Kotlin y Android superfácil

1 Relación entre Kotlin, Java y Android

Kotlin es un lenguaje creado en 2010 por JetBrains que es la empresa que hizo IntelliJ IDEA, uno de los mejores IDE de desarrollo Java. Permite generar código (se traduce automáticamente) para Java, javascript y hasta ejecutables nativos. Los programas Kotlin tienen extensión .kt

Su nombre proviene del nombre de una isla. Kotlin es una "versión mejorada" de Java y aunque se pueden hacer aplicaciones de escritorio (inicialmente fue para eso) actualmente es famoso por ser el lenguaje OFICIAL de Android.

El motivo de que sea el lenguaje oficial de Android se encuentra en una demanda de oracle a todo aquel que usase Java sin pagar una contraprestación a Oracle. Para evitar el problema, así como problemas futuros, Google anunció que Kotlin sería su lenguaje oficial de desarrollo en Android, aunque Java sigue estando disponible para desarrollar Android.

Lo "normal" cuando programas en Kotlin en realidad lo vas a hacer para Android, aunque podrías hacer un programa Kotlin que se ejecutase sin ser para Android, del mismo modo que puedes hacer un programa en Python.

Las extensiones de Kotlin para Android (llamadas androidx) te hacen programar de una forma concreta, con una estructura de recursos y ficheros preestablecida dentro del Android Studio. Es así. Y con Kotlin hay cosas que son más sencillas porque lo han preparado en algunos aspectos para llevarse bien con todo ese tinglado.

A veces necesitarás imports de estas extensiones, como este ejemplo:

```
import androidx.appcompat.app.AppCompatActivity
```

2 El lenguaje Kotlin

2.1 Sintaxis Kotlin

Aquí solo vamos a ver algunas cosas "nuevas" respecto a Java y otros lenguajes

- no hay punto y coma. Las líneas no lo necesitan
- no es como python. Tiene llaves y la indentación no tiene significado. Se parece a python en algunas cosas de su sintaxis pero en eso no.
- Los tipos de las variables se ponen después de declararlas (a la derecha) , al contrario de la mayoría de lenguajes que se ponen a la izquierda.

```
var miVariable: Int
```

- **El tipado** es fuerte (no puede cambiar el tipo de una variable como en Python) pero puedes no declarar el tipo y la variable toma como tipo el del primer valor que le asignes.
- **Null safe**: la bondad de esta característica es cuestionable. Como normalmente una aplicación se pone en producción sin depurar suficiente, han creado la posibilidad de evitar que las variables tomen valores null, a menos que lo indiques explícitamente. Eso evita que la aplicación mal programada tenga este tipo de fallos durante la ejecución.

```
var miVariable: String?
miVariable = null
println("mi variable es $miVariable") // Imprime null
```

- **Println** : la función se llama igual que la de Java pero para imprimir variables se usa el símbolo "$" dentro de la cadena. En java tendrias que poner System.out.println pero en kotlin basta con println

```
var name = "platano"
println("hola $name")
```

- las operaciones binarias en kotlin usan una sintaxis que se aleja de las operaciones lógicas, posiblemente para no cometer errores con los & logicos

kotlin	java

`val andResult  = a and b`
`val orResult  = a or b`
`val xorResult  = a xor b`
`val rightShift = a shr 2`
`val leftShift  = a shl 2`
	`final int andResult  = a & b;`
	`final int orResult  = a | b;`
	`final int xorResult  = a ^ b;`
	`final int rightShift = a >> 2;`
	`final int leftShift  = a << 2;`

- Los **enum** de kotlin son clases , a diferencia de los enum de C o de java. Al ser clases tienen dos atributos por defecto: "name" y "ordinal" , de modo que no son simples constantes como en C sino que se puede ver el texto y el número de un objeto enum. Además se le pueden añadir propiedades que pueden tomar valores diferentes segun el valor principal que tomen

kotlin	java	C

`enum class Direction {`
`  NORTH, SOUTH, WEST,`
`EAST`
`}`
 	
`// Si queremos añadir una`
	`enum Color`
	`{`
	`  ROJO, VERDE,`
	`AZUL;`
	`}`
		`enum Week{`
		`  MON = 10,`
		`  TUE,`
		`  WED,`
		`  THUR,`
		`  FRI = 10,`

propiedad extra:
 	
