

---

# Kotlin Object Oriented Programming

Eko Kurniawan Khannedy

---

# License

- Dokumen ini boleh Anda gunakan atau ubah untuk keperluan non komersial
- Tapi Anda wajib mencantumkan sumber dan pemilik dokumen ini
- Untuk keperluan komersial, silahkan hubungi pemilik dokumen ini

# Eko Kurniawan Khannedy

- Technical architect at one of the biggest ecommerce company in Indonesia
- 10+ years experiences
- [youtube.com/c/ProgrammerZamanNow](https://youtube.com/c/ProgrammerZamanNow)



---

# Apa itu Object Oriented Programming?



# Sebelum Belajar

- Kotlin Dasar
- <https://www.udemy.com/course/pemrograman-kotlin-pemula-sampai-mahir/?referralCode=98BE2E779EB8A0BEC230>

---

# Apa itu Object Oriented Programming?

- Object Oriented Programming adalah sudut pandang bahasa pemrograman yang berkonsep “objek”
- Ada banyak sudut pandang bahasa pemrograman, namun OOP adalah yang sangat populer saat ini.
- Ada beberapa istilah yang perlu dimengerti dalam OOP, yaitu: Object dan Class

---

# Apa itu Object?

- Object adalah data yang berisi properties (fields atau attributes) dan functions (methods atau behavior)
- Semua data di Kotlin adalah object, dari mulai Number, Boolean, Character, String dan yang lainnya

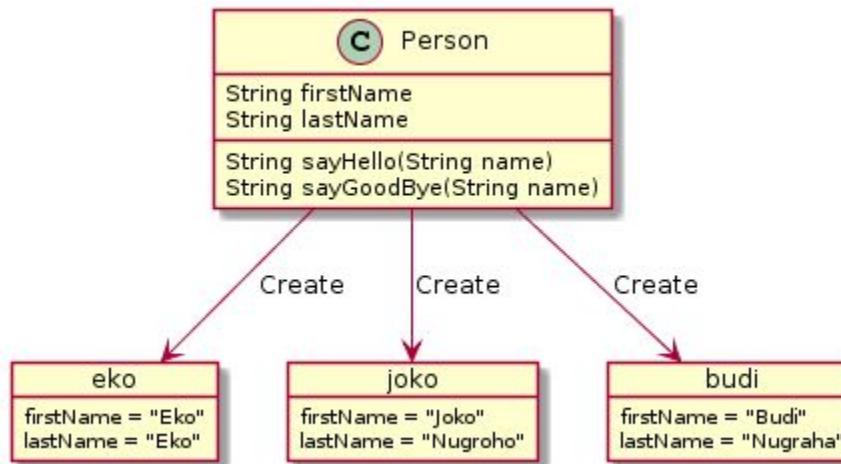
---

# Apa itu Class?

- Class adalah blueprint, prototype atau cetakan untuk membuat Object
- Class berisikan deklarasi semua properties dan functions yang dimiliki oleh Object
- Setiap Object selalu dibuat dari Class
- Dan sebuah Class bisa membuat Object tanpa batas

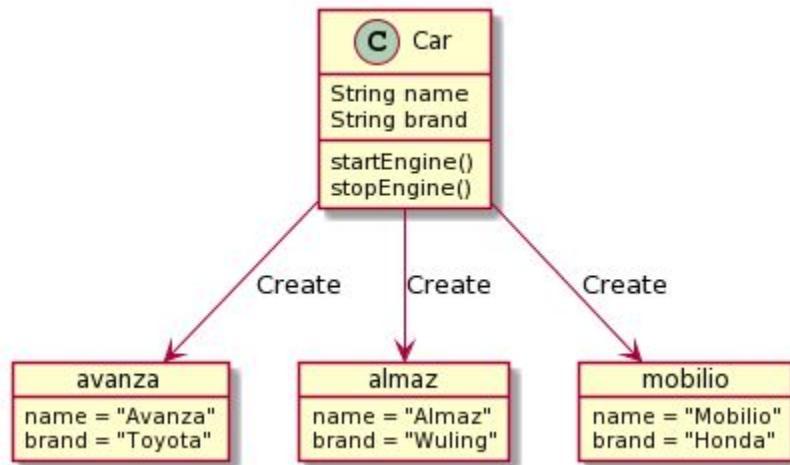
---

# Class dan Object : Person



---

# Class dan Object : Car



---

# Class

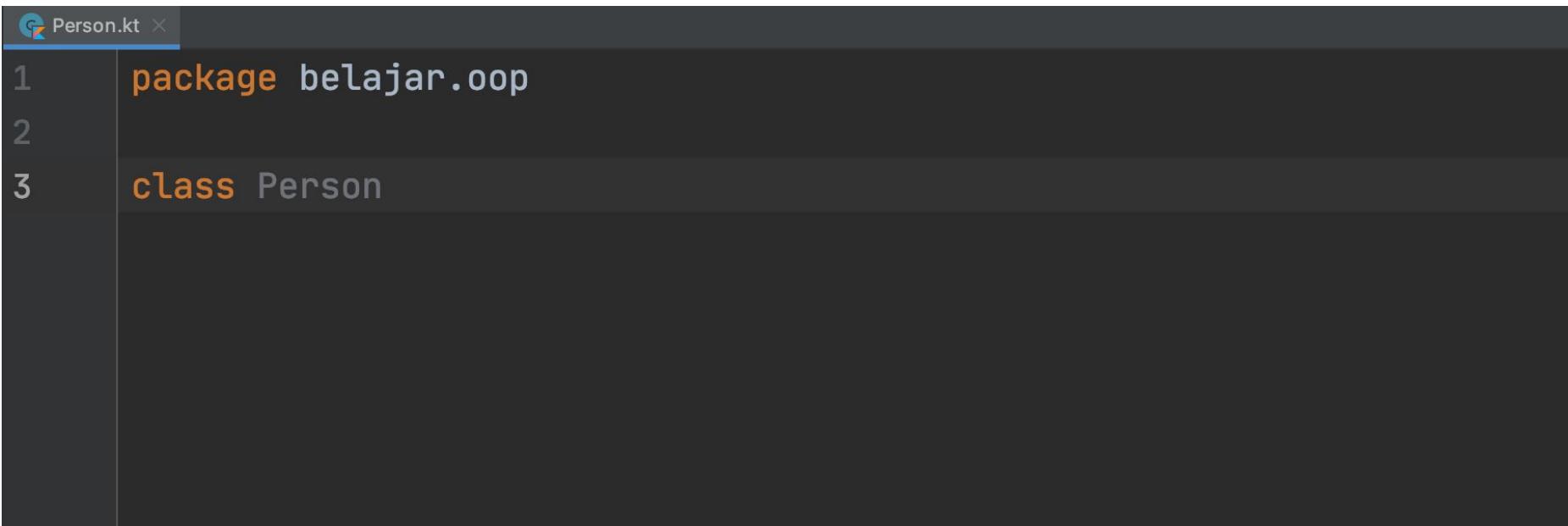
---

# Membuat Class

- Untuk membuat class di Kotlin, kita bisa menggunakan kata kunci class
- Membuat class di Kotlin tidak ada aturan harus sama dengan nama file seperti di Java
- Namun agar kodennya rapih dan mudah untuk dimengerti, disarankan untuk membuat nama class dan nama file sama. Misal class Person di file Person.kt

---

# Kode : Class



A screenshot of a code editor showing a file named `Person.kt`. The code contains the following content:

```
1 package belajar.oop
2
3 class Person
```

The code editor interface includes a tab bar with the file name, a status bar at the bottom, and a vertical scrollbar on the right side.

---

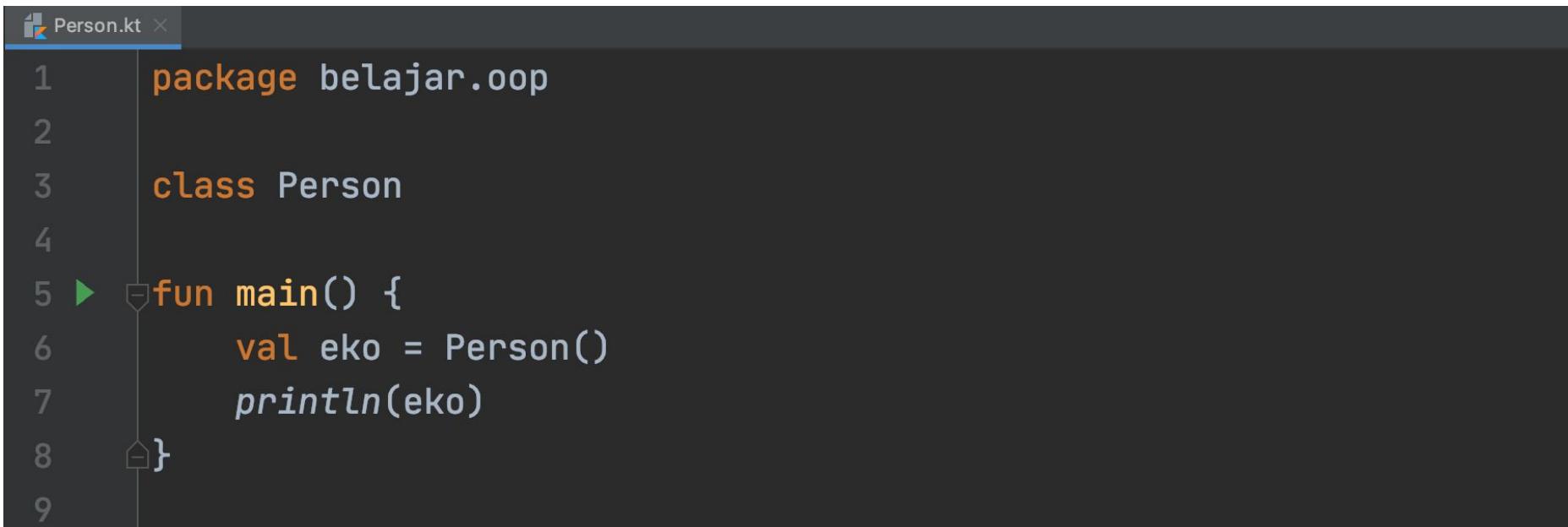
# Object

---

# Membuat Object

- Membuat Object di Kotlin sangat mudah, mirip seperti memanggil function, dengan menggunakan nama class
- Di Kotlin, tidak butuh kata kunci khusus untuk membuat Object, misal jika di Java kita butuh menggunakan kata kunci new untuk membuat Object

# Kode : Object



The screenshot shows a code editor window with a dark theme. The title bar says "Person.kt". The code is as follows:

```
1 package belajar.oop
2
3 class Person
4
5 ▶ fun main() {
6     val eko = Person()
7     println(eko)
8 }
9
```

A green play button icon is next to the "fun main()" line, indicating it is the entry point of the program.

---

# Properties

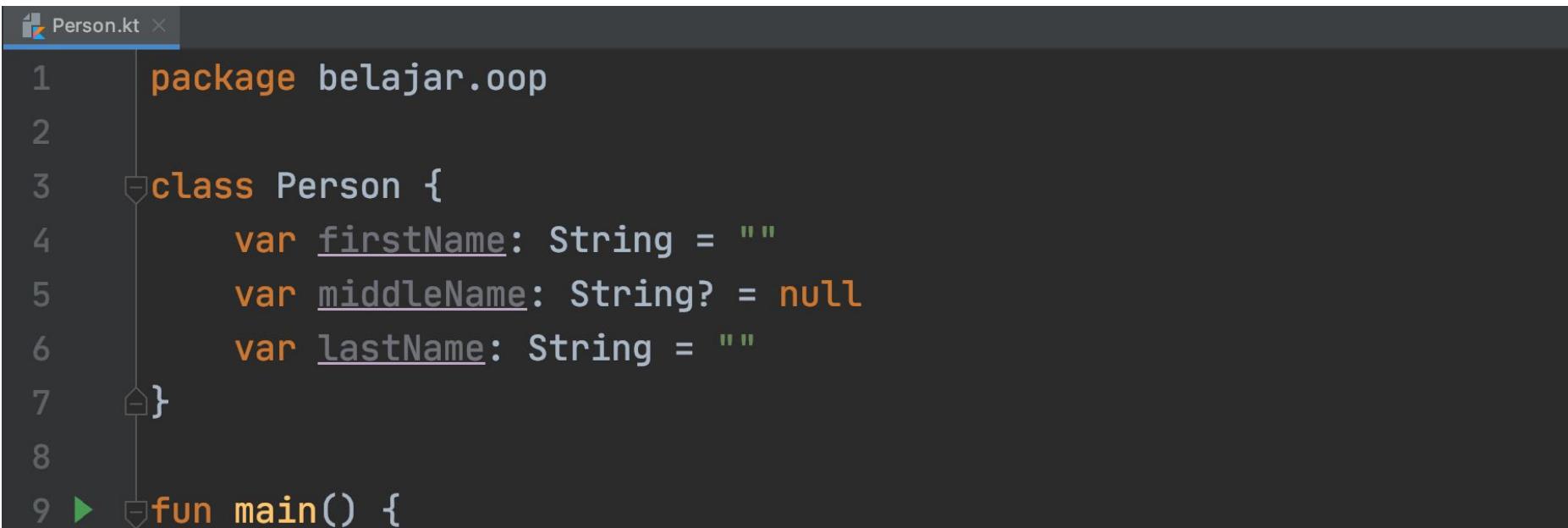
---

# Properties

- Properties / Fields / Attributes adalah data yang bisa kita sisipkan di dalam Object
- Namun sebelum kita bisa memasukkan data di Properties, kita harus mendeklarasikan data apa aja yang dimiliki object tersebut di dalam deklarasi class-nya
- Membuat Properties sama seperti membuat variable, bisa mutable atau immutable

---

# Kode : Properties



```
Person.kt
1 package belajar.oop
2
3 class Person {
4     var firstName: String = ""
5     var middleName: String? = null
6     var lastName: String = ""
7 }
8
9 ► fun main() {
```

The image shows a screenshot of a code editor with a dark theme. The file tab at the top is labeled "Person.kt". The code itself is written in Kotlin. It defines a class named "Person" with three properties: "firstName", "middleName", and "lastName". The "firstName" and "lastName" properties are of type String and have an initial value of an empty string. The "middleName" property is of type String? and has an initial value of null. The code ends with a main function call at the bottom.

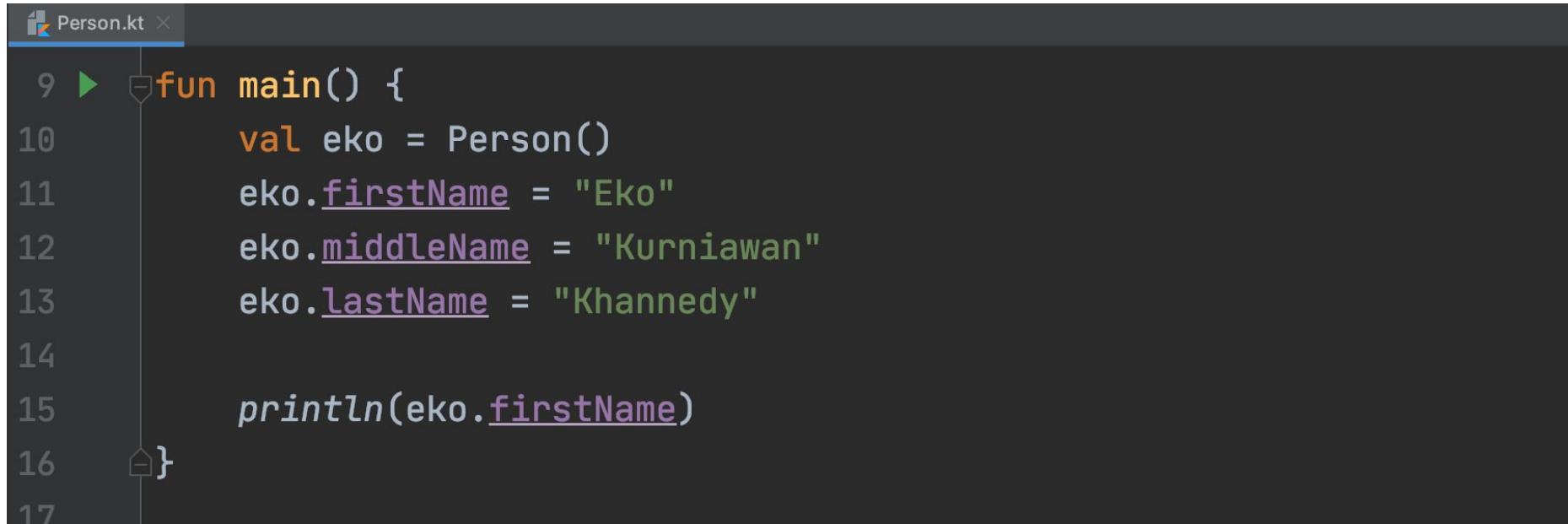
---

# Manipulasi Properties

- Properties yang ada di object, bisa kita manipulasi. Tergantung mutable atau immutable.
- Jika mutable, berarti kita bisa mengubah data properties nya, namun jika immutable, kita hanya bisa mengambil data properties nya saja
- Untuk memanipulasi data properties, sama seperti cara pada variable
- Untuk mengakses properties, kita butuh kata kunci . (titik) setelah nama object dan diikuti nama properties nya

---

# Kode : Manipulasi Properties



The screenshot shows a code editor window with a dark theme. The title bar says "Person.kt". The code is as follows:

```
9 >  fun main() {
10    val eko = Person()
11    eko.firstName = "Eko"
12    eko.middleName = "Kurniawan"
13    eko.lastName = "Khannedy"
14
15    println(eko.firstName)
16 }
17
```

Line 9 has a green play icon and a grey outline. Line 15 has a grey outline.

---

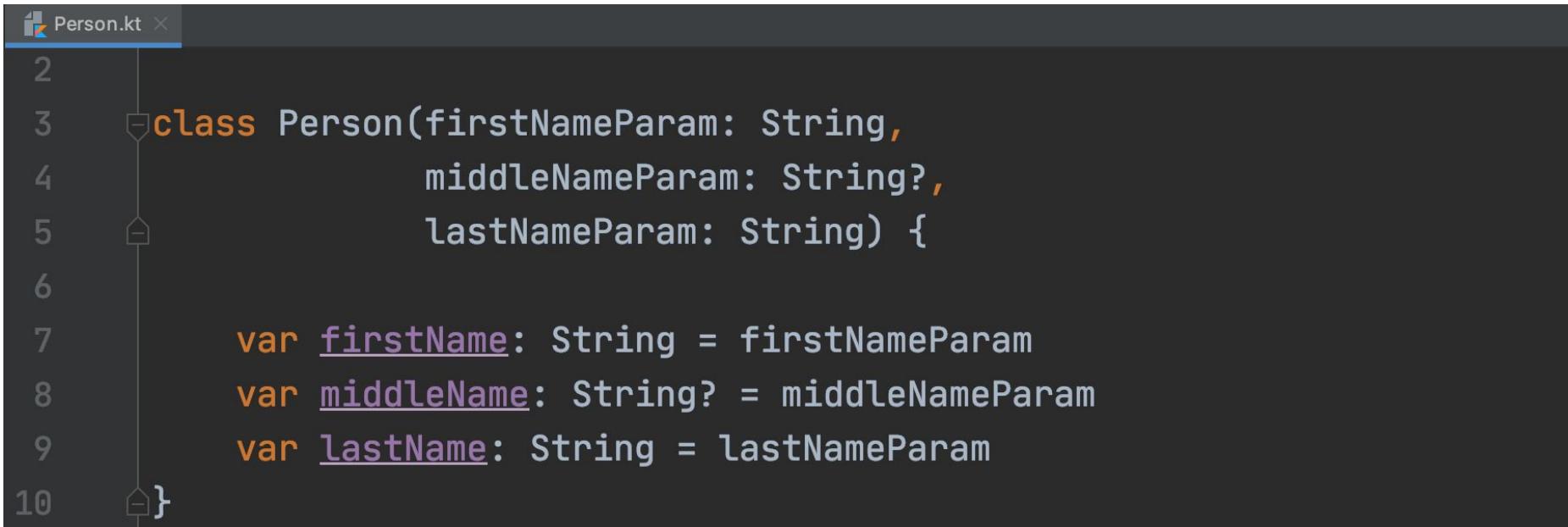
# Constructor

---

# Constructors

- Saat kita membuat Object, maka kita seperti memanggil sebuah function
- Di dalam class Kotlin, kita bisa membuat Constructors, Constructors mirip seperti function yang akan dipanggil saat pertama kali Object dibuat.
- Mirip seperti di Function, kita bisa memberi parameter pada Constructors

# Kode : Constructors



```
Person.kt
2
3 class Person(firstNameParam: String,
4               middleNameParam: String?,
5               lastNameParam: String) {
6
7     var firstName: String = firstNameParam
8     var middleName: String? = middleNameParam
9     var lastName: String = lastNameParam
10}
```

The image shows a screenshot of a code editor with a dark theme. A file named "Person.kt" is open. The code defines a class "Person" with three parameters: "firstNameParam", "middleNameParam", and "lastNameParam". Inside the class, three corresponding variables are declared using the variableName syntax, which is highlighted in purple. The code editor has a vertical line on the left side with icons indicating code navigation or structure.

---

# Kode : Menggunakan Constructors



The screenshot shows a code editor window with a dark theme. The file is named "Person.kt". The code defines a main function that creates a Person object with three parameters: firstNameParam, middleNameParam, and lastNameParam, all set to specific strings. The code then prints the value of the firstName parameter.

```
12 fun main() {
13     val eko = Person(
14         firstNameParam: "Eko",
15         middleNameParam: "Kurniawan",
16         lastNameParam: "Khannedy"
17     )
18
19     println(eko.firstName)
```

---

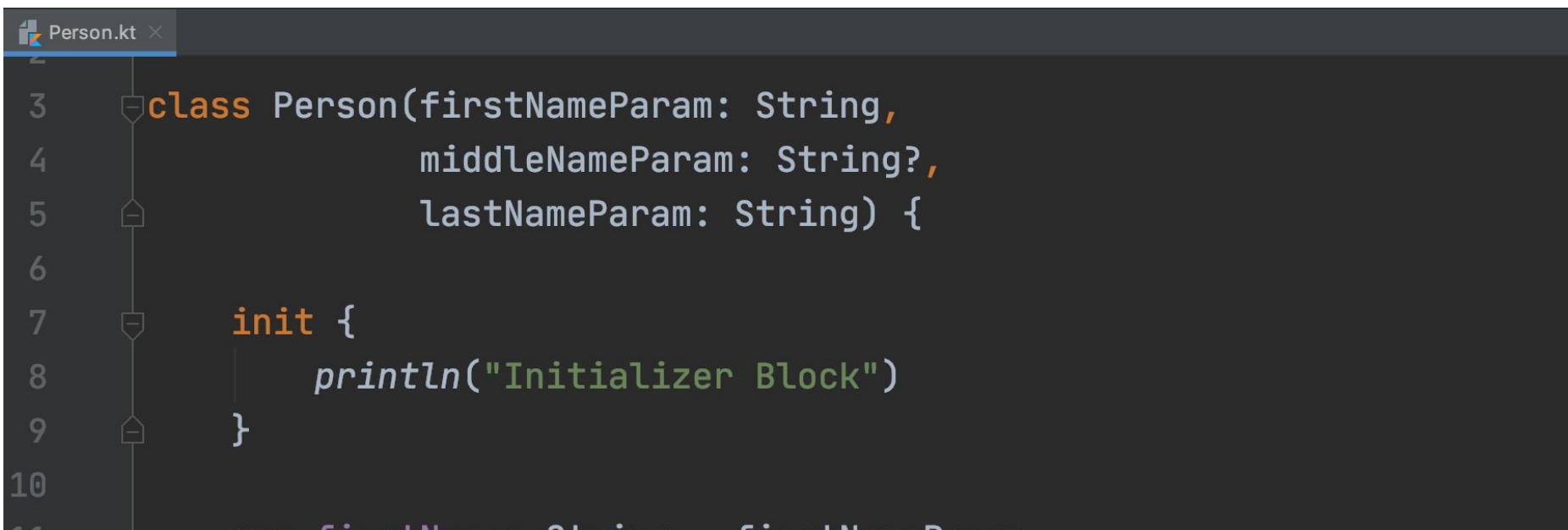
# Initializer Block

---

# Initializer Block

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

# Kode : Initializer Block



```
Person.kt
1
2
3     class Person(firstNameParam: String,
4                     middleNameParam: String?,
5                     lastNameParam: String) {
6
7         init {
8             println("Initializer Block")
9         }
10    }
```

---

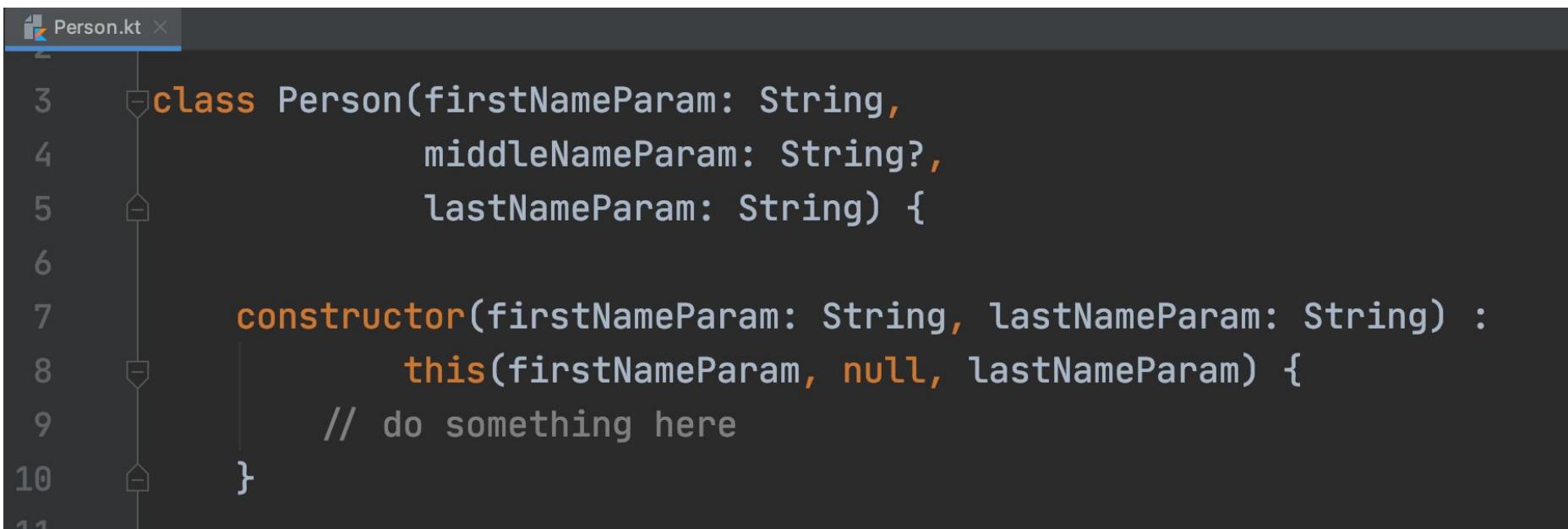
# Secondary Constructor

---

## 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

# Kode : Secondary Constructor



The screenshot shows a code editor window with a dark theme. The file is named "Person.kt". The code defines a class "Person" with three primary constructors:

```
1 class Person(firstNameParam: String,  
2                 middleNameParam: String?,  
3                 lastNameParam: String) {  
4  
5     constructor(firstNameParam: String, lastNameParam: String) :  
6         this(firstNameParam, null, lastNameParam) {  
7             // do something here  
8         }  
9     }  
10 }
```

The first constructor takes three parameters: "firstNameParam" (String), "middleNameParam" (String?), and "lastNameParam" (String). The second constructor takes "firstNameParam" (String) and "lastNameParam" (String), and calls the first constructor with null for "middleNameParam". The third constructor has an empty body and is preceded by a comment "do something here". Line numbers 1 through 11 are visible on the left side of the code.

