Konsep OOP di Kotlin

Praktikum 3

Tujuan :

Mahasiswa mengenal dan mengimplementasikan :

- Kelas dan Objek
- Enkapsulasi
- Inheritance
- Polymorphism
- Visibility Modifier
- Access Modifier
- Kelas Abstrak
- Interfaces
- Nested dan Inner Class
- Data Class

Hasil Praktikum :

- Contoh program praktikum.

Kelas dan Objek

Kelas dapat diumpamakan seperti spesifikasi atau *blueprint* (cetakan) untuk membuat sebuah objek. Anggota kelas (*class member*) terdiri dari, **konstruktor** dan **initializer block**, **fungsi**, **property**, **nested** dan **inner class**, dan **deklarasi objek**. Kelas di Kotlin sedikit berbeda implementasinya dengan Java, Kotlin secara default mendeklarasikan anggota kelas dengan akses modifier *final* dan *visibility modifier public*. Deklarasi sebuah kelas terdiri dari 1) keyword ***class***; 2) sebuah identifier sebagai ***nama kelas***; 3) opsional ***header***; 4) dan opsional ***body***. Berikut adalah contoh cara mendefenisikan kelas Person :

```
class Person() {}
```

Header ditandai dengan tanda kurung berwarna merah dan biasanya didalam *header* tersebut terdapat parameter yang selanjutnya disebut juga dengan *primary constructor*. Pasangan kurung kurawal berwarna biru adalah bagian tubuh (*body*) suatu kelas. *Header* dan *body* bersifat opsional artinya keberadaanya boleh ada atau tidak. Adapun untuk menginstansiasi kelas Person, kita dapat menulisnya seperti berikut :

```
val person = Person()
```

Catatan : Kotlin tidak memiliki keyword *new* seperti yang ada di Java

Contoh penggunaan kelas di Kotlin :

Buatlah *package* baru pada folder *src* dengan nama "*id.ac.polbeng.depandi.test_kelas*". Pada *package* yang baru dibuat, buatlah kelas baru dengan nama Greeter.

Nama file : Greeter.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas


class Greeter {
    fun greet() {
        println("Hello object world!")
    }
}
```

Nama file : GreeterTest.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas


fun main() {
    val greeter = Greeter()
    greeter.greet()
}
```

Output :

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Hello object world!

Process finished with exit code 0
```

Ubahlah kode program Greeter.kt, tambahkan *class member* yaitu sebuah atribut dengan nama text dan dua buah method dengan nama *greet* yang mempunyai satu parameter bertipe String dan *with_ret_val* yang memiliki satu parameter bertipe String dengan nilai kembali String. Sehingga kode program menjadi seperti berikut:

Nama file : Greeter.kt	Package : id.ac.polbeng.dependi.test_kelas

```
package id.ac.polbeng.dependi.test_kelas

class Greeter {
    var text: String = ""
    fun greet() {
        println("Hello object world!")
    }
    fun greet(name: String) {
        println("$text $name")
    }
    fun with_ret_val(name: String): String {
        return "$text $name"
    }
}
```

Ubahlah kode program GreeterTest, berikan nilai text dengan “Hi” kemudian panggillah method *greet* dari kelas Greeter dan masukkan nilainya dengan “Anton” dan “Budi”. Baris selanjutnya tambahkan nilai text dengan “Hello programmer” dan panggil fungsi *with_ret_val* dengan parameter String “Dono”.

Nama file : GreeterTest.kt	Package : id.ac.polbeng.dependi.test_kelas

```
package id.ac.polbeng.dependi.test_kelas

fun main() {
    val greeter = Greeter()
    greeter.greet()
    greeter.text = "Hi"
    greeter.greet("Anton")
    greeter.greet("Budi")
    greeter.text = "Hello programmer"
    println(greeter.with_ret_val("Dono"))
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Hello object world!
Hi Anton
Hi Budi
Hello programmer Dono

Process finished with exit code 0
```

Contructor

Sebuah kelas di Kotlin dapat memiliki satu konstruktor utama (*primary constructor*) dan satu atau lebih konstruktor tambahan (*secondary constructor*). Konstruktor utama adalah bagian dari *header* suatu kelas, ia didefenisikan setelah nama kelas dan opsional tipe parameter.

```
class ClassName [visibility_modifiers] constructor(firstName: String) { /*[Class Body]*/ }
```

Jika konstruktor utama tidak memiliki apapun notasi atau *visibility modifier*, maka *keyword constructor* dapat dihilangkan hingga kode menjadi ringkas seperti berikut:

```
class Person(firstName: String) { /*...*/ }
```

Konstruktor utama tidak berisi kode apapun. Inisialisasi nilai *default* atribut kelas dapat ditaruh didalam *initializer block*.

Initializer Block

Blok inisialisasi diawali dengan keyword **init**. Selama instansiasi sebuah objek, urutan eksekusi blok inisialisasi sama dengan urutan yang ada pada tubuh kelas, berselangkan inisialisasi property-nya.

Contoh kode program :

Nama file : InitOrder.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class InitOrder(name: String) {
    val firstProperty = "First property: $name".also(::println)
    init {
        println("First initializer block that prints ${name}")
    }
    val secondProperty = "Second property: ${name.length}".also(::println)
    init {
        println("Second initializer block that prints ${name.length}")
    }
}

fun main(){
    val initOrder = InitOrder("RPL")
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
First property: RPL
First initializer block that prints RPL
Second property: 3
Second initializer block that prints 3

Process finished with exit code 0
```

Parameter dari konstruktor utama dapat digunakan dalam blok inisialisasi. Ia juga dapat digunakan untuk menginisialisasi property yang dideklarasikan dalam tubuh kelas.

