ANDROID
CON KOTLING

OBJETIVOS

- Comenzar a trabajar con Android y Kotlin
- Kotlin, XML y UI Designer
- Usar Android Studio y definir la estructura del proyecto
- Comenzar a trabajar con Layouts y Material Design
- Crear vistas con CardView y ScrollView
- Ciclo de vida de Android
- Variables, operadores y expresiones con Kotlin
- Estructuras de decisión y bucles con Kotlin
- Funciones Kotlin
- Programación Orientada a Objetos
- Herencia en Kotlin
- Conectar Kotlin con el interfaz de usuarios

OBJETIVOS

- Uso de widgets en Android Studio para enriquecer el interfaz de usuario
- Ventanas de diálogo Android
- Manejar datos y generar números aleatorios
- Adaptadores y Reciclers
- Persistir y compartir datos
- Localización
- Animaciones e Interpolaciones
- Dibujar gráficos
- Diseño Avanzado del IU : Paginación, deslizamiento, navegación y fragmentación
- Bases de Datos Android

EXAMINAR EL LOG DE SALIDA

- Cuando se ejecuta la aplicación en un

COMPONENTES DE LA APLICACIÓN

- Bloques de construcción de una app Android
- Cada componente es un punto de entrada en el sistema
- Cuatro tipos:
  - Actividades
  - Servicios
  - Receptores de emisores
  - Proveedores de Contenido

ACTIVIDADES

- Punto de entrada de una interación con el usuario
- Representa una pantalla individual con un UI
- Cada Activity es independiente de las demás pero puede ser iniciada por otra.
- A diferencia de otros paradigmas de programación en el que la aplicación se inicia con un método main(), Android inicia el código en una instancia de Activity.

SERVICIOS

- Es un punto de entrada general
- Permite mantener la ejecución de la aplicación en segundo plano (reproducir música, captura de datos…)
- No proporciona una interfaz de usuario
- Se ejecuta cuando una aplicación indica que quiere usarlos

RECEPTORES DE EMISORES

- Permite al sistema entregar a la aplicación eventos que no están en el flujo del usuario.
- Permite a la aplicación responder a los emisores de anuncios de todo el sistema (por ejemplo que se apagó la pantalla, nivel de carga de la batería...)
- No tiene una interfaz de usuario pero si crea una notificación en la barra de estado

PROVEEDOR DE CONTENIDO

- Administra un conjunto de datos compartidos que pueden ser almacenados:
  en un sistema de archivos, en una base de datos SQLite, en la Web o en otros sistemas de almacenamiento

EL ARCHIVO MANIFIESTO

- AndroidManifest.xml
- La aplicación declara todos sus componentes en este archivo
- Identifica los permisos de usuarios requeridos por la app
- Declara el nivel de API mínimo que requiere la app
- Declara las características de HW y SW que la app usa o necesita
- Declara bibliotecas de la API que necesita (además de las API del marco de trabajo de Android)

RECURSOS DE LA APLICACIÓN

- En Android, una aplicación no solo está compuesta de código, además requiere recursos como: imágenes, archivos de audio, menús, estilos, colores, diseños de interfaces de activities.
- Son archivos XML
- Para cada recurso, el compilador SDK define un ID único puede ser usado para hacer referencia al mismo desde el código o desde otros recursos definidos en el XML

RECURSOS DE APLICACIÓN (2)

- Ejemplo: Si la app tiene un archivo de imagen llamado log.png en el directorio res/drawable/ SDK genera un ID de recurso llamado R.dragable.logo

- Nos permite los valores de los recursos a otros idiomas

  res/values-fr/
  res/values-eu/

DECLARAR UNA ACTIVITY

- Para que una app use activities, deben ser declaradas en el manifiesto
- En manifests/AndroidManifest.xml

```
<manifest ... >
      <application ... >
          <activity android:name=".ExampleActivit
          ...
      </application ... >
      ...
</manifest >
```

KOTLIN

COMENTARIOS

- Comentario de una línea

```
// este es un comentario
```

- Comentario de varias líneas

```
/*
 Este es un comentario
 de varias líneas
*/
```

VARIABLES - NOMBRE

- Reglas nombrado:
  - Podemos asignar cualquier nombre a una variable siempre que no sea una palabra reservada de Kotlin
  - Por convenio, el nombre de la variable empieza por minúsculas. Si esta formada por varias palabras, la primera empieza en minúsculas y las siguiente empiezan por mayúsculas
  - Ejemplos:

```
peso
mensajesNoLeidos
nombreContacto
```

VARIABLES – TIPOS

- En Kotlin todo son objetos
- Tipos básicos:
  - Numéricos: Byte, Short, Int, Long
  - Decimales: Float, Double
  - Caracteres: Char
  - Booleanos: Boolean
  - String
  - Array
  - Otras clases

DECLARAR E INICIALIZAR VARIABLES

- Variables locales de solo lectura

  se definen y asigna su valor una vez durante la programación y no cambiar durante la ejecución

```
val nombreContacto: String = “Luis Ramos”
nombreContacto = “Carlos Ross” // causa un error
```

- Asignación del tipo

```
val a: Int = 1    // asignación inmediata
val b = 2          // el tipo Int se difiere
val c: Int          //  es obligatorio indicar el tipo
cuando no se inicializa
c = 3                  // se inicializa posteriormente
```

DECLARAR E INICIALIZAR VARIABLES

- Variables locales modificables

```
var  nombreContacto: String = “Luis Ramos”
nombreContacto = “Marco Aurelio”     //ok
```

```
var x = 5  // el tipo es inferido
x += 1
```

DEFINIR EL TIPO

- En la declaración de la variable, si especificamos el tipo de dato explícitamente, este ya no puede cambiar

```
var categoria = “Finanzas”
var categoria = 2 // Error
```

PUNTO Y COMA (;)

- Las instrucciones Kotlin no necesitan terminar en ;
- Si usamos ; el compilador no protesta

```
var peso = 3.0; // OK pero no es necesario
```

TIPOS NUMÉTICOS

Type	Size (bits)	Min value	Max value
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
Long	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

- Literales numéticos y sus tipos:

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

TIPOS DECIMALES

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
Float	32	24	8	6-7
Double	64	53	11	15-16

- Literales decimales y sus tipos:

```
val pi = 3.14 // Double
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float, 2.7182817
```

PLANTILLAS STRING

- Son Piezas de código que se evalúan y sus resultados se concatenan en la cadena.
- Una expresión de plantilla comienza con un signo de dólar ($) y consta de un nombre simple
- Ejemplo

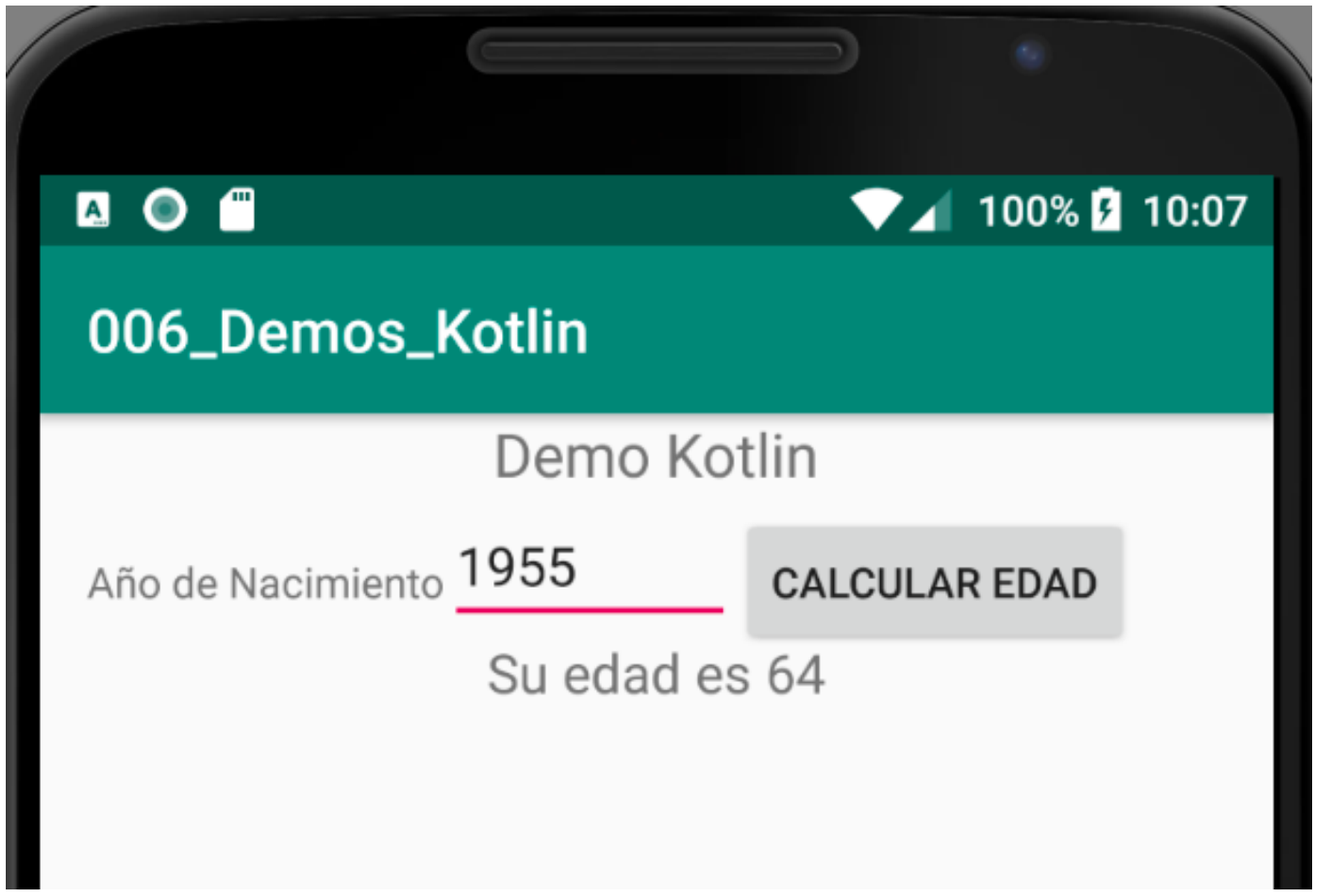
```
val i = 10
println("i = $i") // "i = 10”
val s = "abc"
println("$s.length es ${s.length}”) //"abc.length es 3"
```