---

# Kode : Menggunakan Secondary Constructor



The screenshot shows a code editor window with a dark theme. At the top, there's a tab labeled "Person.kt". The code itself is as follows:

```
17 > fun main() {
18     val eko = Person("Eko", "Kurniawan", "Khannedy")
19     val joko = Person("Joko", "Nugroho")
20
21     println(eko.firstName)
22     println(joko.lastName)
23 }
```

The code defines a `Person` class with three parameters in its constructor. It then creates two `Person` objects, `eko` and `joko`, and prints their `firstName` and `lastName` respectively.

---

## Kode : Tanpa Primary Constructor

```
class Person {  
  
    constructor(firstNameParam: String, lastNameParam: String) {  
        firstName = firstNameParam  
        lastName = lastNameParam  
    }  
  
    constructor(firstNameParam: String, middleNameParam: String, lastNameParam: String) {  
        firstName = firstNameParam  
        middleName = middleNameParam  
        lastName = lastNameParam  
    }  
}
```

---

# Properties di Constructor

---

# Properties di Constructor

- Kotlin mendukung deklarasi properties langsung di primary constructor
- Ini sangat berguna untuk mempersingkat saat kita ingin membuat properties, dan mengisi datanya lewat constructor

---

## Kode : Properties di Constructor

```
2
3  class Person(val firstName: String,
4      val middleName: String? = null,
5      val lastName: String) {
6
7
8 ▶ fun main() {
9     val eko = Person("Eko", "Kurniawan", "Khannedy")
10    val joko = Person(firstName = "Joko", lastName = "Nugroho")
11
```

---

# Function

---

# Function

- Selain Parameters / Fields / Attributes, di dalam Class, kita juga bisa mendeklarasikan Function
- Function yang kita deklarasikan di dalam Class, secara otomatis menjadi behaviour si object yang dibuat dari class tersebut

# Kode Function



The image shows a screenshot of a code editor with a dark theme. The file tab at the top left is labeled "Student.kt". The code itself is as follows:

```
1 package belajar.oop
2
3 class Student(val name: String) {
4
5     fun sayHello(yourName: String): Unit {
6         println("Hello $yourName, my name is $name")
7     }
8
9 }
```

The code defines a package named "belajar.oop" and a class named "Student" that takes a "name" parameter. It contains a single method "sayHello" that prints a greeting message using string interpolation.

---

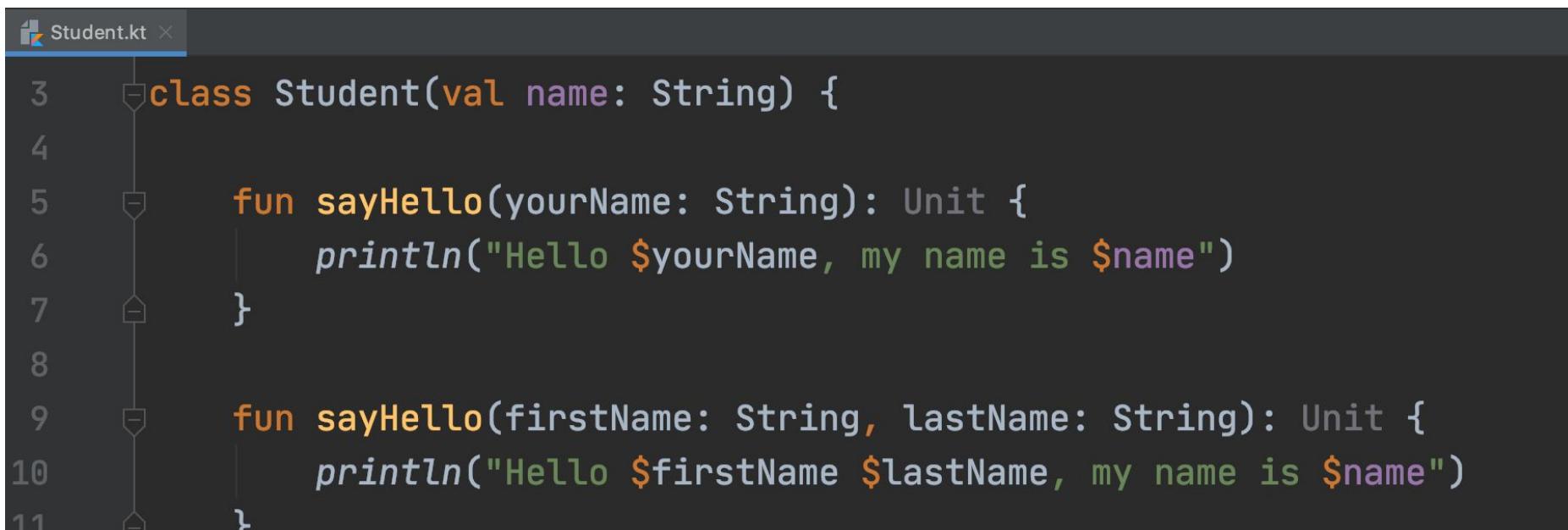
# Function Overloading

---

# 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

# Kode : Function Overloading



The image shows a screenshot of a code editor with a dark theme. The file is named "Student.kt". The code defines a class "Student" with two functions: "sayHello" and "sayHello". The first "sayHello" function takes a single parameter "yourName" and prints "Hello \$yourName, my name is \$name". The second "sayHello" function takes two parameters, "firstName" and "lastName", and prints "Hello \$firstName \$lastName, my name is \$name". The code is color-coded: class and fun keywords are orange, variable names are purple, and strings are green.

```
1 Student.kt
2
3 class Student(val name: String) {
4
5     fun sayHello(yourName: String): Unit {
6         println("Hello $yourName, my name is $name")
7     }
8
9     fun sayHello(firstName: String, lastName: String): Unit {
10        println("Hello $firstName $lastName, my name is $name")
11    }
12 }
```

---

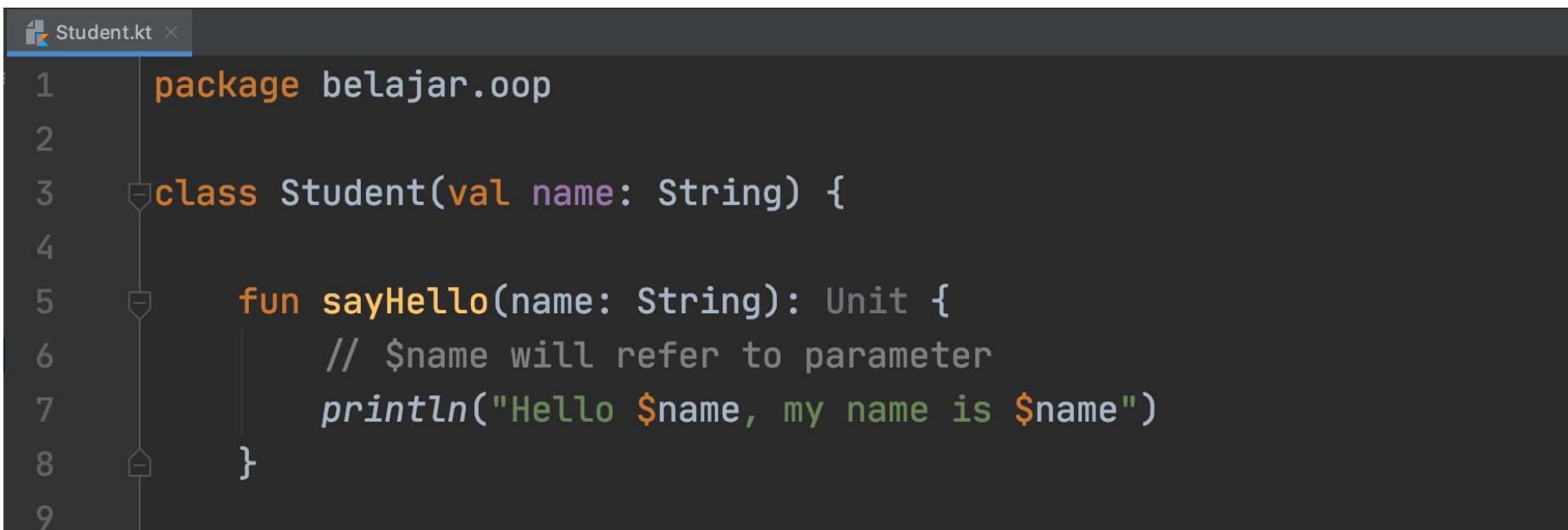
# This Keyword

---

# This Keyword

- this adalah keyword yang bisa digunakan untuk mereferensikan object saat ini
- this hanya bisa digunakan di dalam class itu sendiri
- Biasanya, this digunakan untuk mengakses properties yang tertutup oleh parameter dengan nama yang sama

# Kode : Tanpa This



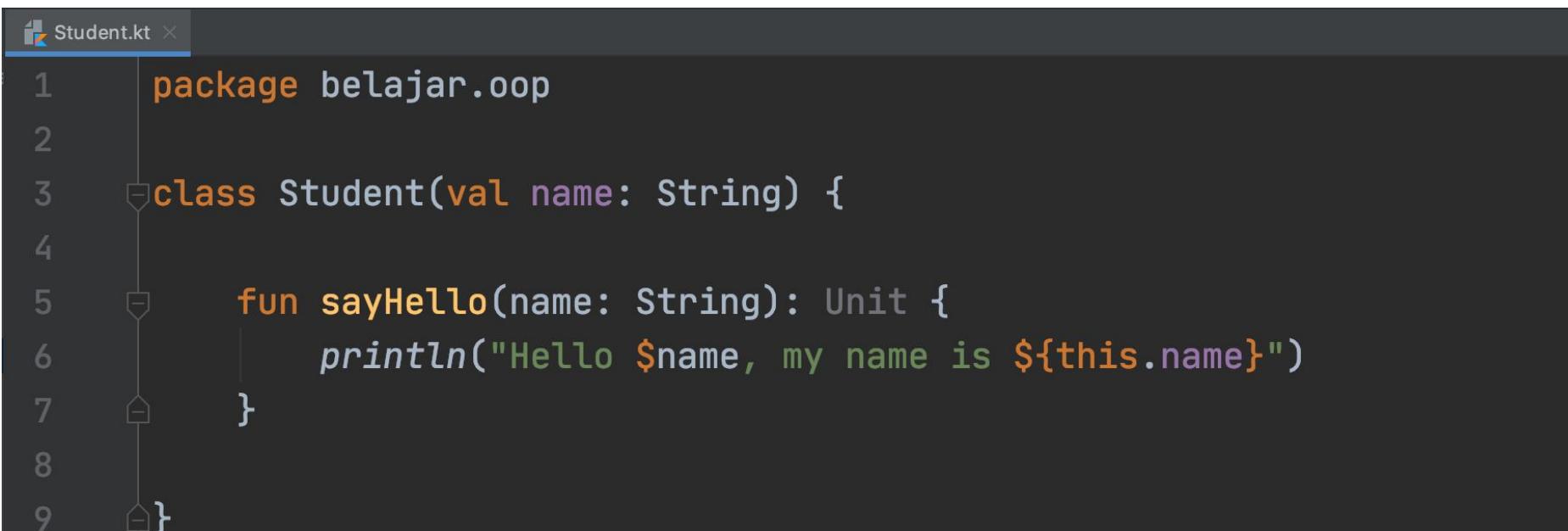
The screenshot shows a code editor window with a dark theme. The title bar says "Student.kt". The code is as follows:

```
1 package belajar.oop
2
3 class Student(val name: String) {
4
5     fun sayHello(name: String): Unit {
6         // $name will refer to parameter
7         println("Hello $name, my name is $name")
8     }
9 }
```

The code defines a class named "Student" with a constructor parameter "name". It contains a single method "sayHello" that prints a greeting using the parameter "name". The code is numbered from 1 to 9 on the left.

---

# Kode : Dengan This



The image shows a screenshot of a code editor with a dark theme. The file tab at the top is labeled "Student.kt". The code itself is as follows:

```
1 package belajar.oop
2
3 class Student(val name: String) {
4
5     fun sayHello(name: String): Unit {
6         println("Hello $name, my name is ${this.name}")
7     }
8
9 }
```

The code defines a class named "Student" with a constructor parameter "name". It contains a single method "sayHello" that prints a greeting message using the "this.name" reference.

---

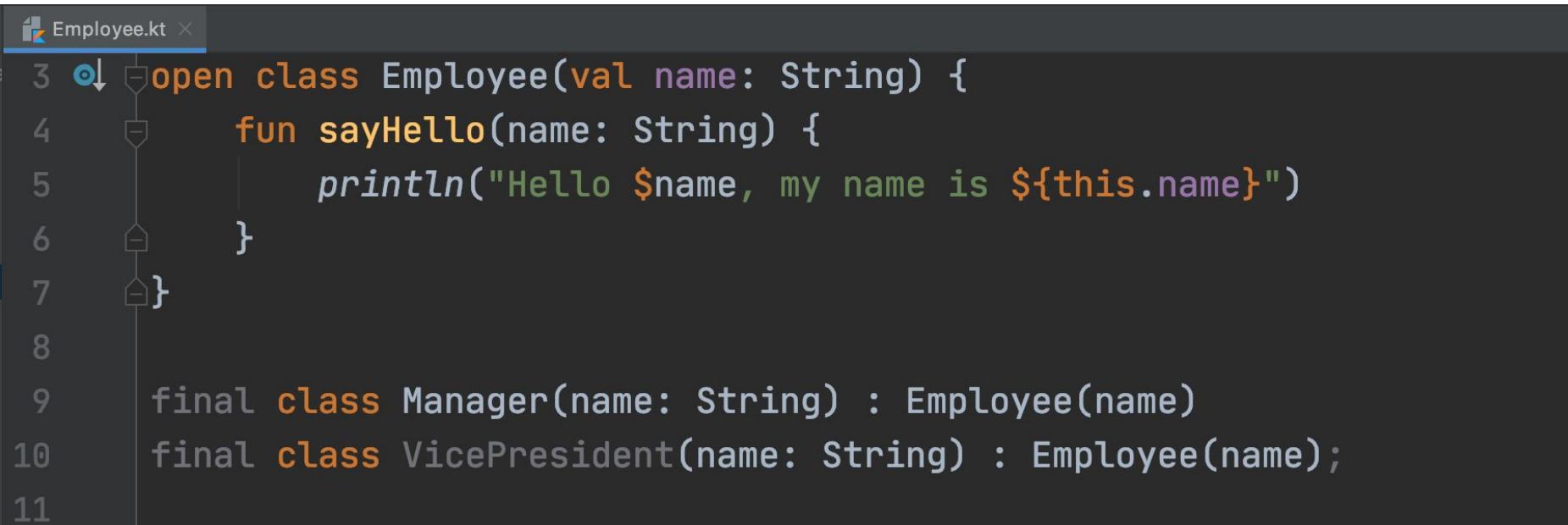
# Inheritance



# Inheritance

- Inheritance atau pewarisan adalah kemampuan untuk menurunkan sebuah class ke class lain
- Dalam artian, kita bisa membuat class Parent dan class Child
- Di Kotlin, tiap class Child, hanya bisa punya satu class Parent, namun satu class Parent bisa punya banyak class Child
- Secara standar, di class yang dibuat di Kotlin adalah final (tidak bisa diwariskan), agar bisa diwariskan, kita harus menggunakan kata kunci open

# Kode : Inheritance

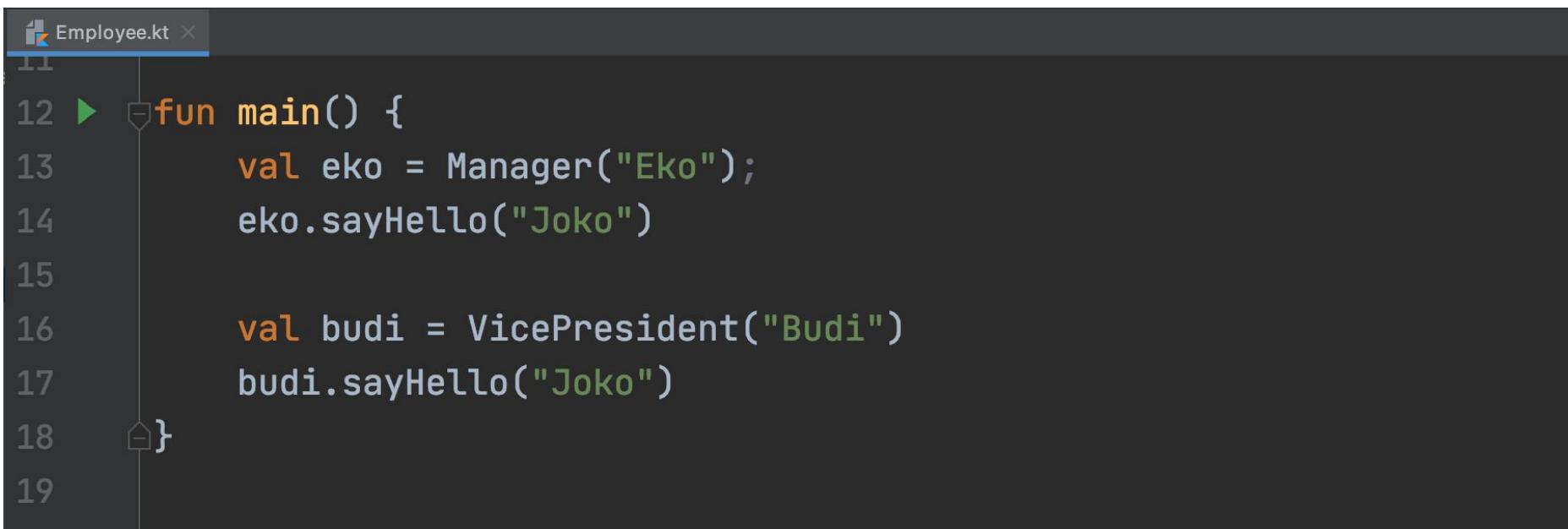


The screenshot shows a code editor window for a file named "Employee.kt". The code defines an open class "Employee" with a constructor parameter "name" and a method "sayHello". It also defines two final subclasses, "Manager" and "VicePresident", both inheriting from "Employee". The code is numbered from 1 to 11 on the left.

```
Employee.kt
1
2
3 3 o↓ open class Employee(val name: String) {
4      fun sayHello(name: String) {
5          println("Hello $name, my name is ${this.name}")
6      }
7  }
8
9  final class Manager(name: String) : Employee(name)
10 final class VicePresident(name: String) : Employee(name);
11
```

---

# Kode : Mengakses Behavior Parent



The screenshot shows a code editor window with a dark theme. The title bar says "Employee.kt". The code is as follows:

```
11  
12 ► fun main() {  
13     val eko = Manager("Eko");  
14     eko.sayHello("Joko")  
15  
16     val budi = VicePresident("Budi")  
17     budi.sayHello("Joko")  
18 }  
19
```

The code defines a main function that creates instances of Manager and VicePresident classes and calls their sayHello method.

---

# Function Overriding

---

# Function Overriding

- Function Overriding adalah kemampuan membuat ulang function yang sudah ada di class Parent
- Secara standar, function di class adalah final, tidak bisa dibuat ulang di class Child
- Agar function bisa dibuat ulang di class Child, kita harus menggunakan kata kunci open

# Kode : Function Overriding

```
Employee.kt
```

```
4  open fun sayHello(name: String) {  
5      println("Hello $name, my name is ${this.name}")  
6  }  
7  }  
9  final class Manager(name: String) : Employee(name) {  
10     override fun sayHello(name: String) {  
11         println("Hello $name, my name is manager ${this.name}")  
12    }  
}
```

---

## Final Override Function

- Saat kita membuat ulang function di class Child, secara standar, function tersebut bersifat open, yang artinya bisa dibuat ulang di class Child dibawahnya lagi
- Jika ingin membuat override function tidak bisa dibuat ulang oleh class Child dibawahnya lagi, kita harus menggunakan kata kunci final

# Kode : Final Override Function

The screenshot shows a code editor with two files open. The top file is `Employee.kt`, which contains the following code:

```
1 package com.example  
2  
3 class Employee(name: String) {  
4     open fun sayHello(name: String) {  
5         println("Hello $name, my name is ${this.name}")  
6     }  
7 }  
8  
9 open class Manager(name: String) : Employee(name) {  
10    final override fun sayHello(name: String) {  
11        println("Hello $name, my name is manager ${this.name}")  
12    }  
13 }
```

The code uses Java-style annotations (`open`, `final`) and Kotlin's `override` keyword. The `Employee` class has an `open` function `sayHello`. The `Manager` class extends `Employee` and overrides the `sayHello` function as `final`. The IDE interface includes a toolbar at the top and a status bar at the bottom.

