dalam modifier public.



```
class Anak : Bapak(){  
    //private adalah salah satu Access Modifier  
    private val name: String = "Sukinem"  
    private val umur: Int = 98  
  
    override fun darahTinggi() {  
        super.darahTinggi()  
    }  
}
```



Penulisan Access Modifier di Kotlin itu seperti ini :

```
<Access Modifier> <val/var> <nama variable>:<tipe data>
```

Atau kalau di run koding-nya, akan tampak seperti ini...



```
<Access Modifier> <val/var> <nama variable>: <tipe data>
private val name: String = "Sabrina"
```





Berikut adalah Access Modifier yang berlaku di Kotlin :

Modifier	Class	Subclass	Module	World
Public	✓	✓	✓	✓
Private	✓	✗	✗	✗
Protected	✓	✓	✗	✗
Internal	✓	✓	✓	✗

Keterangan:

- Subclass artinya class anak
- World artinya bisa diakses dari mana saja dalam satu project



Ternyata proses **encapsulation** dan access modifier yang berlaku di Kotlin cukup mudah dipahami yaa~

Sekarang kita pelajari tentang konsep OOP selanjutnya, yaitu **Inheritance**.

Kalau kamu mau intip cuplikan penjelasannya dulu, boleeh kok~ [disini](#)

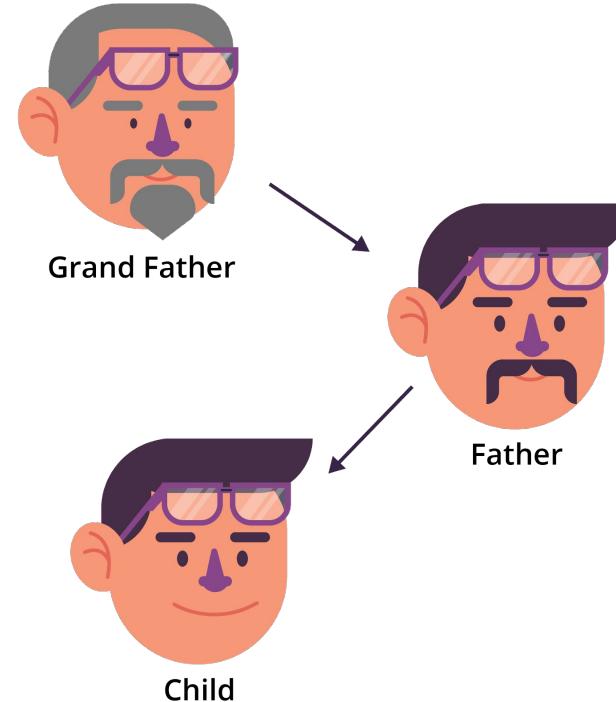


## Memanggil Satu Secondary Constructor dari Constructor Lain

**Inheritance (Pewarisan)** adalah sebuah konsep pewarisan class yang kita terapkan di pemrograman **OOP**.

Cara kerjanya mirip dengan DNA yang diwariskan dari orang tua ke anak-anaknya.

Kita main analogi lagi yaa~



**Sulistyo**

- Kulit coklat
- Mata bulat
- Suka dengan celana berwarna coklat
- Suka dengan baju berwarna hijau
- Rambut warna Coklat tua

**Sukinem**

- Kulit putih
- Mata bulat
- Suka dengan rok berwarna abu
- Suka dengan baju berwarna pink
- Rambut warna coklat muda

**Basuki**

- Kulit putih
- Mata bulat
- **Suka dengan celana berwarna coklat**
- **Rambut warna coklat muda**
- Suka dengan sepatu warna tosca

**Darsini**

- Kulit putih
- Mata bulat
- Suka dengan baju berwarna pink
- **Rambut warna coklat tua**
- Suka dengan rambut yang dikucir

## Perkenalkan, ini Keluarga Sulistyo

Ini lah keluarga bahagia Sulistiyo~

- **Sulistyo** sebagai **suami atau ayah**
- **Sukinem** sebagai **istri atau ibu**
- **Basuki** sebagai **anak laki-laki**
- **Darsini** sebagai **anak perempuan**

**Sulistyo**

- Kulit coklat
- Mata bulat
- Suka dengan celana berwarna coklat
- Suka dengan baju berwarna hijau
- Rambut warna Coklat tua

**Sukinem**

- Kulit putih
- Mata bulat
- Suka dengan rok berwarna abu
- Suka dengan baju berwarna pink
- Rambut warna coklat muda

**Basuki**

- **Kulit putih**
- **Mata bulat**
- **Suka dengan celana berwarna coklat**
- **Rambut warna coklat muda**
- Suka dengan sepatu warna tosca

**Darsini**

- **Kulit putih**
- **Mata bulat**
- **Suka dengan baju berwarna pink**
- **Rambut warna coklat tua**
- Suka dengan rambut yang dikucir

## Perkenalkan, ini Keluarga Sulistyo

Sebagai keluarga, mereka memiliki banyak ciri-ciri yang serupa.