OPERADORES

- Operador de Asignación. =
- Operadores aritméticos básicos +, -, /, *
- Operador de incremento ++
- Operador de decremento --

CALCULA EDAD

- Crear un nuevo proyecto usando la plantilla Empty Activity con nombre 006_Demo_Kotli
- UI



CALCULA EDAD

- Código

```
fun calcularEdad(view: View){

    val anio: EditText = findViewById(R.id.cmpAnioNac)
    val resultado: TextView = findViewById(R.id.resultado)
    val sAnio: String = anio.text.toString()

    val hoy = Date()
    val anioActual = hoy.year + 1900

    var edad = anioActual - sAnio.toInt()

    Log.println(Log.INFO, "edad", "Su edad es $edad")

    resultado.text = "Su edad es $edad".subSequence(0,  "Su edad es
$edad".length )

}
```

OPERADORES DE COMPARACIÓN

- Igualdad ==
- No igualdad !=
- Mayor >
- Menor <
- Mayor o igual >=
- Menor o igual <=

OPERADORES LÓGICOS

- AND &&
- OR ||
- NOT !

EXPRESIÓN IF

- Uso de la expresión if

```
val time = 9

val amOrPm = if(time < 12) {
   "am"
} else {
   “pm"
}

Log.i("It is ", amOrPm)
```

ESTRUCTURAS DE CONTROL – IF ELSE

- Ejemplo:

```
val time = 13

if(time < 12) {
   // Execute some important morning task here
} else {
   // Do afternoon work here
}
```

devuelve un valor de retorno (en este caso true o false), pero no hacemos nada con ello

ESTRUCTURAS DE CONTROL – IF ELSE IF

- Ejemplo:

```
var activo: Boolean
var stock: Int
var cantidad: Int

if(activo && stock > cantidad){
    // haz algo
}else if(activo && stock < cantidad) {
    // haz otra cosa
}else{
    // haz en caso de no cumplirse lo anterior
}
```

ESTRUCTURAS DE CONTROL – WHEN

- Permite tomar decisiones y ejecutar diferentes secciones de código en función de una gama de posibles resultados
- Ejemplo:

```
val rating:Int = 4
when (rating) {
  1 -> Log.i("Oh dear! Rating = ", "$rating stars")
  2 -> Log.i("Not good! Rating = ", "$rating stars")
  3 -> Log.i("Not bad! Rating = ", "$rating stars")
  4 -> Log.i("This is good! Rating = ", "$rating stars")
  5 -> Log.i("Amazing! Rating = ", "$rating stars")

  else -> {
    Log.i("Error:", "$rating is not a valid rating")
  }
}
```

CALCULAR SI ES MAYOR DE EDAD

- En el proyecto 006_Demo_Kotlin usar una estructura when de forma que evalue la variable nombre :
  - Para los valores del Atlántico, Pacífico o Ártico, se ejecuta :

```
Log.i("Encontrado:", "$nombre es un oceano”)
```

  - Para los valores de Ebro, Nilo o Amazonas, se ejecuta :

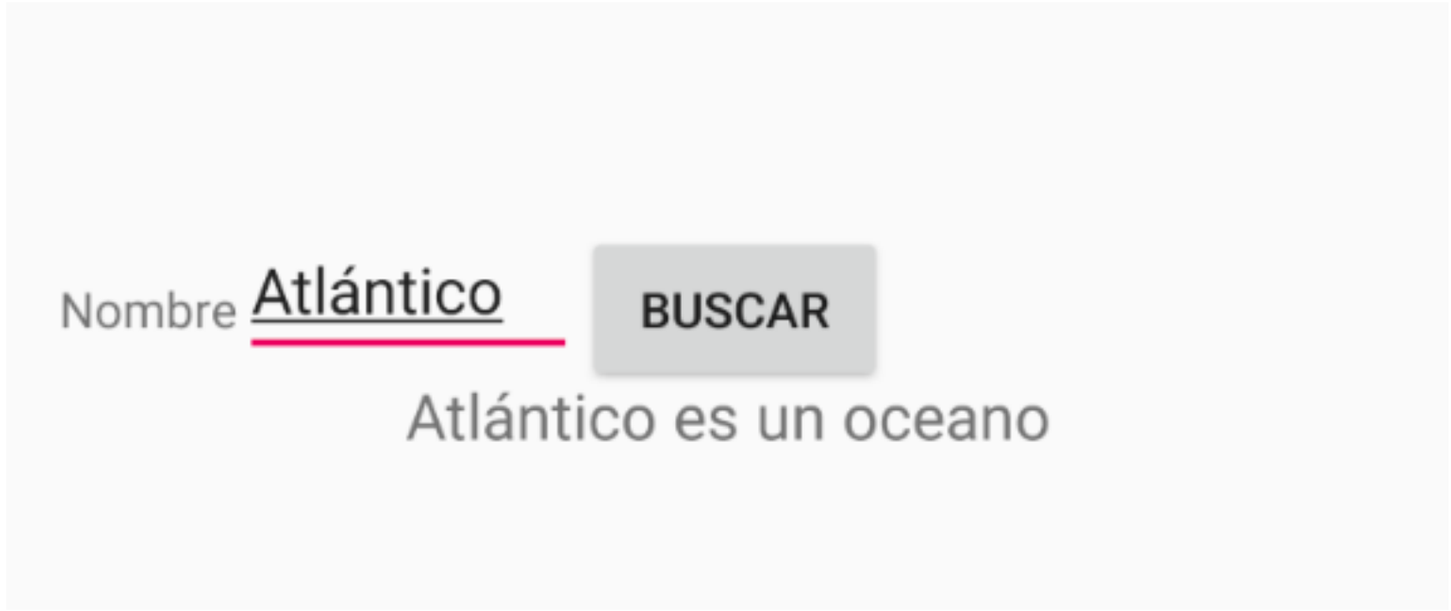
```
Log.i("Encontrado:", "$nombre es un río”)
```

  - Si ninguno coincide la aplicación se ejecuta:

```
Log.i("No encontrado:", "$nombre no está en la
base de datos”)
```

CALCULAR SI ES MAYOR DE EDAD (2)

- El interfaz



CALCULAR SI ES MAYOR DE EDAD (3)

- El código

```
fun buscarPorNombre(view: View){
    val editNombre: EditText = findViewById(R.id.cmpNombre)
    val resultado: TextView = findViewById(R.id.resultado2)
    val nombre: String = editNombre.text.toString()
    var tipo : String

    when (nombre) {
        "Atlántico","Pacífico", "Artico" -> {
            Log.i("Encontrado:", "$nombre es un oceano")
            tipo = "$nombre es un oceano"
        }
        "Ebro","Nilo", "Amazonas" -> {
            Log.i("Encontrado:", "$nombre es un río")
            tipo = "$nombre es un río"
        }
        else -> {
            Log.i("No Encontrado:", "$nombre no está en la base de datos")
            tipo = "$nombre no está en la base de datos"
        }
    }
    resultado.text = tipo.subSequence(0,  tipo.length )
}
```

ESTRUCTURAS DE REPETICIÓN – WHILE

- Ejemplo:

```
var x = 10
while(x > 0) {
  Log.i("x=", "$x")
  x--
}
```

ESTRUCTURAS DE REPETICIÓN - DO WHILE

- Ejemplo:

```
var y = 10
do {
  y++
  Log.i("In the do block and y=","$y")
}
while(y < 10)
```

RANGOS

- Un rango define un intervalo cerrado en el sentido matemático
- Ejemplos:
  - Comprobar si un número está dentro de un rango:

```
val x = 10
val y = 9
if (x in 1..y+1) {
    Log.println(“Está en el rango")
}
```

  - Comprobar si un número no esta en un rango

```
val list = listOf("a", "b", "c")
if (-1 !in 0..list.lastIndex) {
    println("-1 está fuera del rango”)
}
//list.size
//list.indices
```

RANGOS

- Ejemplos:
  - Iterar por un rango

```
for (x in 1..5) {
    print(x)
}
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

ESTRUCTURAS DE REPETICIÓN - FOR

- Ejemplo:

```
val lista = 1..10
for (i in lista)
    Log.i(“Iterando por la lista”,"Valor es $i")
```

BREAK

- Ejemplo:

```
var countDown = 10
while(countDown > 0){
    if(countDown == 5)break

    Log.i("countDown =","$countDown")
    countDown --
}
```

CONTINUE

- Ejemplo:

```
var countUp = 0
while(countUp < 10){
  countUp++

  if(countUp > 5)continue

  Log.i("Inside loop","countUp = $countUp")
}
Log.i("Outside loop","countUp = $countUp")
```

BREAK Y CONTINUE CON ETIQUETAS

- Cualquier expresión en Kotlin puede marcarse con una etiqueta.
- Tienen la forma de identificador seguido del signo @, por ejemplo: abc@, fooBar@
- Para etiquetar una expresión, solo ponemos una etiqueta delante de ella

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

FUNCIONES

- Declaración básica

```
fun diHola(nombre: String){
  Log.i(“Message=","Hola $nombre")
 }
fun printAreaCircle(radius: Float){
  Log.i("Area =","${3.14 * (radius * radius)}")
}


 diHola(“Laura”)
 printAreaCirculo(23.4f)
```

RETURN

- Se puede declarar que las funciones tienen un tipo de retorno.