Contoh kode program :

Nama file : PersonA.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class PersonA (_firstName: String, _lastName: String, _age: Int) {
    var firstName:String
    var lastName:String
    var age: Int
    init {
        firstName = _firstName
        lastName = _lastName
        age = _age
    }
}

fun main(){
    val budi = PersonA("Budi", "Gunawan", 21)
    println("Name : ${budi.firstName} ${budi.lastName}")
    println("Age : ${budi.age}")
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Name : Budi Gunawan
Age : 21

Process finished with exit code 0
```

Untuk lebih meringkas kode program kelas Person diatas maka kita dapat menggabungkan proses deklarasi dan menginisialisasi sebuah variabel pada blok inisialisasi kedalam satu *statement* seperti berikut:

Nama file : PersonB.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class PersonB (_firstName: String, _lastName: String, _age: Int) {
```

```
    var firstName:String = _firstName
    var lastName:String = _lastName
    var age: Int = _age
}

fun main(){
    val budi = PersonB("Budi", "Gunawan", 21)
    println("Name : ${budi.firstName} ${budi.lastName}")
    println("Age : ${budi.age}")
}
```

Untuk lebih ringkasnya lagi, deklarasi dan inisialisasi property dapat dilakukan hanya didalam parameter konstruktor utama, seperti contoh kode program berikut:

Nama file : PersonC.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class PersonC (var firstName: String, var lastName: String, var age: Int) {}

fun main(){
    val budi = PersonC("Budi", "Gunawan", 21)
    println("Name : ${budi.firstName} ${budi.lastName}")
    println("Age : ${budi.age}")
}
```

Secondary Constructor

Konstruktor tambahan (*secondary constructor*) didefenisikan didalam tubuh kelas. Jika property kelas tidak diinisialisasi pada saat mendeklarasikannya maka property tersebut harus di inisialisasi didalam konstruktor tambahan atau blok inisialisasi. Konstruktor tambahan memerlukan *keyword constructor* untuk mendefenisikannya. Selain itu konstruktor tambahan boleh tidak memiliki tubuh kelas dalam pendefenisiannya. Berikut contoh kode program kelas Person yang memiliki konstruktor tambahan tapi tidak memiliki konstruktor utama.

Nama file : PersonD.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class PersonD {
    var firstName: String
    var lastName: String
    var age: Int
    constructor(_firstName: String, _lastName: String, _age: Int){
        firstName = _firstName
        lastName = _lastName
        age = _age
    }
```

```
}

fun main(){
    val budi = PersonD("Budi", "Gunawan", 21)
    println("Name : ${budi.firstName} ${budi.lastName}")
    println("Age : ${budi.age}")
}
```

Konstruktor tambahan bisa didefenisikan lebih dari satu.

Contoh kode program :

Nama file : PersonE.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class PersonE {
    var firstName: String
    var lastName: String
    var age: Int = 25
    constructor(_firstName: String, _lastName: String){
        firstName = _firstName
        lastName = _lastName
    }
    constructor(_firstName: String, _lastName: String, _age: Int): this(_firstName, _lastName){
        age = _age
    }
    fun cetakInfo(){
        println("Name : ${firstName} ${lastName}")
        println("Age : ${age}")
    }
}

fun main(){
    val anton = PersonE("Frank", "Lampard")
    anton.cetakInfo()
    println()
    val budi = PersonE("Budi", "Gunawan", 21)
    budi.cetakInfo()
}
```

Note: Keyword ***this*** merujuk pada konstruktor kelas saat ini yang bersesuaian. Pada contoh diatas ***this(_firstName, _lastName)*** merujuk pada konstruktor yang berwarna kuning dan akan memanggilnya terlebih dahulu sebelum kode didalam konstruktor yang berwarna hijau dieksekusi.

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Name : Frank Lampard
Age : 25

Name : Budi Gunawan
Age : 21

Process finished with exit code 0
```

Jika sebuah kelas terdapat konstruktor utama dan tambahan secara bersamaan, maka setiap konstruktor tambahan harus mendelegasi (*delegate*) konstruktor utama, baik secara langsung maupun tidak langsung melalui konstruktor tambahan. Delegasi ke konstruktor lain di kelas yang sama dilakukan menggunakan keyword **this**.

Contoh kode program:

Nama file : PersonF.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class PersonF (val firstName: String, val lastName: String) {
    init {
        println("Sedang inisialisasi wak!")
    }
    constructor(_firstName: String, _lastName: String, _age:Int): this(_firstName, _lastName){
        println("Name : $_firstName $_lastName")
        println("Age : $_age")
        println()
    }
    constructor(_firstName: String, _lastName:String, _country:String): this(_firstName, _lastName){
        println("Name : $_firstName $_lastName")
        println("Country : $_country")
        println()
    }
    constructor(_firstName: String, _lastName:String, _age:Int, _country:String): this(_firstName,
_lastName, _age){
        println("Name : $_firstName $_lastName")
        println("Age : $_age")
        println("Country : $_country")
    }
}

fun main(){
    val anton = PersonF("Anton", "Sejati")
    val budi = PersonF("Budi", "Gunawan", 21)
    val caca = PersonF("Caca", "Andika", "Indonesia")
    val dono = PersonF("Dono", "Putri", 25, "Indonesia")
}
```

Note: Terdapat 1 konstruktor utama berwarna kuning dan 3 buah konstruktor tambahan berwarna hijau, merah dan biru. Konstruktor tambahan pertama dan kedua mendelegasikan konstruktor utama. Sedangkan konstruktor tambahan ketiga medelegasikan konstruktor tambahan pertama yang secara tidak langsung juga mendelegasikan konstruktor utama.

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Sedang inisialisasi wak!
Sedang inisialisasi wak!
Name : Budi Gunawan
Age : 21

Sedang inisialisasi wak!
Name : Caca Andika
Country : Indonesia

Sedang inisialisasi wak!
Name : Dono Putri
Age : 25

Name : Dono Putri
Age : 25
Country : Indonesia

Process finished with exit code 0
```

Enkapsulasi

Enkapsulasi adalah konsep OOP yang memungkinkan kita untuk menyembunyikan informasi dari suatu kelas sehingga anggota-anggota kelas tersebut tidak bisa diakses dari luar. Biasanya di Java dilakukan dengan memberikan akses control *private* ketika mendeklarasikan suatu atribut/property dan menambahkan 2 method untuk mengakses atribut tersebut yaitu method *Getter* dan *Setter*.