---

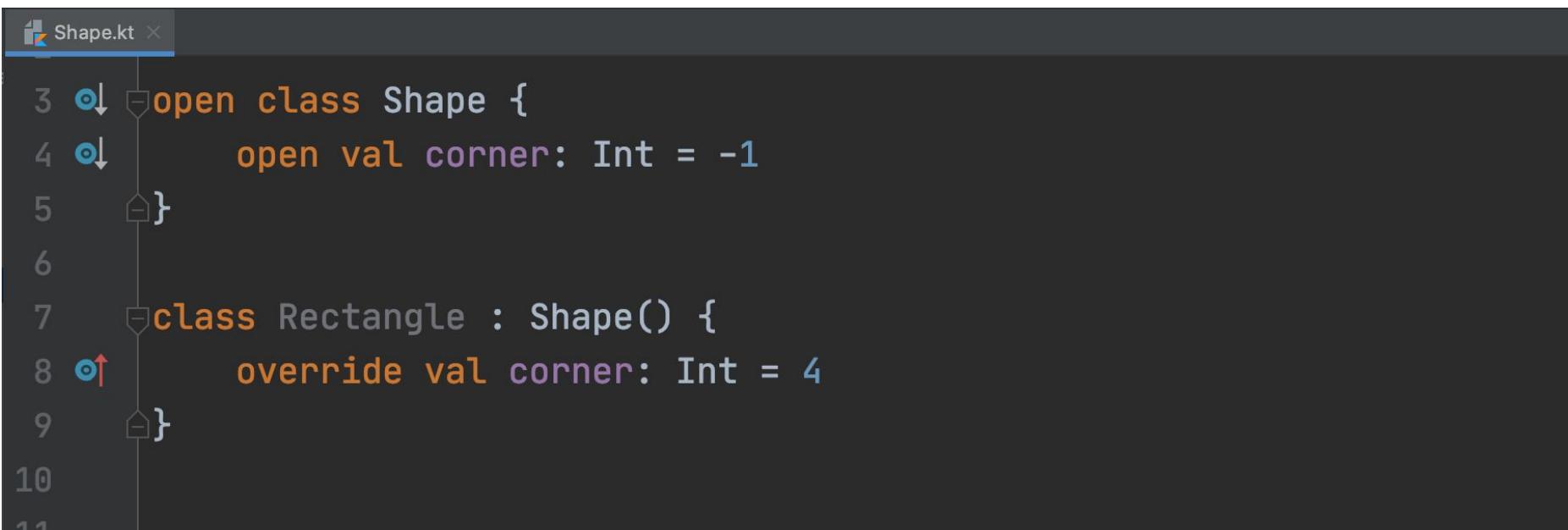
# Properties Overriding

---

# Properties Overriding

- Selain membuat ulang function di class Child, di Kotlin juga kita bisa membuat ulang properties
- Secara standar, properties di class bersifat final, tidak bisa dibuat ulang di class Child nya, agar bisa dibuat ulang, kita harus menggunakan kata kunci open

# Kode : Properties Overriding



The screenshot shows a code editor window with a dark theme. The file is named "Shape.kt". The code defines an open class "Shape" with an open val "corner" set to -1. It then defines a class "Rectangle" that extends "Shape" and overrides the "corner" property to 4.

```
1 Shape.kt
2
3 open class Shape {
4     open val corner: Int = -1
5 }
6
7 class Rectangle : Shape() {
8     override val corner: Int = 4
9 }
10
11 }
```

---

# Super Keyword

---

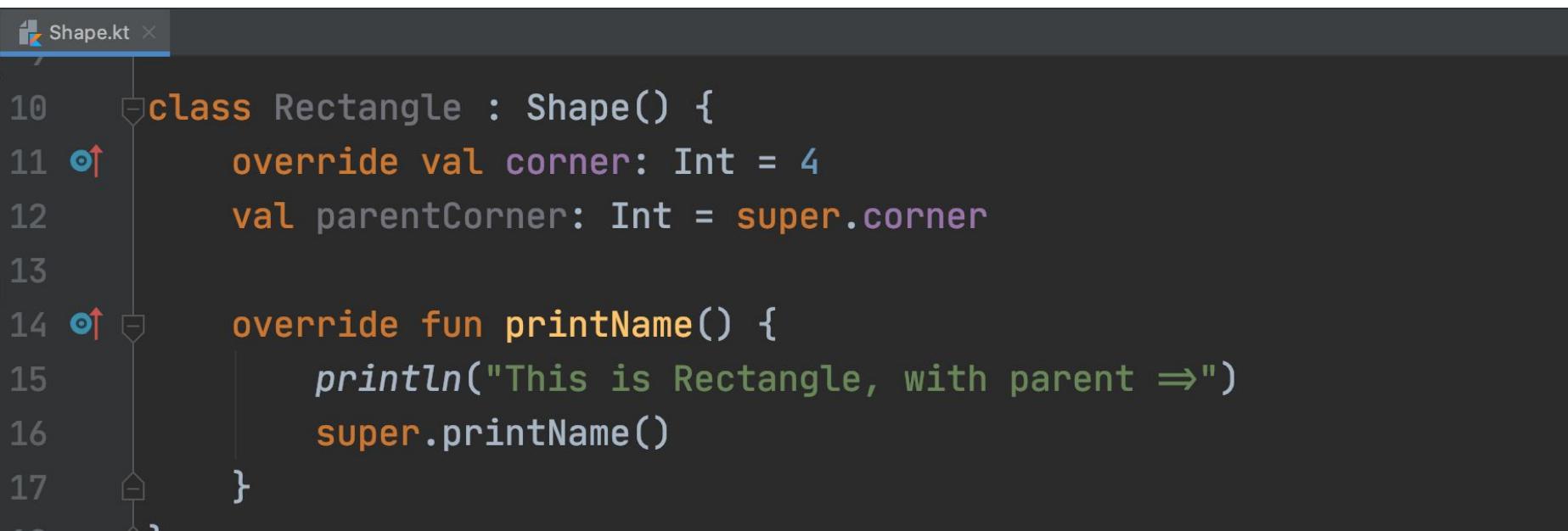
# Super Keyword

- Kadang kita ingin mengakses function atau properties milik class Parent yang sudah dibuat ulang oleh class Child
- Untuk mengakses function atau properties milik class Parent, kita bisa menggunakan kata kunci super

# Kode : Super Properties

```
1 @file:OptIn(ExperimentalContracts::class)
2 konst Shape: Contract {
3     open class Shape {
4         open val corner: Int = -1
5     }
6
7     class Rectangle : Shape() {
8         override val corner: Int = 4
9         val parentCorner: Int = super.corner
10    }
11}
```

# Kode : Super Function



```
Shape.kt
10 class Rectangle : Shape() {
11     override val corner: Int = 4
12     val parentCorner: Int = super.corner
13
14     override fun printName() {
15         println("This is Rectangle, with parent =>")
16         super.printName()
17     }
18 }
```

---

# Super Constructors

---

# Super Constructor

- Kata kunci super tidak hanya bisa digunakan untuk mengakses function atau properties di class Parent
- Kata kunci super juga bisa digunakan untuk mengakses constructor class Parent
- Mengakses constructor class Parent hanya bisa dilakukan di dalam constructor class Child

# Kode : Parent Constructor

```
ape.kt ×

package belajar.oop

open class Shape(val name: String, val shape: Int, val color: String) {
    constructor(name: String, shape: Int) : this(name, shape, "")
    constructor(name: String) : this(name, -1)
}

class Rectangle : Shape {
```

# Kode : Super Constructor

```
Shape.kt
1
2
3 open class Shape(val name: String, val shape: Int, val color: String)
4     constructor(name: String, shape: Int) : this(name, shape, "")
5     constructor(name: String) : this(name, -1)
6 }
7
8 class Rectangle : Shape {
9     constructor() : super("Rectangle", 4)
10    constructor(color: String) : super("Rectangle", 4, color)
11 }
```

---

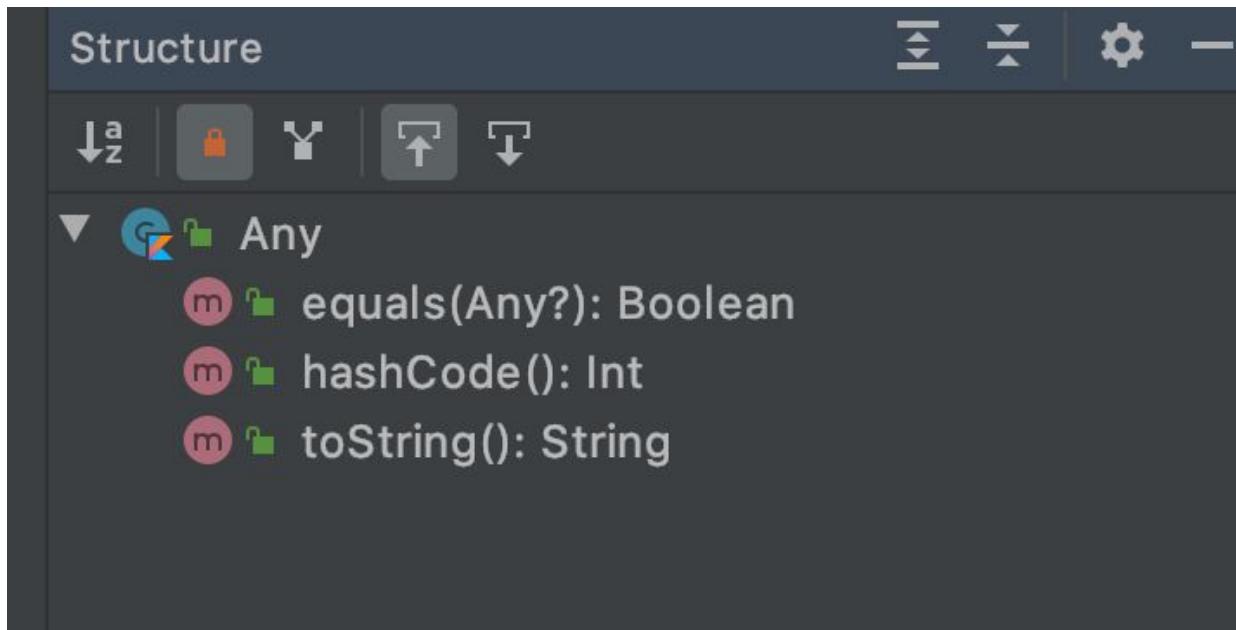
# Any Class

---

# Any Class

- Di Kotlin, semua class yang kita buat tanpa class Parent, sebenarnya secara otomatis dia akan menjadi class child dari class Any
- Any adalah superclass untuk semua class yang kita buat di Kotlin

# Struktur Any Class



# Kode : Any Class



The screenshot shows a Java IDE interface with two tabs at the top: "GeneralFunction.kt" and "Any.kt". The "Any.kt" tab is active, indicated by a blue background. The code editor displays the following Kotlin code:

```
1 package belajar.oop
2
3 class Laptop(val brand: String) // => class Laptop : Any()
4 class Computer(val brand: String) // => class Computer : Any()
5
6 ▶ fun main() {
7     val laptop = Laptop("Apple")
8     println(laptop.toString())
9 }
```

The code defines two classes, `Laptop` and `Computer`, both of which inherit from the `Any` class. The `main` function creates an instance of `Laptop` with the brand "Apple" and prints its string representation.

---

# Type Check & Casts

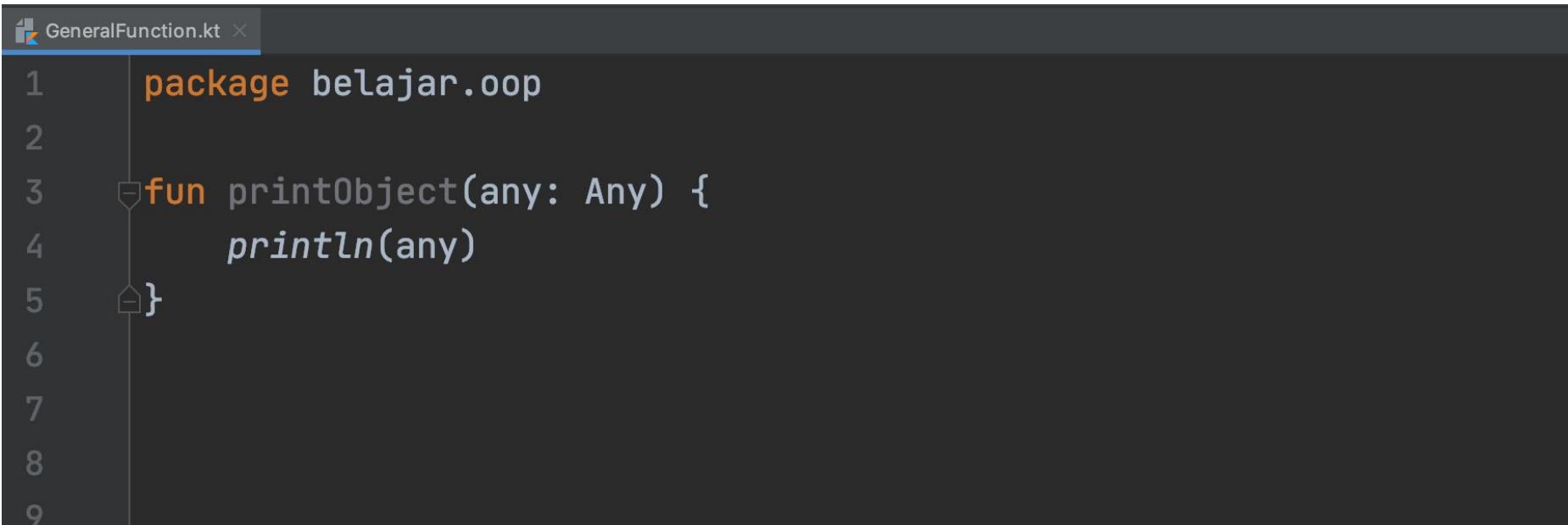
---

# Type Check & Casts

- Dalam Object Oriented Programming, kita akan bertemu dengan banyak sekali tipe data (class) dan pewarisan (inheritance)
- Kadang kita butuh melakukan pengecekan tipe data, atau bahkan melakukan konversi data

---

# Kode : Print Function



The screenshot shows a code editor window with a dark theme. The title bar says "GeneralFunction.kt". The code is as follows:

```
1 package belajar.oop
2
3     fun printObject(any: Any) {
4         println(any)
5     }
6
7
8
9
```

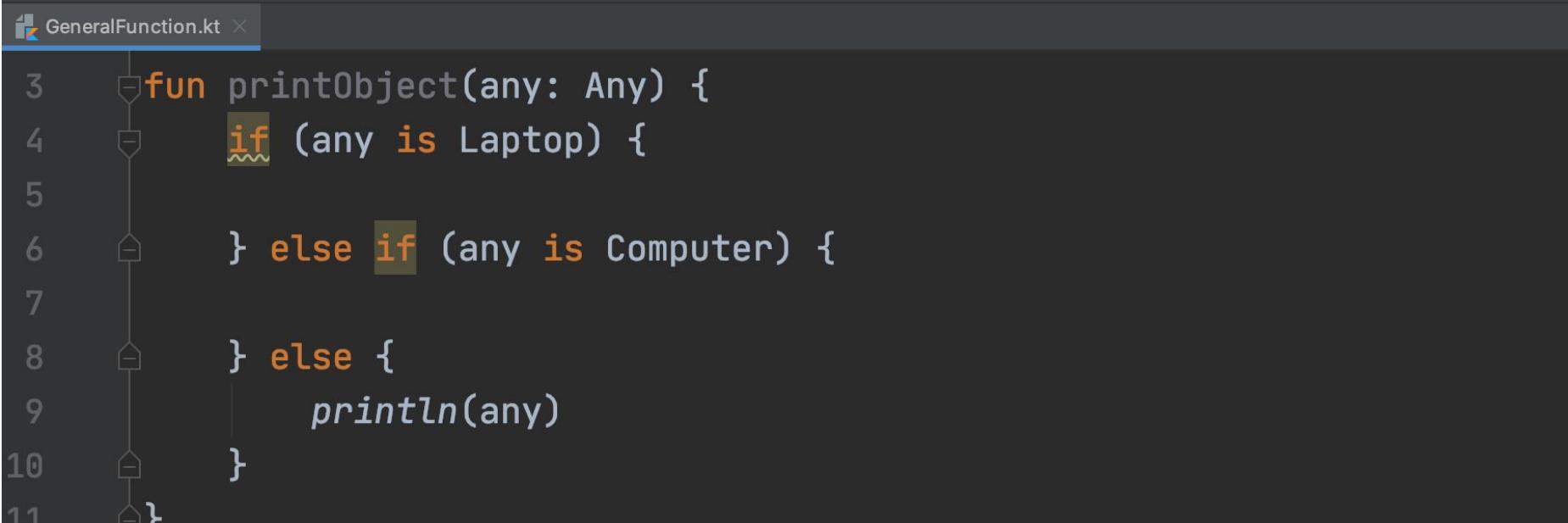
---

## is dan !is Operator

- `is` digunakan untuk melakukan pengecekan apakah sebuah data merupakan tipe data tertentu
- `!is` digunakan untuk melakukan pengecekan apakah sebuah data **bukan** merupakan tipe data tertentu

---

# Kode : Pengecekan



The screenshot shows a code editor window with a dark theme. The file is named "GeneralFunction.kt". The code defines a function "printObject" that checks if an object is a "Laptop" or a "Computer" and prints it accordingly. The code uses Java-style syntax with curly braces and if-else statements.

```
GeneralFunction.kt
1
2
3     fun printObject(any: Any) {
4         if (any is Laptop) {
5
6             } else if (any is Computer) {
7
8             } else {
9                 println(any)
10            }
11        }
```

---

## Smart Casts

- Kotlin memiliki mekanisme konversi data secara otomatis setelah kita melakukan pengecekan menggunakan is
- Setelah kita melakukan pengecekan menggunakan is, maka secara otomatis tipe data yang kita cek akan berubah menjadi tipe data yang kita check

# Kode : Casts

```
GeneralFunction.kt ×  
3     fun printObject(any: Any) {  
4         if (any is Laptop) {  
5             println("Laptop ${any.brand}")  
6         } else if (any is Computer) {  
7             println("Computer ${any.brand}")  
8         } else {  
9             println(any)  
10    }  
11 }
```

---

## Casts di When Expression

- Selain menggunakan If untuk melakukan pengecekan tipe data dan konversi tipe data
- Kita juga bisa menggunakan when expression
- Penggunaan when expression akan lebih sederhana dibanding if expression dalam melakukan pengecekan dan konversi tipe data

# Kode : Casts di When Expression

```
GeneralFunction.kt ×  
3     fun printObject(any: Any) {  
4         when (any) {  
5             is Laptop → println("Laptop ${any.brand}")  
6             is Computer → println("Computer ${any.brand}")  
7             else → println(any)  
8     }  
9 }  
10  
11 }
```

---

# Unsafe Casts

- Kotlin juga mendukung konversi tipe data secara paksa menggunakan kata kunci as
- Namun konversi menggunakan as sangat tidak aman jika ternyata tipe datanya tidak sesuai



# Kode : Unsafe Casts

```
GeneralFunction.kt ×
11  fun printString(any: Any) {
12      val value = any as String
13      println(value)
14  }
15
16 ► fun main() {
17     printString("Eko") // success
18     printString(1) // ClassCastException
19 }
```

---

## Safe Nullable Casts

- Penggunaan as sangat tidak aman, namun kadang bisa menjadikannya aman, dengan menggunakan kata kunci as?
- Dengan menggunakan kata kunci as? secara otomatis jika gagal melakukan konversi, akan diubah menjadi null

---

## Kode : Safe Nullable Casts

```
GeneralFunction.kt ×
11  fun printString(any: Any) {
12      val value: String? = any as? String
13      println(value)
14  }
15
16 ► fun main() {
17     printString("Eko") // success
18     printString(1) // null
19 }
```

---

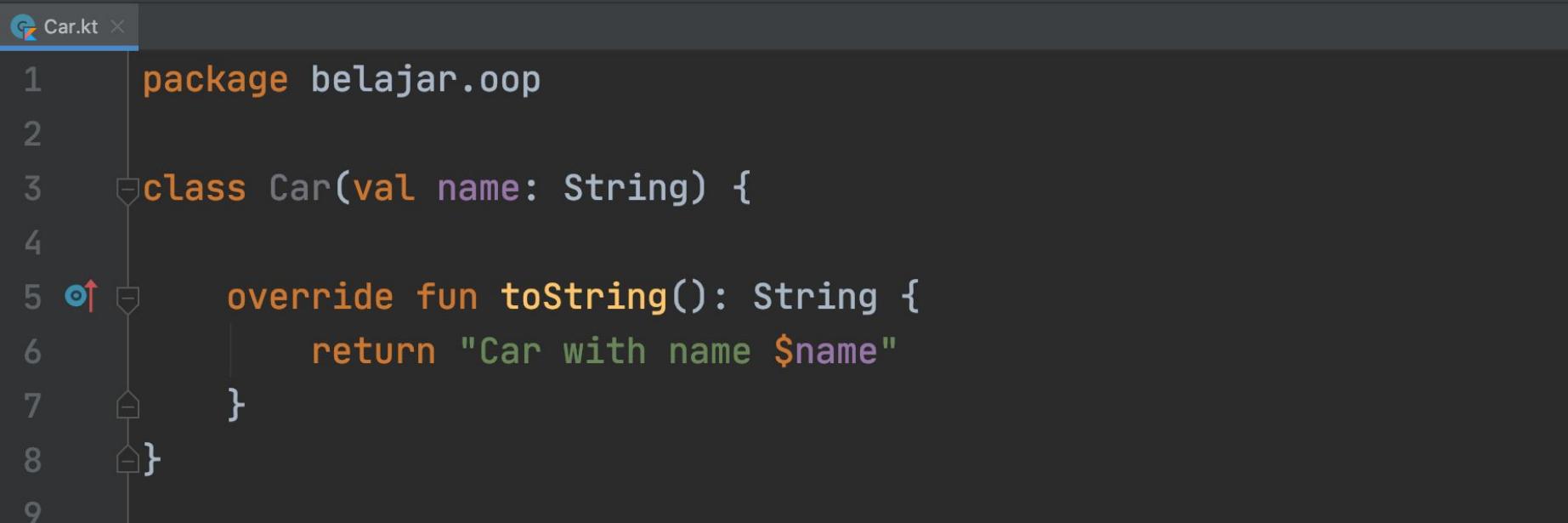
# ToString Function

---

# toString Function

- `toString()` adalah function yang digunakan untuk memberitahu representasi String dari object
- Saat kita melakukan `println` object, sebenarnya function `toString()` akan dipanggil
- Standarnya, function `toString()` akan mengembalikan referensi kode unik dari object
- Kita bisa meng-override function `toString()` jika ingin mengubah representasi dari String pada object kita

# Kode : `toString`



The screenshot shows a code editor window with a dark theme. The file tab at the top is labeled "Car.kt". The code itself is as follows:

```
1 package belajar.oop
2
3 class Car(val name: String) {
4
5     override fun toString(): String {
6         return "Car with name $name"
7     }
8 }
9
```

A red circular icon with a white arrow is positioned next to the opening brace of the `toString()` method. The code editor uses color coding: package names are orange, class names and variable names are purple, and strings are green.

---

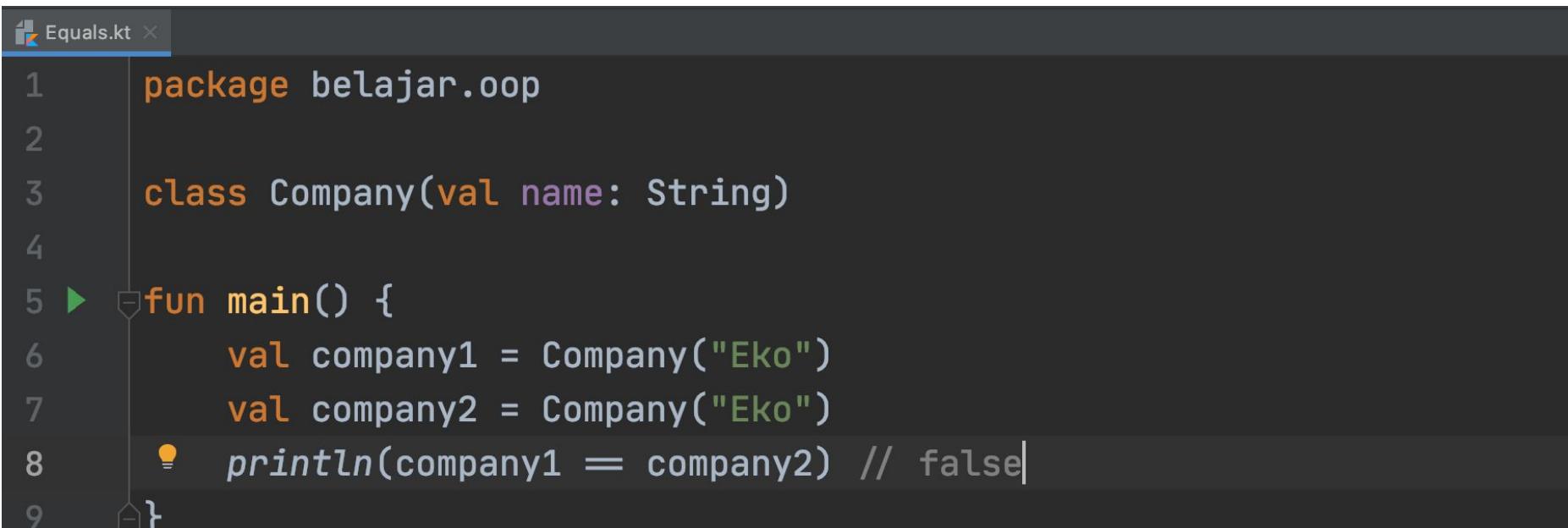
# Equals Function

---

# Equals Function

- Di Kotlin, semua objek bisa dibandingkan menggunakan operasi == atau !=
- Saat kita membandingkan objek menggunakan operasi == atau !=, sebenarnya Kotlin akan menggunakan function equals milik class Any
- Untuk mengubah cara membandingkannya, kita bisa meng-override function equals milik class Any

# Kode : Tanpa Equals

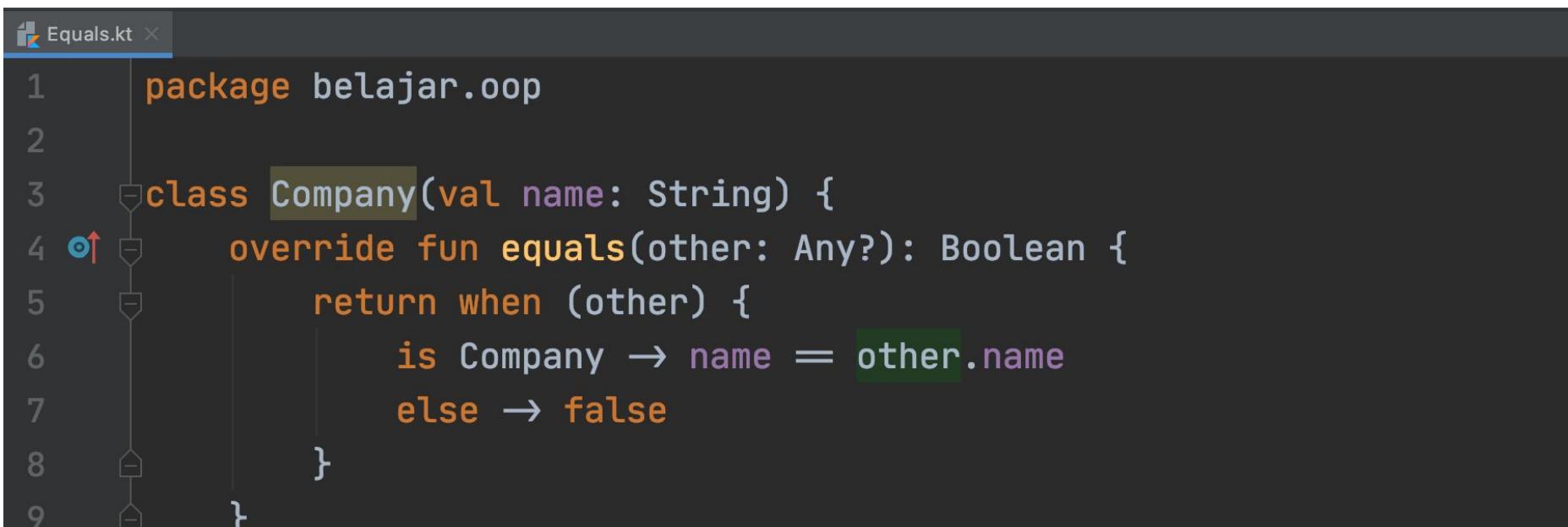


```
1 package belajar.oop
2
3 class Company(val name: String)
4
5 ► fun main() {
6     val company1 = Company("Eko")
7     val company2 = Company("Eko")
8     println(company1 == company2) // false
9 }
```

The screenshot shows a code editor window with a dark theme. The file tab at the top is labeled "Equals.kt". The code itself is as follows:1 package belajar.oop
2
3 class Company(val name: String)
4
5 ► fun main() {
6 val company1 = Company("Eko")
7 val company2 = Company("Eko")
8 println(company1 == company2) // false
9 }The code defines a class `Company` with a constructor parameter `name`. In the `main` function, two instances of `Company` are created with the same name ("Eko"). A `println` statement is used to print the result of the equality comparison between `company1` and `company2`, which is `false`. A yellow lightbulb icon is shown next to the `println` line, indicating a potential issue or warning.