```
fun getSum(a: Int, b: Int): Int {
  return a + b
}

getSum(10, 10)
val answer = getSum(10, 10)

// imprimir el valor de retorno
Log.i("Returned value =","${getSum(10, 10)}")
```

FUNCIONES CON UNA SOLA INSTRUCCIÓN

- Cuando tiene una función con un cuerpo muy simple que contiene una sola expresión, Kotlin nos permite acortar el código mediante el uso de una sintaxis de expresión única

```
fun getSum(a: Int, b: Int) = a + b
```

PARAMETROS DE LA FUNCION

- Parámetro predeterminado: los programadores dan un valor (predeterminado) para el parámetro que se utilizará si en la llama a la función no lo proporciona.

```
fun hacerPedido(costesEnvio: Boolean = false,
                producto: String,
                tipoEnvio: String = "Estandar") {
    //el código
}
```

- Argumento con nombre:el código que llama a una función especifica un nombre junto con un valor. Tenga en cuenta que proporcionar un valor es opcional.

PARAMETROS DE LA FUNCION

- Argumento con nombre:el código que llama a una función especifica un nombre junto con un valor. Tenga en cuenta que proporcionar un valor es opcional.

```
hacerPedido(producto = "Libro")
hacerPedido(true, producto = “Libro")
hacerPedido(true, “Libro", “1 día")
hacerPedido(producto = "Libro", tipoEnvio = "Urgente")
```

ARGUMENTOS VARIABLES

- Un parámetro de una función puede marcarse con el modificador vararg
- Solo un parámetro puede marcarse como vararg.
- Si un parámetro vararg no es el último en la lista, los valores para los siguientes parámetros pueden pasarse usando el argumento nombrado

```
fun sumar(vararg numeros: Int): Int {
    val result = 0
    for (n in numeros) //numeros es un array
        result += n
    return result
}
```

ARRAYS

- La clase Array, que tiene funciones get y set (que se convierten en []) y propiedades size
- Declarar un Array
  - Función arrayOf ()
    arrayOf (1, 2, 3) crea un array [1, 2, 3].
  - Función arrayOfNulls ()
    crear una matriz de un tamaño determinado lleno de elementos nulos.

ARRAYS - EJEMPLOS

- Crea un array de String con los valores ["0", "1", "4", "9", "16"]

```
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { println(it) }
```

- Crear un array de Int con los valores 1,2,3

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

- Crea un Array de int de tamaño 5 con valores[0, 0, 0, 0, 0]

```
val arr = IntArray(5)
```

ARRAYS – EJEMPLOS (2)

- Inicializa los valores del array con un 42

```
val arr = IntArray(5) { 42 }
```

- Inicializa los valores del array usando lambda

```
// Array of int of size 5 with values [0, 1, 2, 3, 4]
(values initialised to their index value)
var arr = IntArray(5) { it * 1 }
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- CLASE
  - La clase es el plano, y hacemos objetos basados en el plano.
  - Podemos usarla para hacer tantos objetos como queramos.
  - Esta formada por funciones, variables, bucles y toda la sintaxis de Kotlin que ya hemos aprendido.
  - Se encuentra en un paquete Kotlin, y la mayoría de los paquetes tendrán múltiples clases.
  - Normalmente, cada nueva clase se definirá en su propio archivo de código .kt con el mismo nombre que la clase

DECLARAR UNA CLASE - VARIABLES (ATRIBUTOS)

```
class Empleado {
    // Variables
    var nombre = "Sin nombre"
    var departamento = "Sistemas"
    var activo = true
}
```

DECLARAR UNA CLASE - FUNCIONES

```
class Empleado {
    // Variables
    var nombre = "Sin nombre"
    var departamento = "Sistemas"
    var activo = true
    // Class function
    fun getEstado(): String {
        var status = "$departamento $nombre"
        if(activo){
            status = "$status no activo!"
        }else{
            status = "$status activo."
        }
       return status
    }
}
```

USO DE LA CLASE

```
val e = Empleado()
Log.i("Nombre =","${e.nombre}")
Log.i("Dpto =","${e.departamento}")
 Log.i("Activo =","${e.activo}")

//cambio valores
 e.activo = false;
 Log.i("Activo =","${e.activo}")

//Llamada a una funcion

 val e2 = Empleado()
 val status = e2.getEstado()
 Log.i("Status",status)
```

CLASES – PROPIEDADES (GETTERS Y SETTERS)

```
class Empleado {

    var salario = 100
        get() {
            Log.i("Getter being used","Value = $field")
            return field
        }
        set(value) {
            field = if (value < 0) 0 else value
            Log.i("Setter being used","New value = $field")
        }

}
```

MODIFICADORES DE VISIBILIDAD

- Public - visible en todas partes
  Si no especifica ningún modificador de visibilidad, public se usa por defecto
- Private , solo es visible dentro de la clase que la declara
- Internal es visible en clases en el mismo paquete
- Protected solo es visible dentro de la clase que la declara y en clases hija por herencia

CONSTRUCTORES

- Cuando declaramos una clase, Kotlin proporciona una función especial llamada constructor que prepara la instancia.

- Ejemplo

```
val empleado = Empleado()
```

CONSTRUCTOR PRIMARIO

- Se declara con la declaración de clase
- Ejemplo

```
  class Book(val title: String, var copiesSold: Int) {
   //Aquí ponemos nuestro código como normal
   // Pero title y copiesSold son propiedades que
   // ya están declarados e inicializados
   }

// Instanciamos un  Book usando primary constructor
  val book = Book(“La Cabaña", 20000000)
```

CONSTRUCTORES SECUNDARIOS

- Podemos declarar mas de un constructor
- Ejemplo

```
class Reunion(val dia: String, val empleado: Empleado) {
    var hora: String = "Pendiente de decidir"

    constructor(dia: String, empleado: Empleado, hora: String)
            :this(dia, empleado){

        // "this" refers to the current instance
        this.hora = hora
        // hora (the property) now equals hora
        // that was passed in as a parameter
    }
}
```

USO DE LOS CONSTRUCTORES

```
val e = Empleado()

var libro = Book("La Cabaña", 30000);
val reunion = Reunion("Lunes", e)
val otraReunion = Reunion("Lunes", e, "18:00")

Log.i("reunion " , “$reunion")
```

BLOQUES DE INICIALIZACIÓN

- Se ejecuta siempre que se construye un objeto de la clases
- Independiente del constructor que se elija

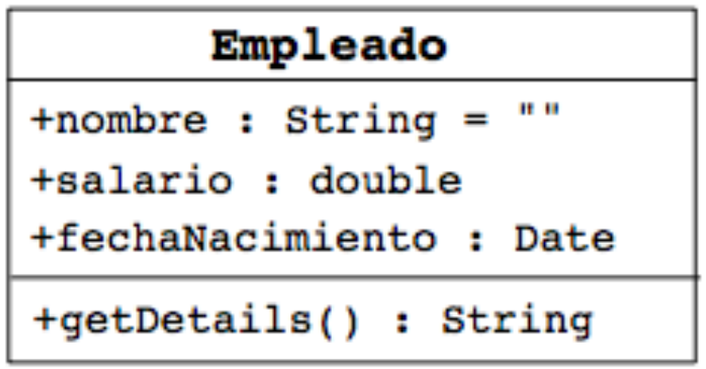
```
init{
   // This code runs when the class is instantiated
   // and can be used to initialize properties
}
```

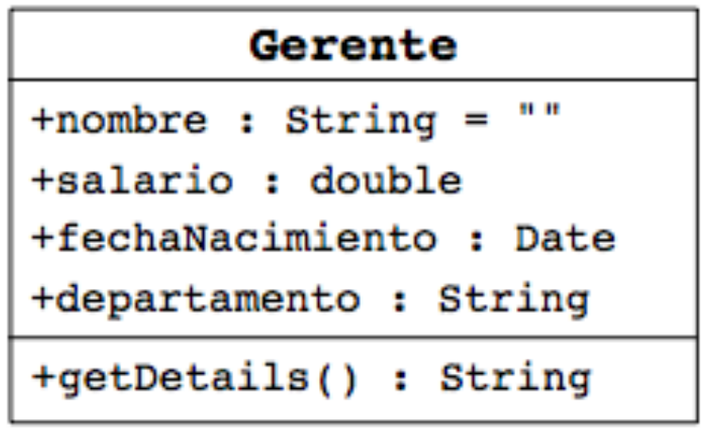
HERENCIA IMPLÍCITA

- Todas las clases Kotlin tienen una superclase común Any
- Any tiene tres métodos: equals(), hashCode() y toString()
- Ejemplo:

```
class Empleado // Implícitamente hereda de Any
```