Tetapi, ada juga ciri-ciri yang muncul karena keunikan masing-masing.

**Sulistyo**

- Kulit coklat
- Mata bulat
- Suka dengan celana berwarna coklat
- Suka dengan baju berwarna hijau
- Rambut warna Coklat tua

**Sukinem**

- Kulit putih
- Mata bulat
- Suka dengan rok berwarna abu
- Suka dengan baju berwarna pink
- Rambut warna coklat muda

**Basuki**

- **Kulit putih**
- **Mata bulat**
- **Suka dengan celana berwarna coklat**
- **Rambut warna coklat muda**
- Suka dengan sepatu warna tosca

**Darsini**

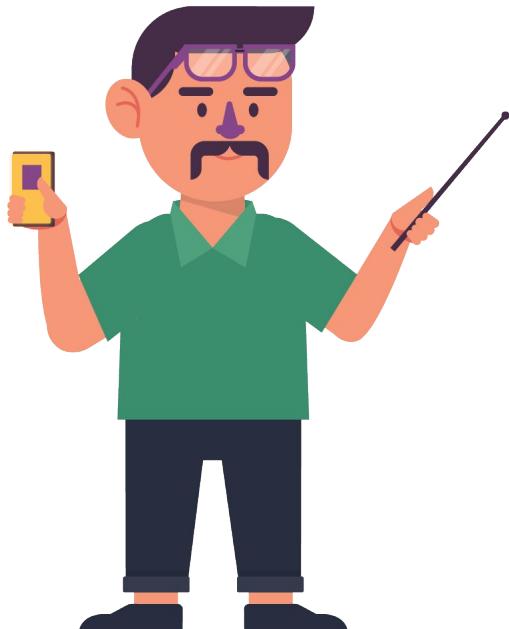
- **Kulit putih**
- **Mata bulat**
- **Suka dengan baju berwarna pink**
- **Rambut warna coklat tua**
- Suka dengan rambut yang dikucir

## Contoh-nya gini ...

Ciri Basuki dan Darsini yang dicetak dengan warna tulisan orange adalah ciri yang diwariskan dari Sulistyo dan Sukinem sebagai orang tua mereka.

Tetapi ada ciri Basuki dan Darsini yang tidak dimiliki oleh Sulistyo dan Sukinem.

Nah, peristiwa inilah yang disebut dengan Inheritance, atau pewarisan karakteristik orang tua ke anaknya~



## Dari Analogi tadi, kita akan kenal istilah Inheritance yang penting :

### 1. Super class atau Parent class

Class yang semua fiturnya diwariskan kepada *class* turunannya.

### 2. Sub-class atau Child class

Class turunan yang mewarisi semua fitur dari class lain. Sub-class dapat menambah field dan *method*-nya sendiri yang berbeda dengan *parent class*.

### 3. Reusability

Ketika kita ingin membuat *class* baru, kita bisa menurunkan atau menggunakan kembali field dan *method* dari *class* yang sudah ada.



## Terus, gimana penerapan Inheritance dalam pemrograman Kotlin?

Perhatikan ilustrasi berikut ini :

Seorang Anak pasti memiliki kemiripan sifat dari ayahnya.  
Maka, **class Anak juga memiliki method & attribute milik Ayah.**

```
open class Bapak {  
    open fun ngantukan(){  
    }  
  
    open fun lupaan(){  
    }  
  
    open fun darahTinggi(){  
    }  
}
```

```
class Anak : Bapak(){  
    override fun ngantukan() {  
        super.ngantukan()  
    }  
  
    override fun lupaan() {  
        super.lupaan()  
    }  
  
    override fun darahTinggi() {  
        super.darahTinggi()  
    }  
}
```



Jadi, **child class** bisa menggunakan method yang ada pada Parentnya dengan menggunakan cara **Overriding**.

Eits, tapi jangan lupa untuk menambahkan syntax **open** pada method yang akan di **override** ya!

```
● ● ●  
open class Bapak {  
    open fun ngantukan(){  
    }  
  
    open fun lupaan(){  
    }  
  
    open fun darahTinggi(){  
    }  
}
```

```
● ● ●  
class Anak : Bapak(){  
    override fun ngantukan() {  
        super.ngantukan()  
    }  
  
    override fun lupaan() {  
        super.lupaan()  
    }  
  
    override fun darahTinggi() {  
        super.darahTinggi()  
    }  
}
```



Nah, tadi udah disinggung tuh, untuk menerapkan konsep **Inheritance**, ada method yang harus diketahui ...