`enum class Direction (val dir : Int) {`
`    NORTH (dir:1), SOUTH(dir:-1),`
`WEST(dir:1), EAST(dir:-1)`
`}`
		`  SAT = 16,`
		`  SUN`
		`};`

- **clases**: todas heredan de la clase “Any”, al igual que en java heredan de “Object”. Estos dos programas hacen lo mismo y el de kotlin es más compacto

kotlin	**java**

`class Programmer(name: String, age: Int): Person (name, age) {`
`  // ...`
`}`
 	
`// A diferencia de java, en kotlin la clase Person tiene que tener el modificador "open" para indicar que se puede extender`
	`class Programmer extends Person`
	`{`
	`  String name;`
	`  int age;`
	`  // Constructor`
	`  Programmer(String name, int age) {`
	`  this.name= name;`
	`  this.age=age;`
	`  }`
	`}`

- **Keyword “is” vs “instanceof”:** en kotlin tenemos la palabra clave “is”

kotlin	**java**
`if (x is Int) { }`	`if(x instanceof Integer){ }`

- **Keyword “in”: en kotlin podemos usar “in”, como hacemos en python**

```
if (x in 0..10) { }
```

- **Keyword when: en vez del “switch case” convencional, se utiliza “when”:** esto es realmente mejor, es una sintaxis clara y más compacta. No necesita break

```
when (variable) {

valor 1-->  {
```

```
   // Instrucciones
  }

valor2 → println ("hola") // Como solo es una instrucción no
hacen falta llaves

else -->  {}
}
```

- **Interfaces**: en kotlin se usa la misma sintaxis que para la herencia. Es más parecido a C++ que a java

kotlin	java	C++

`interface MyInterface {`
`  fun bar()`
`  fun foo() {`
`  // optional body`
`  }`
`}`
 	
`class Child: MyInterface`
`{`
`  override fun bar() {`
`  // body`
`  }`
`}`
	`interface printable {`
	`  void print();`
	`}`
	`class A6 implements printable {`
	`  public void print(){`
	`  System.out.println("Hello");`
	`}`
		Usa la sintaxis de la herencia. La clase madre declara algún método como “virtual”, de modo que se transforma en una clase llamada “abstracta”.

- **Keywords modificadores de visibilidad** para variables, funciones y clases internas de una clase. Son los mismos que en java (en Java no hay un “internal” como tal, pero no poner nada tendría ese efecto).

  - **public**: todos pueden acceder. Valor por defecto.
  - **private**: no se puede acceder desde fuera de la clase.
  - **protected**: desde una clase interna se puede acceder pero no se puede acceder desde fuera.
  - **internal**: no se puede acceder desde otros módulos (hechos por terceros), pero para el resto de cosas es como public.

- **Miembros static vs companion object**

  Al igual que en C++ y java, en kotlin se pueden declarar miembros de una clase para que sean compartidos por todas las instancias de dicha clase, pero no existe la palabra

static. En su lugar hay que meter los atributos estáticos dentro de un “companion object” , que funcionan igual: se puede acceder a ellos sin instanciar la clase y son los mismos para todas las instancias de la clase.

kotlin	**java**

`class miclase {`
`  var name: String`
`  companion object{`
`  var miatributo: Int`
`  }`
`}`
 	
`miclase.nacionalidad = "español" //todos los miembros son españoles`
	`class miclase {`
	` String name;`
	` static int miatributo;`
	`}`

- **Data clases**: es un concepto nuevo, sirve simplemente para ahorrar líneas de código, que sea más compacto.

  Tiene funciones por defecto que son copy (para copiar objetos) y toString, para poder imprimir todos sus atributos. También permite usar el operador == y automáticamente se comparan todas sus propiedades

```
data class Worker (val name: String = "pepe", val age: Int = 0, work: String){
// La inicialización admite 3 variables (name, age y work)
// Hay 2 valores por defecto por si no se provee valor en la inicialización
  val lastwork: String  // Esta variable no se inicializa al crear el objeto
}

val alguien = worker("manolo", 20, "programador")
println(alguien.toString())
val otro = alguien.copy(age = 20)
if (alguien == otro) println("son iguales en todo")

// Kotlin permite asignaciones en tuplas
val (name, age) = alguien // Asigna cada campo a una propiedad
println(name)
```