Atribut atau property di Kotlin dapat dideklarasikan menggunakan keyword val atau var.

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

Sedangkan untuk mengakses property sebuah objek dapat dilakukan dengan cara menggunakan notasi titik (‘.’) berikut:

```
fun copyAddress(address: Address): Address {
```

```
        val result = Address() // there's no 'new' keyword in Kotlin
        result.name = address.name // accessors are called
        result.street = address.street
        // ...
        return result
}
```

Getter dan Setter

Sintaks lengkap untuk mendeklarasikan sebuah property adalah seperti berikut :

```
var <property name>:[<property type>][=<initializer>]
        [<getter>]
        [<setter>]
```

Inisialisasi sintaks *getter* dan *setter* di Kotlin bersifat opsional, karena secara sefault Kotlin telah men-*generate* method *getter* dan *setter* untuk property yang *mutable* (var) dan hanya *getter* untuk property yang *read-only* (val). Secara default kode yang ter-*generate* secara otomatis tersebut adalah seperti pada contoh berikut :

Nama file : EnkapsulasiA.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class Employee(_id: Int, _name: String, _age: Int) {
    val id: Int = _id
        get() = field

    var name: String = _name
        get() = field
        set(value) {
            field = value
        }

    var age: Int = _age
        get() = field
        set(value) {
            field = value
        }
}

fun main() {
    val emp = Employee(1101, "Jono", 25)
    println("Id : ${emp.id}, Nama : ${emp.name}, Umur : ${emp.age}")
}
```

Terdapat dua *keyword* asing yaitu value dan field. Value adalah nama parameter setter. Field atau disebut juga backing fields membantu kita untuk merujuk property didalam method getter dan setter. Field ini sangat dibutuhkan, jika kita menggunakan property secara langsung didalam method getter dan setter maka kita akan memanggil program secara rekursif dan menyebabkan error *StackOverflowError*.

Kita juga dapan merubah method setter dan getter default, sesuai dengan keinginan kita seperti berikut:

Nama file : EnkapsulasiB.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class EmployeeA(_id: Int, _name: String, _age: Int) {
    val id: Int = _id
        get() = field

    var name: String = _name
        get(){
            return field.toUpperCase()
        }
        set(value) {
            field = value
        }

    var age: Int = _age
        get() = field
        set(value) {
            field = if(value > 0) value else throw IllegalArgumentException("Age must be greater than
zero")
        }
}

fun main() {
    val emp = EmployeeA(1101, "Jono", 25)
    println("Id : ${emp.id}, Nama : ${emp.name}, Umur : ${emp.age}")
    emp.age = -1
}
```

Output :

```
Id : 1101, Nama : JONO, Umur : 25
Exception in thread "main" java.lang.IllegalArgumentException: Age must be greater than zero
    at id.ac.polbeng.depandi.test_kelas.EmployeeA.setAge(EEnkapsulasiB.kt:18)
    at id.ac.polbeng.depandi.test_kelas.EEnkapsulasiBKt.main(EEnkapsulasiB.kt:25)
    at id.ac.polbeng.depandi.test_kelas.EEnkapsulasiBKt.main(EEnkapsulasiB.kt)

Process finished with exit code 1
```

Note : Perhatikan hasil keluaran program lakukan analisa terhadap property nama dan umur .

Inheritance

Inheritance adalah salah satu konsep OOP. Inheritance mengizinkan sebuah kelas untuk mewariskan fitur kelas (property dan method) ke kelas lainnya. Kelas yang mewarisi fitur kelas lainnya dipanggil *child class* atau *derived class* atau *subclass* dan kelas yang fiturnya diwariskan dipanggil *parent class* atau *base class* atau *super class*. Semua kelas di Kotlin memiliki base class yang dipanggil Any. Ia berkorespodensi dengan kelas Object di Java. Setiap kelas yang dibuat di Kotlin secara implisit pewarisan dari kelas Any.

```
class Person // Implicitly inherits from the default Super class - Any
```

Kelas Any berisi tiga method yaitu equals() , hashCode() dan toString(). Semua kelas di Kotlin mewarisi ketiga method dari kelas Any dan dapat di override.

Membuat kelas parent dan child

Secara default semua kelas di Kotlin memiliki akses modifier final, artinya tidak dapat di wariskan. Agar dapat diwariskan ke kelas anak, kelas induk harus memiliki akses modifier open. Namun kelas anak memiliki tanggung jawab untuk menginisialisasi kelas induknya. Jika kelas anak memiliki sebuah *primary constructor*, maka ia harus menginisialisasi kelas induknya di bagian kanan kelas header dengan melewatkan parameter ke *primary constructor* nya.

Berikut adalah contoh kode program :

Nama file : InheritanceA.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

// Parent class
open class Computer(val name: String,
                    val brand: String) {
}

// Child class (initializes the parent class)
class Laptop(name: String,
             brand: String,
             val batteryLife: Double) : Computer(name, brand) {
    fun info(){
        println("Name : $name")
        println("Brand : $brand")
        println("Battery Life : $batteryLife")
    }
}

fun main(){
```

```
    val myLaptop = Laptop("Acer Aspire 4738", "Acer", 2.5)
    println(myLaptop.info())
}
```

Jika di kelas anak tidak memiliki primary constructor, maka semua secondary constructor yang di inisialisasi kelas induk dipanggil dengan keyword *super* secara langsung atau mendelegasikan ke konstruktor yang lain.

Contoh kode program:

Nama file : InheritanceB.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

// Child class (initializes the parent class)
class LaptopB : Computer{
    val batteryLife: Double

    // Calls super() to initialize the Parent class
    constructor(name: String, brand: String, batteryLife: Double): super(name, brand) {
        this.batteryLife = batteryLife
    }

    // Calls another constructor (which calls super())
    constructor(name: String, brand: String): this(name, brand, 0.0) {

    }

    fun info(){
        println("Name : $name")
        println("Brand : $brand")
        println("Battery Life : $batteryLife")
    }
}

fun main(){
    val myLaptop = LaptopB("Asus Think Pad", "Asus")
    println(myLaptop.info())
}
```

Overriding Function

Kotlin membutuhkan eksplisit akses modifier untuk meng-*override* fungsi pada kelas induk yaitu akses modifier *open*.

