

Training Kotlin

...

Kotlin Object Oriented Programming

OOP Kotlin Introduction

Mungkin kita sudah pernah mendengar beberapa paradigma seperti *imperative*, *object-oriented*, *procedural*, dan *functional*. Paradigma pemrograman itu apa *sih*? Paradigma pemrograman adalah gaya atau cara kita menulis program.

Pada pemrograman dengan menggunakan pendekatan OOP kita akan menemui beberapa istilah atau pembahasan seperti Class, Attribute dan Function. Perlu kita ingat, bahwa setiap bahasa pemrograman memiliki istilah-istilah tersendiri.

Dalam penerapan OOP kita menggabungkan kumpulan-kumpulan fungsi atau atribut yang memiliki kesamaan dalam sebuah unit yang kita sebut sebagai objek.

Class merupakan sebuah blueprint yang dapat dikembangkan untuk membuat sebuah objek. Blueprint ini merupakan sebuah template yang di dalamnya menjelaskan seperti apa perilaku dari objek itu (berupa properti ataupun function).

Class & Object

- Pembuatan class pada kotlin dengan kata kunci **class**
- Usahakan pemberian nama class sama dengan nama file nya
- Dalam pembuatan sebuah **Object** di kotlin sama hal nya dengan pemanggilan sebuah function dengan memanggil nama class nya
- Jika di java membutuhkan kata kunci **new**, di Kotlin kita tidak memerlukannya

```
1 package oop
2
3 class Animal {}
```

```
1 package oop
2
3 class Animal {}
4
5 fun main() {
6
7     val cat = Animal()
8     println(cat)
9 }
```

```
1 package oop
2
3 class Animal {
4     var name: String = ""
5     var type: String = ""
6     var feet: Int? = null
7 }
8
9 fun main() {
10     val cat = Animal()
11     cat.name = "Pussy"
12     cat.type = "Omnivora"
13     cat.feet = 4
14     println(cat.name)
15     println(cat.type)
16     println(cat.feet)
17 }
```

Data Class

```
1 data class Cat (  
2     var color: String = "",  
3     var height: Double = 0.0,  
4     var length: Double = 0.0,  
5     var weight: Double = 0.0  
6 )
```

```
1 fun main() {  
2     val persian = Cat()  
3     persian.color = "Putih"  
4     persian.height = 24.0  
5     persian.length = 46.0  
6     persian.weight = 2.0  
7     val bengal = Cat("Coklat", 22.0, 39.0, 2.3)  
8     val anggora = Cat("Abu-abu", 25.0, 41.0, 2.4)  
9  
10    println(persian)  
11    println(bengal)  
12    println(anggora)  
13 }
```

Uji coba jika kata **data** dihilangkan, apa yang terjadi?

Constructor



```
1 class Student(firstName: String, middleName: String?, lastName: String) {  
2  
3     var firstName: String = firstName;  
4     var middleName: String? = middleName;  
5     var lastName: String = lastName;  
6  
7 }  
8  
9 fun main() {  
10  
11     val jution = Student("Jution", "", "Kirana")  
12     println("First Name: ${jution.firstName}")  
13     println("Middle Name: ${jution.middleName}")  
14     println("Last Name: ${jution.lastName}")  
15 }
```

Constructor

```
1 package oop
2
3 class Hero(var name: String, var hp: Double, var damage: Double) {
4
5     fun printInfo() {
6         println(this)
7         println("My Hero is a name: ${this.name}, hp: ${this.hp}, damage: ${this.damage}")
8     }
9 }
10
11 fun main() {
12     val gundala = Hero("Gundala Jagoan Petir", 10.0, 100.0)
13     gundala.printInfo()
14 }
```

Initializer Block

- Initializer Block adalah blok kode yang akan dieksekusi ketika constructor dipanggil
- Kita bisa memasukkan kode program di dalam initializer block

```
1 class Student(firstName: String, middleName: String?, lastName: String) {  
2  
3     var firstName: String = firstName;  
4     var middleName: String? = middleName;  
5     var lastName: String = lastName;  
6  
7     init {  
8         println("Initializer Block")  
9     }  
10  
11 }  
12  
13 fun main() {  
14  
15     val jution = Student("Jution", "", "Kirana")  
16     println("First Name: ${jution.firstName}")  
17     println("Middle Name: ${jution.middleName}")  
18     println("Last Name: ${jution.lastName}")  
19 }
```

Initializer Block



```
1 class Student(firstName: String, middleName: String?, lastName: String) {  
2  
3     init {  
4         println("First Name: $firstName")  
5         println("Middle Name: $middleName")  
6         println("Last Name: $lastName")  
7     }  
8 }  
9  
10 fun main() {  
11  
12     val jution = Student("Jution", "", "Kirana")  
13 }
```