HERENCIA

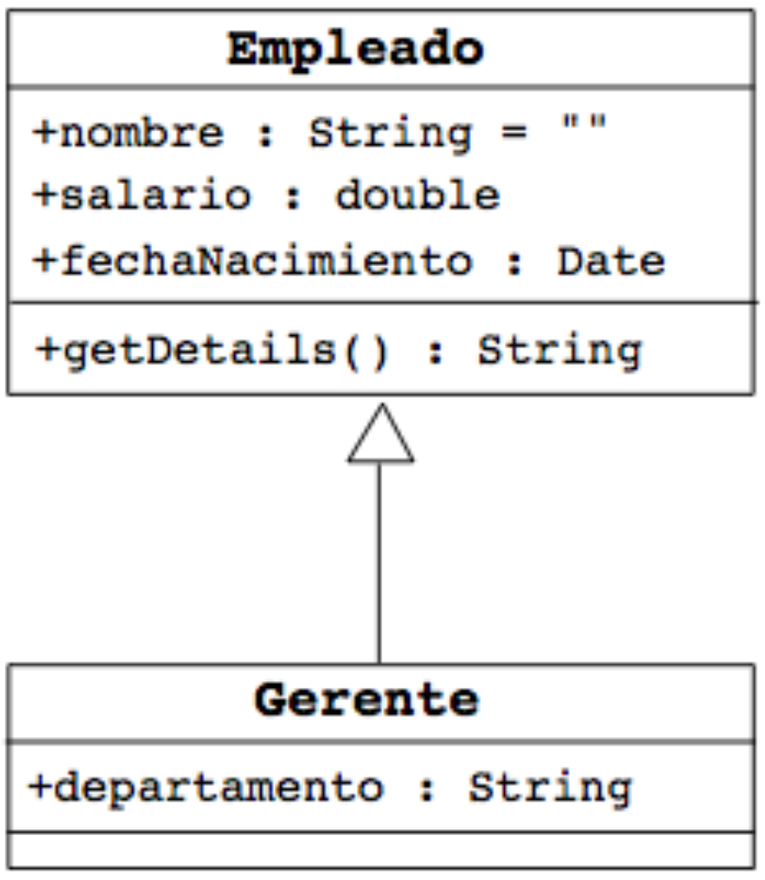





```
class Empleado {

    var  nombre : String = ""
    var  salario: Double = 0.0
    var  fechaNacimiento : Date? = null

    fun  getDetails():String  {
      return "Empleado $nombre salario $salario "
    }

}
```

```
class Gerente {

    var  nombre : String = ""
    var  salario: Double = 0.0
    var  fechaNacimiento : Date? = null
    var departamento: String = ""

    fun  getDetails():String  {
        return "Gerente $nombre salario
          $salario dpto $departamento"
    }
}
```

HERENCIA (2)




```
Open class Empleado {

    var  nombre : String = ""
    var  salario: Double = 0.0
    var  fechaNacimiento : Date? = null

    fun  getDetails():String  {
      return "Empleado $nombre salario $salario "
    }

}
```

```
class Gerente: Empleado() {

        var departamento: String = ""

}
```

HERENCIA BÁSICA

- Por defecto no podemos heredar de una clase (final class)
- Para permitirla herencia debemos declarar a la clase como clase padres explícitamente:
- Ejemplo:

```
open  class Empleado{}
class Gerente : Empleado(){}
```

SOBREESCRIBIR UN MÉTODO

- Podemos cambiarla implementación de un método heredado
- Ejemplo:

```
open  class Empleado{
    …
     open fun trabajar() {
         Log.i("empleado" , "Trabajo en el departamento
     $departamento")
     }
}

class Gerente : Empleado(){
   var zona: String= “Central”

   override fun trabajar() {
     //super.trabajar()
     Log.i("empleado", "Dirijo el departamento $departamento
             de la zona  $zona")
   }
}
```

POLIMORFISMO

```
var e : Empleado = Gerente();


// Intento no permitido de asignar un
 // atributo de Gerente
e.departamento ="Logística"
```

LLAMADA A MÉTODOS VIRTUALES

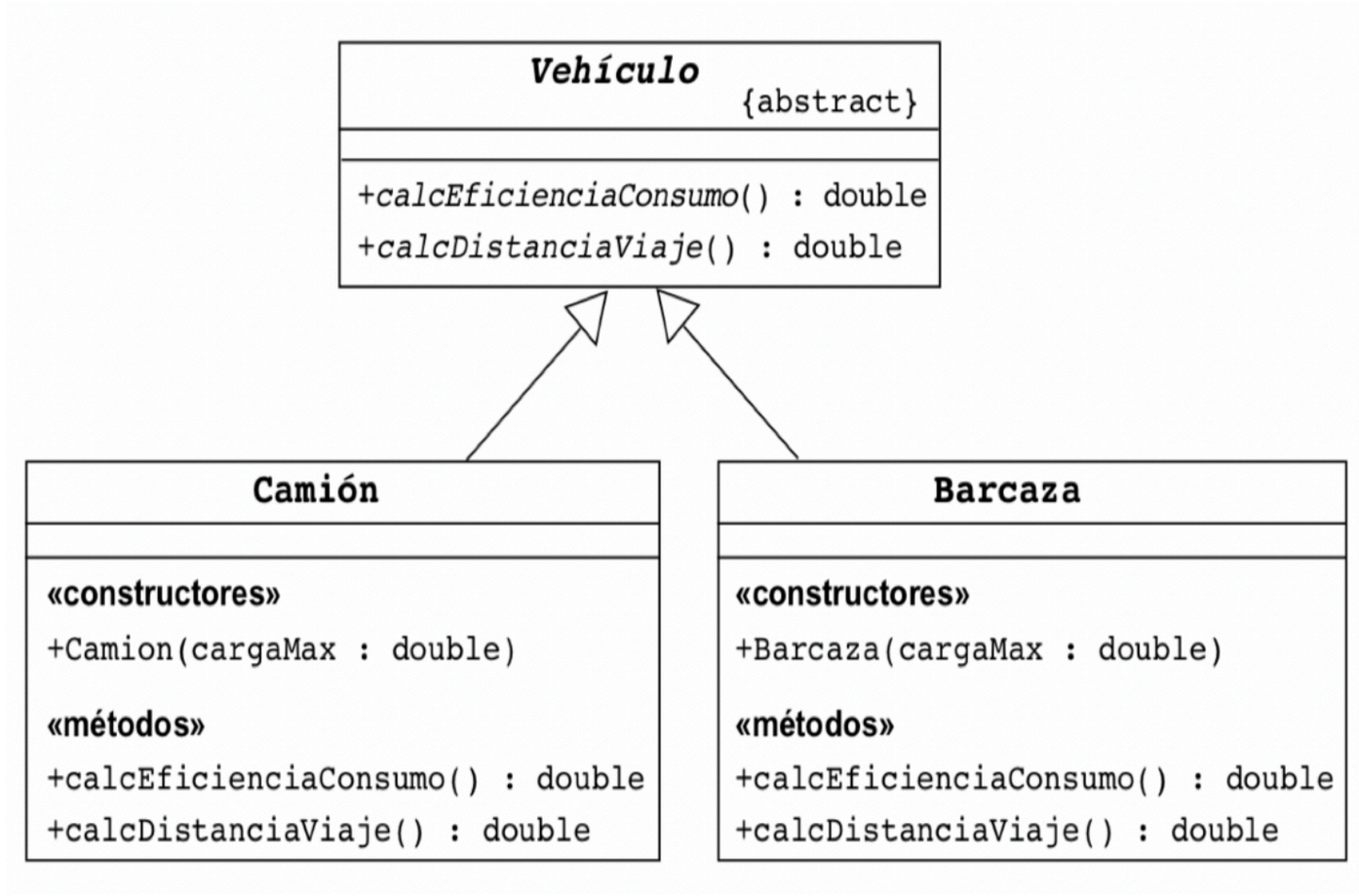
```
var empleado : Empleado = Empleado();
var gerente: Gerente = Gerente();
empleado.getDetails()
gerente.getDetails()

//POLIMORFISMO

var e : Empleado = Gerente();

e.getDetails()
```

COLECCIONES HETEROGÉNEAS

```
val lista : Array<Empleado?> = arrayOfNulls(4)
lista[0] = Empleado()
lista[1] = Empleado()
lista[2] = Gerente();
lista[3] = Gerente()
```

```
lista.forEach {

    if (it!= null){
        Log.i("empleado","${it.nombre} ")
        it.getDetails()
    }

}
```