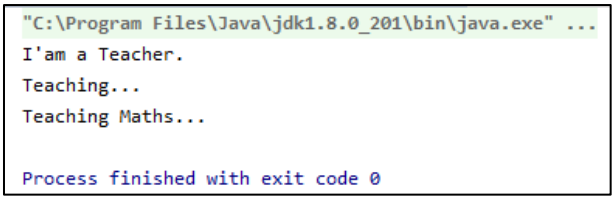
Contoh kode program :

Nama file : InheritanceC.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas
```

```
open class Teacher {
    // Must use "open" modifier to allow child classes to override it
    open fun teach() {
        println("Teaching...")
    }

    fun info(){
        println("I'am a Teacher.")
    }
}

class MathsTeacher : Teacher() {
    // Must use "override" modifier to override a base class function
    override fun teach() {
        println("Teaching Maths...")
    }
}

fun main() {
    val teacher = Teacher()
    val mathsTeacher = MathsTeacher()
    mathsTeacher.info()
    teacher.teach()
    mathsTeacher.teach()
}
```

Output:



Overriding Property

Sama halnya seperti fungsi, property di Kotlin pada kelas induk dapat di override dengan menggunakan akses modifier open.

Contoh kode program :

Nama file : InheritanceD.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

import java.text.NumberFormat
import java.util.*
```

```
open class EmployeeD {
    // Use "open" modifier to allow child classes to override this property
    open val baseSalary: Int = 3000000
}

class Programmer : EmployeeD() {
    // Use "override" modifier to override the property of base class
    override val baseSalary: Int = 5000000
}

class SoftwareEngineer : EmployeeD() {
    // Use "override" modifier to override the property of base class
    override val baseSalary: Int = 6000000
}

class ProjectManager : EmployeeD() {
    // Use "override" modifier to override the property of base class
    override val baseSalary: Int = 8000000
}

fun toRupiahFormat(baseSalary: Int) : String{
    val localeID =  Locale("in", "ID")
    val numberFormat = NumberFormat.getCurrencyInstance(localeID)
    return numberFormat.format(baseSalary).toString()
}

fun main() {
    val employee = EmployeeD()
    println("Rata-rata gaji ${employee.javaClass.simpleName} = ${toRupiahFormat(employee.baseSalary)}")

    val programmer = Programmer()
    println("Rata-rata gaji ${programmer.javaClass.simpleName} =
${toRupiahFormat(programmer.baseSalary)}")

    val softwareEngineer = SoftwareEngineer()
    println("Rata-rata gaji ${softwareEngineer.javaClass.simpleName} =
${toRupiahFormat(softwareEngineer.baseSalary)}")

    val projectManager = ProjectManager()
    println("Rata-rata gaji ${projectManager.javaClass.simpleName} =
${toRupiahFormat(projectManager.baseSalary)}")
}
```

Output :

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Rata-rata gaji EmployeeD = Rp3.000.000,00
Rata-rata gaji Programmer = Rp5.000.000,00
Rata-rata gaji SoftwareEngineer = Rp6.000.000,00
Rata-rata gaji ProjectManager = Rp8.000.000,00

Process finished with exit code 0
```

Overriding Property's Getter/Setter method

Kita dapat meng-override property kelas menggunakan inisialisasi atau menggunakan custom setter dan getter function.

Contoh kode program :

Nama file : InheritanceE.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

open class NewPerson {
    open var age: Int = 1
}

class CheckedPerson: NewPerson() {
    override var age: Int = 1
        set(value) {
            field = if(value > 0) value else throw IllegalArgumentException("Age can not be negative")
        }
}

fun main() {
    val person = NewPerson()
    person.age = -5 // Works

    val checkedPerson = CheckedPerson()
    checkedPerson.age = -5  // Throws IllegalArgumentException : Age can not be negative
}
```

Memanggil property dan fungsi dari kelas induk

Ketika kita meng-*override property* atau fungsi anggota kelas dari kelas induk, maka dari kelas anak kita dapat mengakses *property* atau fungsi anggota dari kelas induknya. Caranya adalah dengan menggunakan keyword *super()*, contohnya adalah seperti berikut :

Nama file : InheritanceF.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

open class EmployeeF {
    open val baseSalary: Double = 10000.0
```

```
    open fun displayDetails() {
        println("I am an Employee")
    }
}

class Developer: EmployeeF() {
    override var baseSalary: Double = super.baseSalary + 10000.0
    override fun displayDetails() {
        super.displayDetails()
        println("I am a Developer")
    }
}

fun main(){
    val employeeF = EmployeeF()
    println("${employeeF.javaClass.simpleName} base salary is ${employeeF.baseSalary}")
    println(employeeF.displayDetails())
    val developer = Developer()
    println("${developer.javaClass.simpleName} base salary is ${developer.baseSalary}")
    println(developer.displayDetails())
}
```

Menurunkan urutan inisialisasi kelas turunan

Selama pembuatan sebuah instance baru dari kelas turunan, inisialisasi kelas induk diselesaikan terlebih dahulu pada langkah pertama (hanya mendahulukan evaluasi dari argument untuk konstruktor kelas induk) dan demikian terjadi sebelum inisialisasi logika dari kelas turunan yang berjalan.

Contoh kode program :

Nama file : InheritanceG.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

open class Base(val name: String) {
    init {
        println("Initializing Base")
    }
    open val size: Int = name.length.also {
        println("Initializing size in Base: $it")
    }
}

class Derived(name: String, val lastName: String) :
    Base(name.capitalize().also { println("Argument for Base: $it") }) {
    init {
```

```
        println("Initializing Derived")
    }
    override val size: Int = (super.size + lastName.length).also {
        println("Initializing size in Derived: $it")
    }
}

fun main(){
    val turunan = Derived("Depandi", "Enda")
    turunan.size
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Argument for Base: Depandi
Initializing Base
Initializing size in Base: 7
Initializing Derived
Initializing size in Derived: 11

Process finished with exit code 0
```

Aturan Overriding

Di Kotlin, implementasi inheritance diatur dengan aturan (*rule*) : jika sebuah kelas mewarisi implementasi method yang sama atau ganda dari kelas induk perantara, maka ia mesti di override dan menyediakan implementasinya. Untuk menunjukan supertype dari kelas yang diwarisi oleh anggota dapat dilakukan dengan menggunakan keyword super<Base>.

