
GALAXY
TRAINING

ANDROID DESDE 0

Sesión 2

Parte 1

Ing. Marco Estrella
Instructor en Tecnologías Java y Android
Github @jmarkstar

Temas a tratar

Paquetes e importaciones

Igualdad

Clases abstractas

Clases Anidadas

Interfaces

Clases de tipo ***data***

Expresiones y declaraciones **object**

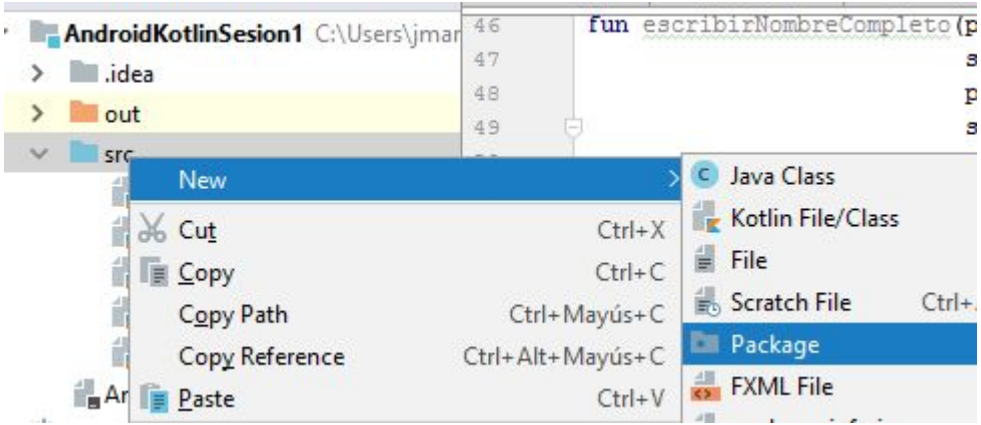
Enums

Paquetes

Usar paquetes es la forma de cómo organizar las en clases.

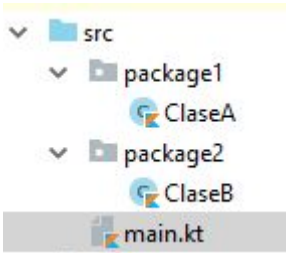
Para crear un paquete se sigue los sgtes pasos:

- Click derecho en src o en un paquete.
- Seleccionar **new**
- Click la opción **Package**.



Importaciones

Cuando creamos nuestras clases en diferentes paquetes, tenemos que usar ***import*** para importar los aquellas clases a la clase donde se quiere usar.



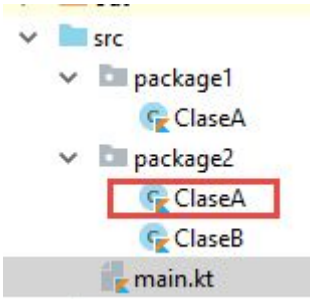

```
import package1.ClaseA
import package2.ClaseB

fun main(args: Array<String>){

    val claseA = ClaseA( property1: "abc")
    val claseB = ClaseB( propertyB: "aaa")

    claseA.showPropertyValue()
    claseB.showPropertyBValue()
}
```

Importar 2 clases con el mismo nombre




```
import package1.ClaseA
import package2.ClaseB
import package2.ClaseA as ClassA

fun main(args: Array<String>){

    val claseA = ClaseA( property1: "abc")
    val claseB = ClaseB( propertyB: "aaa")
    val claseA2 = ClassA( property: 12)

    claseA.showPropertyValue()
    claseB.showPropertyBValue()
}
```

Existen 2 tipos de igualdad.

== ó equals()

Compara el valor de 2 variables..

```
fun main(args: Array<String>){

    val string = "MyString"

    val string2 = "Mystring"

    println("its equals = ${string == string2}")
    println("its equals = ${string.equals(string2)}")
    println("its equals = ${string.equals(string, ignoreCase = true)}")
}
```

===

Compara las referencias, si 2 variables apuntan al mismo objeto en memoria, será true.

```
fun main(args: Array<String>){

    val string = "MyString"

    val string2 = string

    println("its equals = ${string === string2}")
    //true
}
```

Clases abstractas

Una clase abstracta es una clase que no puede ser instanciada. Creamos una clase para proveer una plantilla común para que otras clases puedan extender y usarlos.