Namanya itu adalah **Method Overriding** dan **Overloading** ↪



## Method Overriding

**Method overriding**, berasal dari kata **override** yang artinya mengesampingkan atau **mengabaikan**.

Dapat dilihat di contoh, bahwa class Anak bisa overriding method dari class Bapak.

```
● ● ●  
  
open class Bapak {  
    open fun ngantukan(){  
    }  
  
    open fun lupaan(){  
    }  
  
    open fun darahTinggi(){  
    }  
}  
  
class Anak : Bapak(){  
    override fun ngantukan() {  
        super.ngantukan()  
    }  
  
    override fun lupaan() {  
        super.lupaan()  
    }  
  
    override fun darahTinggi() {  
        super.darahTinggi()  
    }  
}
```



Dengan begitu, jika kita memanggil method **ngantukan** dari class Anak, akan terpanggil method dari class **Anak**. Bukan dari class Bapak.

Karena method **ngantukan** dari class **Bapak** sudah di-override oleh method **ngantukan** dari class **Anak**.

```
● ● ●  
open class Bapak {  
    open fun ngantukan(){  
    }  
  
    open fun lupaan(){  
    }  
  
    open fun darahTinggi(){  
    }  
}
```

```
● ● ●  
class Anak : Bapak(){  
    override fun ngantukan() {  
        super.ngantukan()  
    }  
  
    override fun lupaan() {  
        super.lupaan()  
    }  
  
    override fun darahTinggi() {  
        super.darahTinggi()  
    }  
}
```



## Method Overloading

**Overloading method** adalah nama method-nya sama namun parameternya beda. Pada contoh di samping, dalam *class* Lingkaran terdapat dua method bernama **luas**.

Disinilah keunikan **Overloading**, kamu bisa menggunakan nama method yang sama dalam satu kelas.

Coba perhatikan lagi, method luas yang pertama memiliki parameter tipe data **Float**. Sedangkan *method* luas yang kedua memiliki parameter **Double**.



```
class Lingkaran {  
  
    // method menghitung luas dengan jari-jari  
    fun luas(r: Float): Float {  
        return (Math.PI * r * r).toFloat()  
    }  
  
    // method menghitung luas dengan diameter  
    fun luas(d: Double): Double {  
        return (1 / 4 * Math.PI * d)  
    }  
}
```



Cukup mudahkan caranya Overriding  
dan Overloading method?

Next step, kita belajar satu konsep OOP  
lainnya, yaitu **Polymorphism!**

Untuk bahan bacaan extra, silahkan  
cek dulu [disini~](#)



## Apa itu Polymorphism?

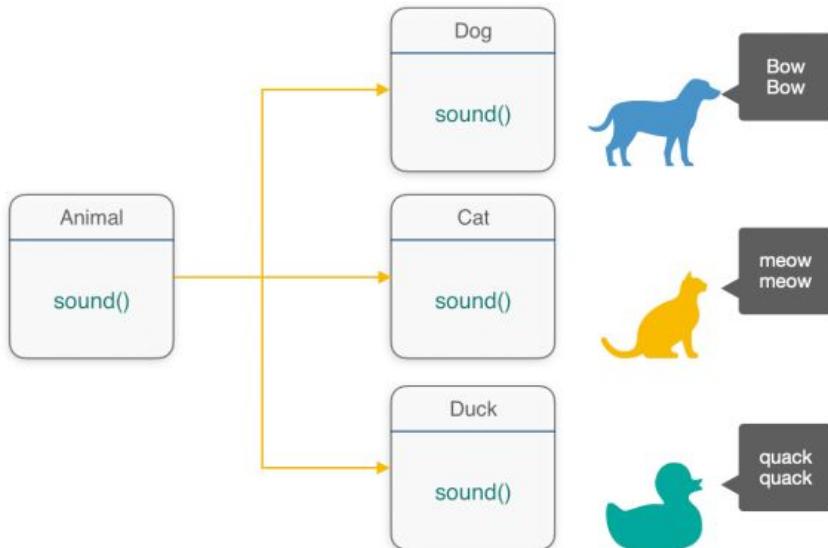
Secara harfiah, **Poly** artinya banyak, **morfisme** artinya bentuk.

Jadi **Polymorphism** adalah sebuah prinsip dalam biologi dimana organisme atau spesies dapat memiliki banyak bentuk atau tahapan (stages).



Atau bisa diartikan **sebuah prinsip di mana class dapat memiliki banyak “bentuk” method yang berbeda-beda meskipun namanya sama.**

“Bentuk” di sini dapat kita artikan: **isinya berbeda, parameternya berbeda, dan tipe datanya berbeda.**



**Supaya lebih mudah, perhatikan flow dari pemrograman suara hewan di samping~**

Jika diterjemahkan ada sebuah *class* Animal yang memiliki 3 *method* dengan nama yang sama, yaitu “**sound()**”. Tetapi, masing-masing memiliki fungsi yang berbeda-beda.

Method yang pertama akan memproses suara anjing, method kedua akan memproses suara kucing, dan method ketiga akan memproses suara bebek.