---

# Kode : Dengan Equals



```
1 package belajar.oop
2
3 class Company(val name: String) {
4     override fun equals(other: Any?): Boolean {
5         return when (other) {
6             is Company → name == other.name
7             else → false
8         }
9     }
}
```

The image shows a screenshot of a code editor with a dark theme. The file is named 'Equals.kt'. The code defines a class 'Company' with a constructor taking a 'name' parameter of type String. It overrides the 'equals' method to compare two companies based on their names. A red arrow points to the 'override' keyword in the 'equals' declaration. The code editor has syntax highlighting and shows line numbers from 1 to 9.

---

# HashCode Function

---

# HashCode Function

- hashCode adalah function yang digunakan sebagai representasi angka unit untuk objek yang kita buat
- Function hashCode sangat berguna saat kita ingin mengkonversi objek kita menjadi angka
- Salah satu penggunaan hashCode yang banyak dilakukan adalah di struktur data, misal untuk memastikan tidak ada data duplicate, agar lebih mudah, di cek nilai hashCode nya, jika hashCode sama, maka dianggap objectnya sama

---

## Kode : Tanda hashCode

```
Equals.kt x
11
12 ► fun main() {
13     val company1 = Company("Eko")
14     val company2 = Company("Eko")
15
16     println(company1.hashCode() == company2.hashCode()) // false
17 }
18
19
```

# Kode : Dengan hashCode

```
Equals.kt x
1 package belajar.oop
2
3 class Company(val name: String) {
4     override fun hashCode(): Int {
5         return name.hashCode()
6     }
7
8     override fun equals(other: Any?): Boolean {
9         return when (other) {
```

---

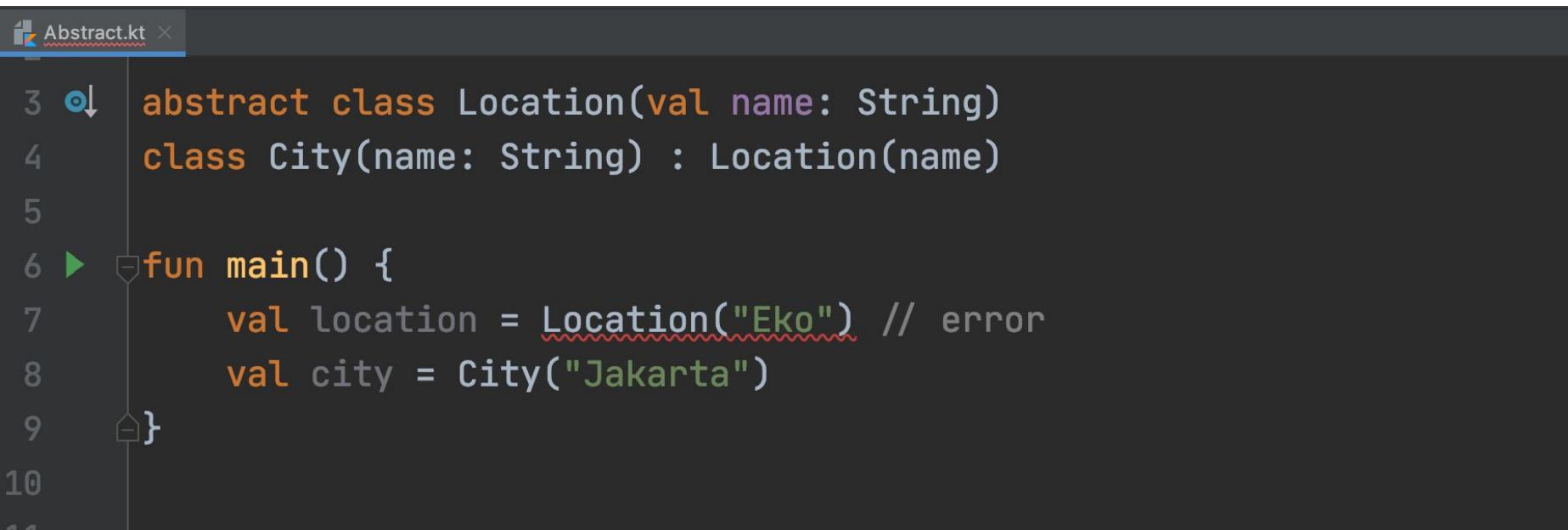
# Abstract Class

---

# Abstract Class

- Saat kita membuat class, kita bisa menjadikan sebuah class sebagai abstract
- Abstract class artinya, class tersebut tidak bisa dibuat sebagai object, hanya bisa diturunkan

# Kode : Abstract Class



```
Abstract.kt ×
3 ⚒ abstract class Location(val name: String)
4 class City(name: String) : Location(name)
5
6 ► fun main() {
7     val location = Location("Eko") // error
8     val city = City("Jakarta")
9 }
10
```

---

# Abstract Properties & Function

---

# Abstract Properties & Function

- Saat kita membuat class yang abstract, kita bisa membuat properties abstract dan function abstract di dalam class tersebut
- Properties dan function yang bersifat abstract, artinya wajib dibuat ulang di class Child nya

# Kode : Abstract Properties & Function



The screenshot shows a code editor window with a dark theme. The title bar says "AbstractPropertyFunction.kt". The code is as follows:

```
1 package belajar.oop
2
3 abstract class Animal {
4     abstract val name: String
5     abstract fun run(): Unit
6 }
7
8 class Cat : Animal() {
9     override val name: String = "Cat"
```

Annotations are present in the code:

- Line 3: A blue circle with a dot is next to the word "abstract".
- Line 4: A green circle with a dot is next to the word "val".
- Line 5: A green circle with a dot is next to the word "fun".
- Line 9: A red circle with a dot is next to the word "override".

# Kode : Extends Abstract

AbstractPropertyFunction.kt ×

```
7  
8     class Cat : Animal() {  
9         override val name: String = "Cat"  
10        override fun run() {  
11            println("Cat run!")  
12        }  
13    }  
14  
15
```

---

# Getter dan Setter

---

# Getter dan Setter

- Di bahasa pemrograman Java, ada sebuah konsep yang bernama Getter dan Setter saat membuat properties
- Getter adalah function yang dibuat untuk mengambil data properties
- Setter ada function untuk mengubah data propertie
- Di kotlin, kita tidak perlu manual untuk membuat function Getter dan Setter nya, karena sudah difasilitasi sehingga lebih mudah

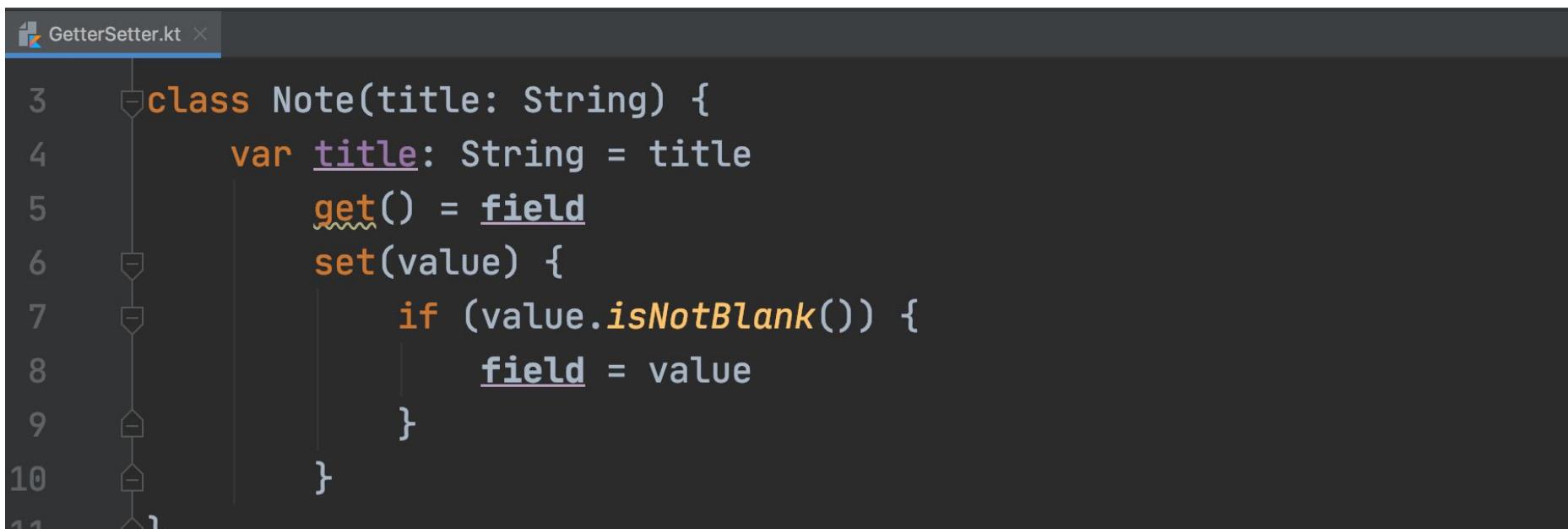
---

# Kode : Java Getter dan Setter

c Contact.java ×

```
3  public class Contact {  
4      private String name;  
5      public String getName() {  
6          return name;  
7      }  
8      public void setName(String name) {  
9          this.name = name;  
10     }  
11 }
```

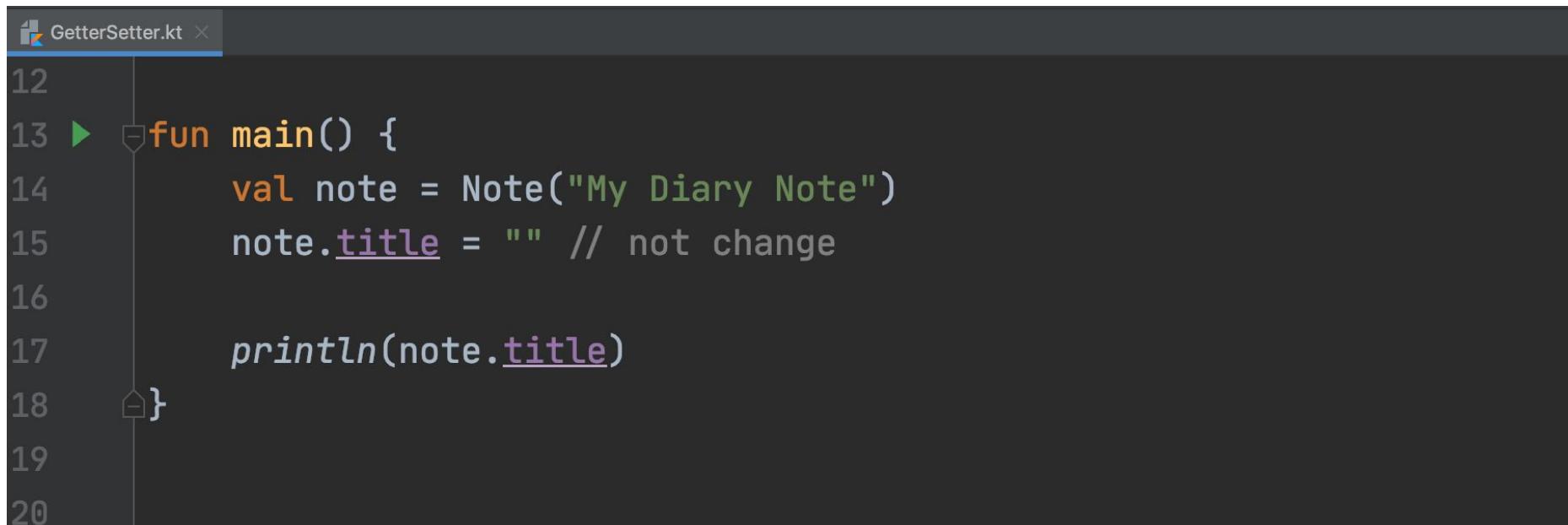
# Kode : Kotlin Getter dan Setter



```
1 GetterSetter.kt ×
2
3     class Note(title: String) {
4
4         var title: String = title
5             get() = field
6             set(value) {
7                 if (value.isNotBlank()) {
8                     field = value
9                 }
10            }
11    }
```

---

# Kode : Menggunakan Getter dan Setter



The screenshot shows a code editor window with a dark theme. The file tab at the top is labeled "GetterSetter.kt". The code itself is as follows:

```
12
13 > fun main() {
14     val note = Note("My Diary Note")
15     note.title = "" // not change
16
17     println(note.title)
18 }
19
20
```

The code defines a `Note` class with a private `title` field and a public `title` property. The `main` function creates a `Note` object and attempts to change its `title` property to an empty string, which is commented out as not changing it. The code editor highlights the `title` property in purple.

---

# Optional Getter dan Setter

- Getter dan Setter tidak wajib dideklarasikan semua di Kotlin
- Kita bisa hanya mendeklarasikan hanya Getter atau hanya Setter

# Kode : Kotlin Getter

```
GetterSetter.kt ×
13     class BigNote(val title: String) {
14         val bigTitle: String
15             get() = title.toUpperCase()
16     }
17
18 ►  fun main() {
19     val bigNote = BigNote("My Diary Note")
20     println(bigNote.title)
21     println(bigNote.bigTitle)
```

---

# Late-Initialized Properties

---

# Late-Initialized Properties

- Standarnya, properties di Kotlin wajib di inisialisasi di awal saat deklarasi properties tersebut
- Namun di Kotlin kita juga bisa menunda inisialisasi data para properties
- Dengan menggunakan kata kunci lateinit, kita bisa membuat properties tanpa harus langsung mengisi datanya
- Kata kunci lateinit hanya bisa digunakan di var, tidak bisa digunakan di val

# Kode : Late-Initialized Properties

LateinitializedProperties.kt ×

```
1 package belajar.oop
2
3 class Television {
4     lateinit var brand: String
5
6     fun initTelevision() {
7         brand = "Samsung"
8     }
9 }
```

---

# Kode : Mengakses Lateinit Properties

```
LateInitializedProperties.kt ×
11 ►  fun main() {
12     val television = Television()
13     println(television.brand) // error
14
15     television.initTelevision()
16     println(television.brand) // success
17 }
18
```

---

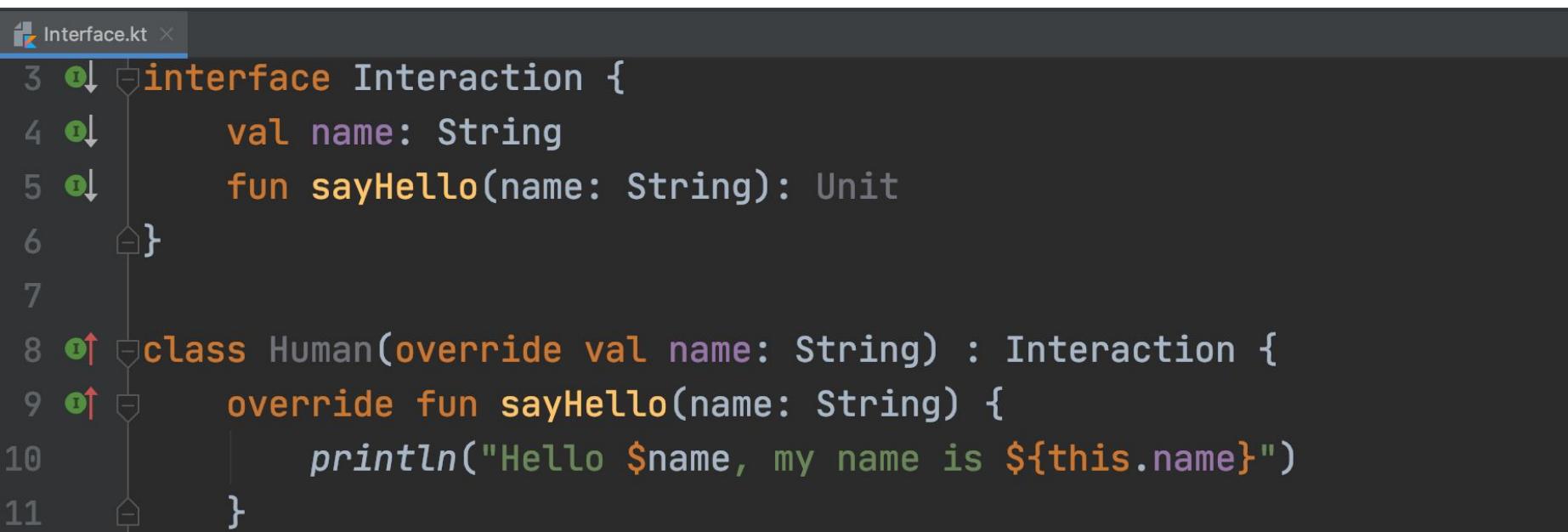
# Interface

---

# Interface

- Di kotlin, deklarasi type tidak hanya dalam bentuk class, ada juga interface
- Interface adalah blueprint, prototype atau cetakan di Kotlin
- Berbeda dengan Class, Interface tidak bisa langsung dibuat sebagai Object
- Interface hanya bisa diturunkan, dan biasanya Interface digunakan sebagai kontrak untuk class - class turunannya
- Secara standar, semua properties dan function di Interface adalah abstract

# Kode : Interface



The screenshot shows a code editor window titled "Interface.kt". The code defines an interface named "Interaction" with a single method "sayHello". It then provides an implementation for "Interaction" in the "Human" class, which overrides the "sayHello" method to print a greeting message.

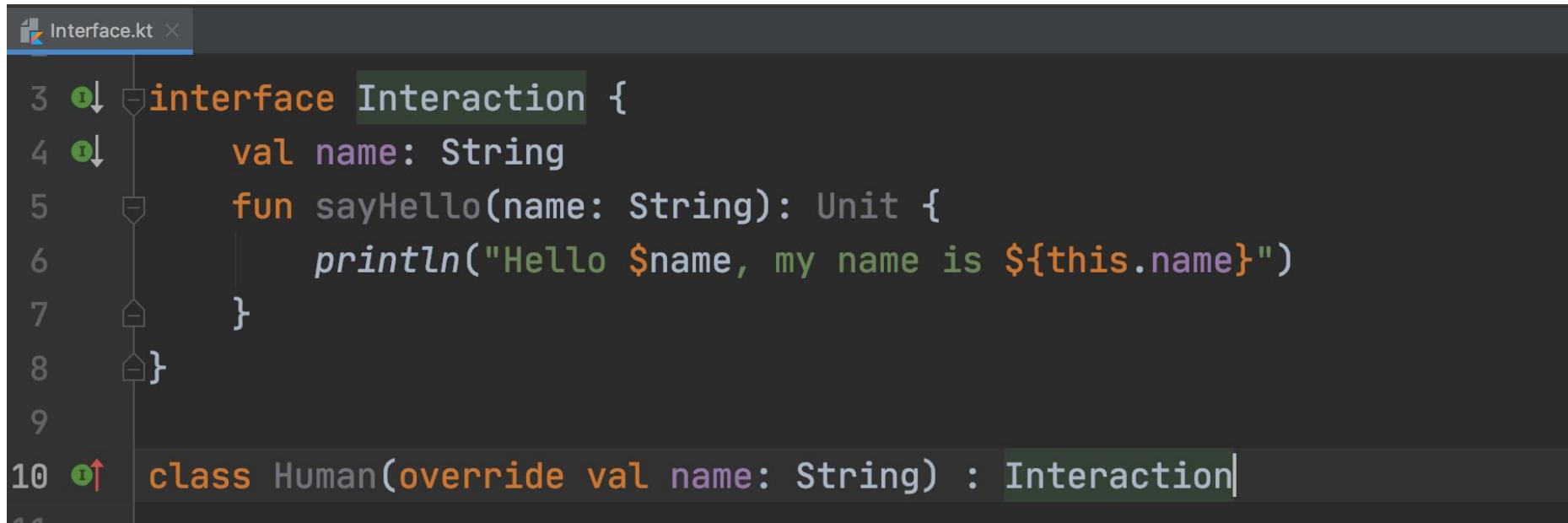
```
1  interface Interaction {  
2      val name: String  
3      fun sayHello(name: String): Unit  
4  }  
5  
6  class Human(override val name: String) : Interaction {  
7      override fun sayHello(name: String) {  
8          println("Hello $name, my name is ${this.name}")  
9      }  
10 }
```

---

## Concrete Function di 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

# Kode : Concrete Function di Interface



The screenshot shows a code editor window titled "Interface.kt". The code defines an interface named "Interaction" with a single concrete function "sayHello". This function prints a greeting message using the current object's name. Below the interface, a class "Human" is defined, which implements the "Interaction" interface and overrides the "name" variable.

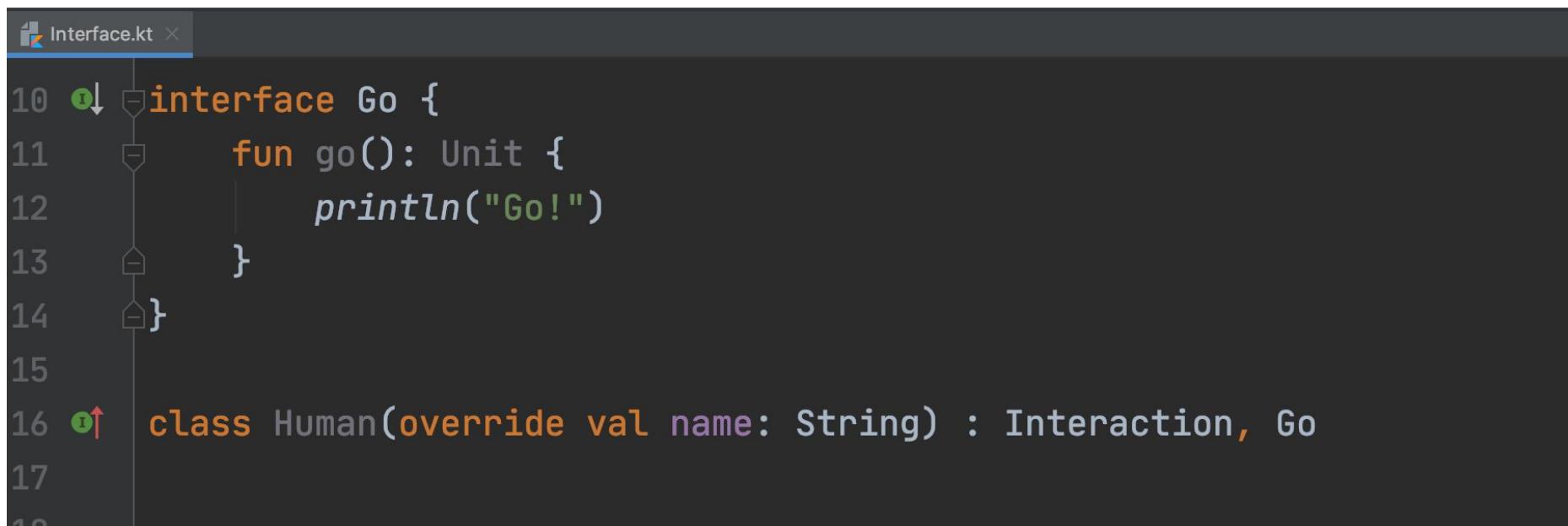
```
1  package com.example  
2  
3  interface Interaction {  
4      val name: String  
5      fun sayHello(name: String): Unit {  
6          println("Hello $name, my name is ${this.name}")  
7      }  
8  }  
9  
10 class Human(override val name: String) : Interaction
```

---

## Multiple Inheritance dengan Interface

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

# Kode : Multiple Inheritance



The screenshot shows a code editor window with a dark theme. The title bar says "Interface.kt". The code is as follows:

```
10  interface Go {  
11      fun go(): Unit {  
12          println("Go!")  
13      }  
14  }  
15  
16  class Human(override val name: String) : Interaction, Go
```

Annotations in the code editor:

- Line 10: A green circle with a question mark icon is next to the word "interface".
- Line 16: A green circle with a question mark icon is next to the word "Human".
- Line 16: A red arrow icon is next to the word "Interaction".