Puede contener métodos y propiedades abstractos y no abstractos.

tenemos que usar el keyword ***abstract*** para declarar una clase como abstract, una propiedad o una función.

```
abstract class Vehicle(val name: String,
                       val color: String,
                       val weight: Double) {

    abstract var maxSpeed: Double

    abstract fun start()
    abstract fun stop()

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

Clases abstractas

Toda subclase que extiende de una clase abstracta tiene que implementar todas las funciones y propiedades abstractas, a

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

    override fun start() {
        println("Car Started")
    }

    override fun stop() {
        println("Car Stopped")
    }
}
```

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

    override fun start() {
        println("Bike Started")
    }

    override fun stop() {
        println("Bike Stopped")
    }
}
```

Clases anidadas

Una clase anidada es aquella que puede ser creada dentro de otra clase.

Sin usar *inner*

No puede acceder a propiedades y/o funciones de la clase anfitrión.

No es necesario crear una instancia del anfitrión para crear un objeto de esta clase anidada,

```
fun main(args: Array<String>){

    val anidada = Anfitrion.ClaseAnidada()

     anidada.funcion()
}
```

```
class Anfitrion {

    private val variable: Int = 1000

    fun funcion(){
        println("Show!!!")
    }

    class ClaseAnidada {
        fun funcion(){
            println("Nested class!!!")
        }
    }
}
```

Usando *inner*

Una clase anidada marcada con ***inner*** puede acceder a las propiedades y funciones de la clase anfitrión.

Necesita el objeto de la clase anfitrión para crear un object de la clase anidada.

```
class Anfitrion {

    private val variable: Int = 1000

    fun funcion(){
        println("Show!!!")
    }

    inner class ClaseAnidada {
        fun funcion(){
            println("Nested class!!! $variable")
            this@Anfitrion.funcion()
        }
    }
}

fun main(args: Array<String>){

    val objectAnfitrion = Anfitrion()
    val objetoClaseAnindada = objectAnfitrion.ClaseAnidada()

    objetoClaseAnindada.funcion()
}
```

Interfaces

Las interfaces pueden contener declaraciones de métodos abstractos, pero también métodos implementados. Lo que lo hace diferente a las clases abstractas es que las interfaces no pueden almacenar el estado.

Creando...

```
interface MyInterface {

    val propiedad: Int
    val propiedad2: String

    fun funcionAbstracta()

    fun funcionImplementada(){
        println("Mensaje de la funcion implementada")
    }
}
```

Implementando...

```
class MyClass(override val propiedad: Int,
              override val propiedad2: String)

    : MyInterface {

    override fun funcionAbstracta() {
        println("Mensaje de la funcion abstracta")
    }

    override fun funcionImplementada() {
        super.funcionImplementada()
        println("$this funcionImplementada()")
    }
}
```

Usando...

```
fun main(args: Array<String>){

    val myclass = MyClass( propiedad: 1,  propiedad2: "a")
    myclass.funcionImplementada()
    myclass.funcionAbstracta()
}
```

Herencia de Interfaces

Las interfaces también pueden extender de otras interfaces y, por lo tanto. proporcionar implementaciones para sus miembros y declarar nuevas funciones y propiedades abstractas.

```
interface Eatable {

    fun comer()
}

interface Drinkable {
    fun beber()
}
```

```
interface Human: Eatable, Drinkable {

    val name: String
    val age: Int

    fun nacer()

    override fun comer() {
        println("Comiendo...")
    }
}
```

```
class Person(override val name: String,
             override val age: Int)
    : Human {

    override fun beber() {

    }

    override fun nacer() {

    }
}
```

El keyword ***super*** nos permite llamar métodos de la clase padre en la herencia y de interfaces que han sido implementados.