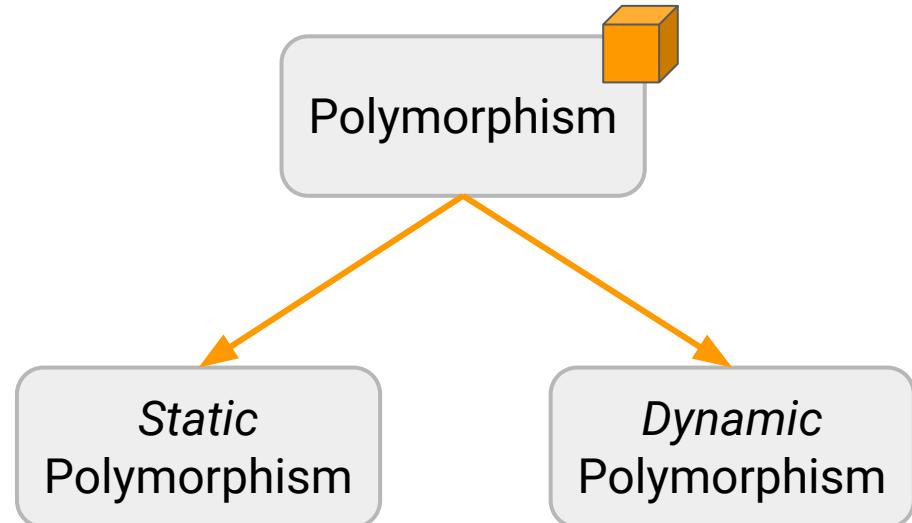
**Terus, gimana penerapan  
Polymorphism dalam pemrograman  
Kotlin?**



## Terus, gimana penerapan konsep Polymorphism di Kotlin?

Eiits, tunggu dulu...

Sebelum kita masuk ke praktek deklarasi, kita harus tau dulu kalau Polymorphism terbagi menjadi 2 macam, seperti gambar di samping ini ➔





## Static Polymorphism

**Static Polymorphism** menggunakan **method overloading**.

Ayo, ingat-ingat lagi pembahasan sebelumnya 😊

Method overloading terjadi pada sebuah class yang memiliki **nama method yang sama** tapi memiliki **parameter** dan **tipe data** yang **berbeda**.

```
class Lingkaran {  
  
    // method menghitung luas dengan jari-jari  
    fun luas(r: Float): Float {  
        return (Math.PI * r * r).toFloat()  
    }  
  
    // method menghitung luas dengan diameter  
    fun luas(d: Double): Double {  
        return (1 / 4 * Math.PI * d)  
    }  
}
```

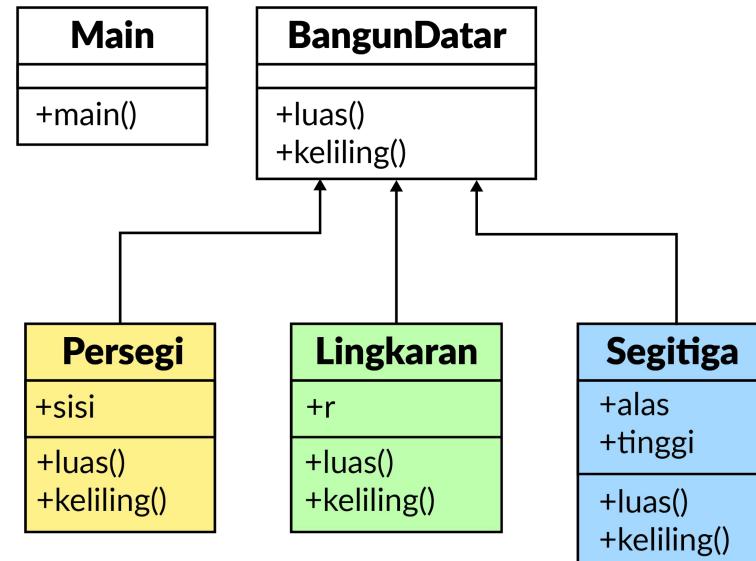


## Dynamic Polymorphism

Dynamic polymorphism juga menggunakan method **overriding**, biasanya terjadi saat kita menggunakan **pewarisan (inheritance)** dan **implementasi interface**.

Pada pewarisan, kita bisa mewariskan *method & attribute* dari *class ayah* ke *class anak*.

*Class anak* akan memiliki nama *method* yang sama dengan *class ayah*, begitu juga dengan anak-anak yang lainnya.