2.2 Constructor by lazy

En kotlin existe una forma de evitar trabajo a un constructor, de modo que solo se ejecuten ciertos procesos e inicializaciones en el momento en el que se acceda a ciertos atributos y no antes. A eso se le llama “lazy” (https://kotlination.com/kotlin-lazy-initialization/)

Ejemplo: creas una persona y hasta que no accedes a su miembro books, no se llama a la función loadBooks().

```
package com.kotlination.lazyinit

class BookManager {
  companion object {
    fun loadBooks(person: Person): List<String> {
      println("Load books for ${person.name}")
      return listOf("Master Kotlin at Kotlination", "Be Happy to
become Kotlineer")
    }
  }
}

data class Person(val name: String) {
  val books by lazy { BookManager.loadBooks(this) }
}
```

2.3 Palabras clave let, run, also, apply

Estas palabras clave sí que son bastante “nuevas” en relación a otros lenguajes. He sacado unos ejemplos ilustrativos de https://www.journaldev.com/19467/kotlin-let-run-also-apply-with

- **let**: usando esta palabra clave desde un objeto, ejecutas un bloque de código donde dicho objeto es reemplazado por la expresión “it”. Esto sirve para ahorrarte crear una función. Es decir, es una forma de compactar más el código. Se pueden anidar, sacando más jugo a su funcionalidad.

```
fun main(args: Array<String>) {
    var str = "Hello World"
    str.let { println("$it!!") }
    println(str)

}
// El resultado en consola:
// Hello World!!
// Hello World
```

- **run:** es igual que let, pero retorna la última sentencia. “let” se puede equiparar a un procedimiento y “run” a una función. Ojo porque run no soporta “it”

```
var tutorial = "This is Kotlin Tutorial"
    println(tutorial) //This is Kotlin Tutorial
    tutorial = run {
        val tutorial = "This is run function"
        tutorial
    }
    println(tutorial) //This is run function
```

- **also**: es como **let** y soporta "it". La diferencia es que "**also**" retorna el objeto. Con este ejemplo vemos la diferencia

```
data class Person(var name: String, var tutorial : String)
var person = Person("Pepe", "Kotlin")

var l = person.let { it.tutorial = "Android" }
var al = person.also { it.tutorial = "Android" }

println(l)  // kotlin.unit
println(al) // Person (name="Pepe", tutorial="Android")
println(person) // Person (name="Pepe", tutorial="Android")
```

- **apply**: es como **also** pero en lugar de it , usa **this** para referirse al objeto. Es decir, cambia el significado de this dentro de el fragmento de código. Lógicamente la palabra this se puede obviar

```
data class Person(var n: String, var t: String)
var person = Person("Pepe", "Kotlin")

person.apply { t = "Swift" }
println(person) // Person (name="Pepe", tutorial="Swift")

person.also { it.t = "Kotlin" }
println(person) // Person (name="Pepe", tutorial="kotlin")
```

2.4 Recursos para aprender Kotlin

Te recomiendo el video de 10 lecciones de mouredev para empezar. Lo encontrarás en el apartado de videos. Ponlo a velocidad 2x.

2.4.1 Recursos online

para probar ejemplos kotlin con un editor online sin necesidad de instalar nada

- https://play.kotlinlang.org

Referencia oficial del lenguaje:

- https://kotlinlang.org/docs/home.html

Comparaciones Java vs Kotlin con ejemplos de muchas cosas. Muy interesante y cortito:

- https://fabiomsr.github.io/from-java-to-kotlin/

Estos dos son de la misma colección y el segundo usa conceptos del primero:

- https://tutorialesprogramacionya.com/kotlinya/
- https://www.tutorialesprogramacionya.com/kotlinparaandroidya/

Este es en español, buena pinta:

- https://www.toptal.com/software/introduccion-a-kotlin-programacion-de-android-para-seres-humanos

Curso con 20 temas en español:

- https://cursokotlin.com/curso-programacion-kotlin-android/

2.4.2 Libros gratis

Libro en inglés, recomendado. titulado “kotlin notes for professionals”. Además pone ejemplos de cómo se escribe en Java y en Kotlin.

Tiene 94 páginas y es un pdf con navigation panel:

- https://books.goalkicker.com/KotlinBook/

2.4.3 Videos

Curso de Kotlin para principiantes (10 lecciones) e intermedio (6 lecciones):

- https://www.youtube.com/playlist?list=PLNdFk2_brsReZeIQ1-2r783GWus0ZZ5io

La playlist contiene las 3 partes, que en total son 21 videos.