---

# Inheritance antar Interface

- Tidak hanya Class yang punya kemampuan Inheritance
- Interface juga bisa melakukan Inheritance dengan Interface lain
- Namun Interface tidak bisa melakukan Inheritance dengan Class

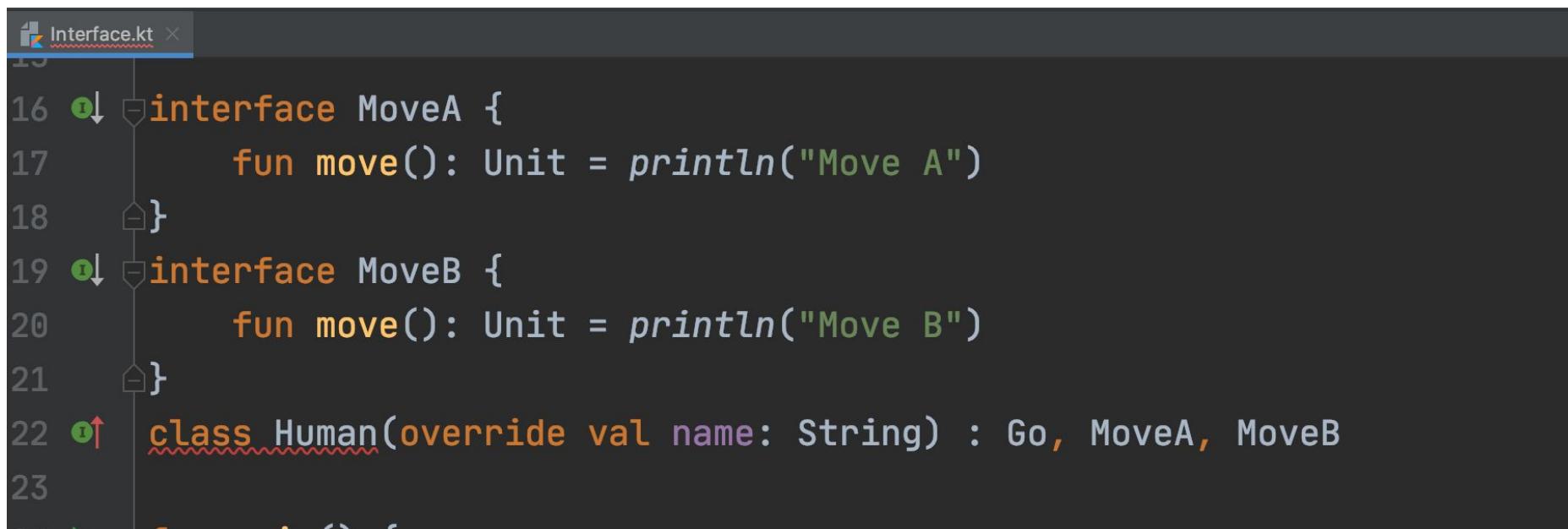
# Kode : Inheritance antar Interface



The screenshot shows a code editor window with a dark theme. The file is named "Interface.kt". The code defines an interface "Go" and a class "Human" that implements it.

```
10  interface Go : Interaction {  
11      fun go(): Unit {  
12          println("Go ${this.name}!")  
13      }  
14  }  
15  
16  class Human(override val name: String) : Go
```

# Konflik di Interface



```
Interface.kt x
15
16 ①↓ interface MoveA {
17      fun move(): Unit = println("Move A")
18  }
19 ①↓ interface MoveB {
20      fun move(): Unit = println("Move B")
21  }
22 ①↑ class Human(override val name: String) : Go, MoveA, MoveB
23
```

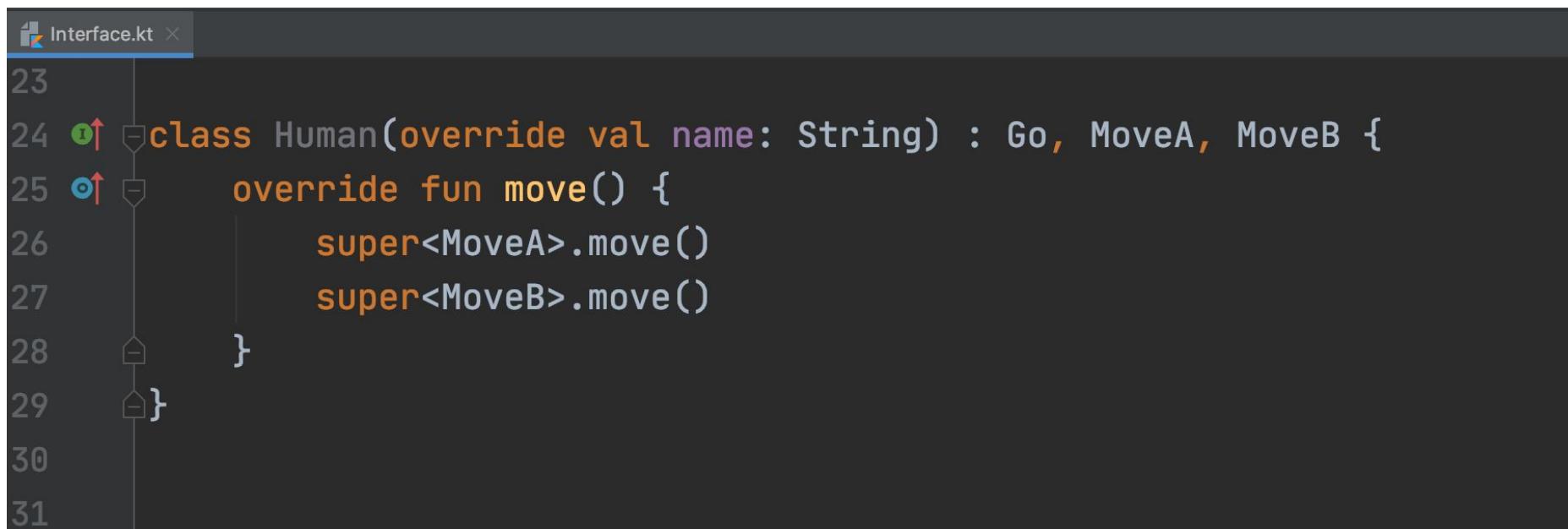
The screenshot shows a code editor window with a dark theme. The title bar says "Interface.kt". The code is as follows:

```
15
16 ①↓ interface MoveA {
17      fun move(): Unit = println("Move A")
18  }
19 ①↓ interface MoveB {
20      fun move(): Unit = println("Move B")
21  }
22 ①↑ class Human(override val name: String) : Go, MoveA, MoveB
23
```

Annotations in the code:

- Line 16: A green circle with a question mark and a downward arrow is positioned next to the word "interface".
- Line 19: A green circle with a question mark and a downward arrow is positioned next to the word "interface".
- Line 22: A red circle with a question mark and an upward arrow is positioned next to the word "Human".

# Memperbaiki Konflik di Interface



The screenshot shows a code editor window titled "Interface.kt". The code is a Kotlin class definition:

```
13
14 class Human(override val name: String) : Go, MoveA, MoveB {
15     override fun move() {
16         super<MoveA>.move()
17         super<MoveB>.move()
18     }
19 }
```

The code editor highlights several parts of the code in orange, including the class name "Human", the interface names "Go", "MoveA", and "MoveB", and the super type reference "super<MoveA>". There are also two small circular icons with arrows pointing up (one green, one blue) positioned near the first two lines of code.

---

# Visibility Modifiers

---

# Visibility Modifiers

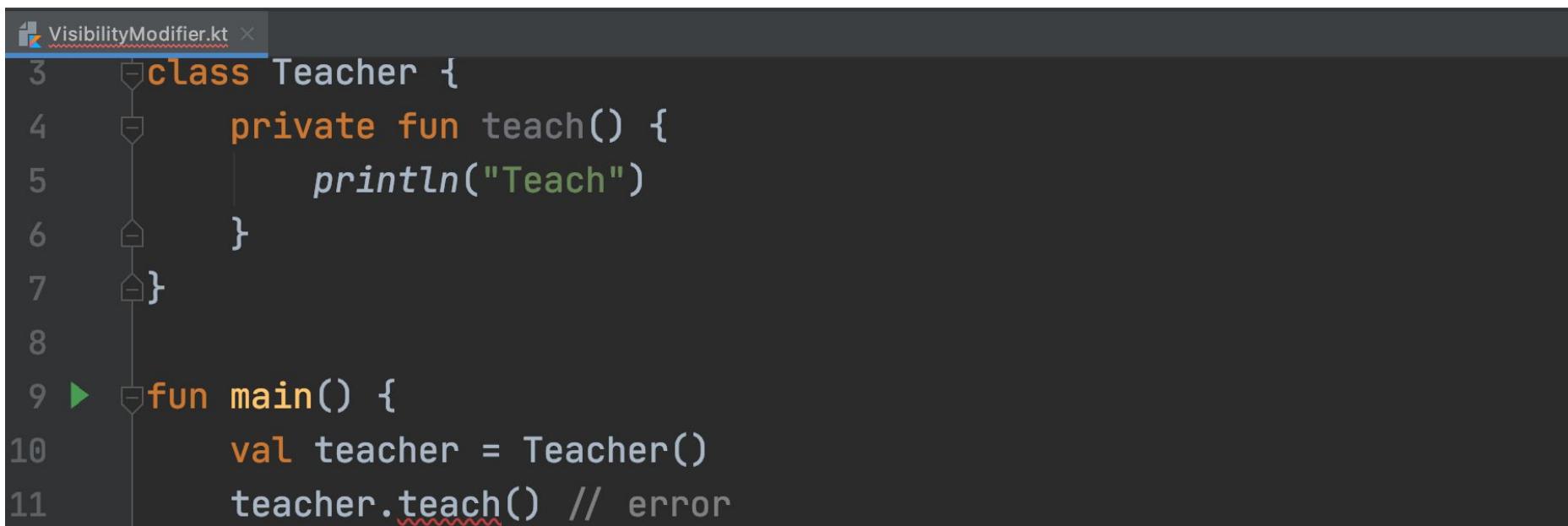
- Class, Interface, Constructor, Function, dan Properties (termasuk Getter dan Setter) bisa memiliki Visibility Modifiers di Kotlin
- Ada 4 visibility modifiers di kotlin, yaitu: public, private, protected, internal
- Secara standar visibility modifiers di kotlin adalah public

---

# Perbedaan tiap 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.

# Kode : Visibility Modifiers



A screenshot of a code editor showing a Kotlin file named `VisibilityModifier.kt`. The code defines a class `Teacher` with a private method `teach()`. In the `main` function, an attempt is made to call `teacher.teach()`, which results in an error.

```
VisibilityModifier.kt
1
2
3     class Teacher {
4         private fun teach() {
5             println("Teach")
6         }
7     }
8
9     fun main() {
10        val teacher = Teacher()
11        teacher.teach() // error
```

---

# Extension Function

---

# Extension Function

- Pada materi Kotlin Dasar kita sudah membahas tentang Extension Function, yaitu menambahkan function pada tipe data yang sudah ada

# Kode : Extension Function



The screenshot shows a code editor window with a dark theme. The file is named "Student.kt". The code defines an extension function "sayGoodBye" and a main function:

```
11  fun Student.sayGoodBye(name: String) {  
12      println("GoodBye $name, my name is ${this.name}")  
13  }  
14  
15 > fun main() {  
16     val eko = Student("Eko")  
17     eko.sayGoodBye("Joko")  
18  }  
19
```

The code uses color coding for syntax: orange for keywords like "fun", "Student", "println", "val", and "this"; green for strings and variable names; and grey for numbers and symbols. Line numbers are visible on the left. A green arrow icon is next to the "main" declaration.

---

## Extension Bukanlah Function di Class

- Perlu diperhatikan, bahwa saat membuat extension function, kita tidak memodifikasi class aslinya
- Extension function hanyalah sebuah function bantuan yang artinya, kita tidak bisa mengakses data private atau protected dari class tersebut

# Kode : Extension Function Error



The screenshot shows a code editor window with a dark theme. The title bar says "Student.kt". The code is as follows:

```
1
2
3 class Student(val name: String, private val age: Int)
4
5 fun Student.sayGoodBye(name: String) {
6     println("GoodBye $name, my age is ${this.age}") // error
7 }
8
9 ► fun main() {
10    val eko = Student("Eko", 12)
```

The code editor highlights the word "age" in the println statement with a red squiggly underline, indicating a syntax error.

---

## Nullable Extension Function

- Secara standar, extension function hanya bisa untuk data yang tidak null
- Jika kita ingin membuat extension function yang bisa digunakan untuk data yang bisa null, kita perlu menggunakan kata kunci ? (tanda tanya)

# Kode : Nullable Extension Function

```
Student.kt ×
5  fun Student?.sayGoodBye(name: String) {
6      if (this != null) {
7          println("GoodBye $name, my age is ${this.name}") // error
8      }
9  }
10 fun main() {
11     val eko: Student? = Student("Eko", 12)
12     eko.sayGoodBye("Joko")
13 }
```

---

# Extension Properties

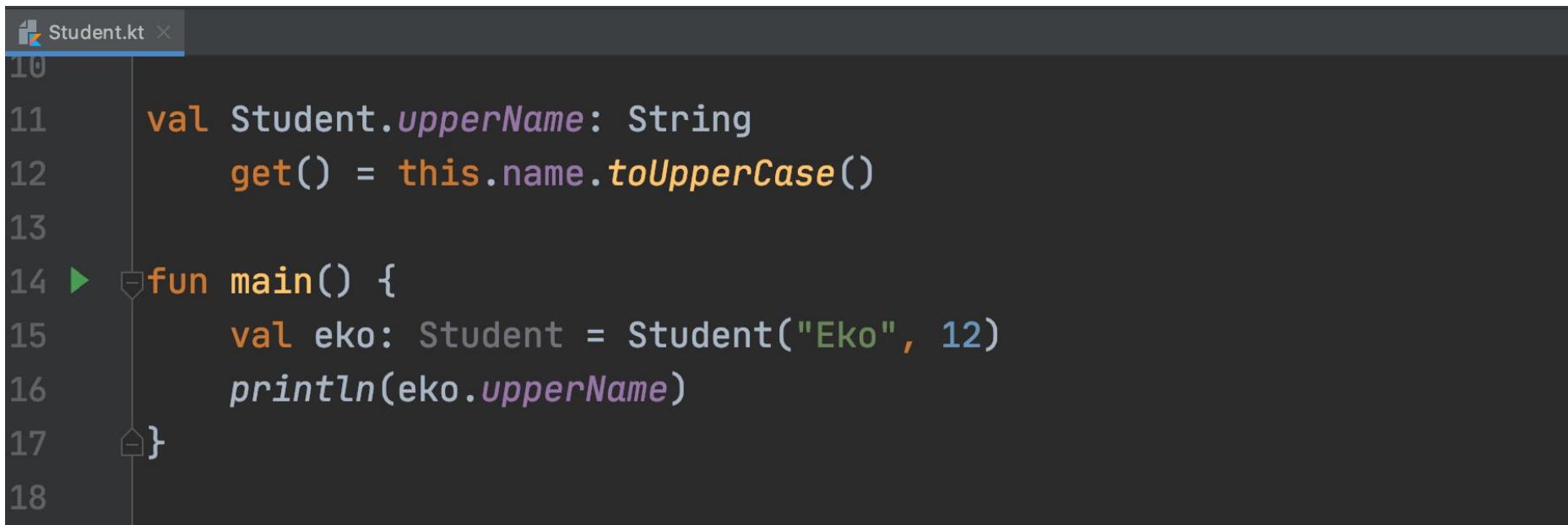
---

# Extension Properties

- Selain function, kita juga bisa membuat extension untuk properties di Kotlin
- Untuk membuat extension properties, kita bisa membuat properties dengan Getter atau Setter

---

# Kode : Extension Properties



The screenshot shows a code editor window with a dark theme. The title bar says "Student.kt". The code is as follows:

```
10
11     val Student.upperName: String
12         get() = this.name.toUpperCase()
13
14 ► fun main() {
15     val eko: Student = Student("Eko", 12)
16     println(eko.upperName)
17 }
18
```

Line 14 has a green play icon to its left, indicating it's the current line of execution.

---

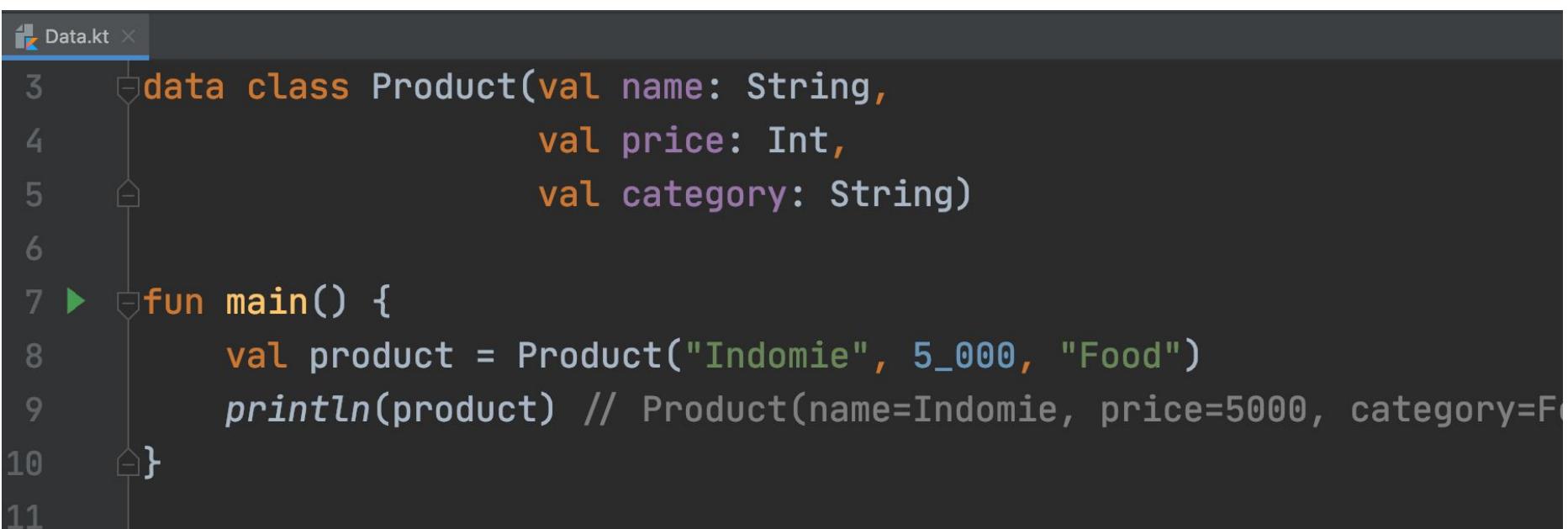
# Data Class

---

# Data Class

- Kadang kita sering membuat class yang hanya digunakan sebagai representasi dari data
- Data Class cocok digunakan dalam hal ini
- Data Class merupakan class yang secara otomatis akan membuatkan function equals, hashCode, toString dan copy dari semua properties yang terdapat di primary constructors yang dimiliki oleh data class

# Kode : Data Class



The screenshot shows a code editor window with a dark theme. The file tab at the top is labeled "Data.kt". The code itself is a simple example of a data class and its usage:

```
1 data class Product(val name: String,  
2                     val price: Int,  
3                     val category: String)  
4  
5  
6  
7 ► fun main() {  
8     val product = Product("Indomie", 5_000, "Food")  
9     println(product) // Product(name=Indomie, price=5000, category=F  
10 }  
11
```

The code uses the `Product` class to create a variable `product` and then prints it to the console. The output will be a string representation of the `Product` object with its properties: name, price, and category.

---

# Copy Data Class

- Data class memiliki function copy yang bisa digunakan untuk menduplikasi object.
- Bahkan metode copy yang terdapat di data class bisa digunakan untuk menduplikasi sekaligus mengubah properties nya

---

## Kode : Copy Data Class



The screenshot shows a code editor window with a dark theme. The tab bar at the top has a file icon and the text "Data.kt x". The main area displays the following Kotlin code:

```
6
7 >  fun main() {
8     val product = Product("Indomie", 5_000, "Food")
9     val product2 = product.copy()
10    val product3 = product.copy(price = 10_000)
11 }
12
13
14
```

The code defines a main function that creates a Product object and then creates two copies of it using the copy() method. The first copy retains the original price, while the second copy sets a new price of 10\_000. Lines 7 through 11 are highlighted with a light gray background.

---

# Sealed Class

---

# Sealed Class

- Sealed class merupakan jenis class yang didesain untuk inheritance
- Sealed class tidak bisa diinstansiasi menjadi object, dan secara standar sealed class merupakan abstract class
- Sealed class sangat cocok digunakan sebagai class Parent

---

## Kode : Sealed Class



The screenshot shows a code editor window with a dark theme. The title bar says "Sealed.kt". The code is as follows:

```
1 package belajar.oop
2
3 sealed class Operation(val name: String)
4     class Plus : Operation("Add")
5     class Minus : Operation("Minus")
```

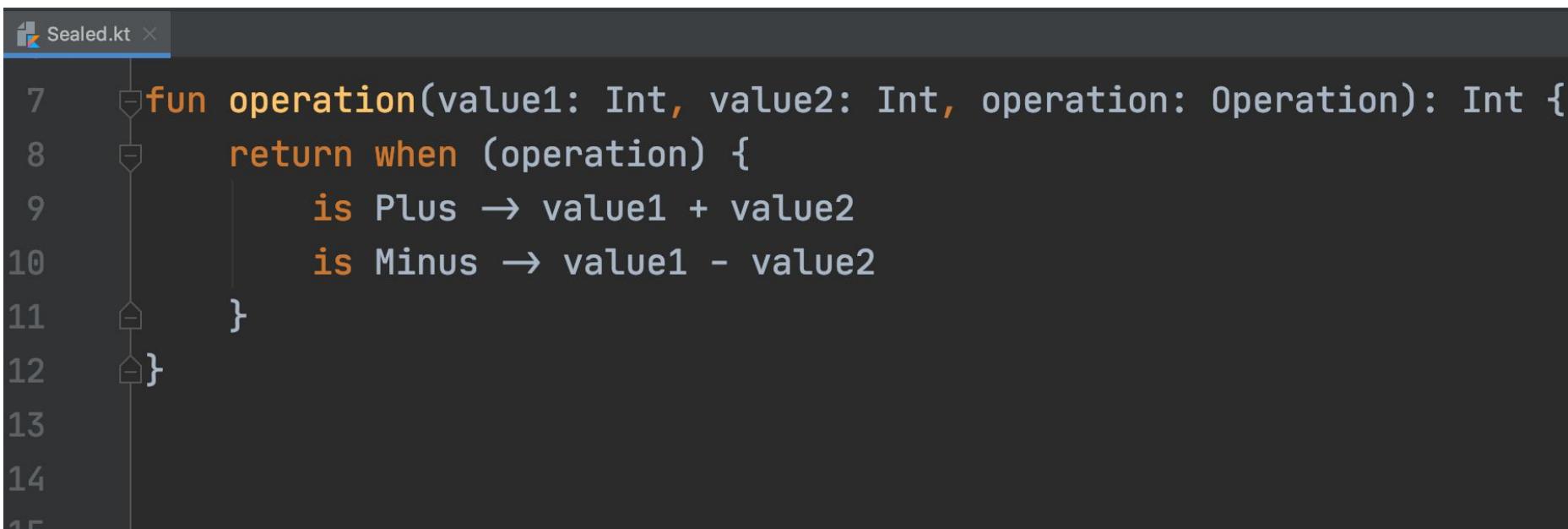
---

## Sealed Class di When Expression

- Sealed class sangat berguna saat kita menggunakan when expression
- Dengan menggunakan sealed class, kita bisa membatasi hanya class turunannya yang perlu di check

---

# Sealed Class di When Expression



The screenshot shows a code editor window with a dark theme. The file tab at the top is labeled "Sealed.kt". The code itself is a sealed class definition:

```
1  sealed class Operation {  
2      object Plus : Operation()  
3      object Minus : Operation()  
4  }  
5  
6  fun operation(value1: Int, value2: Int, operation: Operation): Int {  
7      return when (operation) {  
8          is Plus → value1 + value2  
9          is Minus → value1 - value2  
10     }  
11 }  
12 }  
13  
14  
15 }
```

The code uses a sealed class named `Operation` with two objects: `Plus` and `Minus`. A function `operation` takes three parameters: `value1`, `value2`, and `operation`. It returns the sum or difference of `value1` and `value2` based on the value of `operation`.