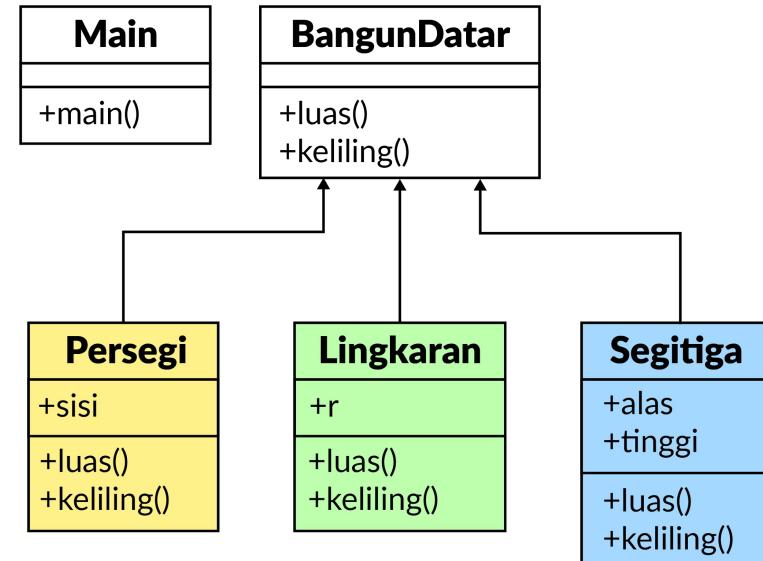


## Untuk mudahnya, Dynamic Polymorphism bisa dianalogikan begini...

Jika diterjemahkan ada 3 buah class yaitu class **Persegi**, **Lingkaran**, dan **Segitiga**. Ketiga class tersebut melakukan pewarisan ke class **BangunDatar** yang memiliki 2 method yaitu :

- **luas()**, dan
- **keliling()**,

sehingga ketiga class tersebut **Persegi**, **Lingkaran** dan **Segitiga** mewarisi 2 method tersebut. Tetapi pada kedua method yang diwarisi tersebut memiliki fungsi yang berbeda beda.



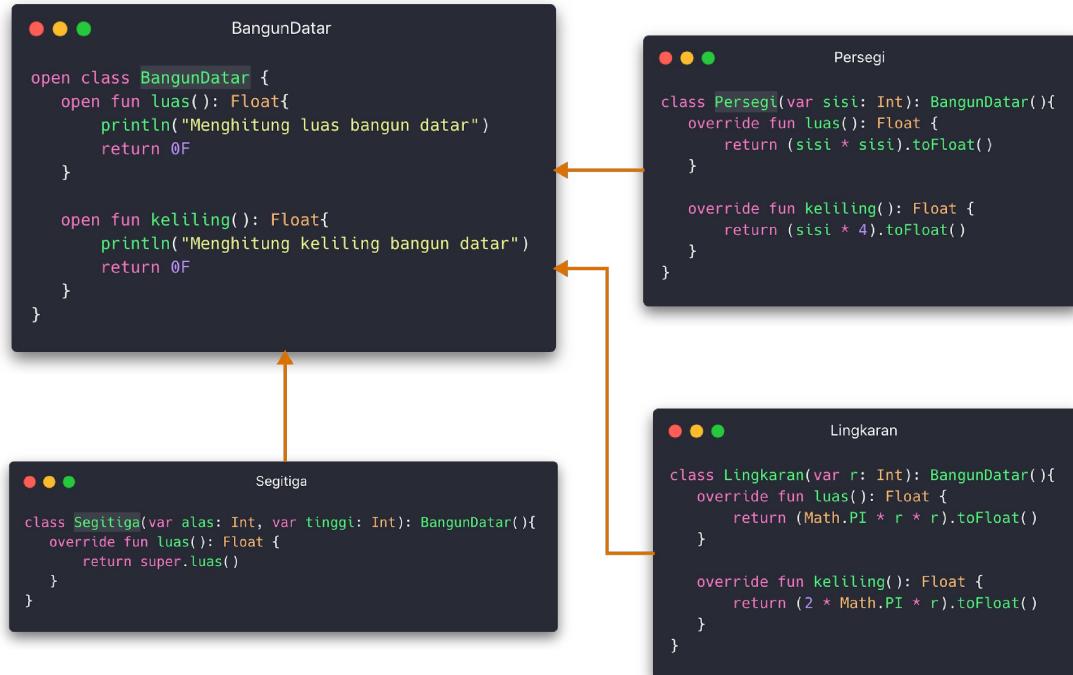


## Sebagai contoh Dynamic Polymorphism, coba deh lirik ke samping~

Disini, kita punya sebuah class **BangunDatar**. Class ini memiliki **open keyword** yang berarti class ini dapat dikembangkan.