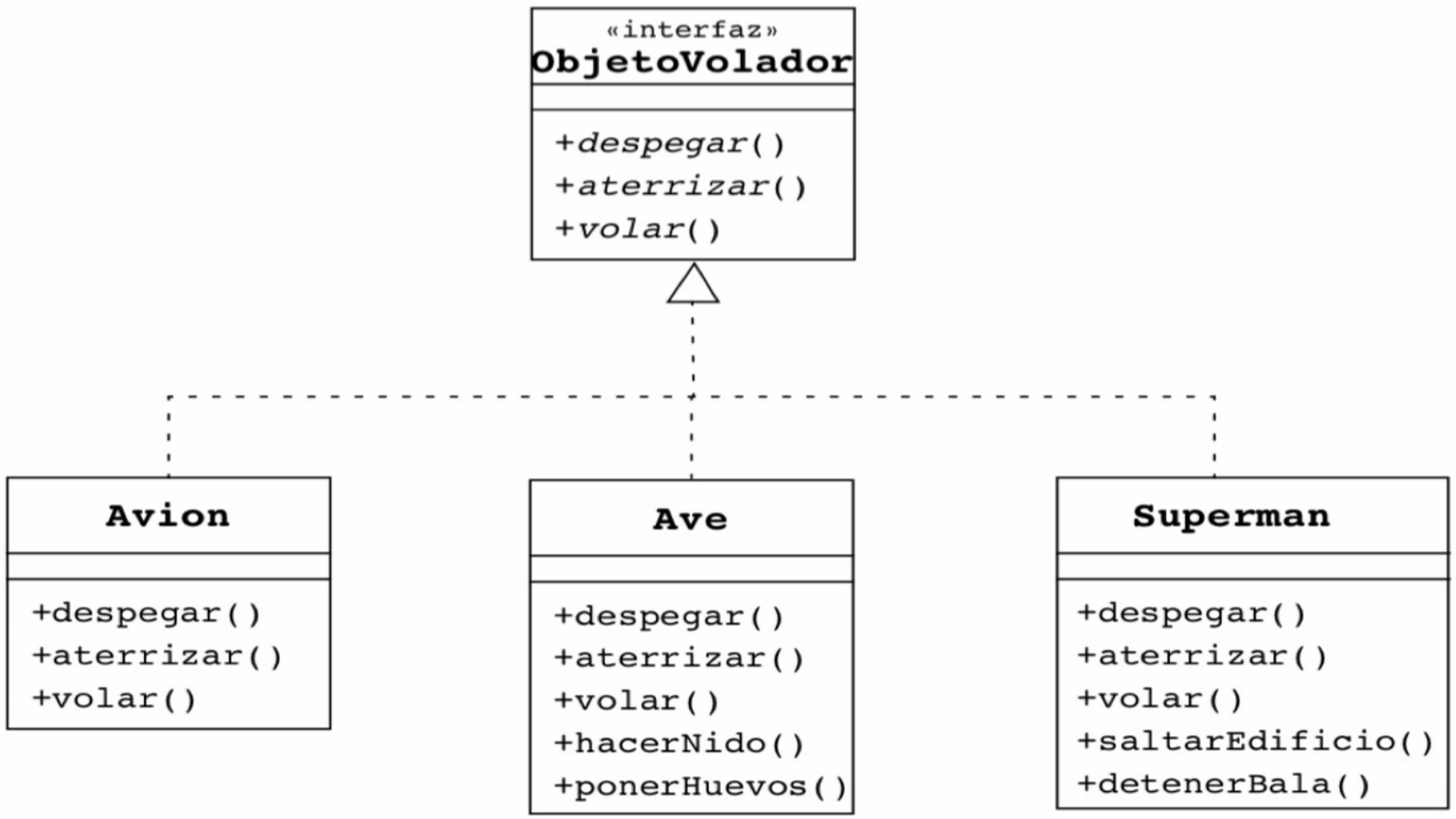
CLASES ABSTRACTAS




```kotlin
abstract class Vehiculo{

    var cargaMax :Double = 0.0

    constructor(cargaMax: Double) {
        this.cargaMax = cargaMax
    }

    abstract fun calcEficienciaConsumo():Double
    abstract fun calcDistanciaViaje(): Double

}
```

```kotlin
class Camion:Vehiculo{

    constructor(cargaMax: Double) : super(cargaMax)

    override fun calcEficienciaConsumo(): Double {
        return cargaMax * 0.2;
    }

    override fun calcDistanciaViaje(): Double {
        return 30.0;
    }
}
```

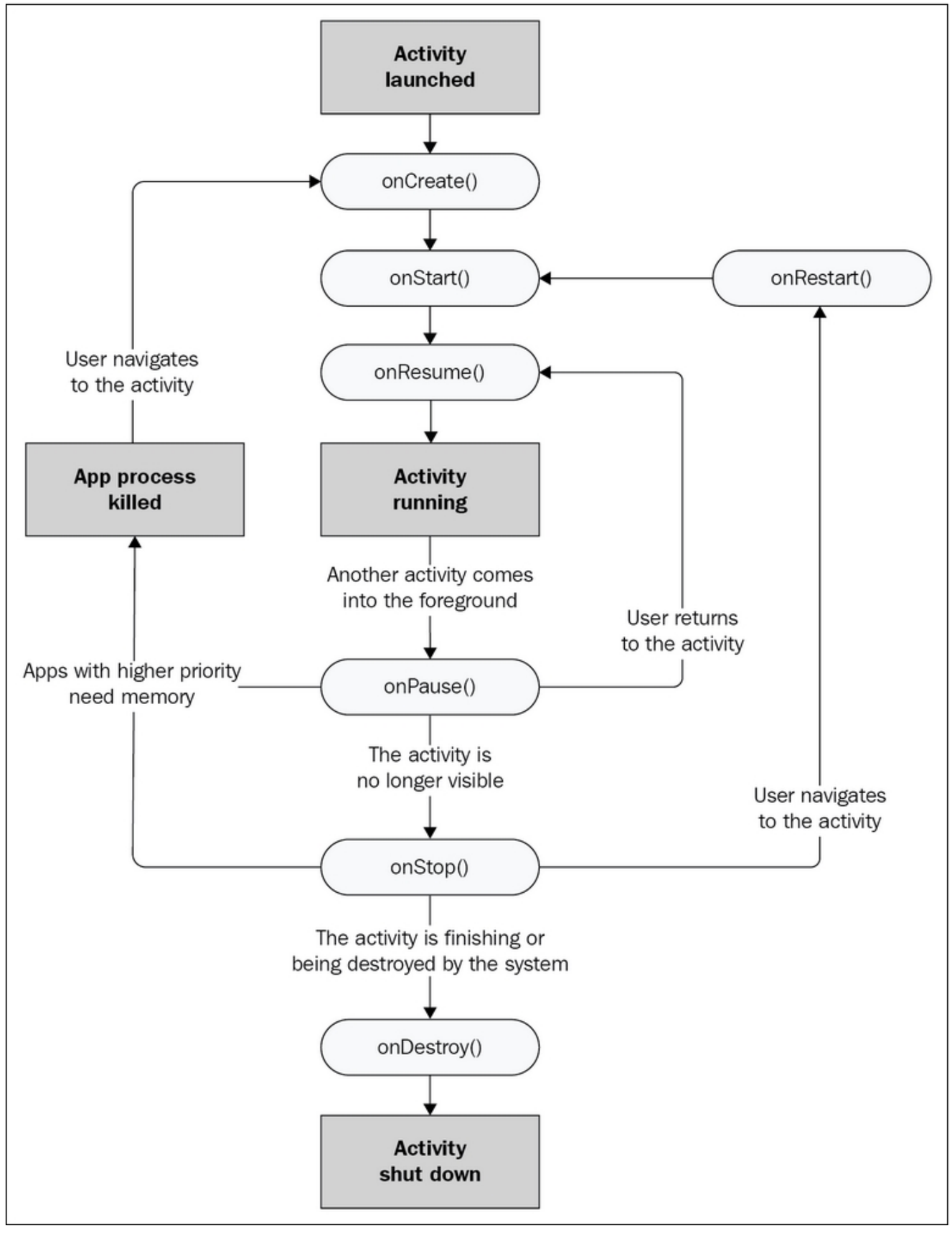
INTERFACES




```
interface ObjetoVolador {

    fun despegar()
    fun aterrizar()
    fun volar()
}
```

```
class Avion: Vehiculo, ObjetoVolador {

    …
     override fun despegar() {
       Log.i("avion","despego en el aeropuerto”)
     }
     override fun aterrizar() {
         Log.i("avion","aterrizo en una pista”)
      }
     override fun volar() {
         Log.i("avion","vuelo a propulsión”)
      }
}
```

CICLO DE VIDA DE ANDROID



DIALOGFRAGMENT

- Ventana de diálogo emergente
- Permite
  - Mostrar al usuario información
  - Pedir confirmación de una acción
- Tiene características más avanzadas que un simple widget

DIALOGFRAGMENT

- Crear una clase MiDialog.kt

```
import android.app.Dialog
import android.os.Bundle
import androidx.appcompat.app.AlertDialog
import androidx.fragment.app.DialogFragment

class MiDialog : DialogFragment() {

}
```

DIALOGFRAGMENT

- Sobreescribir el método

```
    override
    fun onCreateDialog(savedInstanceState: Bundle?): Dialog {

        val builder = AlertDialog.Builder(this.activity!!)

        // MÁS CÓDIGO
    }
```