Contoh kode progam:

Nama file : InheritanceH.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

open class Rectangle {
    open fun draw() {
        println("From Base Class Rectangle")
    }
}

interface Polygon {
    fun draw() { // interface members are 'open' by default
        println("From interface Polygon")
    }
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
```

```
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}

fun main() {
    val square = Square()
    square.draw()
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
From Base Class Rectangle
From interface Polygon

Process finished with exit code 0
```

Polymorphism

Polymorphisme adalah salah satu konsep OOP. Terdapat 2 jenis polymorphisme di Kotlin yaitu :

1. Static (*compile-time*) Polymorphism
2. Dynamic (*run-time*) Polymorphism

Static polymorphism terjadi ketika kita mendefenisikan beberapa fungsi *overloading* dengan nama yang sama tetapi berbeda karakteristik parameter (jenis type dan jumlah parameter). Ia dipanggil *compile-time polymorphisme* karena kompiler dapat menentukan fungsi yang mana dipanggil saat melakukan kompilasi.

Dynamic polymorphism terjadi ketika mendefenisikan beberapa fungsi yang di *override*. Dalam kasus ini fungsi yang dipanggil ditentukan pada saat *runtime*.

Static Polymorphism (Overloading Function)

Contoh kode program fungsi *overloading* :

Nama file : PolymorphismA.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas


fun printNumber(n : Number){
    println("Using printNumber(n : Number)")
    println(n.toString() + "\n")
}

fun printNumber(n : Int){
    println("Using printNumber(n : Int)")
    println(n.toString() + "\n")
```

```
}

fun printNumber(n : Double){
    println("Using printNumber(n : Double)")
    println(n.toString() + "\n")
}

fun printNumber(n:Int, str:String){
    println("Using printNumber(n:Int, str:String)")
    println(str + n.toString() + "\n")
}

fun main(){
    val a: Number = 99
    val b = 1
    val c = 3.1
    printNumber(a)
    printNumber(b)
    printNumber(c)
    printNumber(b, "Overloading function ")
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Using printNumber(n : Number)
99

Using printNumber(n : Int)
1

Using printNumber(n : Double)
3.1

Using printNumber(n:Int, str:String)
Overloading function 1


Process finished with exit code 0
```

Dynamic Polymorphism (Overriding Function)

Contoh kode program fungsi *overriding* :

Nama file : PolymorphismB.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas


open class MyBase {
    // Must use "open" modifier to allow child classes to override it
    open fun think () {
```

```
        println("Hey!! i am thinking ")
    }
}

class MyDerived: MyBase() { // inheritance happens using default constructor
    // Must use "override" modifier to override a base class function
    override fun think() {
        println("I Am from Child")
    }
}
fun main() {
    val myBase = MyBase()
    myBase.think()
    var myDerived = MyDerived()
    myDerived.think()
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Hey!! i am thinking
I Am from Child

Process finished with exit code 0
```

Visibility Modifier

Jenis visibility modifier yaitu:

- *public* : Instansiasi kelas dapat dilakukan dimanapun didalam dan dan luar program. Ini adalah *visibility modifier* default ketika konstruktor dibuat tanpa *visibility modifier*.
- *private* : Instansiasi kelas dapat dilakukan hanya dari dalam setiap kelas dan objek yang sama. Visibility modifier ini sangat berguna jika kita menggunakan konstruktor tambahan.
- *protected* : Pengaturan ini sama dengan visibility modifier private, tetapi instansiasi dapat juga dilakukan dari subclass (konsep *inheritance*).
- *internal* : Instansiasi dapat dilakukan dimanapun didalam modul. Di Kotlin module adalah kumpulan file yang dikompilasi secara bersamaan. Kita dapat menggunakan *visibility modifier* ini jika kita tidak ingin program lain (dari *project* lain) mengakses konstruktor, tapi selain itu kita bisa secara bebas mengakses konstruktor dari kelas dan objek lain didalam program.

Berikut adalah ringkasan singkat dari penjelasan mengenai 4 jenis *visibility modifier* :

Modifier	**Class Member**	**Top-level declaration**

public(default)	Visible everywhere	Visible everywhere
internal	Visible in a module	Visible in a module
protected	Visible in subclass	-
private	Visible in class	Visible in file

Contoh kode program :

Nama file : VisibilityA.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

//Private
private class privateExample {
    private val i = 1
    private fun doSomething() {
        println("Welcome to Psrivate access!")
        println("Private Access : $i")
    }
    fun calldoSomething(){
        doSomething();
    }
}

//Protected
open class C() {
    protected val i = 1
}
class D : C() {
    fun getValue() : Int {
        println("Welcome to Protected access!")
        return i
    }
}

//internal
class internalExample {
    internal val i = 1
    internal fun doSomething() {
        println("Welcome to Internal access!")
        println("Internal Access : $i")
    }
}

//public
class publicExample {
    val i = 1
```

```
    fun doSomething() {
        println("Welcome to Public access!")
        println("Public Access : $i")
    }
}

fun main() {
    val objPrivate = privateExample()
    objPrivate.calldoSomething()
    val objProtected = D()
    println(println("Protected Access : ${objProtected.getValue()}"))
    val objInternal = internalExample()
    objInternal.doSomething()
    val objPublic = publicExample()
    objPublic.doSomething()
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Welcome to Psrivate access!
Private Access : 1
Welcome to Protected access!
Protected Access : 1
kotlin.Unit
Welcome to Internal access!
Internal Access : 1
Welcome to Public access!
Public Access : 1

Process finished with exit code 0
```

Access Modifier

Akses modifier di Kotlin terdiri dari *final*, *open*, *abstract*, dan *override*. Akses modifier di Kotlin berpengaruh terhadap perwarisan (*inheritance*). Secara default seluruh fungsi dan kelas di Kotlin apabila tidak terdapat akses modifier-nya maka secara implisit akses modifier-nya adalah *final*. Sehingga apapun kelas dan method di Kotlin harus memiliki akses modifier *open* untuk menurunkan maupun meng-*override* methodnya. Sangat berbeda dengan Java dimana secara default kelas dan method di Java memiliki akses modifier *open*. Sehingga apapun kelas dapat diturunkan dan methodnya dapat di *override* (ditulis kembali) secara *default*-nya.