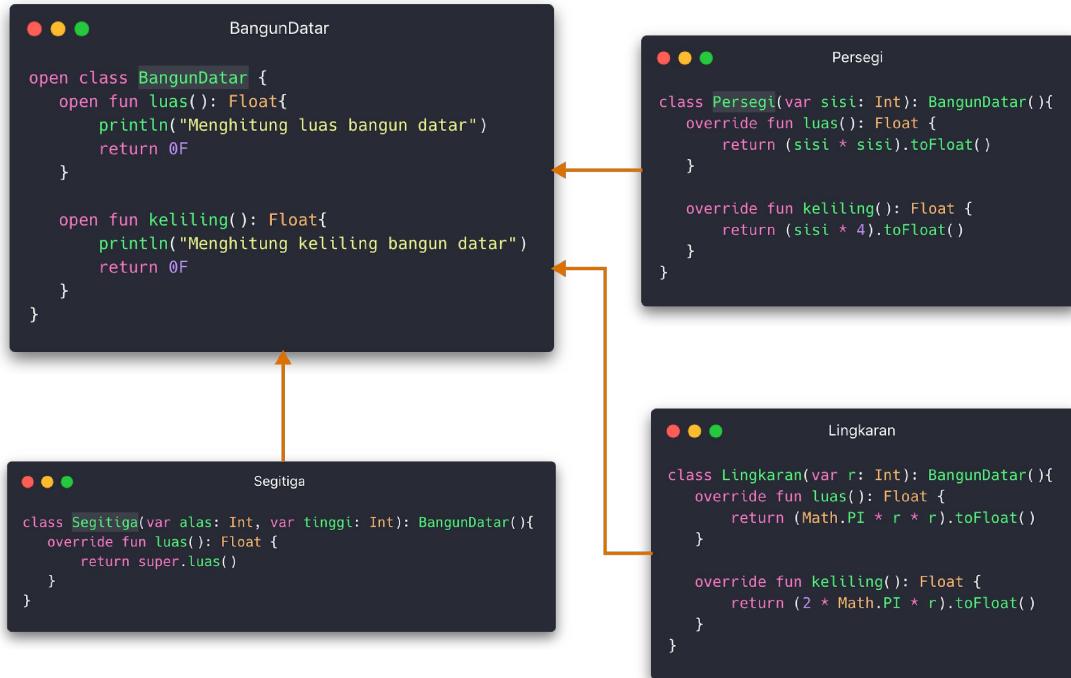
Dikembangkan seperti apa maksudnya?

Class dengan keyword open dapat diturunkan ke class lain.





Contohnya class **BangunDatar** dapat diturunkan ke class **Persegi**, **Lingkaran**, dan **Segitiga**. Penurunan ini dengan cara class Persegi, Lingkaran, dan Segitiga semuanya meng-extends ke class BangunDatar.



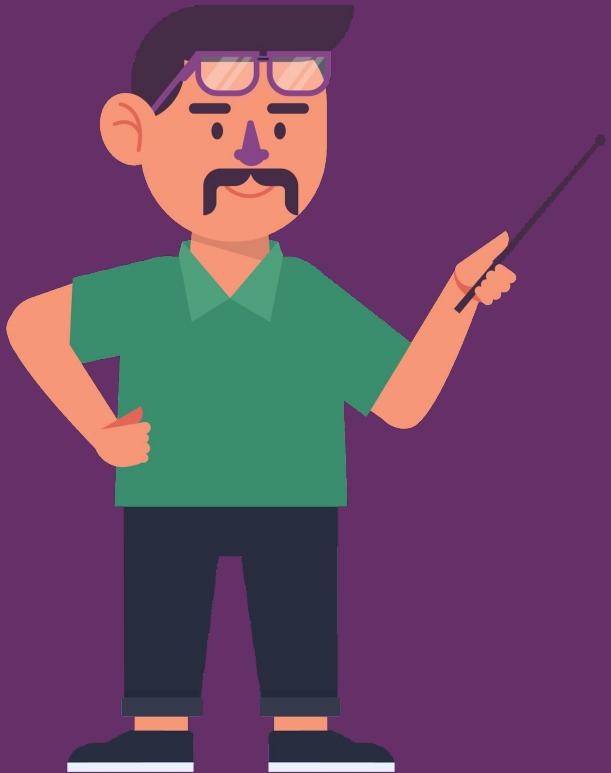


## Berikut Output dari Kode tadi ...

```
● ● ●  
  
fun main() {  
    val bangunDatar = BangunDatar()  
    val persegi = Persegi(4)  
    val segitiga = Segitiga(6,3)  
    val lingkaran = Lingkaran(50)  
  
    //memanggil method luas dan keliling  
    bangunDatar.luas()  
    bangunDatar.keliling()  
  
    println("Luas Persegi : ${persegi.luas()}")  
    println("Keliling Persegi : ${persegi.keliling()}")  
    println("Luas Segitiga : ${segitiga.luas()}")  
    println("Luas Lingkaran : ${lingkaran.luas()}")  
    println("Keliling Lingkaran : ${lingkaran.keliling()}")  
}
```



Output:  
Menghitung luas bangun datar  
Menghitung keliling bangun datar  
Luas Persegi : 16.0  
Keliling Persegi : 16.0  
Menghitung luas bangun datar  
Luas Segitiga : 0.0  
Luas Lingkaran : 7853.9814  
Keliling Lingkaran : 314.15927



Sudah banyak sekali yah konsep OOP yang kita pelajari ...