Todos los videos de la primera parte (10 lecciones =11 videos) están enlazados desde github

- https://github.com/mouredev/KotlinDesdeCero

Primera parte "Kotlin desde cero": son 11 videos (cada video es una lección, salvo una lección)

Segunda parte: "Kotlin intermedio": son 6 videos, el primero es https://www.youtube.com/watch?v=4MoWs6Miq0s&list=PLNdFk2_brsReZeIQ1-2r783GWus0ZZ5io&index=12

Tercera parte: "": son 5 videos

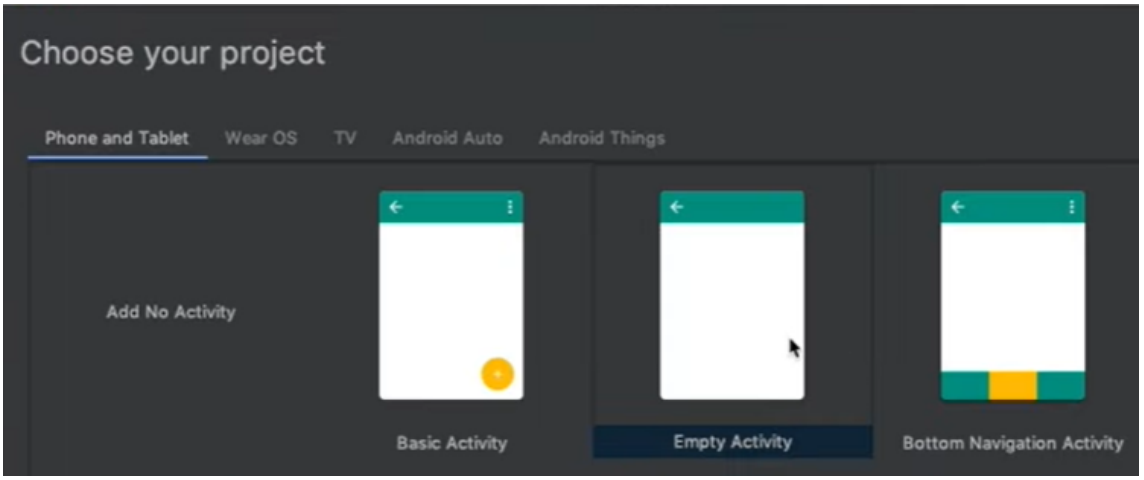
3 Instalar Android studio y crear un emulador

Para crear o abrir un proyecto debes estar dentro de Android Studio. Descargatelo desde https://developer.android.com/studio/

No hay un fichero como en C++ y visual studio (ficheros .sln) que al hacer clic pueda lanzar el visual studio.

Lo que hace Android Studio es entrar en los directorios y buscar en ficheros ocultos , que contienen información del proyecto.

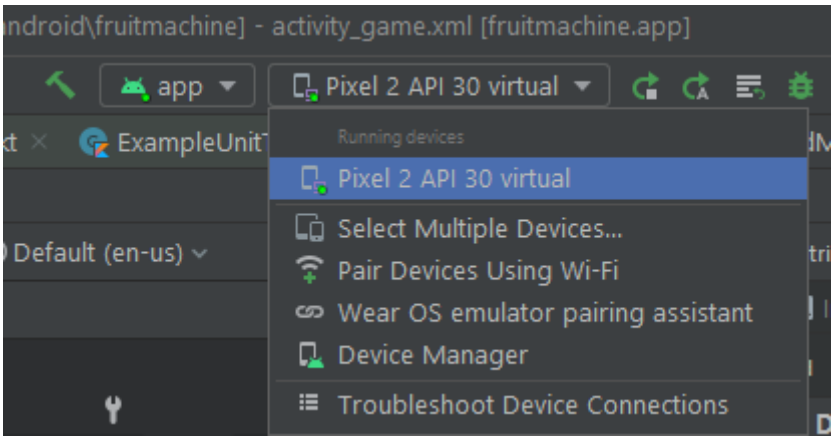
Para crear un proyecto nuevo basta con crear una “empty activity”.



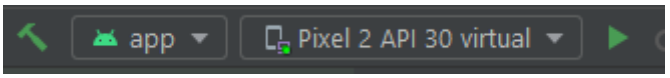
Luego, editamos el fichero **mainActivity.kt** que se encuentra dentro de una estructura de información de la app que verás en la parte izquierda del Android Studio.

