

Desarrollo Android

Clase 04

Classes

Clases



```
class Person { /*...*/ }
```

```
//Si no tiene body (lo que va entre { }), se puede omitir
```

```
class Empty
```

Constructores



```
//Definiendo el constructor primario
```

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

```
//Si el constructor es público se puede omitir la keyword constructor
```

```
//Por defecto los constructores son públicos
```

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

```
//Si el constructor primario tiene anotaciones o modificadores de  
visibilidad (private, protected, etc)
```

```
//se debe usar la keyword constructor
```

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

Constructores



//Los constructores no pueden contener código. Para agregar código de inicialización se usa la keyword init:

```
class Person(firstName: String, lastName: String) {  
    var fullName = ""  
  
    init {  
        fullName = "$firstName $lastName"  
    }  
}
```

//También se puede inicializar directamente al definir la variable

```
class Person(firstName: String, lastName: String) {  
    var fullName = "$firstName $lastName"  
}
```

```
println(p.fullName) //John Doe  
println(p.firstName) //ERROR
```

Constructores



//Se pueden definir los atributos directamente en el constructor primario, incluso asignando valores por defecto

```
class Person(  
    val firstName: String,  
    val lastName: String,  
    var team: String = "Peñarol", //valor por defecto  
    var job: String?, //valor nulleable  
    var age: Int, //trailing comma (coma en el último atributo) => opcional  
)
```

Constructores

```
class Person(  
    val firstName: String,  
    val lastName: String,  
) {  
  
    val teams = mutableListOf<String>()  
  
    init {  
        println("Init: $firstName")  
    }  
  
    //Segundo constructor  
    constructor(team: String): this("Jane", "Dowy") {  
        teams.add(team)  
        println("Segundo constructor: $firstName")  
    }  
}
```

//Orden de ejecución: constructor primario, init, constructor secundario

Constructores



```
// Para crear una instancia
```

```
val person = Person("John")
```

```
//Kotlin no usa la keyword new
```


Herencia



```
//Por defecto, las clases en Kotlin son final (no heredables)  
//Para que sean heredables existe la keyword open
```

```
open class Person
```


```
class Adult: Person()
```

```
//Herencia pasando atributos
```

```
open class Person(val name: String)
```

```
class Adult(name: String, val job: String): Person(name)
```

Herencia



```
fun main() {
    val adult = Adult("John", "Ingeniero")
    val person = Adult("John", "Ingeniero") as Person


    println(adult.name) //John
    println(adult.job) //Ingeniero
    println(person.name) //John; person.job => ERROR!
}

open class Person(val name: String)

class Adult: Person {
    var job: String = ""

    constructor(name: String, currentJob: String): super(name) {
        job = currentJob
    }
}
```

Herencia



```
open class Person(val name: String) {  
    //open en los métodos que pueden sobrescribirse  
    open fun sayHi() {           para que se puede hacer override tiene que tener open  
        println("Hola, soy una persona")  
    }  
  
    //no es open => no tiene override  
    fun noOverrideFunction() { /*...*/}  
}  
  
class Adult: Person {  
    var job: String = ""  
  
    constructor(name: String, currentJob: String): super(name) {  
        job = currentJob  
    }  
  
    override fun sayHi() {  
        super.sayHi() //ejecuta primero el sayHi de person  
        println("Hola, soy un adulto")  
    }  
}
```

Herencia

```
open class Person(val name: String) {  
    open val isAdult = false //variables heredables  
  
    open fun sayHi() {  
        println("Adulto? $isAdult")  
    }  
}  
  
class Adult: Person {  
    var job: String = ""  
    override val isAdult = true //se debe utilizar override  
  
    constructor(name: String, currentJob: String): super(name) {  
        job = currentJob  
    }  
  
    override fun sayHi() {  
        println("Adulto? $isAdult")  
    }  
}
```


Clases abstractas



```
abstract class Polygon {    Es similar a la interfaz y no se coloca implementación
    abstract fun draw() //No lleva implementación en la clase abstracta sino
    en la clase que lo aplica (Rectangle en este caso)
}

class Rectangle : Polygon() {
    override fun draw() {
        //Implementación del método
    }
}
```

Interfaces



```
//Las interfaces pueden tener métodos abstractos o con implementación
interface MyInterface {
    fun bar()                ya asume si es del tipo abstracto o si es de tipo open
    fun foo() {
        //implementación
    }
}

class Child : MyInterface {
    override fun bar() {
        // implementación
    }
}
```

Interfaces

```
interface Person {
    val name: String //es abstract (debe hacerse override)
    val canBreathe: Boolean //es open (puede o no hacerse override)
    get() = false

    fun sayHi() { //es open (puede o no hacerse override)
        println("Hola, soy $name. Puedo respirar: $canBreathe")
    }

    fun goToSleep() //es abstract (debe hacerse override)
}

class Adult(
    override val name: String,
    //override val canBreathe: Boolean, //se puede implementar a nivel del
    constructor
) : Person {

    override val canBreathe: Boolean = true //se puede implementar al
    inicializar la variable

    override fun goToSleep() { println("Chau, me voy a dormir") }

    override fun sayHi() {
        super.sayHi()
        println("Soy adulto. Puedo respirar: $canBreathe")
    }
}
```