DIALOGFRAGMENT

- Definir Texto. Y Botons

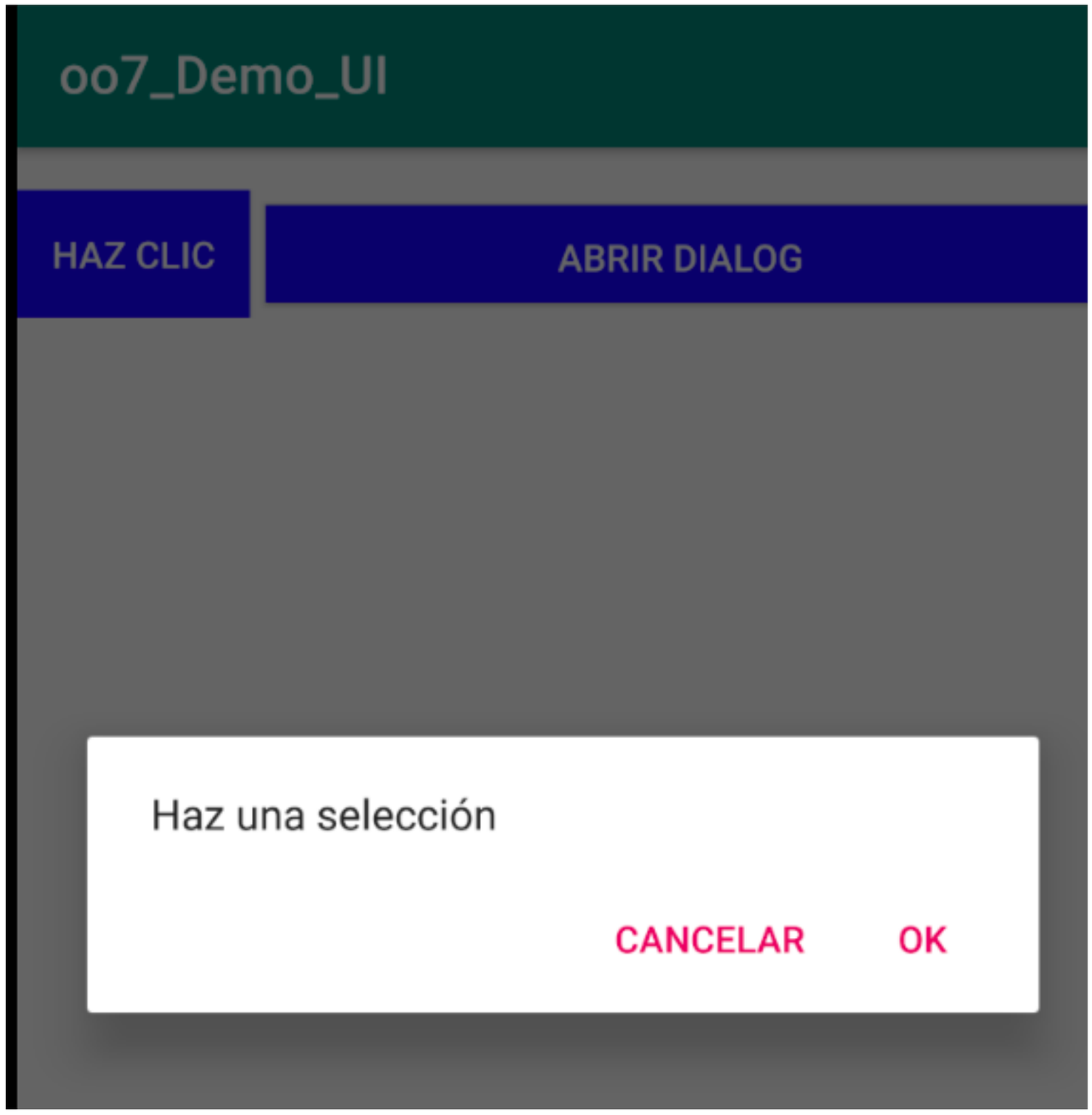
```
override
fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
 builder.setMessage("Haz una selección")
 .setPositiveButton("OK", { dialog, id ->
    // codigo si ok
    Toast.makeText(this.context, "OK !", Toast.LENGTH_SHORT).show()
 })
 .setNegativeButton("Cancelar", { dialog, id ->
    // codigo si cancelar
    Toast.makeText(this.context, "CANCELAR !", Toast.LENGTH_SHORT).show()

})

return builder.create();

}
```

DIALOGFRAGMENT

- Ver Dialog

```
val miDialog = MiDialog()
miDialog.show(supportFragmentManager, "123")
```

DIALOGFRAGMENT



CREAR Y APLICAR UN ESTILO

- Abrir el fichero res/values/styles.xml
- Añadir un elemento <style> con un nombre que identifique de forma exclusiva el estilo.
- Añadir un elemento <item> para cada atributo de estilo que desee definir.
- El nombre en cada elemento se usa como atributo XML en el diseño.