Secondary Constructor

- Kotlin mendukung pembuatan constructor lebih dari satu, constructor yang utama yang terdapat di Class, disebut primary constructor, constructor tambahan yang bisa kita buat lagi adalah secondary constructor
- Saat membuat constructor, kita wajib memanggil primary constructor jika ada primary constructor

```
1 class SecondaryConstructor(var name: String) {
2
3     var age: Int = 10
4     init {
5         //ini akan di cetak pertama, dan umur langsung mengambil dari properti yang di buat
6         println("Nama saya $name dan umur saya $age")
7     }
8
9     // secondary constructor
10    // ini akan di eskekusi setelah init
11    constructor(name: String, age: Int): this(name) {
12        this.age = age
13
14        // Buat kondisi dimana, jika nama yang di masukkan kurang dari 5
15        // mengeluarkan peringatan: Nama harus lebih dari 5
16
17        // Untuk umur, jika umur < 17 keluarkan peringatan belum dewasa, jika lebih OK
18    }
19 }
20
21 fun main() {
22     val user = SecondaryConstructor("Jution", 25)
23     println("Nama saya ${user.name} dan umur saya ${user.age}")
24 }
```

Function

```
1 package oop
2
3 class Hero(var name: String, var hp: Double, var damage: Double) {
4
5     fun printInfo() {
6         println(this)
7         println("My Hero is a name: ${this.name}, hp: ${this.hp}, damage: ${this.damage}")
8     }
9 }
10
11 fun main() {
12     val gundala = Hero("Gundala Jagoan Petir", 10.0, 100.0)
13     gundala.printInfo()
14 }
```

Function Overloading

- Function Overloading adalah kemampuan membuat function dengan nama yang sama di dalam class
- Untuk membuat function dengan nama yang sama, kita wajib menggunakan parameter yang berbeda, bisa tipe parameter yang berbeda, atau jumlah parameter yang berbeda

```
1 class Customer(var name: String) {
2
3     fun buySomething(productName: String, productPrice: Int) {
4         println("Customer dengan nama ${this.name} membeli barang $productName dengan harga $productPrice")
5     }
6
7     //overloading
8     fun buySomething(productName: String, productPrice: Int, productQty: Int) {
9         println("Customer dengan nama ${this.name} membeli barang $productName dengan harga $productPrice sejumlah $productQty")
10    }
11 }
12
13 fun main() {
14
15     val budi = Customer("Budi")
16     budi.buySomething("Laptop", 10_000_000)
17
18     val joni = Customer("Joni")
19     joni.buySomething("Iphone 12 Pro Max", 25_000_000, 10)
20 }
```

Function Overloading - Hands On

Buat sebuah class yang meng overloading sebuah function untuk mencari luas lingkaran dengan 2 kemungkinan $22/7$ dan 3.14 (inputan)

Hasil Output:

Area dengan phi ditentukan:

Area dengan phi $22/7$:

Inheritance

- Inheritance atau pewarisan sifat.
- Child class hanya dapat mempunyai 1 parent class, tetapi parent class bisa memiliki lebih dari satu child
- Di Kotlin class bersifat final, artinya tidak dapat diturunkan/diwariskan, untuk itu untuk melakukan pewarisan dapat menggunakan kata kunci **open** di depan sebuah class

```
1 package oop
2
3 open class Employee(val name: String) {
4
5     fun sayHello(name: String) {
6         println("Hello $name, my name is ${this.name}")
7     }
8 }
9
10 class Manager(name: String): Employee(name)
11 class VicePresident(name: String): Employee(name)
12
13 fun main() {
14     val jution = Manager("Jution")
15     jution.sayHello("Arif")
16
17     val doni = VicePresident("Doni")
18     doni.sayHello("Arif")
19 }
```

```
1 package oop
2
3 open class Employee(val name: String) {
4     open fun sayHello(name: String) {
5         println("Hello $name, my name is ${this.name}")
6     }
7 }
8
9 class Manager(name: String): Employee(name) {
10     override fun sayHello(name: String) {
11         println("Hello $name, my name is manager ${this.name}")
12     }
13 }
14
15 fun main() {
16     val jution = Manager("Jution")
17     jution.sayHello("Arif")
18 }
```


Inheritance in Constructor

```
1 open class Parent(name: String) {
2     init {
3         println("$name dari class parent")
4     }
5     constructor(name: String, address: String): this(name) {
6         println("$address dari secondary constructor class parent")
7     }
8 }
9
10 class Child: Parent {
11     // Disini katakunci super di ambil dari class parent dengan constructor primary
12     constructor(name: String, age: Int): super(name) {
13         println("$age dari primary constructor class child")
14     }
15     // Disini katakunci super di ambil dari class parent dengan constructor secondary nya
16     constructor(name: String, age: Int, address: String): super(name, address) {
17         println("$age dari secondary constructor class child")
18     }
19 }
20
21 fun main() {
22     val child = Child("Jution", 25)
23     val child2 = Child("Doni", 20, "Jakarta Selatan")
24 }
```