Akses modifier abstract di Kotlin sama penggunaannya dengan Java, ia dapat dipakai untuk kelas dan method. Ketika sebuah kelas ditandai sebagai abstrak, maka secara implisit ia juga memiliki akses modifier *open* jadi kita tidak perlu memberikan akses modifier open ketika mendeklarasikan fungsi didalam kelas abstrak tersebut.

Berikut adalah ringkasan singkat dari penjelasan mengenai 4 jenis akses modifier :

Modifier	Corresponding Member	Comments
final(default)	Can't be overriden	Used by default for class member
open	Can be overriden	Should be specified explicitly
abstract	Must be overriden	Can be used only in abstract classes; abstract member can't have an implementation
override	Overrides a member in a superclass or interface	Overriden member is open by default, if not marked final

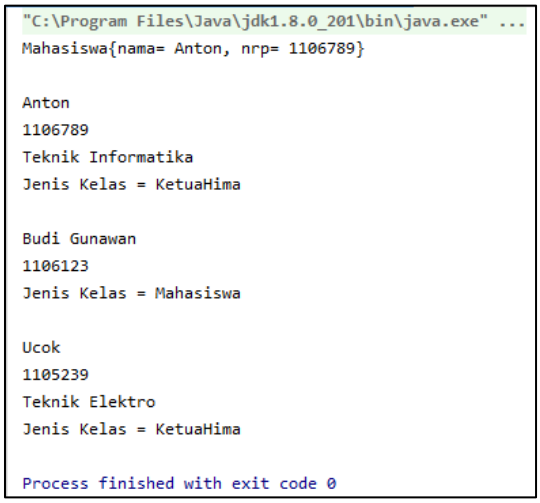
Contoh kode program:

Nama file : AccessModifierA.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

open class Mahasiswa(var nama: String, var nrp: Int){
    open fun info() {
        println(nama +'\n'+ nrp);
    }
    fun toString(): String {
        return "Mahasiswa{nama= $nama, nrp= $nrp}"
    }
}

class KetuaHima(nama: String, nrp: Int, val jurusan: String) : Mahasiswa(nama, nrp) {

    @Override
    override fun info(){
        println(nama +'\n'+ nrp +'\n'+ jurusan);
    }
}

fun main(){
    val budi = Mahasiswa("Budi Gunawan", 1106123)
    val anton = KetuaHima("Anton", 1106789, "Teknik Informatika")
    println(anton.toString());

    println()
    anton.info()
    println("Jenis Kelas = " + anton.javaClass.simpleName)

    println()
    budi.info()
    println("Jenis Kelas = " + budi.javaClass.simpleName)
```

```
    val ucok = KetuaHima("Ucok", 1105239, "Teknik Elektro")
    println()
    ucok.info()
    println("Jenis Kelas = " + ucok.javaClass.simpleName)
}
```

Output:



Kelas Abstrak (Abstract Class)

Kelas Abstrak adalah kelas yang tidak bisa di Instansiasi menjadi sebuah objek, akan tetapi ia bisa diwariskan ke kelas yang lainnya. Sebuah kelas abstrak biasanya berisi anggota abstrak (*abstract member*) yang tidak memiliki implementasi dan harus di *override* didalam *subclass*-nya. Anggota abstrak memiliki akses modifier open secara *default*, jadi kita tidak perlu menulisnya secara eksplisit ketika mau di *override*.

Contoh kode program:

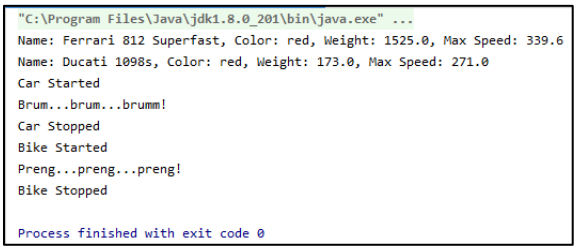
Nama file : AccessModifierA.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

abstract class Vehicle(val name: String,
                       val color: String,
                       val weight: Double) {   // Concrete (Non Abstract) Properties

    // Abstract Property (Must be overridden by Subclasses)
    abstract var maxSpeed: Double

    // Abstract Methods (Must be implemented by Subclasses)
    abstract fun start()
    abstract fun stop()
    abstract fun sound()
```

```
    // Concrete (Non Abstract) Method
    fun displayDetails() {
        println("Name: $name, Color: $color, Weight: $weight, Max Speed: $maxSpeed")
    }
}

class Car(name: String,
          color: String,
          weight: Double,
          override var maxSpeed: Double): Vehicle(name, color, weight) {

    override fun start() {
        // Code to start a Car
        println("Car Started")
    }

    override fun stop() {
        // Code to stop a Car
        println("Car Stopped")
    }

    override fun sound() {
        // Code sound of a Car
        println("Brum...brum...brumm!")
    }
}

class Motorcycle(name: String,
                 color: String,
                 weight: Double,
                 override var maxSpeed: Double): Vehicle(name, color, weight) {

    override fun start() {
        // Code to Start the Motorcycle
        println("Bike Started")
    }

    override fun stop() {
        // Code to Stop the Motorcycle
        println("Bike Stopped")
    }

    override fun sound() {
        // Code sound of a Car
        println("Preng...preng...preng!")
```

```
    }
}

fun main() {
    val car = Car("Ferrari 812 Superfast", "red", 1525.0, 339.60)
    val motorCycle = Motorcycle("Ducati 1098s", "red", 173.0, 271.0)

    car.displayDetails()
    motorCycle.displayDetails()

    car.start()
    car.sound()
    car.stop()
    motorCycle.start()
    motorCycle.sound()
    motorCycle.stop()
}
```

Output:



Interfaces

Sama halnya dengan kelas abstrak, sebuah *interface* tidak dapat di instansiasi karena ia tidak memiliki konstruktor.