CREAR Y APLICAR UN ESTILO

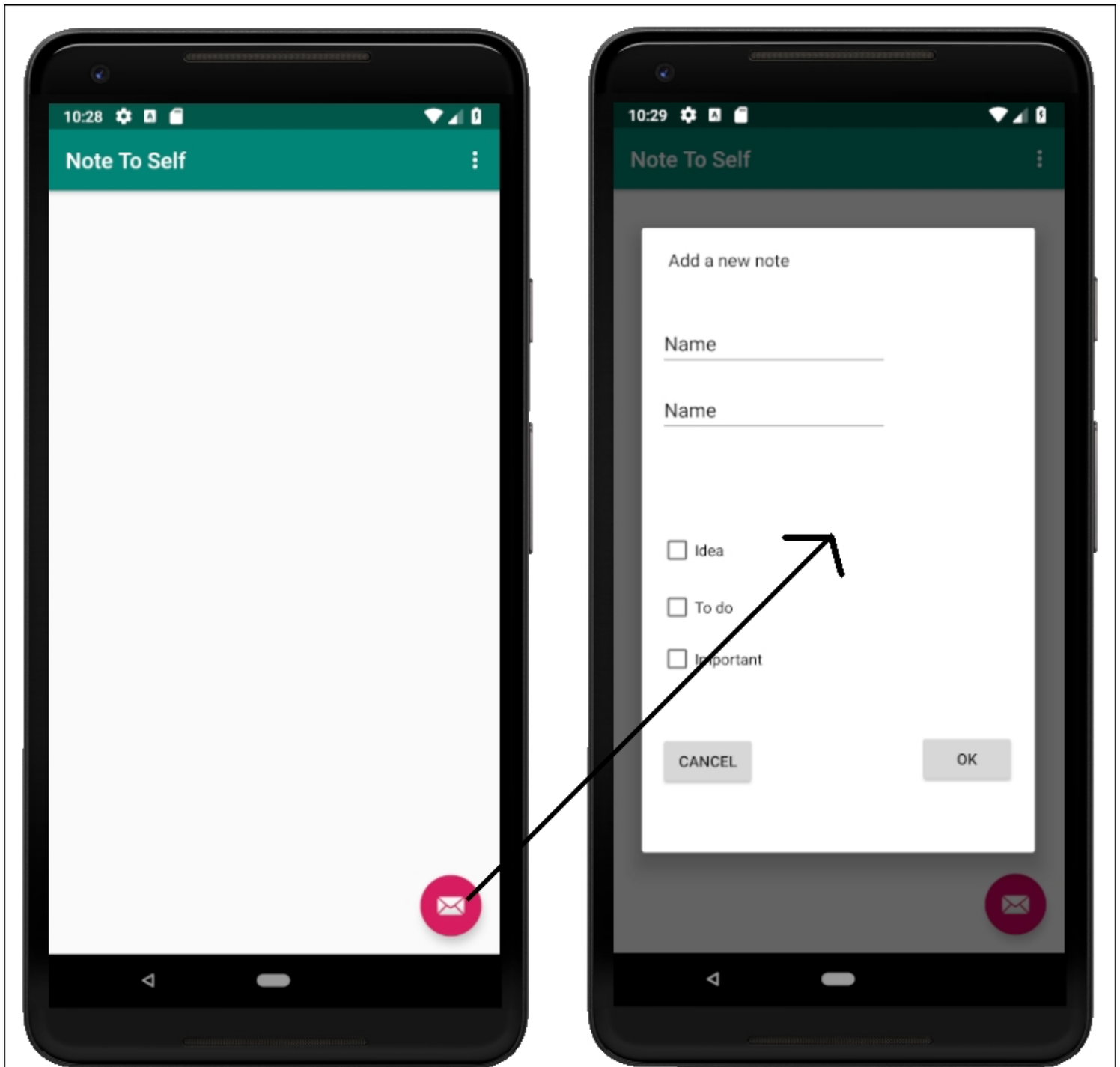
- res/values/styles.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name=“TextoVerde" parent="TextAppearance.AppCompat">
        <item name="android:textColor">#00FF00</item>
    </style>
</resources>
```

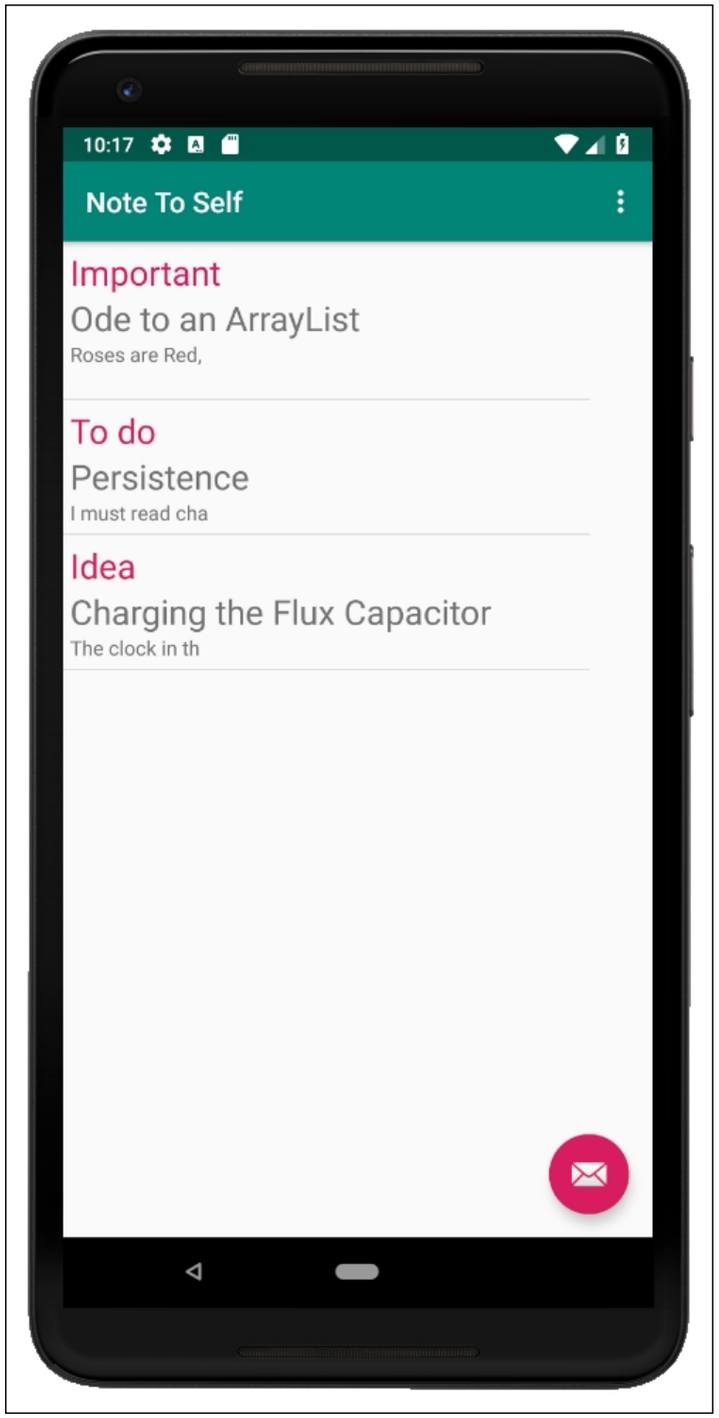
- En el layout

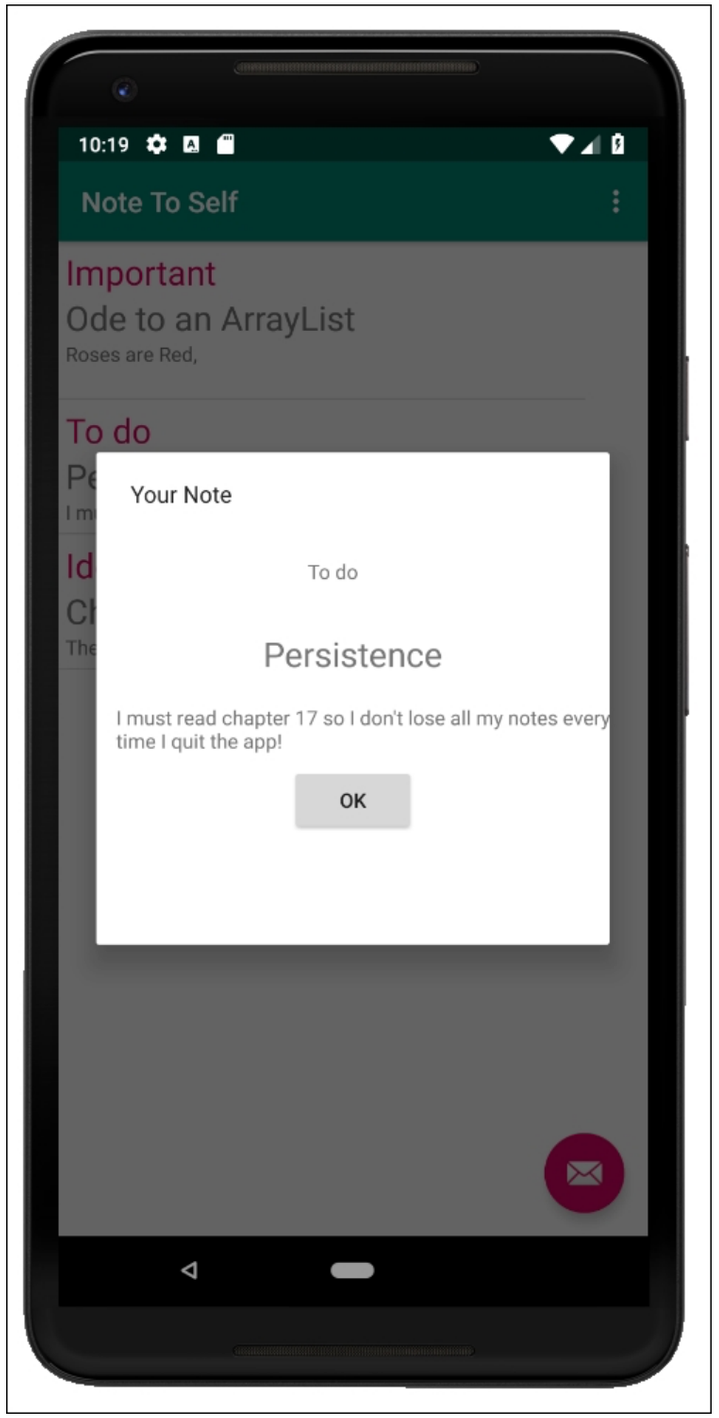
```
<TextView
    style="@style/GreenText"
    ... />
```

APP NOTAS


10:28
Note To Self
10:29
Note To Self
Add a new note
Name
Name
Idea
To do
Important
CANCEL
OK

APP NOTAS


10:17
Note To Self
Important
Ode to an ArrayList
Roses are Red,
To do
Persistence
I must read cha
Idea
Charging the Flux Capacitor
The clock in th


10:19
Note To Self
Important
Ode to an ArrayList
Roses are Red,
To do
Your Note
To do
Persistence
I must read chapter 17 so I don't lose all my notes every time I quit the app!
OK

APP NOTAS – CREAR PROYECTO

- File - > New Adrodproyect
  - Nombre : NotasApp
  - Plantilla : Basic Activity

APP NOTAS. - TEXTOS APLICACIÓN

- Preparar los textos de la aplicación
- Abrir strings.xml del direcotior /res/values

```
<resources>
    <string name="app_name">Notas</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>

    <string name="action_add">add</string>
    <string name="title_hint">Titulo</string>
    <string name="description_hint">Descripción</string>
    <string name="idea_text">Idea</string>
    <string name="important_text">Importante</string>
    <string name="todo_text">Tarea</string>
    <string name="cancel_button">Cancelar</string>
    <string name="ok_button">OK</string>

    <string name="settings_title">Settings</string>
    <string name="theme_title">Theme</string>
    <string name="theme_light">Light</string>
    <string name="theme_dark">Dark</string>
</resources>
```

APP NOTAS. - CLASE NOTA

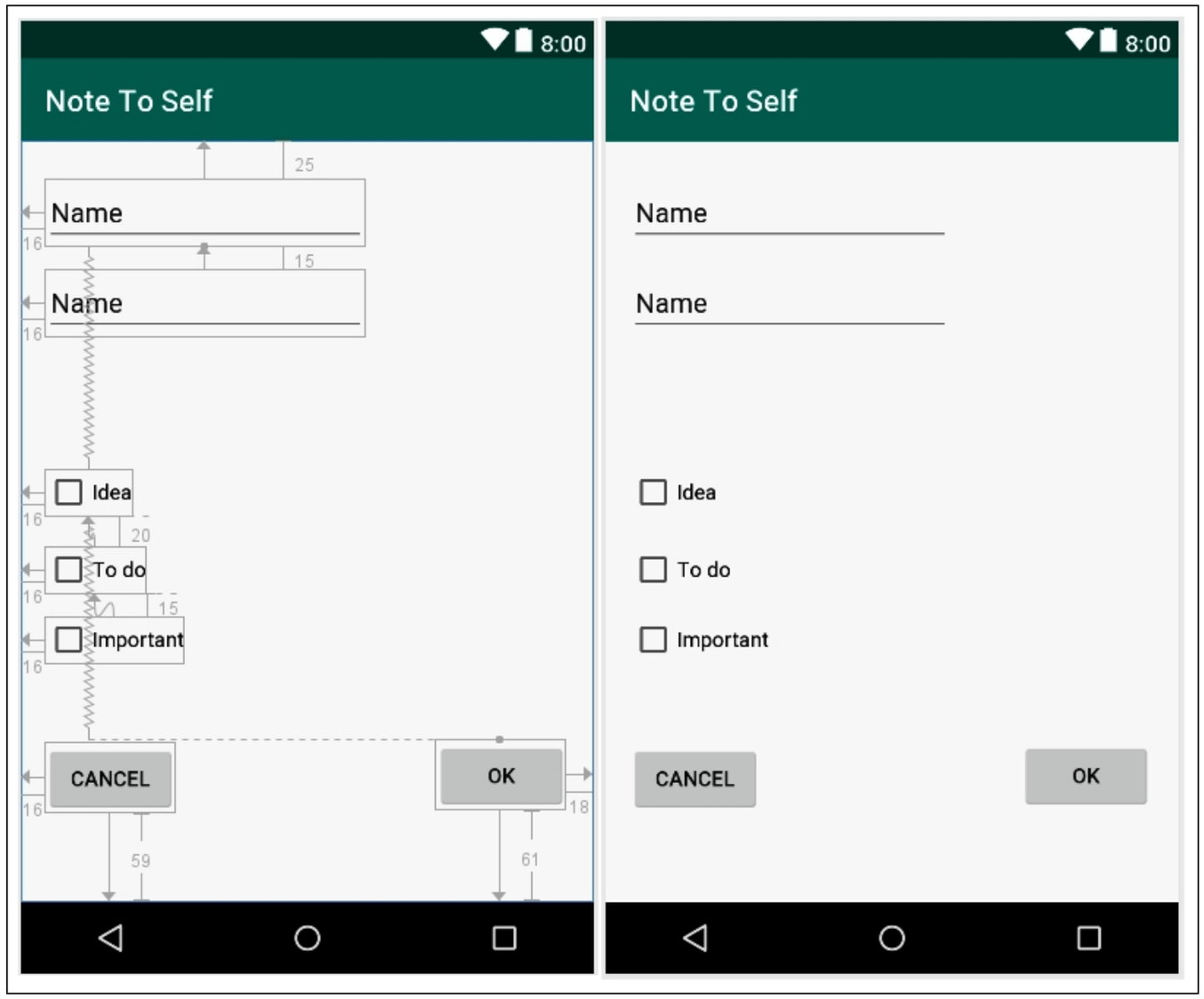
- Crear una nueva clase Kotlin. Llamada Nota