Masih ada loh konsep OOP yang berikutnya, yaitu **Abstraction!**

Penjelasan lebih lanjut bisa kamu cek [disini~](#)



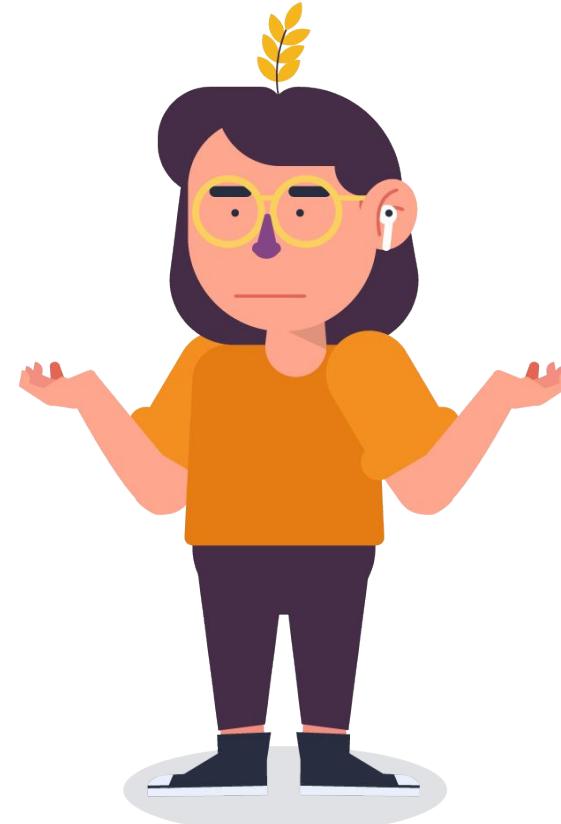
## Apa itu Abstraction

**Abstraction** adalah salah satu konsep inti dari OOP.

Itu bisa digunakan **saat kita tahu fungsi apa yang harus dimiliki class tetapi tidak tahu bagaimana fungsi tersebut harus diimplementasikan.**

Kamu juga bisa menggunakan ini jika fungsionalitas dapat diimplementasikan dalam beberapa cara.

Nah, ketika suatu *class* melakukan *extends* ke *class abstract*, *class* tersebut dapat mengimplementasikan *method abstract* yang ada dalam *class abstract* tadi.





Untuk implementasi konsep Abstraction, cek Koding disamping ini~

```
● ● ●  
  
//Class Abstract sebagai Parent Class :  
abstract class MakhlukHidup (var alatNafas: String = "") {  
    abstract fun bernafas(alatNafas: String)  
}
```

```
● ● ●  
  
//Child Class yang extends ke Class Abstract :  
class Kambing : MakhlukHidup() {  
    override fun bernafas(alatNafas: String) {  
        this.alatNafas = alatNafas  
    }  
}
```

# Saatnya kita Quiz!



1. Perhatikan tabel dibawah ini. Modifier yang tertera [Blank] baik diisi dengan ...

Modifier	Class	Subclass	Module	World
Public	✓	✓	✓	✓
Private	✓	✗	✗	✗
Protected	✓	✓	✗	✗
[Blank]	✓	✓	✓	✗

- A. Internal
- B. Local
- C. Inside



1. Perhatikan tabel dibawah ini. Modifier yang tertera [Blank] baik diisi dengan ...

Modifier	Class	Subclass	Module	World
Public	✓	✓	✓	✓
Private	✓	✗	✗	✗
Protected	✓	✓	✗	✗
[Blank]	✓	✓	✓	✗

- A. Internal
- B. Local
- C. Inside

Yap betul! Pilihan yang tepat adalah **Internal**.

Access modifier internal adalah access modifier baru di Kotlin.



## 2. Class sabrina mewarisi dari class Manusia. Apa Code yang ditutupi agar tidak error?



```
class Sabrina : Manusia() {  
    fun sebutNama( ) {  
        print("Nama saya ${super.nama}")  
    }  
}
```

- A. Manusia
- B. Manusia()
- C. Manusia::class.java



## 2. Class sabrina mewarisi dari class Manusia. Apa Code yang ditutupi agar tidak error?



```
class Sabrina : Manusia() {  
    fun sebutNama( ) {  
        print("Nama saya ${super.nama}")  
    }  
}
```

- A. Manusia
- B. Manusia()
- C. Manusia::class.java

Kotlin, mengharuskan **class** untuk meng-extend pada suatu **constructor** class yang ingin diturunkan. Sehingga biasanya diikuti oleh simbol kurung buka dan kurung tutup.