Beberapa hal yang harus diperhatikan pada suatu interfaces:

1. Sebuah *interface* memiliki fungsi abstrak dan non-abstrak.
2. Sebuah *interface* hanya dapat memiliki property abstrak (data member), non-abstrak property tidak diizinkan.
3. Sebuah kelas dapat mengimplementasikan lebih dari 1 *interface*.
4. Semua abstrak property dan fungsi anggota abstrak dari sebuah *interface* harus di *overridden* didalam kelas yang mengimplementasinya.

Pada *interfaces* kita tidak memerlukan akses modifier *final*, *open* atau abstrak. Anggota *interface* selalu memiliki akses modifier *open*, kita tidak dapat mendeklarasikannya sebagai *final*. Ia abstrak jika ia tidak memiliki tubuh, tetapi *keyword* juga tidak dibutuhkan.

Implementasi Interface

Contoh kode program *interface*:

Nama file : InterfaceA.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

interface MyInterface{
    var str: String
    fun demo()
    fun func(){
        println("Hello")
    }
}

class MyClass: MyInterface{
    override var str: String = "Rekayasa Perangkat Lunak"
    override fun demo() {
        println("Demo Function")
    }
}

fun main() {
    val obj = MyClass()
    obj.demo()
    obj.func()
    println(obj.str)
}
```

Output :

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Demo Function
Hello
Rekayasa Perangkat Lunak

Process finished with exit code 0
```

Interface Inheritance

Contoh kode program :

Nama file : InterfaceB.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

interface Named {
    val name: String
}
```

```
interface Person : Named {
    val firstName: String
    val lastName: String
    override val name: String get() = "$firstName $lastName"
}

data class EmployeeB(
// implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: String
) : Person

fun main(){
    val pegawai = EmployeeB("Ucok", "Baba", "Programmer")
    println(pegawai.name)
}
```

Multiple Interface

Contoh kode program implementasi suatu kelas yang memiliki multiple interface:

Nama file : InterfaceC.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

interface X {
    fun demoX() {
        println("demoX function")
    }
    fun funcX()
}

interface Y  {
    fun demoY() {
        println("demoY function")
    }
    fun funcY()
}

// This class implements X and Y interfaces
class MyClassA: X, Y {
    override fun funcX() {
        println("Hello")
    }
    override fun funcY() {
        println("Hi")
    }
```

```
}

fun main() {
    val obj = MyClassA()
    obj.demoX()
    obj.demoY()
    obj.funcX()
    obj.funcY()
}
```

Output :

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
demoX function
demoY function
Hello
Hi

Process finished with exit code 0
```

Resolve Problem in Multiple Interface Method

Ketika *multiple interface* memiliki method yang sama maka untuk memecahkan masalah tersebut pada kelas yang mengimplementasikan interface tersebut ditambah *keyword super*. Seperti pada contoh berikut:

Nama file : InterfaceD.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

interface A {
    fun demo() {
        println("From interface A")
    }
}

interface B  {
    fun demo() {
        println("From interface B")
    }
}

// This class implements X and Y interfaces
class MyClassB: A, B {
    override fun demo() {
        super<A>.demo()
        super<B>.demo()
    }
}
```

```
fun main() {
    val obj = MyClassB()
    obj.demo()
}
```

Output :

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
From interface A
From interface B

Process finished with exit code 0
```

Nested dan Inner Class

Nested Class

Ketika sebuah kelas dideklarasikan didalam kelas lain, maka ia disebut dengan nested class. Nested class di Kotlin sama dengan static nested class di Java.

Hal yang perlu diperhatikan dalam nested class :

1. Sebuah nested class tidak dapat mengakses anggota dari kelas luarnya.
2. Untuk mengakses anggota dari nested class, kita gunakan (‘.’) operator dengan outer kelasnya.

Berikut adalah contoh cara mengakses anggota kelas dari nested class :

Nama file : NestedInnerClass.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class Outer {
    val a = "Outside Nested class."
    class Nested {
        val b = "Inside Nested class."
        fun callMe() = "Function call from inside Nested class."
    }
}

fun main(args: Array<String>) {
    // accessing member of Nested class
    println(Outer.Nested().b)

    // creating object of Nested class
    val nested = Outer.Nested()
    println(nested.callMe())
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Inside Nested class.
Function call from inside Nested class.

Process finished with exit code 0
```

Inner Class

Inner class adalah nested class yang memiliki kata kunci inner dalam pendeklarasiannya. Inner class berbeda dengan nested class, sebuah inner class dapat mengakses data member dari kelas luar.

Berikut adalah contoh inner class :

Nama file : NestedInnerClassA.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

class OuterA {
    val a = "Outside Nested class."
    inner class Inner {
        fun callMe() = a
    }
}

fun main() {
    val outer = OuterA()
    println("Using outer object: ${outer.Inner().callMe()}")
    val inner = OuterA().Inner()
    println("Using inner object: ${inner.callMe()}")
}
```

Output:

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Using outer object: Outside Nested class.
Using inner object: Outside Nested class.

Process finished with exit code 0
```

Anonymous Class

Konsep membuat sebuah objek *interface* menggunakan objek *runtime* dikenal sebagai *anonymous class*. Berikut adalah contoh *anonymous class* :

Nama file : NestedInnerClassB.kt | Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

fun main() {
    // creating an instance of the interface
    var programmer: Human = object : Human {
        override fun think() { // overriding the think method
            print("I am an example of Anonymous Inner Class ")
```

```
        }
    }
    programmer.think()
}

interface Human {
    fun think()
}
```

Output :

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
I am an example of Anonymous Inner Class
Process finished with exit code 0
```

Data Kelas

Kotlin memiliki solusi yang lebih baik untuk menangani data yaitu dengan menggunakan data kelas. Sebuah data kelas sama seperti kelas biasa tapi memiliki tambahan funsionalitas. Di Kotlin kita tidak perlu menulis atau men-*generate* semua data kelas seperti method *Getter* dan *Setter* di Java, Kotlin telah men-*generate* secara otomatis fungsi *Getter* dan *Setter* pada setiap anggota kelas. Selain itu, ia juga menurunkan implementasi fungsi standar seperti equals(), hashCode() dan toString() dari property yang dideklarasikan didalam data kelas kontruktor utama.