Después de crear nuestra aplicación pulsaremos el botón “play” para ejecutar. En ese momento debemos crear un emulador si aún no lo hemos hecho. Puedes elegir distintos tipos de emuladores con soportes a distintas versiones de Android. Este emulador en realidad necesita el sistema operativo android y por lo tanto te obligará a descargar bastantes megas de datos.

Otra opción es ejecutar directamente en un móvil real, que tendrás que conectar por USB al ordenador, usando el device manager.



Una vez creado podremos lanzar nuestro programa. Android Studio se toma su tiempo. Cuando pulsas el botón “play”, Android Studio lanza una cosa llamada **gradle** que es un compilador basado en la máquina virtual Java y si no tienes errores podrás ver el emulador ejecutándose en tu PC.



Para detener tu aplicación dispones de un botón rojo cuadrado.

Si haces uso de la función println, tendrás que abrir la pestaña run (situada abajo), para ver los mensajes que se mandan a la consola. En la consola verás también los mensajes de ejecución de Android, los errores, etc.

3.1 ¿Qué es gradle?

Gradle es un sistema de compilación/build **basado en JVM** (Java Virtual Machine). Gradle entiende Java y Kotlin y XML, etc y es capaz de crear el APK (el fichero de aplicación para un dispositivo android).

Lo mejor de gradle es que **es un plugin**, lo que facilita su actualización y su exportación de un proyecto a otro.

Gradle se puede invocar desde Android Studio pero también desde consola.

Gradle tiene su propio lenguaje mediante el que se especifican dependencias a la hora de compilar, etc. Es parecido a los ficheros makefile para el programa make, que tienen su propia sintaxis.

3.2 Aplicación hola mundo

La aplicación hola mundo simplemente es meter una línea en la función onCreate() de la clase MainActivity, que es un fichero .kt que verás en la carpeta Java de la estructura de la aplicación que te muestra Android Studio.

Como verás, Android Studio ha escrito ficheros Java, ficheros XML y un montón de cosas que luego veremos. Ha programado mucho más Android Studio que tú. Tú solo has añadido una línea a un fichero.

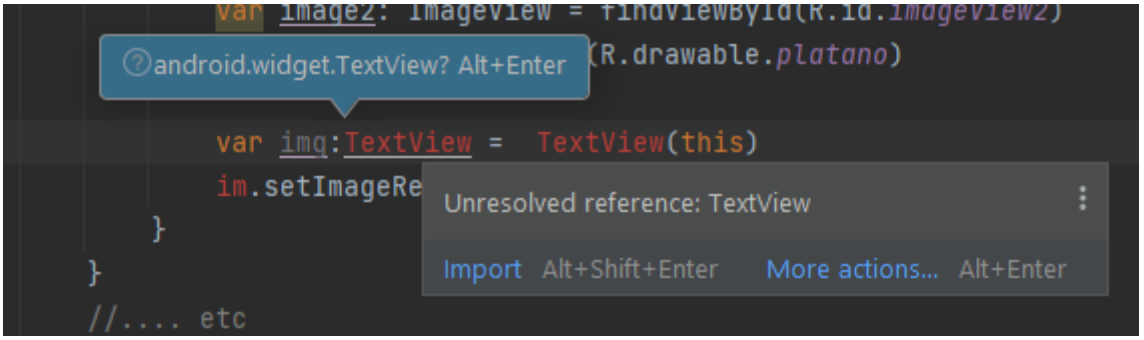
```
package com.example.holamundo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      println("hola mundo")  // Solo esta línea es "tuya"
    }
  }
}
```

3.3 Imports automáticos

Cuando te falte un import o no le guste algo al compilador, te aparecerá el texto en rojo mientras escribes. Si te acercas con el ratón, te permitirá importar la librería que te falta para que funcione, simplemente pulsando Alt+Enter.

¿Y cómo sabe Android Studio que te falta importar eso? Pues porque es muy listo. Probablemente busca el nombre de la clase que has utilizado en las librerías típicas y a partir de ahí, te recomienda el import de alguna de las coincidencias.

También te avisa en rojo cuando escribes una variable que aun no has declarado, etc.



4 Creando aplicaciones

4.1 Estructura de ficheros de un proyecto Android+Kotlin

La aplicación que has construido te crea esta estructura "lógica" que puedes ver desde Android Studio, y una estructura física de ficheros. Como ves, hay muchísimas cosas creadas de forma automática y aunque cada cosa tiene su razón de existir, resulta chocante.