---

# Inner Class

---

# Inner Class

- Di Kotlin, kita bisa membuat class (Inner) di dalam class (Outer)
- Namun walaupun class Inner tersebut berada di dalam class Outer, namun class Inner tidak bisa mengakses data yang ada di dalam class Outer
- Agar class Inner bisa mengakses data yang ada di dalam class Outer, kita bisa menggunakan kata kunci inner

# Kode : Inner Class

```
InnerClass.kt ×
3   class Boss(val name: String) {
4
5       inner class Employee(val name: String) {
6
7           fun hi(){
8               println("Hi, I'm $name, and my boss is ${this@Boss.name}
9           }
10      }
11  }
```

---

## Kode : Membuat Object dari Inner Class

```
InnerClass.kt ×
14 >  fun main() {
15     val eko = Boss("Eko")
16
17     val joko = eko.Employee("Joko")
18     joko.hi()
19
20     val budi = eko.Employee("Budi")
21     budi.hi()
22 }
```

---

# Anonymous Class

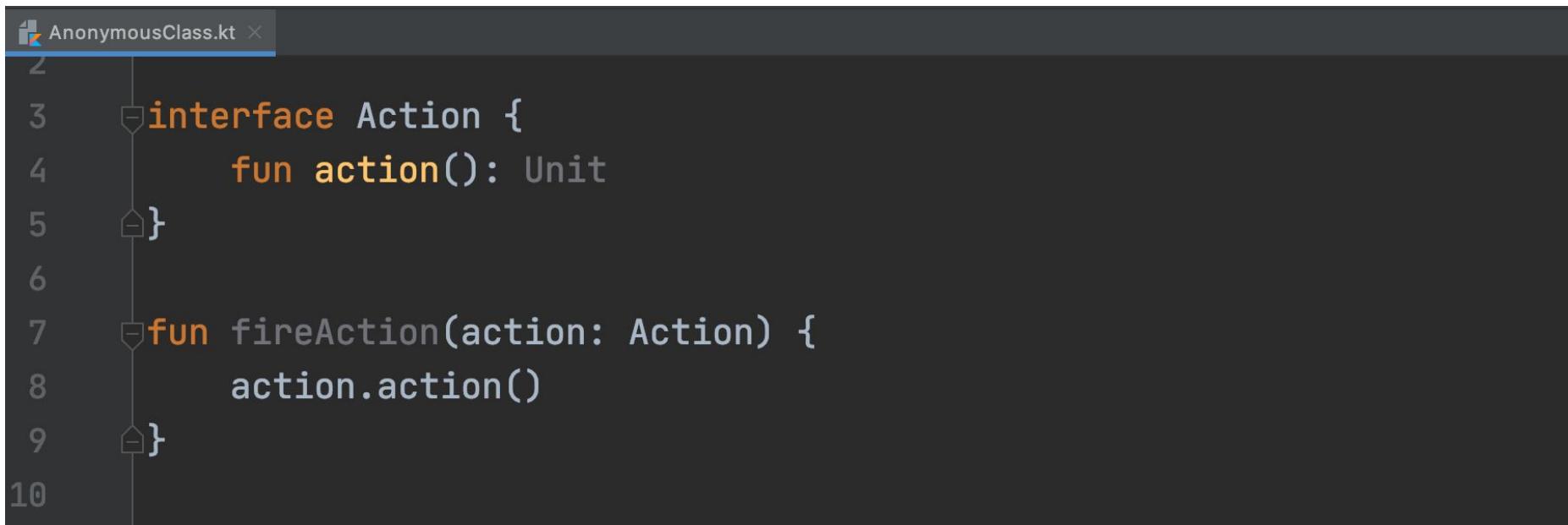
---

# Anonymous Class

- Saat kita ingin membuat object, maka kita diwajibkan untuk menggunakan deklarasi class yang lengkap
- Namun, Kotlin mendukung pembuatan object dari class yang bahkan belum lengkap
- Bahkan, di Kotlin, kita bisa membuat object dari interface
- Kemampuan ini dinamakan anonymous class
- Untuk membuat anonymous class, kita bisa menggunakan kata kunci object diikuti dengan deklarasi class Child seperti pada pewarisan

---

# Kode : Interface Action



A screenshot of a code editor showing an anonymous class definition named `AnonymousClass.kt`. The code defines an interface `Action` with a single function `action()`, and then creates an anonymous class that implements this interface and adds a new function `fireAction`.

```
AnonymousClass.kt
2
3     interface Action {
4         fun action(): Unit
5     }
6
7     fun fireAction(action: Action) {
8         action.action()
9     }
10
```

# Kode : Anonymous Class

AnonymousClass.kt

```
10
11 > fun main() {
12     fireAction(object : Action {
13         override fun action() = println("Action One")
14     })
15     fireAction(object : Action {
16         override fun action() = println("Action Two")
17     })
18 }
```

---

# Enum Class

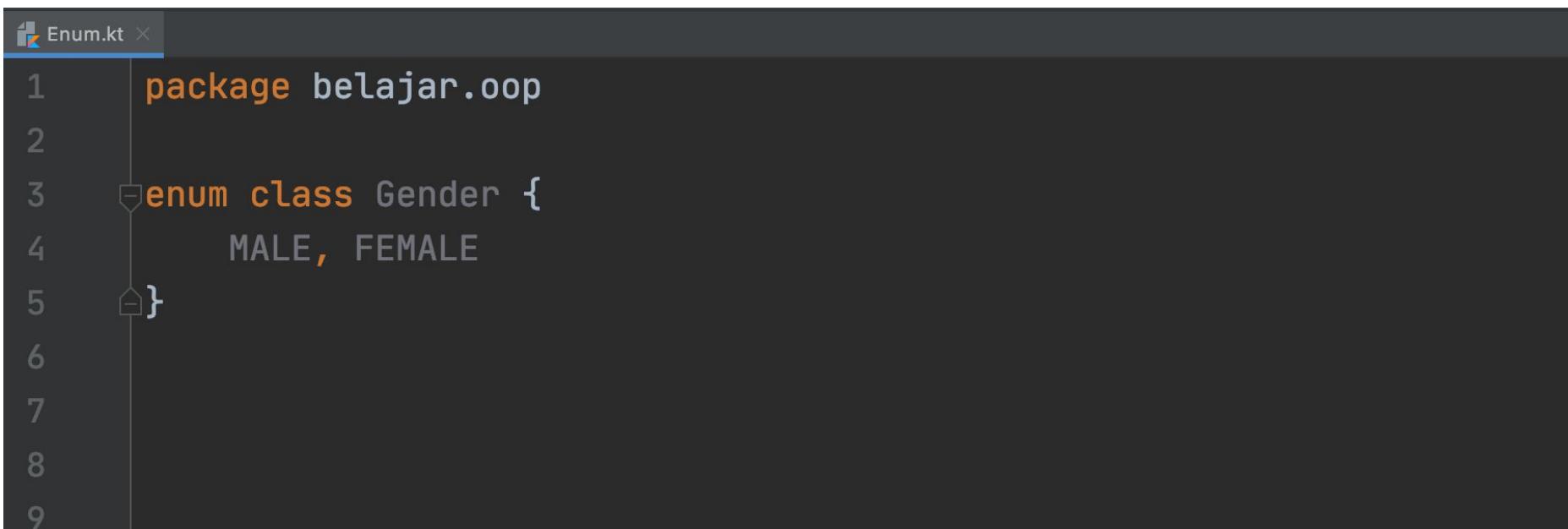
---

# Enum Class

- Enum class merupakan representasi dari class yang sudah tetap nilainya
- Biasanya enum class digunakan untuk jenis data yang sudah baku, seperti jenis kelamin, arah mata angin dan sejenisnya
- Untuk membuat enum class, kita bisa menggunakan kata kunci enum sebelum deklarasi class nya
- Kita tidak bisa membuat object dari class enum, namun kita bisa mendeklarasikan langsung object yang tersedia untuk enum class tersebut

---

# Kode : Membuat Enum Class



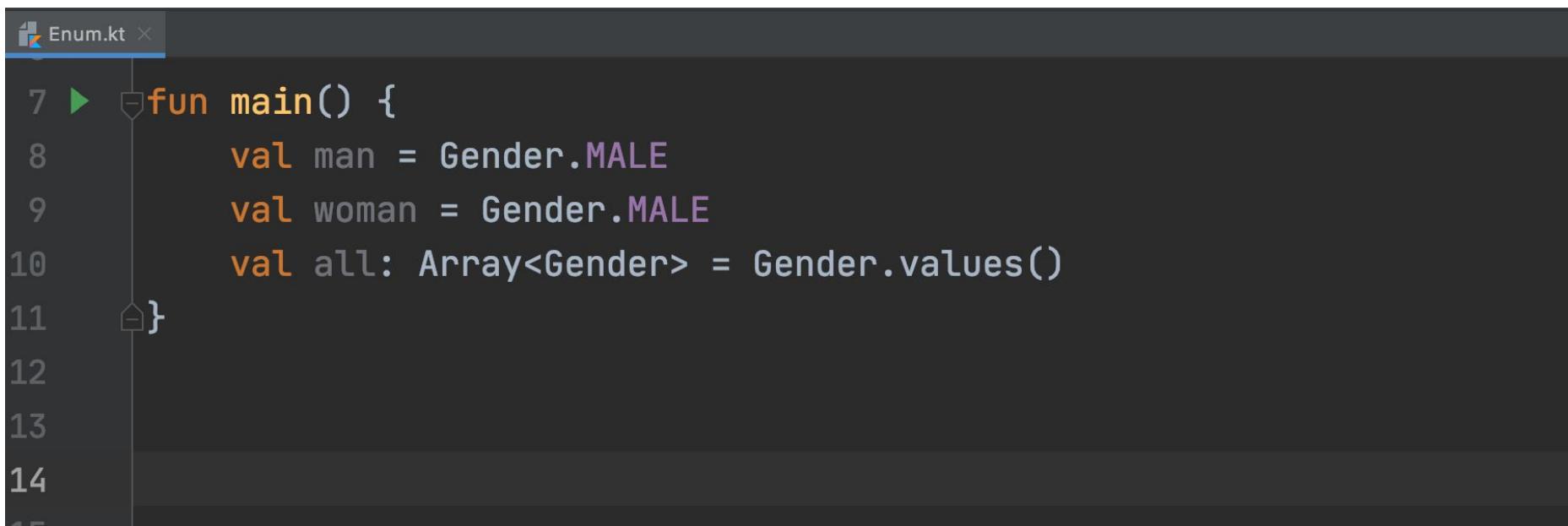
The screenshot shows a code editor window with a dark theme. The title bar says "Enum.kt x". The code itself is as follows:

```
1 package belajar.oop
2
3 enum class Gender {
4     MALE, FEMALE
5 }
```

The code defines an enum class named "Gender" with two values: "MALE" and "FEMALE". The file contains 9 lines of code, with lines 1 through 5 visible in the screenshot.

---

## Kode : Membuat Object Enum Class



The screenshot shows a code editor window with a dark theme. The title bar says "Enum.kt". The code is as follows:

```
1 fun main() {
2     val man = Gender.MALE
3     val woman = Gender.FEMALE
4     val all: Array<Gender> = Gender.values()
5 }
6
7
8
9
10
11
12
13
14
15
```

The code defines a main function that creates three variables: "man" of type Gender initialized to MALE, "woman" of type Gender initialized to FEMALE, and "all" of type Array<Gender> initialized to the values of the Gender enum. The Gender enum is defined elsewhere in the code.

---

## Properties & Function di Enum Class

- Enum class adalah class seperti biasa pada umumnya
- Enum class bisa memiliki properties ataupun function
- Namun jika properties nya di set menggunakan constructors, maka saat pembuatan object enum, wajib diisi, dan jika terdapat abstract function, wajib di override pada saat pembuatan object enum

---

# Kode : Properties & Function di Enum Class



The screenshot shows a code editor window with a dark theme. The file tab at the top left is labeled "Enum.kt". The code itself is as follows:

```
1 enum class Gender(val description: String) {
2     MALE("Male"),
3     FEMALE("Female");
4
5     fun printDescription() {
6         println(description)
7     }
8 }
```

The code defines an enum class named "Gender" with two values: "MALE" and "FEMALE". It includes a function "printDescription" that prints the enum's description.

---

# Singleton Object

---

# Singleton Object

- Salah satu konsep Design Pattern yang sangat populer adalah singleton object
- Singleton object adalah object yang hanya dibuat satu kali
- Di Kotlin, membuat object singleton sangat mudah, hanya dengan menggunakan kata kunci object
- Cara membuat singleton object di Kotlin sama seperti membuat class
- Singleton object mirip dengan class, bisa extends class ataupun interface
- Namun singleton object tidak memiliki constructors

---

# Kode : Singleton Object



The screenshot shows a code editor window with a dark theme. The title bar says "SingletonObject.kt". The code itself is as follows:

```
1 package belajar.oop
2
3 object Utilities {
4     val name = "Utilities"
5     fun toUpper(value: String): String {
6         return value.toUpperCase()
7     }
8 }
9 }
```

The code defines a singletons object named "Utilities". It contains a constant "name" set to "Utilities" and a function "toUpper" which takes a string and returns its uppercase version.

---

# Kode : Mengakses Singleton Object

```
SingletonObject.kt ×
13
14 ►  fun main() {
15     val result = Utilities.toUpperCase("Eko")
16
17     println(result)
18     println(Utilities.name)
19 }
20
21
```

---

# Inner Object

- Di Kotlin, singleton object bisa dibuat di dalam sebuah class
- Namun berbeda dengan inner class, singleton object tidak bisa mengakses properties atau function yang ada di outer class nya

---

## Kode : Inner Object

```
11 class Application(val name: String) {
12     object Utilities{
13         fun hello(name: String): Unit {
14             println("Hello $name")
15         }
16     }
17 }
18 fun main() {
19     Application.Utilities.hello("Eko")
```

---

# Companion Object

---

# Companion Object

- Companion object adalah kemampuan membuat inner object di dalam class, tanpa harus menggunakan nama object
- Companion object secara otomatis akan menggunakan nama Companion, atau bisa langsung diakses lewat nama class nya

---

# Kode : Companion Object

```
11  class Application(val name: String) {  
12      companion object {  
13          fun hello(name: String): Unit {  
14              println("Hello $name")  
15          }  
16      }  
17  }  
18  ► fun main() {  
19      Application.hello("Eko")  
20      Application.Companion.hello("Eko")
```

---

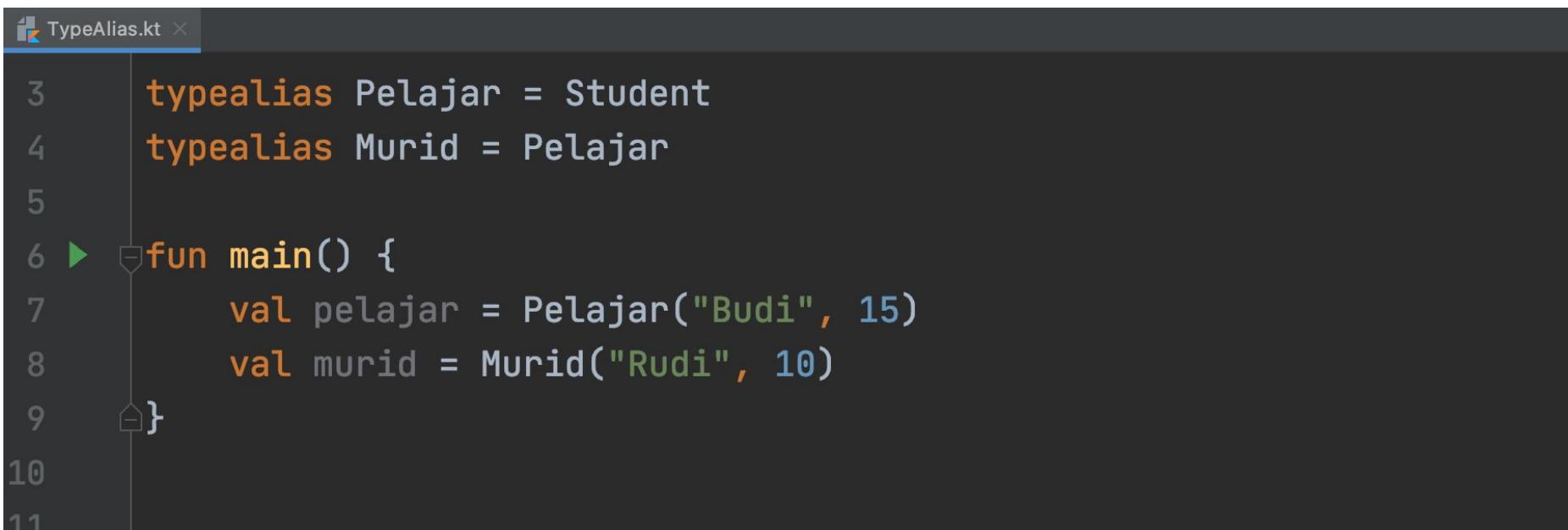
# Type Alias

---

# Type Alias

- Kotlin mendukung Type Alias
- Type Alias adalah membuat nama berbeda dari tipe data yang telah ada
- Biasanya ini digunakan ketika ada tipe data dengan nama yang sama, atau untuk mempersingkat tipe data sehingga kita tidak perlu menulisnya terlalu panjang

# Kode : Type Alias



The screenshot shows a code editor window with a dark theme. The file tab at the top is labeled "TypeAlias.kt". The code itself is as follows:

```
1 typealias Pelajar = Student
2 typealias Murid = Pelajar
3
4
5
6 ► fun main() {
7     val pelajar = Pelajar("Budi", 15)
8     val murid = Murid("Rudi", 10)
9 }
10
11
```

The code defines two type aliases: `Pelajar` and `Murid`, both pointing to the `Student` class. It then uses these aliases in the `main` function to create instances of `Pelajar` and `Murid` with names and ages.

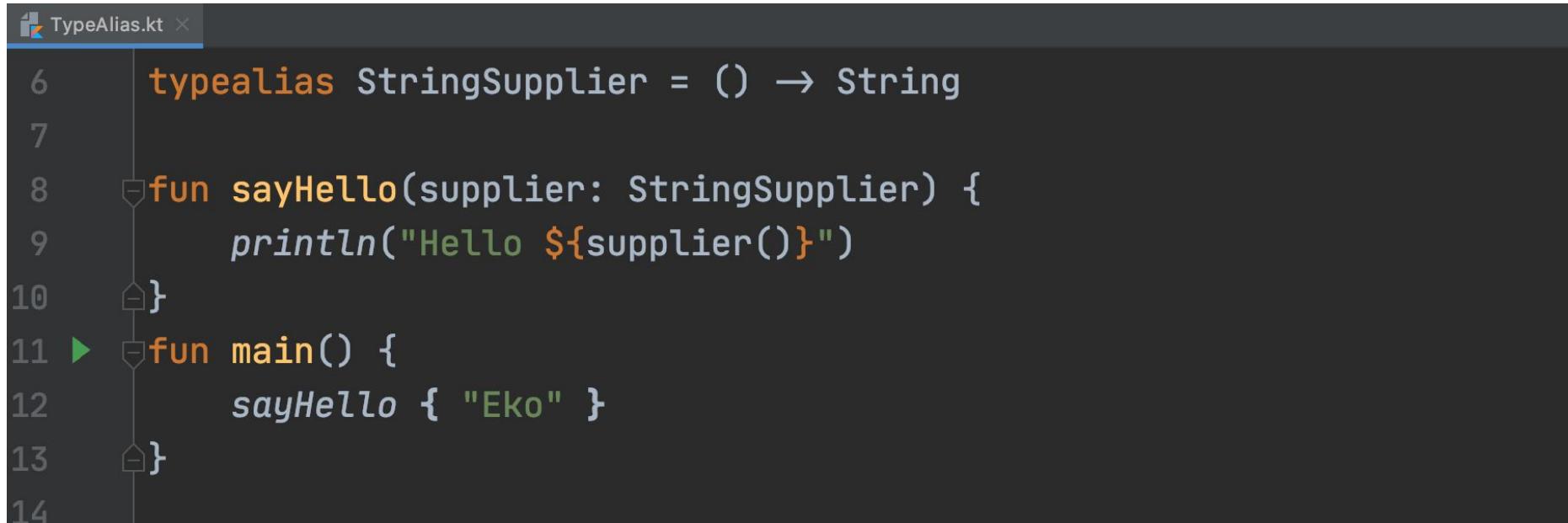
---

# Type Alias untuk Function

- Type Alias tidak hanya bisa digunakan untuk class saja, namun bisa untuk function
- Dengan menggunakan Type Alias untuk function, kita akan lebih mudah saat menggunakan function sebagai parameter, karena lebih sederhana

---

# Kode : Type Alias untuk Function



The screenshot shows a code editor window with a dark theme. The file is named "TypeAlias.kt". The code defines a type alias and a function:

```
6  typealias StringSupplier = () -> String
7
8  fun sayHello(supplier: StringSupplier) {
9      println("Hello ${supplier()}")
10 }
11
12 fun main() {
13     sayHello { "Eko" }
14 }
```

The code editor highlights certain parts of the code in different colors: "StringSupplier" is orange, "Hello" and "Eko" are green, and the brace completion icons are light blue.

---

# Inline Class

---

# Inline Class

- Kadang kita hanya membuat class dengan satu properties
- Terlalu banyak class akan ada konsekuensinya, yaitu memakan konsumsi memory yang lebih banyak ketika program kita berjalan
- Kotlin memiliki kemampuan untuk mengubah class menjadi inline dengan menggunakan kata kunci inline

---

# Kode : Membuat Inline Class



The screenshot shows a code editor window titled "InlineClass.kt". The code defines an inline class "Token" and a main function.

```
1 // Token.kt
2
3 inline class Token(val value: String) {
4     fun toUpper(): String = value.toUpperCase()
5 }
6
7 fun main() {
8     val login = Token("secret token")
9     println(login.toUpper())
10}
11
```

---

# Delegation

---

# Delegation

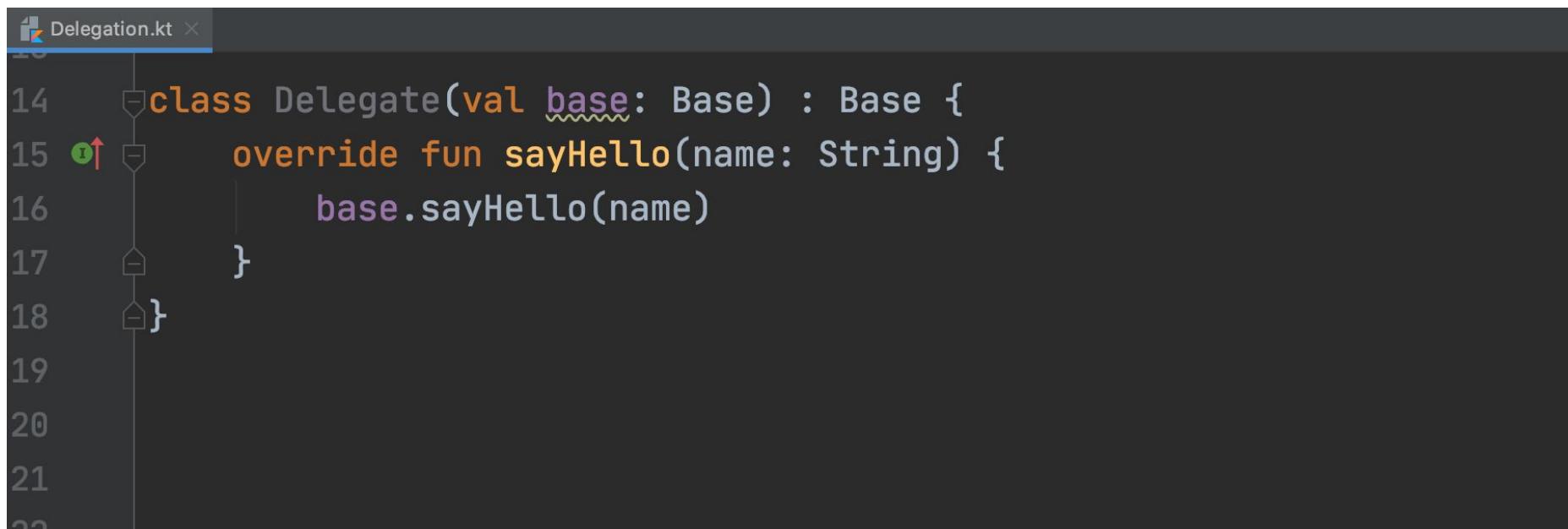
- Salah satu design pattern lain yang populer adalah Delegation
- Delegation sederhananya adalah meneruskan properties atau function ke object yang lain
- Kotlin mendukung delegation tanpa harus membuat kode secara manual

# Kode : Base Class

Delegation.kt

```
3 ①↓ interface Base {  
4   ②↓     fun sayHello(name: String)  
5   }  
6   class MyBase : Base {  
7     ③↑     override fun sayHello(name: String) {  
8       println("Hello $name")  
9     }  
10    }  
11 }
```

# Kode : Delegation Manual



A screenshot of a code editor showing a file named "Delegation.kt". The code defines a class "Delegate" that delegates the "sayHello" method to a "base" object.

```
13
14 class Delegate(val base: Base) : Base {
15     override fun sayHello(name: String) {
16         base.sayHello(name)
17     }
18 }
```

The code editor highlights the "val base" declaration with a blue underline and shows a yellow info icon with an arrow pointing to it. The "base.sayHello(name)" call is also highlighted with a blue underline.