Untuk memastikan data kelas konsisten dan memiliki perilaku yang sesuai dari kode yang digenerate, ada beberapa persyaratan yang harus dipenuhi dalam mendeklarasikan data kelas yaitu :

1) Konstruktor utama harus memiliki paling sedikit 1 buah parameter.
2) Parameter didalam konstruktor utama harus ditandai dengan keyword val (*read-only*) atau var (*read-write*).
3) Kelas tidak bisa memiliki akses modifier *open* dan *abstract*, maupun keyword *inner* dan *sealed class*.
4) Kelas mungkin di ekstend dari kelas lain atau mengimplementasikan *interfaces*. Jika menggunakan Kotlin versi 1.1. kebawah, kelas hanya dapat mengimplementasikan *interfaces*.

Contoh kode program :

Nama file : DataClassA.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas


data class Student(val name: String, val age: Int)
```

```
fun main() {
    val boni = Student("Boni", 21)
    val meri = Student("Meri", 20)
    println("Student Name is: ${boni.name}")
    println("Student Age is:  ${boni.age}")
    println("Student Name is: ${meri.name}")
    println("Student Age is:  ${meri.age}")
}
```

Data class hashCode() and equals() methods

Jika dua buah objek memiliki referensi objek yang sama, maka kedua objek tersebut juga memiliki nilai hashCode yang sama. Kita bisa membandingkan nilai kode hash 2 objek apakah sama atau tidak dengan menggunakan fungsi hashCode(). Sedangkan untuk membandingkan nilai pada 2 buah objek dapat dilakukan dengan menggunakan fungsi equals().

Berikut adalah implementasi fungsi hashCode() dan equals():

Nama file : DataClassB.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

fun main() {
    val boni = Student("Boni", 21)
    val newBoni = Student("Boni", 21)
    val meri = Student("Meri", 20)
    if (boni.equals(newBoni))
        println("boni is equal to newBoni.")
    else
        println("boni is not equal to newBoni.")

    if (boni.equals(meri))
        println("boni is equal to meri.")
    else
        println("boni is not equal to meri.")

    println("Hashcode of boni: ${boni.hashCode()}")
    println("Hashcode of newBoni: ${newBoni.hashCode()}")
    println("Hashcode of meri: ${meri.hashCode()}")
}
```

Data class copy() method

Untuk sebuah data kelas, kita dapat membuat sebuah objek salinan(*copy*) dengan beberapa property yang berbeda menggunakan fungsi *copy()*.

Contoh kode program :

Nama file : DataClassC.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

fun main(){
    val meri = Student("Meri", 20)
    val badu = meri.copy("Badu")
    if(meri.equals(badu))
        println("meri is equal to badu.")
    else
        println("meri is not equal to badu.")
    println("Hashcode of meri: ${meri.hashCode()}")
    println("Hashcode of badu: ${badu.hashCode()}")
}
```

Data class toString() method

Fungsi toString() pada data kelas mengembalikan nilai kembalian String yang merupakan representasi dari objek.

Nama file : DataClassD.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

fun main(){
    val meri = Student("Meri", 20)
    println(meri.toString())
}
```

Data Classes and Destructuring Declarations: The componentN() method

Kita dapat mendestruksi data kelas sebuah objek kedalam sejumlah variabel menggunakan *destructing declarations*.

Contoh kode program *destructing declarations* :

Nama file : DataClassE.kt	Package : id.ac.polbeng.depandi.test_kelas

```
package id.ac.polbeng.depandi.test_kelas

fun main() {
    val meri = Student("Meri", 20)
    // Destructuring Declaration
    val(name, age) = meri
    println("Name = $name")
    println("Age = $age")
}
```

Kotlin juga mendukung destruksi data kelas sebuah objek kedalam sejumlah variabel menggunakan fungsi componentN(). Fungsi component1() mengembalikan nilai property pertama pada sebuah objek, fungsi component2() mengembalikan nilai property kedua pada sebuah objek, begitu selanjutnya hingga N property pada sebuah kelas.

Contoh kode program *destructing declarations* menggunakan componentN() method:

Nama file : DataClassF.kt	Package : id.ac.polbeng.dependi.test_kelas

```
package id.ac.polbeng.dependi.test_kelas


fun main() {
    val meri = Student("Meri", 20)
    // Destructuring Declaration with componentN() method
    val name = meri.component1()
    val age = meri.component2()
    println("Name = $name")
    println("Age = $age")
}
```

Tugas Praktikum

- Tuliskan pemahaman anda mengenai contoh koding program praktikum 3 dan screen shoot hasilnya.
- Buatlah laporan resmi praktikum 3.

Daftar Referensi

Ted Hagos, *Learn Android Studio 3 with Kotlin (Efficient Android App Development)*, Apress, Manila, 2018

Dmitry Jemerov and Svetlana Isakova, *Kotlin in Action*, Manning Publication Co., Shelter Island, 2017

https://www.callicoder.com/kotlin-abstract-classes/

https://beginnersbook.com/2019/03/kotlin-data-class/

https://www.programiz.com/kotlin-programming/data-class

https://www.tutorialspoint.com/kotlin

https://www.ict.social/kotlin/basics

https://kotlinlang.org/docs/reference/

** ☺ Selamat Mengerjakan ☺ ***