Lo que ves desde Android Studio:	Lo que ves en el directorio (en un proyecto al que he llamado "fruitmachine")

app
• manifests
• java
• java (generated)
• res
  ○ drawable
  ○ layout
  ○ mipmap
  ○ navigation
  ○ values
• res (generated)
 	
Gradle Scripts
• build.gradle
• build.gradle
• gradle-wrapper.properties
• proguard-rules.pro
• gradle.properties
• settings.gradle
• local.properties
	fruitmachine
	.gradle
	.idea
	app
	build
	gradle
	
	.gitignore
	build.gradle
	gradle.properties
	gradlew
	gradlew.bat
	local.properties
	README.md
	settings.gradle

4.2 Activities

Una actividad es una pantalla de una aplicación (más o menos). Aunque no es lo mismo, es una aproximación sencilla para entender el concepto. Posiblemente haya actividades que involucren más de una pantalla y pantallas que se puedan dividir en actividades, pero en general se puede decir que una actividad se corresponde con una pantalla de la aplicación. Si la

aplicación tiene 3 pantallas entonces tendremos 3 actividades. Cada actividad escucha los eventos de los botones y elementos gráficos de la pantalla.

Para crear una actividad, hay que ir a File → New → Activity → Empty activity y se creará automáticamente un fichero .kt y un layout asociados a la actividad.

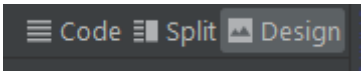
Para lanzar una actividad desde otra actividad se hace así:

Supongamos que la actividad que quiero lanzar se llama **GameActivity**. Simplemente escribimos estas líneas. Verás la palabra "java" en estas líneas de Kotlin porque nos estamos refiriendo a la clase proveniente de Java y no de Kotlin. Es decir, que se tendrá acceso a las propiedades internas de la clase usando la "Java Reflection API" y no la "Kotlin Reflection API".

```
val intent = Intent(this, GameActivity::class.java)
startActivity(intent)
```

4.3 Diseño de layouts

Se pueden hacer en XML o mediante un asistente gráfico de Android Studio. Basta con pulsar el botón "Design" en la parte superior derecha del fichero xml del layout:

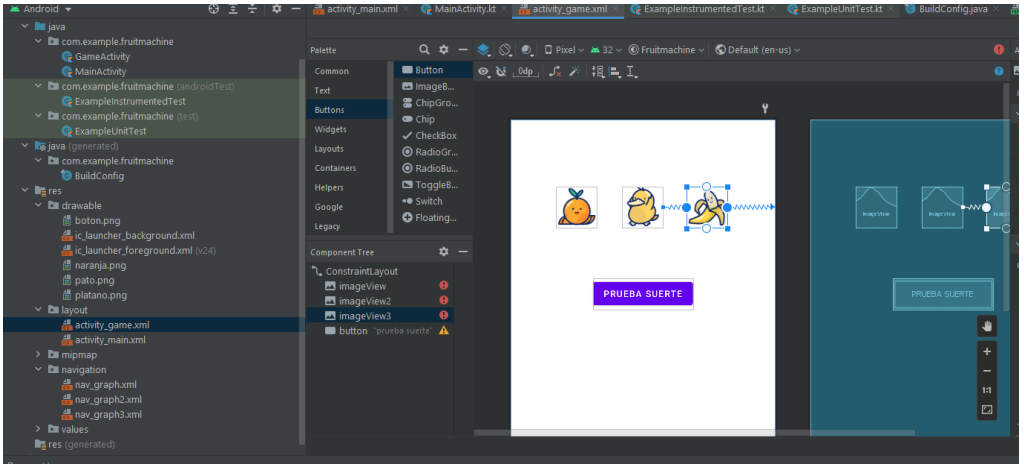


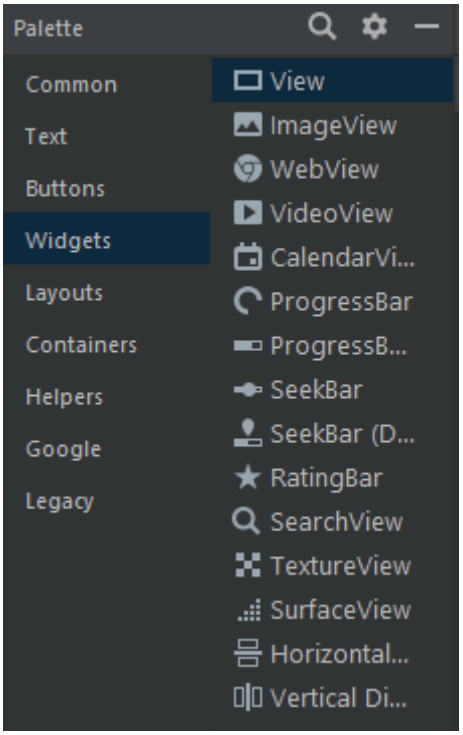
Existen layouts de diferentes tipos. El más común o por defecto es el llamado "ConstraintLayout", que se caracteriza por permitirte colocar los elementos que quieras y ajustarlos mediante restricciones de tamaño y posición. Las restricciones de posición se mostrarán como diferentes dibujos de líneas que salen desde unos círculos que hay en los límites superior, inferior, derecho e izquierdo del widget que estás colocando.

Hay 3 tipos de restricciones de posición:

- Fixed (línea recta |--------|): establece una distancia fija entre dos puntos.
- Wrap content (línea en zigzag simétrico |/\/\/\/\|): hará que el elemento se centre y el tamaño del elemento sea ajustado al contenido.
- Match constraints (línea en diente de sierra |/|/|/|/|/|): hará que el tamaño del elemento se agrande hasta el final.

Si no aplicas restricciones, los elementos serán libres de ser dispuestos en cualquier lugar de la pantalla según le apetezca al compilador, a pesar de que en la pantalla de diseño se vean en una posición concreta. **Ojo: debes ajustar los 4 muelles de cada elemento!**



	Dentro del layout tienes muchos elementos posibles para añadir, que se encuentran en la paleta clasificados por categoría. Por ejemplo, dentro de la categoría button tenemos el button, el ImageButton, el radioButton, etc. Dentro de los widgets tenemos el imageView, videoView, etc.

4.4 Botones con imágenes y recursos

Si quieres usar imagenes en tus layouts, dentro de botones (ImageButton) o simplemente como imágenes (imageview) tienes que meterlas en:

<projectname>/app/src/main/res/drawable

Una vez que las metas las podrás usar dentro de tus layouts. Hay muchos sitios donde encontrar dibujos para tus layouts, pero un sitio donde encontrar botones y elementos gráficos gratis organizados por categorias es en https://www.iconfinder.com/

¿Por qué tienen que estar ahí? Pues porque Android Studio las necesita ahí. No te lo cuestiones porque te va a dar igual, ni trates de meterlas en otro sitio.

4.5 Captura de eventos

Hay al menos cuatro formas de capturar los eventos de los elementos de un layout. Vamos a ver dos de ellos:

- **Atributo onclick:** está bien para una prueba pero si pones onclick= prueba y la función prueba está en un archivo kt, no sabrás quien invoca a dicha función a menos que revises todos los elementos de todos los layouts buscando quien la invoca.
- **Listener de eventos**: puedes hacer que la actividad herede de la clase "view.OnClickListener".

```
class MainActivity : AppCompatActivity() , View.OnClickListener{
```

Y después en el OnCreate creas variables para los botones y les asignas como listener a "this".

```
var boton_juega : ImageButton = findViewById(R.id.boton_juega)
var boton_platano : ImageButton = findViewById(R.id.boton_platano)

boton_juega.setOnClickListener(this)
boton_platano.setOnClickListener(this)
```

Después, creas el listener dentro de la clase que tiene la actividad así:

```
override fun onClick(v:View) {
  when (v.getId()) {
    R.id.boton_juega -> {
      // Do something for button 1 click
      println("juega")
      var caca: String?
      caca = null
      println("caca es $caca")
    }

    R.id.boton_platano -> {
      // Do something for button 2 click
      println("platano")
    }
     //... etc
}
```

4.6 Actuar sobre las propiedades de un elemento de layout

Es muy sencillo. cada elemento tiene muchas funciones y puedes actuar sobre ellos, simplemente debes crear una variable para acceder al elemento y justo después invocas sus funciones

En este ejemplo le cambio la imagen a un elemento llamado “imageView1”

La imagen plátano está en el directorio de recursos de imágenes, concretamente en **<projectname>/app/src/main/res/drawable** y como esta ahi (platano.png), cuando escribes el punto tras la palabra drawable, una de las opciones que te aparece es “platano”**. Es decir, detrás del editor android studio está trabajando muchísimo buscando cosas, lo tiene todo catalogado para facilitar la programación.**

```
var image1: ImageView = findViewById(R.id.imageView1)
image1.setImageResource(R.drawable.platano)
```

4.7 Creación dinámica de elementos en el layout

Para hacerlo debes darle un id al layout , que lo puedes hacer desde la vista de diseño del layout. yo lo he llamado “milayout”.