```
class Nota {
    var titulo: String? = null
    var descripcion: String? = null
    var idea: Boolean = false
    var tarea: Boolean = false
    var importante: Boolean = false
}
```

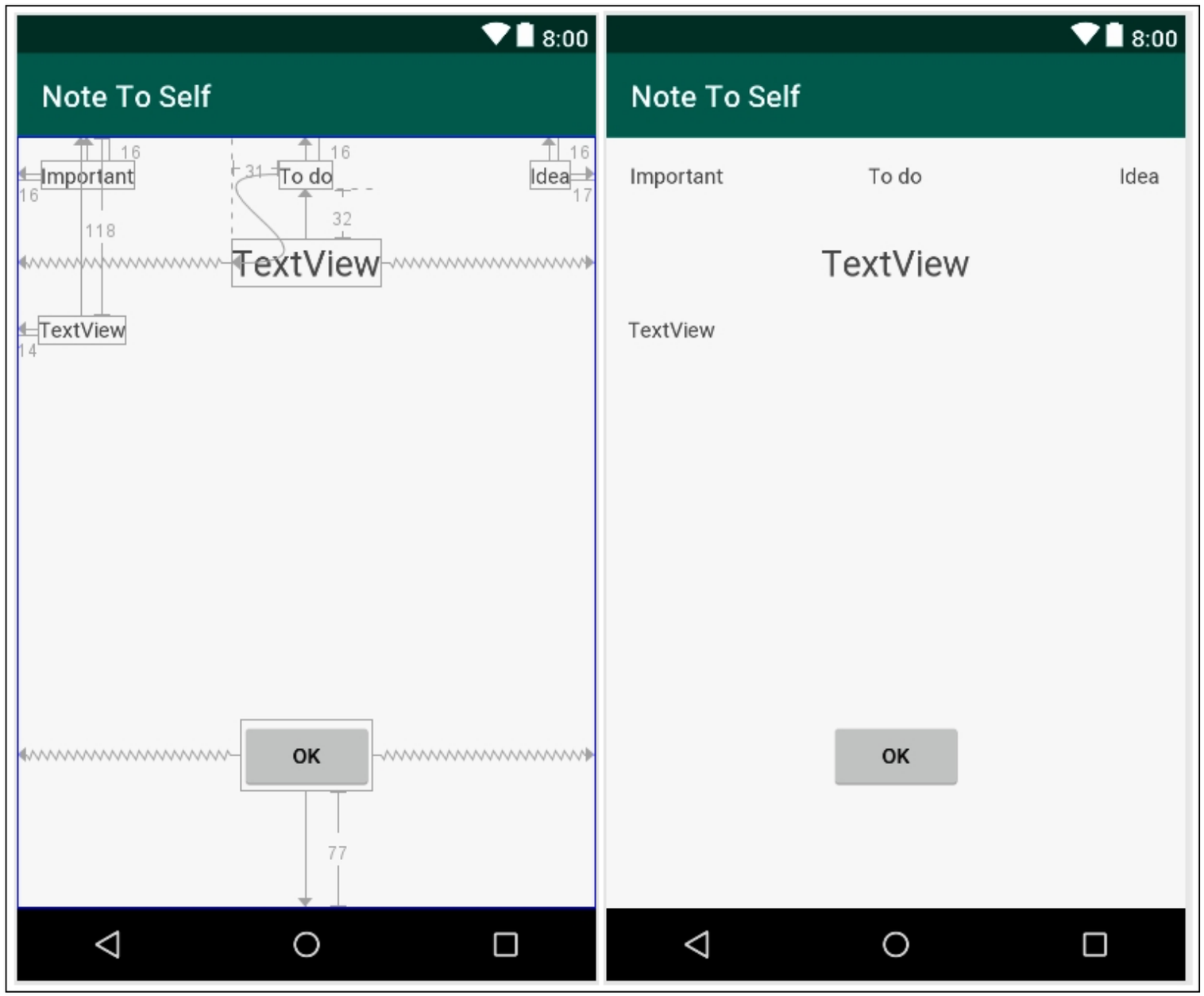
APP NOTAS. – DISEÑAR LA VENTANA DE DIALOGO NUEVA NOTA

- Crear un nuevo layout Kotlin
- Ir a la carpeta /layout y crear un nuevo Layout resourceFile.
- Nombre dialog_nueva_nota
- Root: androidx.constraintlayout.widget.ConstraintLayout.

APP NOTAS. - DISEÑAR LA VENTANA DE DIALOGO NUEVA NOTA



APP NOTAS. - DISEÑAR LA VENTANA DE DIALOGO VER NOTA



INTENT

- Intent es un objeto de mensajería
- Se usa para solicitar una acción a otro compomente de una app
- Usos principales
  - Iniciar un Activity
  - Iniciar un Service
  - Transmitir una emisión

INTENT – INICIAR UN ACTIVITY

- Ejemplo

```
// Declarar e inicializar un objeto Intent llamado miIntent
val myIntent = Intent(this, SettingsActivity::class.java)

// Cambiar  a SettingsActivity
startActivity(myIntent)
```

INTENT – PASAR DATOS ENTRE ACTIVITIES

- Ejemplo

```
val username = “Carlos”

val myIntent = Intent(this, SettingsActivity::class.java)

myIntent.putExtra("USER_NAME", username)

startActivity(myIntent)
```

SHAREDPREFERENCES

- Permite guardar información para usarla en otro momento
- Los datos se almacenan en pares de **clave** y **valor** en ficheros
- Pueden ser :
  - privados **Context.MODE_PRIVATE**
  - Públicos **Context.MODE_WORLD_READABLE** o **Context.MODE_WORLD_WRITEABLE**

PERSISTIR DATOS CON SHAREDPREFERENCES

- Obtener el fichero de preferencia por defecto:

```
val prefs = getSharedPreferences(Context.MODE_PRIVATE)
```

- Obtener el fichero de preferencia por su nombre:

```
//obtener el contexto de laaplicación
Context context = this.getApplicationContext();
//obtenemos un SharedPreferences y creamos un fichero Privado bajo el
//nombre definido con la clave preference_file_key en string.xml
var prefs = context.getSharedPreferences(
            getString(R.string.preference_file_key),
            Context.MODE_PRIVATE);
```

PERSISTIR DATOS CON SHAREDPREFERENCES

- Grabar datos

```
// una instancia de SharedPreferences.Editor para escribir datos
val editor = prefs.edit()

editor.putString("usuario", “luis”)
editor.putInt("cantidad", 2)
editor.putBoolean("activo", true)

// Grabar todos los datos
editor.apply()
```

PERSISTIR DATOS CON SHAREDPREFERENCES

- Leer datos

```
val usuario  = prefs.getString("usuario", “nuevo usuario")

val cantidad  = prefs.getInt("cantidad", 1)

val activo = prefs.getBoolean("activo", false)
```

JSON

- JavaScript Object Notation
- Ejemplo

```
{
  "nombre":"John",
  "edad":30,
  "direccion": {
    “calle":"Los chopos, 1”,
    "CP":"48000",
    "Localidad":"Bilbao"
  }
}
```

EXCEPCIONES

- Definir que un método puede lanzar una excepción
- Ejemplo:

```
@Throws(someException::class)
fun miFunction() {
    // el código con riesgo de producir una excepcion
}
```

- Ahora, cualquier código que use miFunction necesitará manejar la excepción

EXCEPCIONES – TRY, CATCH, FINALLY

- Capturar una excepción
- Ejemplo:

```
 try {
  …
  miFunction()
  …
} catch (e: Exception) {
   Log.e("Exception!",
         “falló miFunction failure", e)
}finally{ // aqui  }
```