---

# Kode : Delegation



The screenshot shows a code editor window with a dark theme. The title bar says "Delegation.kt". The code is as follows:

```
13
14     class Delegate(val base: Base) : Base by base
15
16 ► fun main() {
17     val myBase = MyBase()
18     val delegate = Delegate(myBase)
19
20     delegate.sayHello("Eko")
21 }
```

The code defines a class `Delegate` that delegates to a `Base` object. It also contains a `main` function that creates a `MyBase` instance and a `Delegate` instance, then calls the `sayHello` method on the `Delegate` instance with the argument "Eko".

---

# Override Delegation

- Dalam delegation, properties dan function secara otomatis akan didelegasikan ke object yang dipilih
- Namun kita tetap bisa meng-override properties dan function jika kita mau

# Kode : Override Delegation



The screenshot shows a code editor window titled "Delegation.kt". The code is a Kotlin file demonstrating delegation:

```
13
14 class Delegate(val base: Base) : Base by base {
15     override fun sayHello(name: String) {
16         println("Delegate Hello $name")
17     }
18 }
19
20 fun main() {
21     val myBase = MyBase()
```

A green arrow-shaped icon with the number "1" is positioned above the opening brace of the "Delegate" class definition at line 14. The code uses the "by" keyword to delegate the "sayHello" method to the "base" property.

---

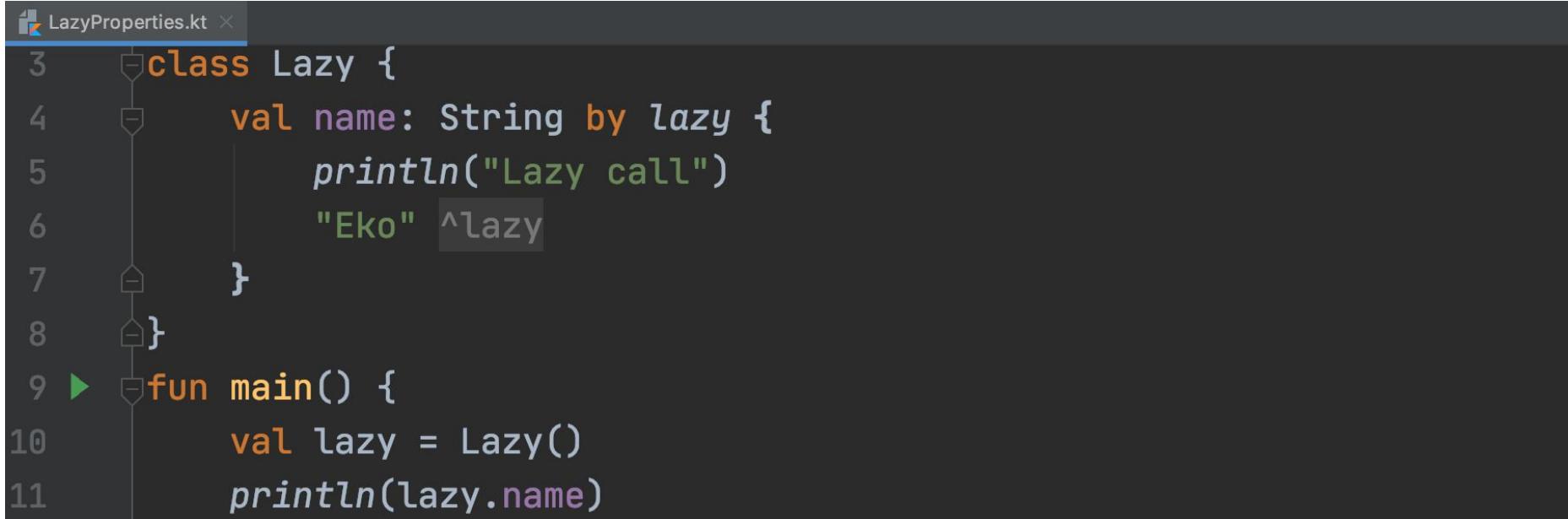
# Lazy Properties

---

# Lazy Properties

- Selain delegating di class, Kotlin mendukung delegating di properties. Namun materi ini tidak akan dibahas disini, karena butuh mengerti dahulu tentang Generic Programming. Materi Delagating di Properties akan dibahas di series Kotlin Generic
- Salah satu implementasi delegating properties yang sudah disediakan oleh Kotlin adalah, lazy properties
- Lazy adalah standar library yang telah disediakan agar properties baru diinisialisasi ketika properties itu diakses

# Kode : Lazy Properties



A screenshot of a code editor showing a file named `LazyProperties.kt`. The code defines a class `Lazy` with a lazy property `name` and a `main` function.

```
1  package com.example
2
3  class Lazy {
4      val name: String by lazy {
5          println("Lazy call")
6          "Eko" ^lazy
7      }
8  }
9  fun main() {
10     val lazy = Lazy()
11     println(lazy.name)
```

---

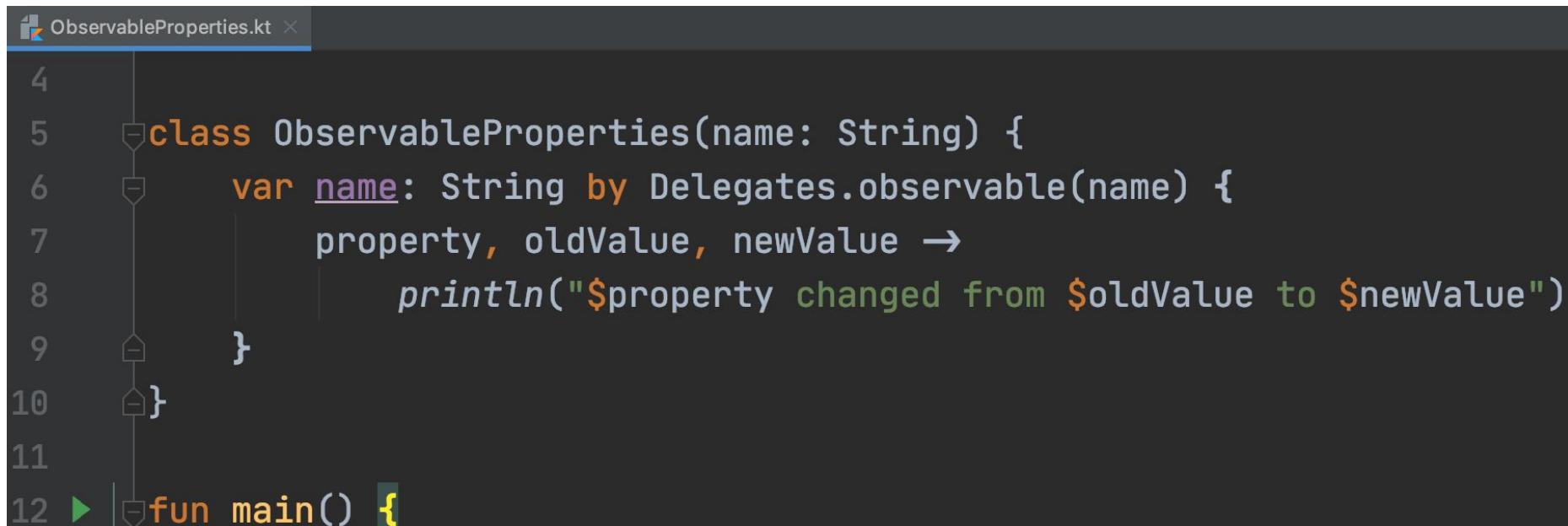
# Observable Properties

---

# Observable Properties

- Salah satu standar library yang disediakan oleh Kotlin untuk delegating di properties adalah observable properties
- Dengan observable properties, kita bisa tahu properties, value sebelum dan value setelah ketika diubah

# Kode : Observable Properties



A screenshot of a code editor showing a file named `ObservableProperties.kt`. The code defines a class `ObservableProperties` with a single observable property `name`. The `main` function is also shown.

```
1
2
3
4
5    class ObservableProperties(name: String) {
6        var name: String by Delegates.observable(name) {
7            property, oldValue, newValue →
8                println("Property changed from $oldValue to $newValue")
9        }
10    }
11
12 fun main() {
```

---

# Destructuring Declarations

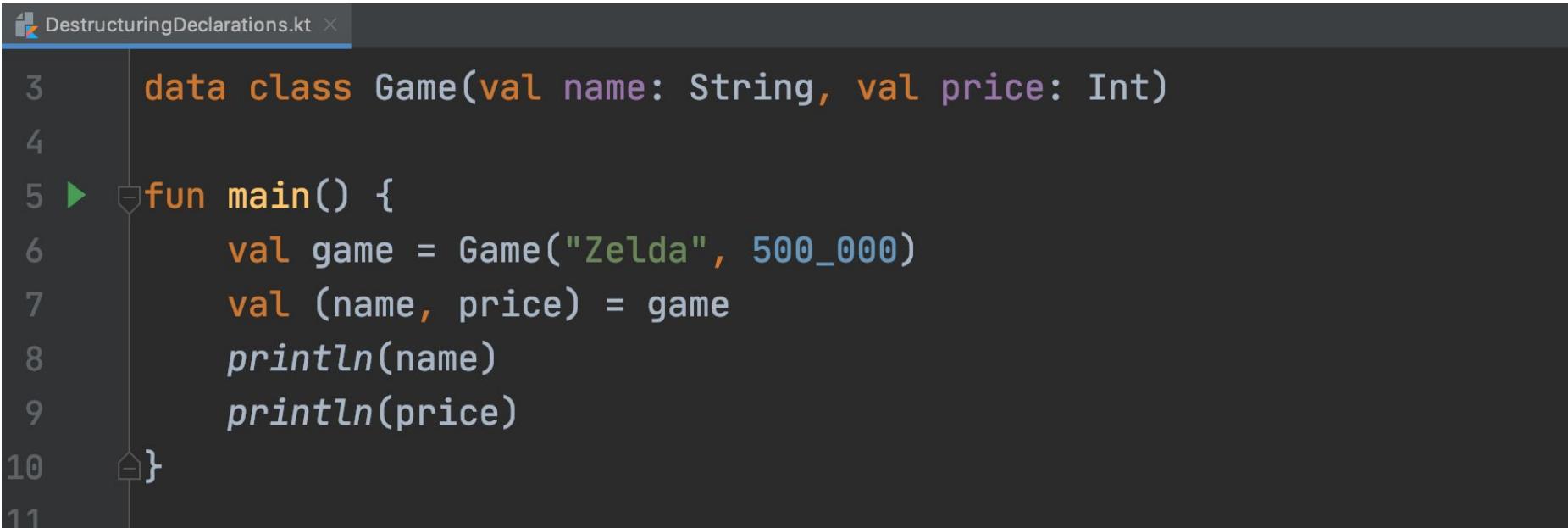
---

# Destructuring Declaration

- Destructuring declaration adalah membuat multiple variables dari sebuah object
- Destructuring tidak bisa dilakukan untuk semua object, hanya objek yang memiliki function componentX() yang bisa dilakukan destructuring
- Saat kita membuat data class, secara otomatis akan dibuatkan function componentX() sesuai dengan jumlah parameter nya



# Kode : Destructuring Declarations



A screenshot of a code editor showing a file named "DestructuringDeclarations.kt". The code demonstrates how to use destructuring declarations with a data class.

```
1  package com.example.kotlinplayground
2
3  data class Game(val name: String, val price: Int)
4
5  fun main() {
6      val game = Game("Zelda", 500_000)
7      val (name, price) = game
8      println(name)
9      println(price)
10 }
11 }
```



# Kode : ComponentX



A screenshot of a code editor showing a file named "DestructuringDeclarations.kt". The code demonstrates the use of destructuring declarations in a main function:

```
1  ━━━━━━━━━━ DestructuringDeclarations.kt ━━━━━━━━━━
2
3  fun main() {
4      val game = Game("Zelda", 500_000)
5      // val (name, price) = game
6      val name = game.component1()
7      val price = game.component2()
8
9      println(name)
10     println(price)
11 }
```

The code uses the `Game` class to demonstrate how to extract components from a tuple-like object using destructuring declarations. Lines 5 and 6 show an alternative way to declare the variables using a regular assignment followed by a comment. Lines 7 and 8 then use the `component1()` and `component2()` methods to access the individual components.

---

# Destructuring di Function

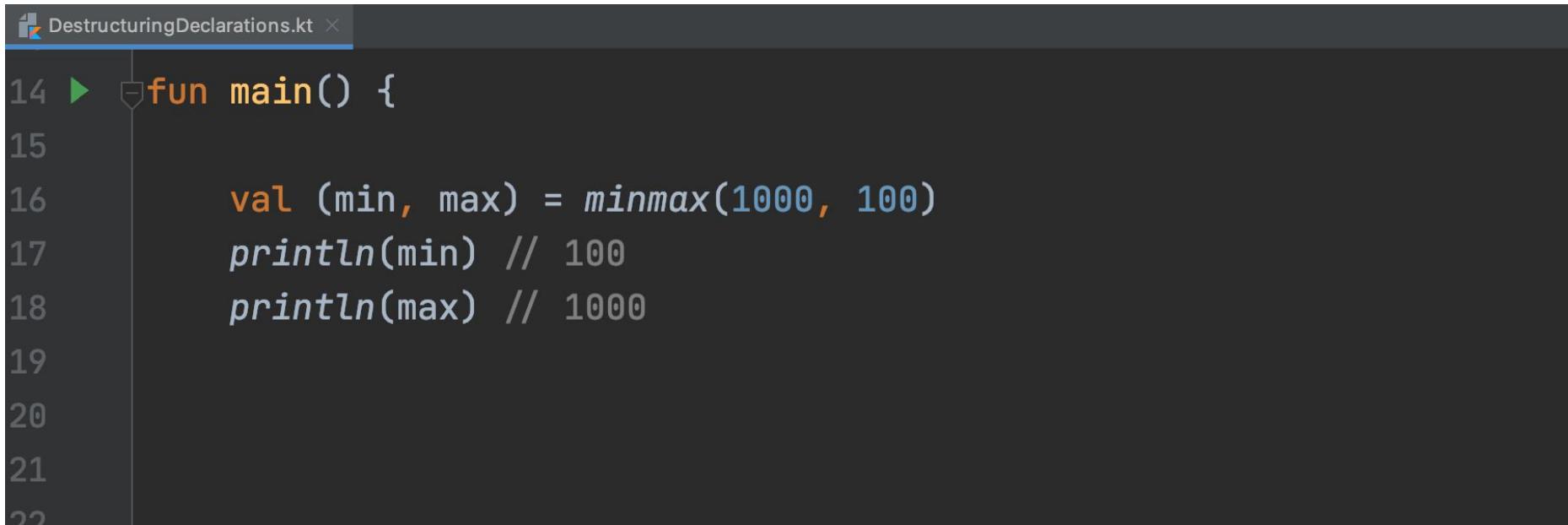
- Destructuring declarations juga bisa dilakukan ketika memanggil function
- Dengan ini, seakan-akan kita bisa mengembalikan multiple return value pada function

# Kode : Destructuring di Function

```
DestructuringDeclarations.kt ✘  
2  
3     data class MinMax(val min: Int, val max: Int)  
4  
5     fun minmax(value1: Int, value2: Int): MinMax {  
6         return when {  
7             value1 > value2 → MinMax(value2, value1)  
8             else → MinMax(value1, value2)  
9         }  
10    }
```

---

# Kode : Destructuring Return Function



The screenshot shows a code editor window with a dark theme. At the top, there's a tab labeled "DestructuringDeclarations.kt". The main area contains the following code:

```
14 > fun main() {  
15  
16     val (min, max) = minmax(1000, 100)  
17     println(min) // 100  
18     println(max) // 1000  
19  
20  
21  
22
```

The code defines a main function that calls a minmax function with arguments 1000 and 100. It then prints the minimum value (100) and the maximum value (1000) using println statements.

---

# Underscore untuk Variable Tidak Digunakan

- Kadang kita hanya ingin melakukan destructuring pada beberapa variable, dan menghiraukan variable lainnya
- Untuk menghiraukan suatu variable, kita bisa menggunakan kata kunci `_` (underscore)



# Kode : Underscore



A screenshot of a code editor showing a file named "DestructuringDeclarations.kt". The code demonstrates destructuring declarations:

```
14 > fun main() {  
15  
16     val (min, _) = minmax(1000, 100)  
17     println(min) // 100  
18  
19  
20  
21  
22
```

---

# Destructuring Lambda Parameter

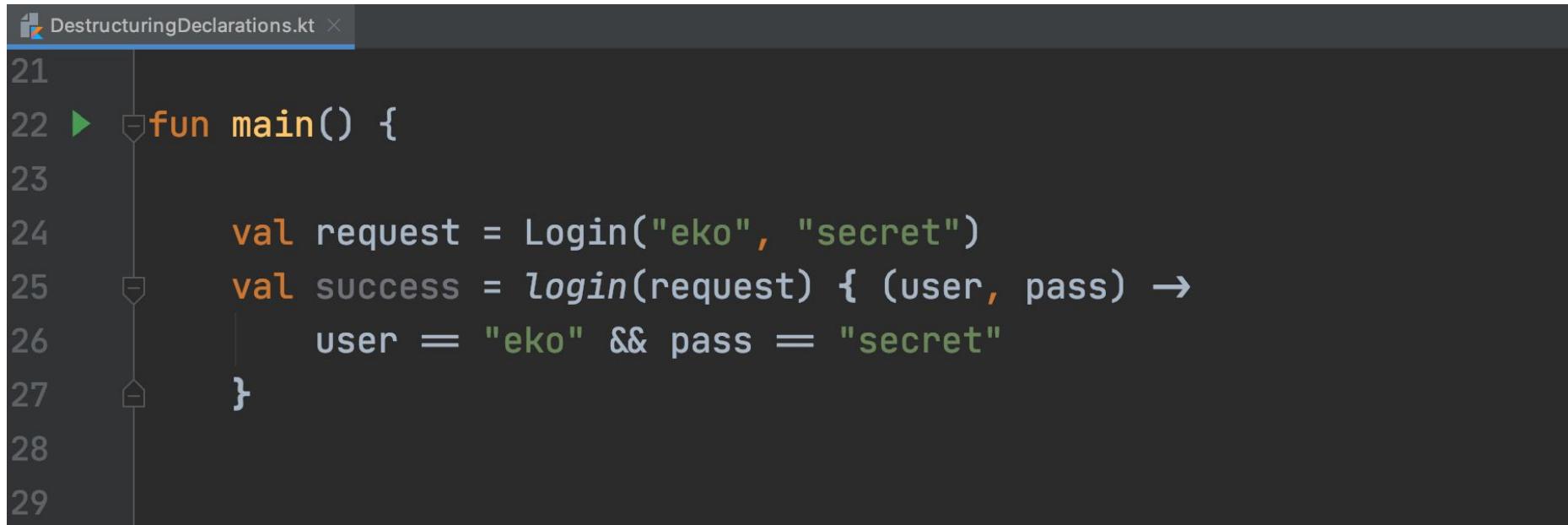
- Destructuring juga bisa dilakukan di lambda parameter
- Hal ini bisa mempermudah kita saat ingin mengakses data yang ada di parameter

# Kode : Lambda Function

```
DestructuringDeclarations.kt ✘
14
15  data class Login(val user: String, val pass: String)
16  typealias LoginCallback = (Login) → Boolean
17
18  fun login(login: Login, callback: LoginCallback): Boolean {
19      return callback(login)
20  }
21
22
```

---

# Kode : Destructuring Lambda Parameter



The screenshot shows a code editor window with a dark theme. The title bar says "DestructuringDeclarations.kt". The code is as follows:

```
1 21
2 22 > fun main() {
3
4 23
5 24     val request = Login("eko", "secret")
6 25     val success = login(request) { user, pass ) ->
7 26         user = "eko" && pass = "secret"
8 27     }
9 28
10 29 }
```

The code demonstrates the use of destructuring declarations within a lambda parameter. Line 25 shows a lambda expression where the parameters are destructured from the variable `request` (which is of type `Login`). The lambda body then uses the variables `user` and `pass` to check if they are both equal to their respective values.

---

# Operator Overloading

---

# Operator Overloading

- Kotlin mendukung operator overloading, artinya kita bisa membuat function dari operator-operator seperti operator matematika, dan lain-lain
- Kemampuan ini membuat kita bisa melakukan operasi apapun di object seperti layaknya tipe data Integer
- Ada banyak sekali operator yang bisa di override di Kotlin



# Unary Prefix Operator

Expression	Translated to
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

---

# Increment & Decrement

Expression	Translated to
------------	---------------

a++	a.inc() + see below
-----	---------------------

a--	a.dec() + see below
-----	---------------------



# Aritmatik Operator

Expression	Translated to
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b), a.mod(b) (deprecated)
a..b	a.rangeTo(b)



# In Operator

Expression	Translated to
a in b	b.contains(a)
a !in b	!b.contains(a)



# Index Acces Operator

Expression	Translated to
$a[i]$	$a.get(i)$
$a[i, j]$	$a.get(i, j)$
$a[i_1, \dots, i_n]$	$a.get(i_1, \dots, i_n)$
$a[i] = b$	$a.set(i, b)$
$a[i, j] = b$	$a.set(i, j, b)$
$a[i_1, \dots, i_n] = b$	$a.set(i_1, \dots, i_n, b)$



# Invoke Operator

Expression	Translated to
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)



# Assignment Operator

Expression	Translated to
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.remAssign(b), a.modAssign(b)</code> (deprecated)

---

## Equality dan Inequality Operator

Expression	Translated to
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))



# Comparison Operator

Expression	Translated to
$a > b$	<code>a.compareTo(b) &gt; 0</code>
$a < b$	<code>a.compareTo(b) &lt; 0</code>
$a \geq b$	<code>a.compareTo(b) \geq 0</code>
$a \leq b$	<code>a.compareTo(b) \leq 0</code>

---

# Membuat Operator Overloading

- Untuk membuat operator overloading, kita bisa menggunakan kata kunci operator sebelum deklarasi function nya

# Kode : Operator Overloading

OperatorOverloading.kt

```
3  data class Fruit(val total: Int) {  
4      operator fun plus(fruit: Fruit): Fruit {  
5          return Fruit(total + fruit.total)  
6      }  
7  }  
8  
9  ► fun main() {  
10     val fruit = Fruit(100) + Fruit(100)  
11     println(fruit)
```

---

# Null Safety

---

# Null Safety

- Jika teman-teeman sebelumnya pernah belajar bahasa pemrograman Java, di Java kita sering mengenal error bernama NullPointerException. Error ini terjadi ketika kita mengakses properties atau functions di null object
- Di Kotlin, hal ini sangat jarang terjadi, karena sejak awal di Kotlin tidak direkomendasikan untuk menggunakan nullable type.
- Walaupun akan menggunakan nullable type, di Kotlin kita memberitahu secara eksplisit menggunakan karakter ? (tanda tanya)
- Kali ini kita akan bahas cara-cara agar terhindar dari error null di Kotlin sehingga tidak sering kejadian seperti di Java

---

# Kode : Checking for Null



The screenshot shows a code editor window with a dark theme. The title bar says "NullSafety.kt". The code is as follows:

```
1 package belajar.oop
2
3 data class Friend(val name: String)
4
5 fun sayHello(friend: Friend?): Unit {
6     if (friend != null) {
7         println("Hello ${friend.name}")
8     }
9 }
```

The code defines a package named "belajar.oop" and a data class "Friend" with a single parameter "name". A function "sayHello" takes a nullable "Friend" parameter. Inside the function, an "if" statement checks if the parameter is not null, and if true, it prints a greeting using string interpolation.