Interfaces vs abstract classes



```
interface Person { //no es posible hacer Person()
    val name: String //es abstract (debe hacerse override)
    val canBreathe: Boolean //open
        get() = false

    fun sayHi() { println("Hola, soy $name. Puedo respirar: $canBreathe") }
//open

    fun goToSleep() //es abstract (debe hacerse override)
}

abstract class Person2 { //no es posible hacer Person2()
    abstract val lastName: String //es abstract (debe hacerse override)
    val age: Int = 20 //es final => no es overrideable
        no se le puede cambiar el contenido
    abstract fun studyKotlin()

    fun howOldAmI() { println("Tengo $age") } //es final => no es
overrideable
}
```


Interfaces vs abstract classes

```
//se pueden implementar los atributos a nivel del constructor o al
//inicializar la variable (abstracts y opens)
class Adult(
    override val name: String,
    //override val canBreathe: Boolean,
    override val lastName: String,
) : Person, Person2() {

    override val canBreathe: Boolean = true

    override fun goToSleep() { //de la interface
        println("Chau, me voy a dormir")
    }

    override fun sayHi() { //de la interface
        super.sayHi()
        println("Soy adulto. Puedo respirar: $canBreathe")
    }

    override fun studyKotlin() { //de la abstract class
        println("Voy a estudiar interfaces!!")
    }

}
```

Conflictos en override

```
//Tanto interface como abstract class tienen misma open fun sayHi()
interface Person {
    fun sayHi() { println("Hola, soy una person") }
}

abstract class Person2 {
    open fun sayHi() { println("Hola, soy una person2") }
}

class Adult: Person, Person2() {
    override fun sayHi() {
        //Para identificar cuál llamar se pone super<ClassName>.funName()
        super<Person>.sayHi()      se debe añadir el tipo, para poder saber que clase o
        super<Person2>.sayHi()      interfaz es
        println("Soy adulto")
    }
}
```

Data class



//Es una clase con el propósito es contener un modelo de datos, sin funcionalidades complejas. Se definen con la keyword data

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

/*Algunas características:

- 1) El constructor primario debe tener mínimo un parámetro
- 2) Todos los parámetros del cosnstructor deben ser val o var
- 3) No pueden ser abstract, open, sealed o inner

*/

```
data class Person(val name: String, val age: Int = 20)
```

Data class

```
data class Person3(val name: String, var age: Int, val occupation: String)

val person = Person3("John", 20, "Ingeniero")

val personB = person.copy() //Crea una nueva instancia con los mismos
    atributos del copiado
val personC = person.copy()
    personC.age = 30 //copio y cambio la edad

val areEqualsWithB = person.equals(personB)
println("Son iguales? $areEqualsWithB") //Son iguales? true
val areEqualsWithC = person.equals(personC)
println("Son iguales? $areEqualsWithC") //Son iguales? false

//permite tomar datos especificos de una clase
val (name, age) = person //destructuring
println("$name - $age") //John - 20

println(person.toString()) //Person3(name=John, age=20,
    occupation=Ingeniero)
println(personC.toString()) //Person3(name=John, age=30,
    occupation=Ingeniero)

println(person.hashCode()) //1779650459
```

Data class



//Los atributos declarados en el body no se consideran para el equals, toString, hashCode y copy

```
data class Person(val name: String, val occupation: String) {  
    var age: Int = 10  
}
```

```
val person = Person("John", "Ingeniero")  
val personB = person.copy() //Crea una nueva instancia con los mismos  
atributos del copiado  
person.age = 20  
personB.age = 30
```

```
println(person.equals(personB)) //true  
println(person.toString()) //Person(name=John, occupation=Ingeniero)
```

```
val personC = person.copy(name="Jane") //Otra forma de copiar  
println(personC.toString()) //Person(name=Jane, occupation=Ingeniero)  
si se quiere copiar y cambiar solo un atributo
```

Sealed class



ejemplo respuestas de servidores

```
sealed class ServerResponse //es abstracta, no se puede instanciar
class ResponseSuccess(val message: String): ServerResponse()
class ResponseError(val error: Exception): ServerResponse()

var response: ServerResponse

response = ResponseSuccess("Éxito!")
response = ResponseError(Exception("Error :("))

//Ventaja: al usar en when, verifica que todos los que heredan de esa clase,
estén cubiertos => no es necesario el when
val msg = when(response) {
    is ResponseSuccess -> response.message
    is ResponseError -> response.error.message
}
println(msg)
```

Enum class



```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}
```

```
enum class Color(val rgb: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00),  
    BLUE(0x0000FF)  
}
```

```
//Cada elemento es un object, por lo que puedo acceder a sus propiedades  
Color.RED.desc // "rojo"
```