Overriding

```
1 open class Development {
2     open var name: String = "Basic Programming with Kotlin"
3     open fun skill() {
4         println("Android")
5     }
6 }
7
8 class WebDev: Development() {
9     override var name: String = "Basic HTML"
10    //panggil ini jika ingin memanggil name di super class
11    //get() = super.name
12    override fun skill() {
13        super.skill()
14        println("Front End")
15    }
16 }
17
18 fun main() {
19     val webDev = WebDev()
20     println(webDev.name)
21     webDev.skill()
22 }
```

Abstract Class



```
1 abstract class Location(val name: String)
2
3 class City(name: String) : Location(name)
4
5 fun main() {
6     // Error
7     // val location = Location("sss")
8     val city = City("Jakarta")
9     println(city.name)
10 }
```


Getter and Setter

Secara default di kotlin sudah menerapkan Getter dan setter secara otomatis, jika kita membuat sebuah properti **var**, tetapi jika kita membuat properti **val** hanya dapat getter saja.

Kata kunci Getter adalah **Get() = field**

Kata kunci Setter adalah **Set(value)**

```
1 class Note(title: String) {
2     //Kalau pake var, bisa get dan set, kalau val hanya bisa get
3     var title: String = title
4     //field adalah kata kunci, dalam hal ini isinya adalah title
5     get() = field.toUpperCase()
6     //value juga kata kunci
7     set(value) {
8         if (value.isNotBlank()) {
9             field = value
10        }
11    }
12 }
13
14 fun main() {
15     val note = Note("Catatanku")
16     note.title = ""
17     println(note.title)
18 }
```

Lateinit

- Properties di kotlin wajib di inisialisasi di awal deklarasi
- Dengan kata kunci **lateinit**, kita bisa membuat properties tanpa harus langsung mengisi datanya
- lateinit hanya bisa digunakan di var

```
1 class Television {
2     lateinit var brand: String
3
4     fun initTelevision(brand: String) {
5         this.brand = brand
6     }
7 }
8
9 fun main() {
10     val tv1 = Television()
11     //Error karena fun initTelevision belum di call
12     // println(tv1.brand)
13     tv1.initTelevision("LG")
14     println(tv1.brand)
15 }
```

Interface

```
1 package oop
2
3 interface Engine {
4
5     fun starEngine()
6 }
```

```
1 package oop
2
3 class ElectricEngine: Engine {
4
5     override fun starEngine() {
6         println("Mesin listrik menyala...")
7     }
8 }
```

```
1 package oop
2
3 fun main() {
4
5     val electricEngine = ElectricEngine()
6     electricEngine.starEngine()
7     val gasolineEngine = GasolineEngine()
8     gasolineEngine.starEngine()
9
10 }
```

```
1 package oop
2
3 class GasolineEngine: Engine {
4     override fun starEngine() {
5         println("Mesin gasoline menyala...")
6     }
7 }
```

Concrete Function in Interface

- Function di Interface memiliki pengecualian, tidak harus abstract
- Kita bisa membuat concrete function di Interface, artinya function tersebut tidak wajib untuk dibuat ulang di child Class nya

```
1 interface Interaction {  
2     // wajib di override  
3     val name: String  
4     fun home(address1: String, address2: String)  
5     // tidak wajib di override  
6     fun sayHello(name: String) {  
7         println("Hello $name, my name is ${this.name}")  
8     }  
9 }
```

Concrete Function in Interface [2]



```
1 class HumanInteraction(override val name: String): Interaction {
2     override fun home(address1: String, address2: String) {
3         println("Hi $name, I Live in $address1, $address2")
4     }
5 }
6
7 fun main() {
8     val humanInteraction = HumanInteraction("Jution")
9     humanInteraction.home("Kemayoran", "Jakarta Pusat")
10 }
```

Multiple Interface

- Inheritance di Class hanya boleh memiliki satu class Parent
- Di Interface, sebuah class Child bisa memiliki banyak interface Parent

```
1 interface Interaction {
2     // wajib di override
3     val name: String
4     fun home(address1: String, address2: String)
5     // tidak wajib di override
6     fun sayHello(name: String) {
7         println("Hello $name, my name is ${this.name} :from concrete function interface")
8     }
9 }
10
11 interface Go {
12     fun go()
13 }
14
15 class HumanInteraction(override val name: String): Interaction, Go {
16     override fun home(address1: String, address2: String) {
17         println("Hi $name, I Live in $address1, $address2")
18     }
19
20     override fun go() {
21         println("Go!")
22     }
23 }
24
25 fun main() {
26     val humanInteraction = HumanInteraction("Jution")
27     humanInteraction.go()
28     humanInteraction.sayHello("Doni")
29     humanInteraction.home("Kemayoran", "Jakarta Pusat")
30 }
```