### 3. Di bawah ini, manakah dua prinsip dalam OOP yang memiliki kegunaan yang sama?

- A. Abstraction dan Polymorphism merupakan prinsip yang memiliki kegunaan yang sama
- B. Inheritance dan Encapsulation merupakan prinsip yang memiliki kegunaan yang sama
- C. Encapsulation dan Abstraction merupakan prinsip yang memiliki kegunaan yang sama

**3. Di bawah ini, manakah dua prinsip dalam OOP yang memiliki kegunaan yang sama?**

- A. Abstraction dan Polymorphism merupakan prinsip yang memiliki kegunaan yang sama
- B. Inheritance dan Encapsulation merupakan prinsip yang memiliki kegunaan yang sama
- C. Encapsulation dan Abstraction merupakan prinsip yang memiliki kegunaan yang sama

**Encapsulation** menyembunyikan fitur object dan mengikat semua property di dalam satu class. Dan **abstraction** adalah prinsip yang menunjukkan data yang hanya diperlukan kepada pengguna.



#### 4. Di bawah ini manakah method yang didukung oleh polymorphism?

- A. Oversearching.
- B. Overloading.
- C. Overclocking.



#### 4. Di bawah ini manakah method yang didukung oleh polymorphism?

- A. **Oversearching.**
- B. **Overloading.**
- C. **Overclocking.**

Mari kita ingat-ingat lagi materi sebelumnya. **Method overloading** digunakan oleh **static polymorphism**. Method ini terjadi pada sebuah class yang memiliki nama method yang sama tapi memiliki parameter dan tipe data yang berbeda.



**5. Pada syntax di atas, dapat diartikan bahwa kita telah mengimplementasikan salah satu prinsip OOP, yaitu...**



```
class Pisang : Buah() {  
}
```

- A. Inheritance
- B. Abstraction
- C. Polymorphism

5. Pada syntax di atas, dapat diartikan bahwa kita telah mengimplementasikan salah satu prinsip OOP, yaitu...



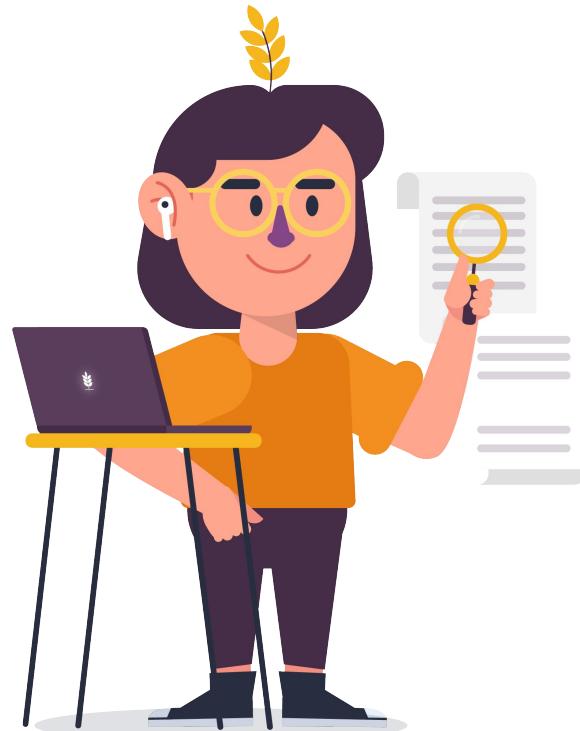
```
class Pisang : Buah() {  
}
```

- A. Inheritance
- B. Abstraction
- C. Polymorphism

**Inheritance** merupakan prinsip yang memanfaatkan pewarisan. Pada Kotlin untuk mengimplementasikan **inheritance** kita hanya cukup menambahkan simbol **titik ":"** dan memasukkan nama class yang ingin kita panggil.

### Referensi dan bacaan lebih lanjut~

1. [Belajar Java OOP: Memahami Prinsip Polimorfisme dalam OOP](#)
2. [Classes | Kotlin](#)
3. [Belajar Kotlin Lanjutan \(OOP\)](#)
4. [Kotlin Class and Objects – Object Oriented Programming \(OOP\)](#)
5. [Kotlin Class and Objects - Programiz](#)
6. [Kotlin Tutorial 12 — Encapsulation And Polymorphism](#)
7. <https://codekey.id/kotlin/class-dan-object-kotlin/>
8. [Android Kotlin Basics – Property Encapsulation | Alex Dunn](#)
9. [Lesson 7 - Inheritance and polymorphism in Kotlin | Itchdemry](#)
10. [Kotlin Abstraction | Tutorial Kart](#)





Nah, selesai sudah pembahasan kita di Chapter 1 Topic 6 ini.

Daaan, ini juga menandakan akhir dari keseluruhan Chapter 1 😊

Selanjutnya, kita siap untuk beresin Challenge paling pertama

See you tomorrow~



# Terima Kasih!



Chapter ✓

completed