```
interface Interface1 {

    fun funcion(){
        println("Interface1.funcion()")
    }
}

interface Intergace2{
    fun  funcion(){
        println("Interface2.funcion()")
    }
}
```

```
class MiClase: Interface1, Intergace2 {

    override fun funcion() {
        super<Interface1>.funcion()
        super<Intergace2>.funcion()
        println("MiClase.funcion()")
    }
}
```

```
fun main(args: Array<String>){

    MiClase().funcion()
}
```

Clases de tipo ***Data***

El keyword ***data*** es usado en los modelos. Lo que hace es implementar automáticamente los métodos getters, setters, equals(), hasCode(), toString() y copy().

```
data class Persona(val nombre: String, val direccion: String, val edad: Int)


fun main(args: Array<String>){

    val persona1 = Persona( nombre: "Jorge",  direccion: "av. arequipa",  edad: 20)

    println(persona1)

    val persona2 = persona1.copy(nombre = "María")

    println(persona2)
}
```

Variables y funciones de clase con ***companion object***

Cuando queremos crear propiedades o funciones estáticas debemos usar el bloque ***companion object***.

Para usar estas propiedades o métodos no es necesario crear un objeto de clase.

```
fun main(args: Array<String>){

    println(" ${MyClaseNormal.variableDeClase}")

    println(" ${MyClaseNormal.funcionDeClase( x: 1, y: 1)}")
}
```

```
class MyClaseNormal {

    var variableDeObjeto: String? = null

    init {
        variableDeObjeto = "Valor"
    }

    fun funcionDeObjeto(){
        println("funcionDeObjeto()")
    }

    companion object {

        val variableDeClase = 123456

        fun funcionDeClase(x: Int, y: Int): Int {
            return x + y
        }
    }
}
```

Clases singleton con clases de tipo ***object***

La clase singleton solo se crea una vez y permanece en la memoria ram.

El keyword object permite crear un singleton.

```
object MySingleton {

    init {
        println("Instancia creada")
    }

    fun fun1(){
        println("fun1()")
    }
}

fun main(args: Array<String>){

    MySingleton.fun1()
    MySingleton.fun1()
    MySingleton.fun1()
}
```

Creando clases anónimas con ***object***

Una clase anónima nos permite crear un objeto que implementa una interfaz en particular y poder usarlo sin tener que definir explícitamente una clase.

Object también nos permite crear clases anónimas.

```
interface OnClickListener {
    fun onClick()
}

class View {

    fun setOnClickListener(onClickListener: OnClickListener){

        //Mas codigo
        onClickListener.onClick()
    }
}

fun main(args: Array<String>){

    val myView = View()

    myView.setOnClickListener(object: OnClickListener{

        override fun onClick() {

            println("View fue clickeado!!")
        }
    })
}
```

Enums

Un Enum es una clase especial que limita la creación de objetos a los especificados explícitamente en la implementación de la clase.

Creación Básica

```
enum class Direccion {
    NORTE, SUR, ESTE, OESTE
}

fun main(args: Array<String>){

    val direccion = Direccion.NORTE

    if(direccion == Direccion.NORTE){
        println("Direccion es Norte")
    }
}
```

Con atributos

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

fun main(args: Array<String>){

    val blueRgb = Color.BLUE.rgb
}
```

Enums

Enum con métodos

```
enum class MembershipType(val price: Double) {

    SILVER( price: 100.0) {
        override fun getPriceWithDiscount(): Double {
            return price - (price/100 * 5)
        }
    }, GOLD( price: 200.0) {
        override fun getPriceWithDiscount(): Double {
            return price - (price/100 * 10)
        }
    }, PLATINUM( price: 350.0) {
        override fun getPriceWithDiscount(): Double {
            return price - (price/100 * 15)
        }
    };

    abstract fun getPriceWithDiscount(): Double
}

fun main(args: Array<String>){

    val priceWithDiscount = MembershipType.GOLD.getPriceWithDiscount()

    println(priceWithDiscount)
}
```

Enum implementando una interface

```
interface ICardLimit {
    fun getCreditLimit(): Int
}

enum class CardType : ICardLimit {
    SILVER {
        override fun getCreditLimit() = 10000
    },
    GOLD {
        override fun getCreditLimit() = 20000
    },
    PLATINUM {
        override fun getCreditLimit() = 30000
    }
}

fun main(args: Array<String>){

    val creditLimit = CardType.PLATINUM.getCreditLimit()
    println(creditLimit)
}
```

Thank you!
Questions?