Inheritance Interface

```
1 interface Interaction {
2     val name: String
3     fun home(address1: String, address2: String)
4     fun sayHello(name: String) = println("Hello $name, my name is ${this.name} :from concrete function interface")
5 }
6
7 interface Go: Interaction {
8     fun go()
9 }
10
11 class HumanInteraction(override val name: String): Go {
12     override fun go() {
13         TODO("Not yet implemented")
14     }
15
16     override fun home(address1: String, address2: String) {
17         TODO("Not yet implemented")
18     }
19 }
20
21
22 fun main() {
23     val humanInteraction = HumanInteraction("Jution")
24     humanInteraction.go()
25     humanInteraction.sayHello("Doni")
26     humanInteraction.home("Kemayoran", "Jakarta Pusat")
27 }
```

Interface Conflict

- Terkadang saat kita membuat sebuah interface memiliki sebuah function/method yang sama
- Ketika meng-inheritance sebuah interface yang memiliki function/method yang sama akan terjadi sebuah konflik

```
1 interface MoveA {
2     fun move() = println("Move A")
3 }
4
5 interface MoveB {
6     fun move() = println("Move B")
7 }
8
9 class Human(): MoveA, MoveB {
10    // Conflict
11    override fun move() {
12        TODO("Not yet implemented")
13    }
14    // Conflict
15    override fun move() {
16        TODO("Not yet implemented")
17    }
18
19 }
```

```
1 interface MoveA {
2     fun move() = println("Move A")
3 }
4
5 interface MoveB {
6     fun move() = println("Move B")
7 }
8
9 class Human(): MoveA, MoveB {
10    // Fix conflict
11    override fun move() {
12        super<MoveA>.move()
13        super<MoveB>.move()
14    }
15 }
```


Visibility Modifiers

Visibility Modifiers	Keterangan
public	Jika tidak menyebutkan, maka secara otomatis visibility modifiers nya adalah public, yang artinya bisa diakses dari manapun
private	Artinya hanya bisa diakses di tempat deklarasinya
protected	Artinya hanya bisa diakses di tempat deklarasi, dan juga turunannya
internal	Artinya hanya bisa diakses di module/project yang sama.

Coroutine

Pengenalan Coroutine

Pengenalan Coroutine

- Coroutine sering diistilahkan sebagai lightweight thread (thread ringan), walaupun sebenarnya coroutine sendiri bukanlah thread.
- Coroutine sebenarnya di eksekusi di dalam thread, namun dengan coroutine sebuah thread bisa memiliki kemampuan untuk menjalankan beberapa coroutine secara bergantian (concurrent)
- Artinya jika sebuah thread menjalankan 10 coroutine, sebenarnya thread akan menjalankan coroutine satu per satu secara bergantian
- Perbedaan lain thread dan coroutine adalah coroutine itu murah dan cepat, sehingga kita bisa membuat ribuan atau bahkan jutaan coroutine secara cepat dan murah tanpa takut kelebihan memory
- Fungsi dari coroutine yaitu mencegah terjadinya blocking yang disebabkan oleh menunggu result yang terlalu lama dan mencegah beban yang berlebih pada thread utama ketika kita menjalani proses di thread utama

Suspend Function

- Suspend computation adalah komputasi yang bisa ditangguhkan (ditunda waktu eksekusinya).
- Sebelumnya kita tahu untuk menangguhkan komputasi di Java, kita biasanya menggunakan `Thread.sleep()`, sayangnya `Thread.sleep()` akan mem-block thread yang sedang berjalan saat ini. Sehingga tidak bisa digunakan.
- Kotlin memiliki sebuah fitur bernama suspending function, dimana kita bisa menangguhkan waktu eksekusi sebuah function, tanpa harus mem-block thread yang sedang menjalankannya.
- Syarat menjalankan suspend function di Kotlin adalah, harus dipanggil dari suspend function lainnya.

Membuat Coroutine

- Coroutine tidak bisa berjalan sendiri, dia perlu berjalan di dalam sebuah Scope.
- Salah satu scope yang bisa kita gunakan adalah GlobalScope (masih banyak scope yang ada, dan akan kita bahas nanti dimateri tersendiri)
- Untuk membuat coroutine, kita bisa menggunakan method launch()
- Dan di dalam coroutine, kita bisa memanggil suspend function