---

# Safe Call Menggunakan ?

NullSafety.kt ×

```
1 package belajar.oop
2
3 data class Friend(val name: String)
4
5 fun sayHello(friend: Friend?): Unit {
6     println("Hello ${friend?.name}")
7 }
8
9 }
```



# Elvis Operator

NullSafety.kt ×

```
1 package belajar.oop
2
3 data class Friend(val name: String)
4
5 fun sayHello(friend: Friend?): Unit {
6     val name = friend?.name ?: ""
7     println("Hello $name")
8 }
9 }
```

---

# !! Operator

- Jika kita sangat mencintai NullPointerException :D
- Dan kita benar-benar yakin bahwa variabel tersebut tidak null
- Maka kita bisa menggunakan kata kunci !! untuk mengkonversi dari data yang nullable menjadi data tidak nullable
- Tapi ingat, konsekuensinya, jika sampai ternyata datanya null, maka akan terjadi error

---

# Menggunakan !! Operator

NullSafety.kt ×

```
1 package belajar.oop
2
3 data class Friend(val name: String)
4
5 fun sayHello(friend: Friend?): Unit {
6     val name = friend!!.name
7     println("Hello $name")
8 }
9
```

---

# Exception

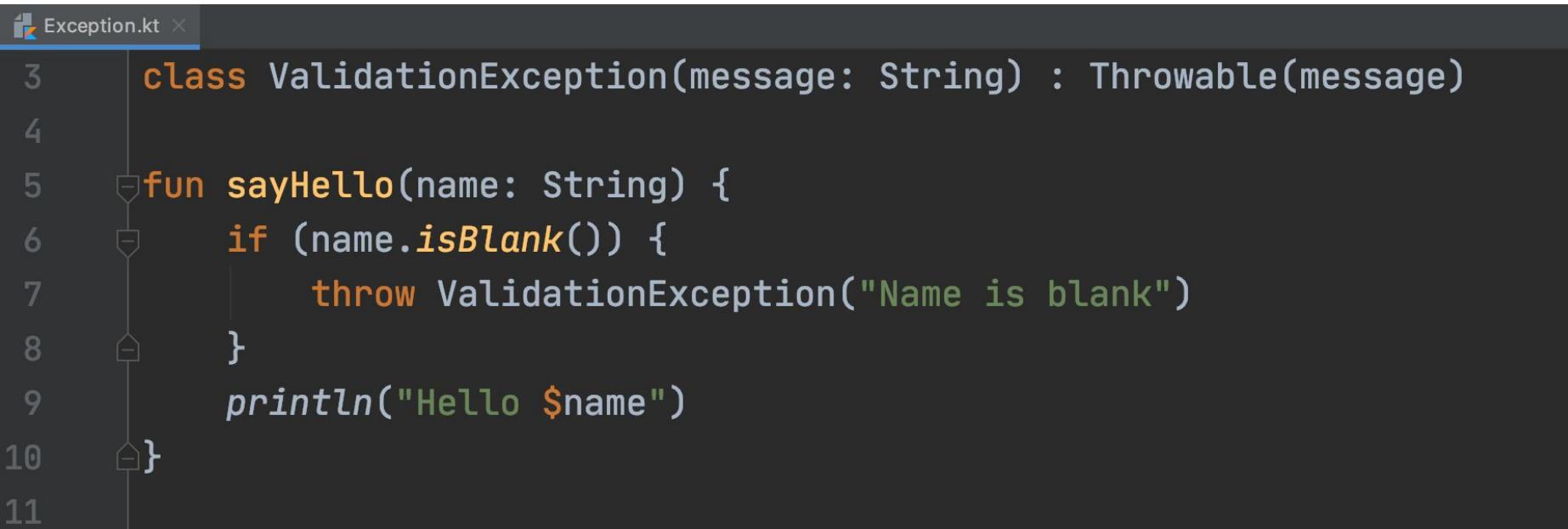
---

# Exception

- Saat membuat program, kita pasti akan selalu berhadapan dengan error
- Banyak sekali jenis error, baik itu error yang sudah diduga, seperti validation error, atau error yang tidak terduga, seperti ada jaringan error, koneksi ke database error, dan lain-lain
- Error di Kotlin direpresentasikan dengan Exception, dimana semua tipe data error pasti akan selalu class turunan dari Throwable
- Untuk membuat error di Kotlin sangatlah mudah, kita bisa menggunakan kata kunci throw, diikuti dengan object error nya

---

# Kode : Membuat Exception



The image shows a screenshot of a code editor with a dark theme. The file is named "Exception.kt". The code defines a class "ValidationException" that extends "Throwable". It contains a single function "sayHello" that checks if a name is blank and throws an exception if it is. If the name is not blank, it prints a greeting message.

```
1 class ValidationException(message: String) : Throwable(message)
2
3
4
5     fun sayHello(name: String) {
6         if (name.isBlank()) {
7             throw ValidationException("Name is blank")
8         }
9         println("Hello $name")
10    }
```

---

# Try Catch

- Jika dalam program Kotlin kita terjadi exception, maka secara otomatis program kita akan terhenti
- Kadang kita tidak ingin sampai kejadian seperti itu
- Di Kotlin, kita bisa menangkap exception, lalu melakukan sesuatu jika terjadi error
- Try Catch di Kotlin digunakan untuk mencoba melakukan sesuatu, jika terjadi error maka akan ditangkap dan kita bisa memberi reaksi sesuai dengan yang kita mau

---

# Kode : Try Catch



The image shows a screenshot of a code editor with a dark theme. A file named "Exception.kt" is open, indicated by a blue tab bar at the top. The code itself is written in Kotlin and contains a main function with a try-catch block:

```
12 fun main() {
13     try {
14         sayHello("Eko")
15         sayHello("")
16     } catch (error: ValidationException) {
17         println("Error with message ${error.message}")
18     }
19 }
```

The code editor uses color coding for syntax: orange for keywords like `fun`, `main`, `try`, `catch`, and `ValidationException`; green for strings; and purple for variable names like `error` and `message`. Line numbers are visible on the left side of the code area.

---

# Multiple Catch

- Saat kita mengeksekusi sebuah kode program di blok Try, bisa dimungkinan tidak hanya terjadi satu jenis exception, bisa saja terjadi beberapa jenis exception
- Di Kotlin, kita bisa menggunakan multiple block Catch, untuk menangkap jenis exception yang berbeda

# Kode : Multiple Catch



The screenshot shows a code editor window with a dark theme. The title bar says "Exception.kt". The code is as follows:

```
17     try {
18         sayHello("Eko")
19         sayHello("")
20     } catch (error: ValidationException) {
21         println("Error with message ${error.message}")
22     } catch (error: NullPointerException) {
23         println("Error with message ${error.message}")
24     }
25 }
```

The code consists of a try block with two catch clauses. The first catch clause handles `ValidationException` and prints the error message. The second catch clause handles `NullPointerException` and also prints the error message. Lines 17 through 25 are visible, while lines 16 and 26 are hidden by scroll bars.

---

## Finally

- Finally adalah block kode yang bisa ditambahkan di Try Catch
- Block finally akan selalu dieksekusi setelah kode program Try Catch di eksekusi, baik itu sukses ataupun gagal
- Ini cocok untuk menempatkan kode yang memang harus dilakukan tidak peduli baik itu kodenya sukses atau gagal

## Kode : Finally



The screenshot shows a code editor window with a dark theme. The title bar says "Exception.kt". The code is as follows:

```
17     try {
18         sayHello("Eko")
19         sayHello("")
20     } catch (error: ValidationException) {
21         println("Error with message ${error.message}")
22     } finally {
23         println("Finally will always be executed")
24     }
25 }
```

Annotations are present above each line of code, consisting of small gray boxes with arrows pointing to specific parts of the code. Line 17 has an annotation above the first brace. Lines 18 and 19 have annotations above both braces. Line 20 has an annotation above the first brace. Line 21 has an annotation above the first brace. Line 22 has an annotation above the first brace. Line 23 has an annotation above the first brace. Line 24 has an annotation above the first brace. Line 25 has an annotation above the first brace.

---

# Annotation

---

# Annotation

- Annotation adalah menambahkan metadata ke kode program yang kita buat
- Tidak semua orang membutuhkan Annotation, biasanya Annotation digunakan saat kita ingin membuat library / framework
- Annotation sendiri bisa diakses menggunakan Reflection, yang akan kita bahas nanti
- Untuk membuat class annotation, kita bisa menggunakan kata kunci annotation sebelum membuat class tersebut
- Annotation hanya boleh memiliki properties via primary constructor, tidak boleh memiliki members lainnya (function atau properties di body)

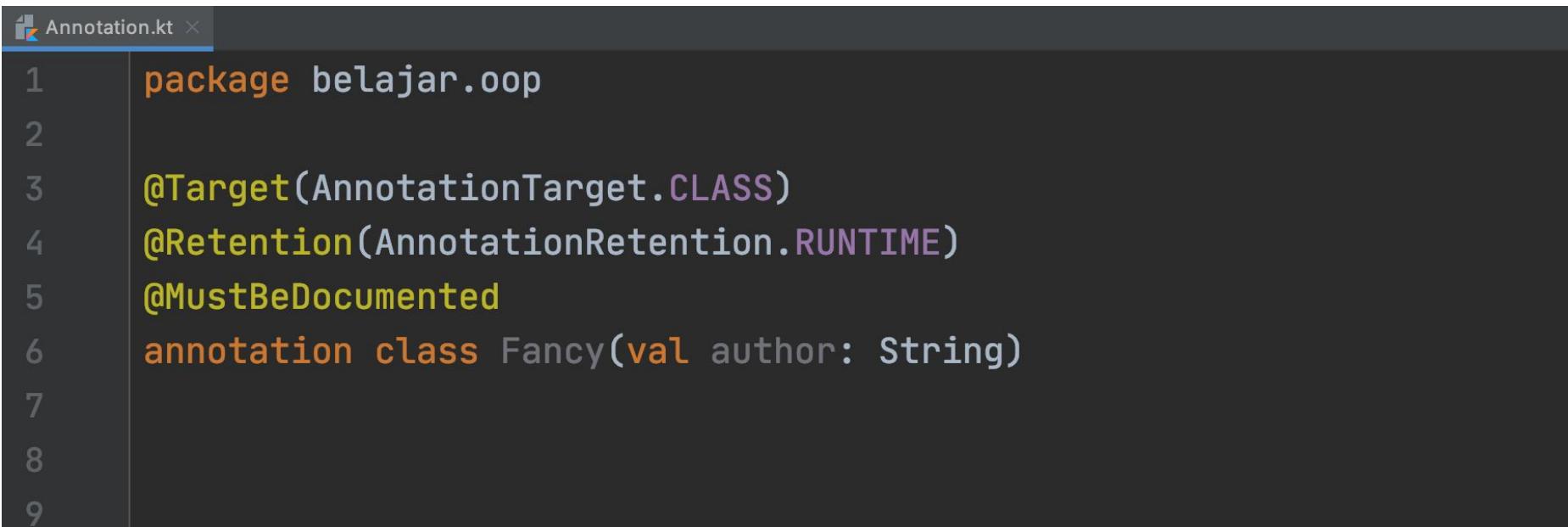
---

# Attribute Annotation

Attribute	Keterangan
@Target	Memberitahu annotation bisa digunakan di mana? Class, function, properties, dan lain-lain
@Retention	Memberitahu annotation apakah disimpan di hasil kompilasi, dan apakah bisa dibaca oleh reflection?
@Repeatable	Memberitahu annotation apakah bisa digunakan lebih dari sekali di target yang sama?
@MustBeDocumented	Memberitahu annotation apakah perlu didokumentasikan di public API

---

# Kode : Membuat Annotation



The screenshot shows a code editor window with a dark theme. The title bar says "Annotation.kt". The code is as follows:

```
1 package belajar.oop
2
3 @Target(AnnotationTarget.CLASS)
4 @Retention(AnnotationRetention.RUNTIME)
5 @MustBeDocumented
6 annotation class Fancy(val author: String)
```

---

# Kode : Menggunakan Annotation



The screenshot shows a code editor window with a dark theme. The file is named "Annotation.kt". The code contains an annotation and a class definition:

```
Annotation.kt
9
10    @Fancy(author = "Eko")
11    class MyApplication(val name: String, val version: Int) {
12
13        fun info(): String = "Application $name-$version"
14    }
15
16
17
```

The annotation "@Fancy" is highlighted in yellow, and its parameter "author" is highlighted in green. The class name "MyApplication" is highlighted in orange, and its parameters "name" and "version" are highlighted in purple. The function "info()" is highlighted in orange, and its return value is highlighted in green. The variable "\$name" and "\$version" within the function body are also highlighted in purple.

---

# Annotation Target

---

# Annotation Target

- Kotlin Annotation berjalan baik dengan Java Annotation
- Namun kadang kita ingin menempatkan posisi annotation sesuai dengan yang kita mau agar bisa terbaca di Java dengan baik
- Biasanya kejadian ini terjadi ketika kita menggunakan Kotlin, namun menggunakan framework Java, sehingga target lokasi Annotation nya harus sesuai dengan yang biasa digunakan framework tersebut di Java
- Di kotlin kita bisa menggunakan @target:NamaAnnotation nya untuk menentukan lokasi target Annotation akan ditempatkan dimana

# Kode : Annotation Target

AnnotationTarget.kt

```
3  @Target(AnnotationTarget.PROPERTY_GETTER,
4      AnnotationTarget.FIELD,
5      AnnotationTarget.VALUE_PARAMETER)
6  annotation class Beta
7
8  class ExampleTarget(@field:Beta val firstName: String,
9      @get:Beta val middleName: String,
10     @param:Beta val lastName: String)
```

---

# Reflection

---

# Reflection

- Reflection adalah kemampuan melihat struktur aplikasi kita pada saat berjalan
- Reflection biasanya sangat berguna saat kita ingin membuat library ataupun framework, sehingga bisa meng-otomatisasi pekerjaan
- Untuk mengakses reflection class dari sebuah object, kita bisa menggunakan kata kunci ::class, atau bisa langsung dari NamaClass::class

---

## Kode : Reflection Class

Reflection.kt X

```
10     val sample = Sample()
11     val clazz = sample::class
12
13     val constructors = clazz.constructors
14     val functions = clazz.memberFunctions
15     val parameters = clazz.memberProperties
16     val annotations = clazz.annotations
17 }
18 }
```

---

# Scope Functions

---

# Scope Functions

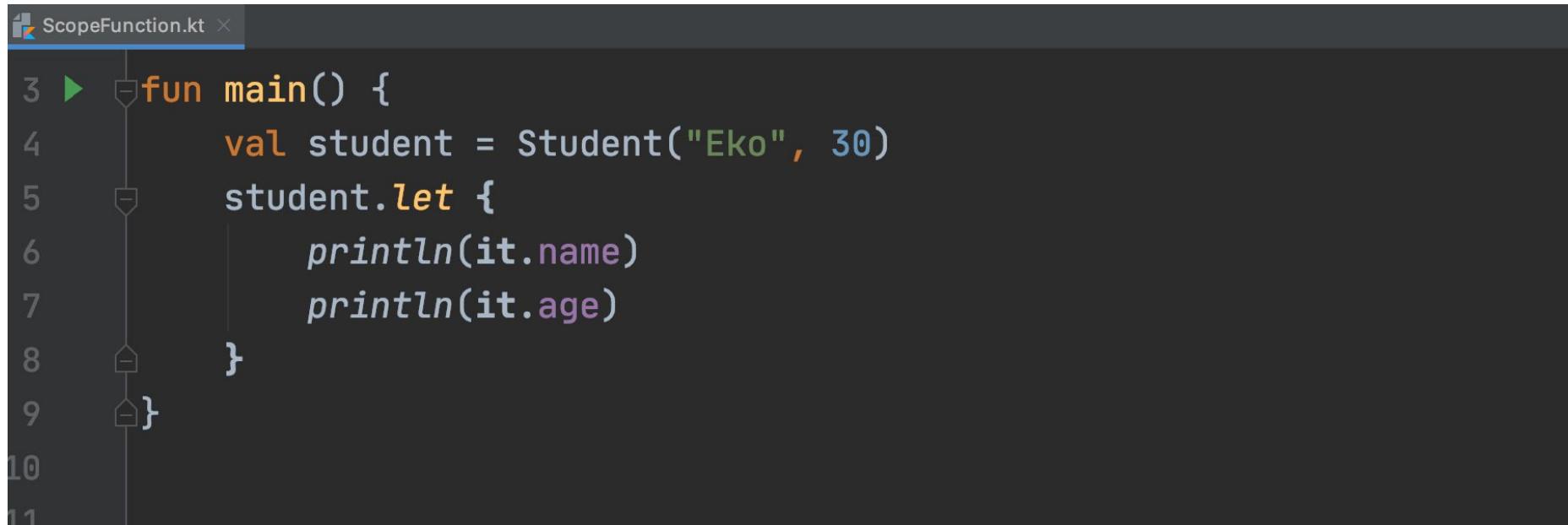
- Kotlin memiliki standar library yang berisikan beberapa function yang bisa digunakan untuk mengeksekusi object dengan mudah memanfaatkan lambda expression
- Function-function tersebut adalah let, run, with, apply, dan also

---

## Let Function

- Let function bisa digunakan sebagai reference block agar kita bisa lebih mudah ketika ingin memanipulasi sebuah object
- Let function memiliki parameter object itu sendiri, sehingga kita bisa menggunakan kata kunci it untuk mendapatkan referensi terhadap object tersebut

# Kode : Let Function



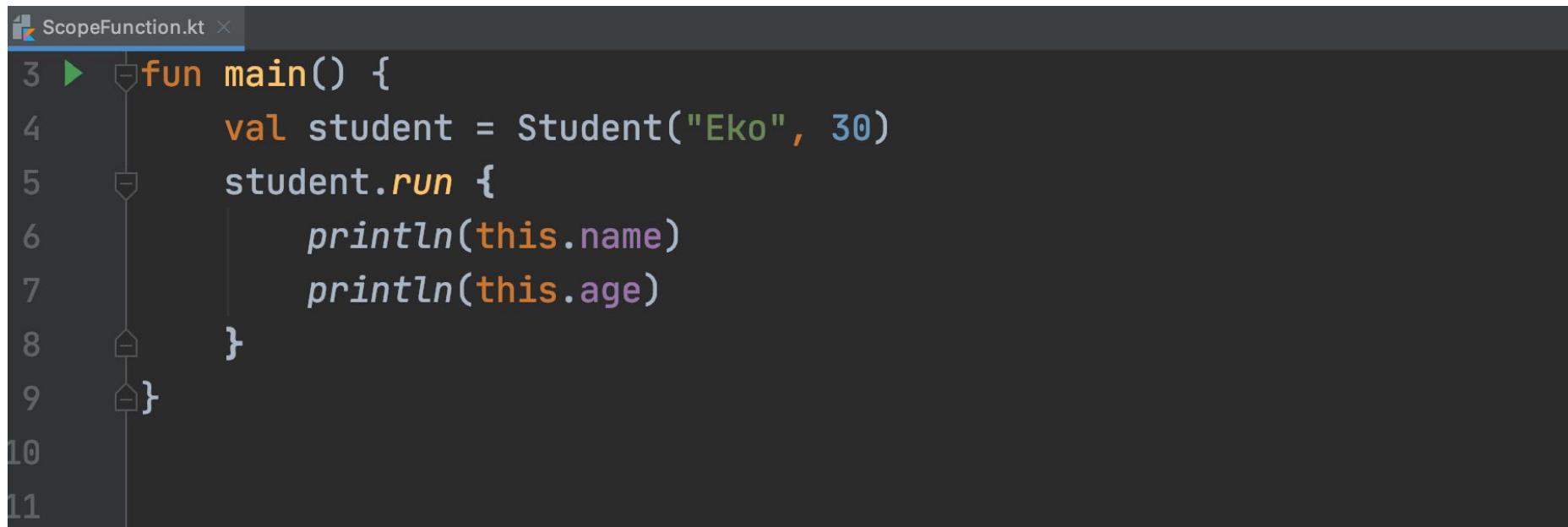
```
ScopeFunction.kt
3 ►  fun main() {
4      val student = Student("Eko", 30)
5      student.let {
6          println(it.name)
7          println(it.age)
8      }
9  }
```

---

## Run Function

- Run function mirip dengan let function
- Yang membedakan adalah, pada run function, tidak ada parameter pada lambda nya, sehingga kita tidak bisa mengakses reference object menggunakan kata kunci it. Namun kita masih bisa menggunakan kata kunci this

# Kode : Run Function



```
ScopeFunction.kt
3 ►  fun main() {
4      val student = Student("Eko", 30)
5      student.run {
6          println(this.name)
7          println(this.age)
8      }
9  }
```

---

# Apply Function

- Apply function hampir sama dengan run function
- Yang membedakan adalah return value dari apply function ada reference object itu sendiri

# Kode : Apply Function



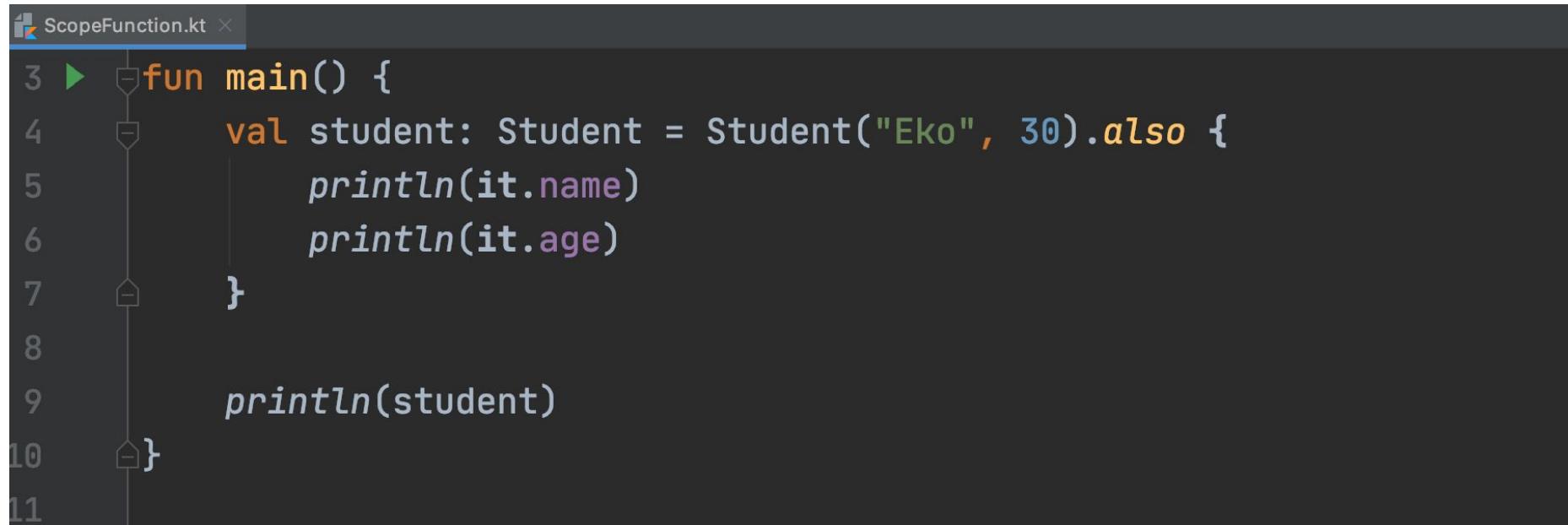
```
ScopeFunction.kt
3 ►  fun main() {
4      val student: Student = Student("Eko", 30).apply {
5          println(this.name)
6          println(this.age)
7      }
8
9      println(student)
10 }
```

---

## Also Function

- Also function mirip dengan let function
- Yang membedakan adalah, pada also function, return value nya adalah reference ke object itu sendiri

# Kode : Also Function



```
ScopeFunction.kt
3 ►  fun main() {
4      val student: Student = Student("Eko", 30).also {
5          println(it.name)
6          println(it.age)
7      }
8
9      println(student)
10 }
```

---

## With : Function

- With function mirip dengan run function, menggunakan reference this
- Yang membedakan adalah, with function bukanlah extension function, jadi tidak bisa digunakan langsung dari object-nya
- Kita harus memanggil with function secara manual

# Kode : With Function



```
ScopeFunction.kt
3 ►  fun main() {
4      val student: Student = Student("Eko", 30)
5      with(student) {
6          println(this.name)
7          println(this.age)
8      }
9  }
```



# Kesimpulan

Function	Object reference	Return value	Is extension function
let	it	Lambda result	Yes
run	this	Lambda result	Yes
run	-	Lambda result	No: called without the context object
with	this	Lambda result	No: takes the context object as an argument.
apply	this	Context object	Yes
also	it	Context object	Yes

---

# Polymorphism

---

# Polymorphism

- Polymorphism berasal dari bahasa Yunani yang berarti banyak bentuk.
- Dalam OOP, Polymorphism adalah kemampuan sebuah object berubah bentuk menjadi bentuk lain
- Polymorphism erat hubungannya dengan Inheritance

# Kode Program : Polymorphism

```
on.kt 6
7 ► fun main() {
8     var employee: Employee = Employee("Eko")
9     employee.sayHello("Budi")
10
11    employee = Manager("Eko")
12    employee.sayHello("Budi")
13
14    employee = VicePresident("Eko")
15    employee.sayHello("Budi")
16 }
```

---

# Materi Selanjutnya

---

# Materi Selanjutnya

- Kotlin Generic
- Kotlin Collection
- Kotlin Coroutine

---

# Eko Kurniawan Khannedy

- Telegram : @khannedy
- Facebook : fb.com/khannedy
- Twitter : twitter.com/khannedy
- Instagram : instagram.com/programmerzamannow
- Youtube : youtube.com/c/ProgrammerZamanNow
- Email : echo.khannedy@gmail.com

---

# Single Abstract Method (Kotlin 1.4)



# Kotlin 1.4

- <https://blog.jetbrains.com/kotlin/2020/03/kotlin-1-4-m1-released/>
- <https://blog.jetbrains.com/kotlin/2020/05/1-4-m2-standard-library/>
- <https://blog.jetbrains.com/kotlin/2020/06/kotlin-1-4-m2-released/>