En este ejemplo creo un TextView dinámicamente. Podría centrarlo, ponerle colores, elegir otro tamaño de fuente, etc. En este caso simplemente lo añado al layout, lo cual simplemente hará que se muestre en la esquina superior izquierda.

```
var txt: TextView =  TextView(this)
txt.setText("FELICIDADES")
txt.setTextSize(40.0F)

val layout: ConstraintLayout =
findViewById(R.id.milayout)
layout.addView(txt)
```

5 Acceso a almacenamiento del teléfono

5.1 Acceso a ficheros como recursos propios

Una forma de acceder a ficheros propios de la aplicación es meterlos en el directorio “res” al crear el proyecto. Después podremos acceder a ellos mediante la clase “resources”.

Este es un ejemplo que se encuentra en:
https://rrtutors.com/tutorials/how-do-i-read-a-text-file-in-android-using-kotlin

En este ejemplo se ha metido un fichero llamado “textfile” dentro de un directorio llamado “raw” creado dentro de “res”. Ojo no vale crear un directorio llamado “miscosas”. Tienes que crear uno de los que conoce la clase resources.

Como se puede ver en el ejemplo, no usa un tipo de datos “File”, sino que crea un InputStream desde la clase resources.

```
var string: String? = ""    // String es inmutable
val stringBuilder = StringBuilder() // StringBuilder es mutable
(como StringBuffer)
val mis: InputStream =
this.resources.openRawResource(R.raw.textfile)
val reader = BufferedReader(InputStreamReader(mis))
while (true) {
    try {
        if (reader.readLine().also { string = it } == null) break
    } catch (e: IOException) {
        e.printStackTrace()
    }
    stringBuilder.append(string).append("\n")
    textView.text = stringBuilder
}
mis.close()
```

5.2 Acceso a “internal storage”

Es privado. Sólo puede acceder la propia aplicación y desde fuera (aplicación ”files”) no vamos a poder verlo. Por eso es un almacenamiento muy “opaco”. El usuario del teléfono no puede verlo, tan solo lo ve la aplicación. **Es útil para ficheros temporales de la aplicación, o para ficheros que contienen información de claves, tokens y cosas así**. No requiere dar permisos especiales a la aplicación.

```
capturedImage = File(getFilesDir() , "My_Captured_Photo.jpg")
if (capturedImage.exists()) {
   capturedImage.delete()
}
capturedImage.createNewFile()
```

5.3 Acceso a “external storage”

Este es el almacenamiento habitual que puede ver el usuario a traves de otras aplicaciones como “files” o de la aplicacion “galeria de fotos”, etc. **Es útil para ficheros que queremos ver desde otras aplicaciones o enviar por correo etc.** Requiere dar permisos especiales a la aplicación, lo cual debemos hacer a través del fichero manifest (después al ejecutarla, el usuario tendrá que aceptar esa petición de permisos).

Añadir estas dos líneas al fichero AndroidManifest.xml , , dentro de <manifest>

```
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

El acceso al external storage se hace a través de un concepto llamado “**provider**” que permite asociar URIs a los ficheros y compartirlos entre aplicaciones de un modo seguro. Parece que hacerlo con URIs es seguro porque se pueden dar permisos de acceso temporales mientras que con un File no se puede. Tienes mas info del concepto provider en https://developer.android.com/guide/topics/providers/content-provider-basics

Meter estas dos líneas dentro de AndroidManifest.xml, dentro del tag <aplication>

```
<provider
   android:name="androidx.core.content.FileProvider"
   android:authorities="${applicationId}.provider"
   android:exported="false"
   android:grantUriPermissions="true">
   <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/provider_paths" />
</provider>
```

En el folder de resources (res) crea una carpeta llamada xml y dentro mete el fichero **provider_paths.xml** al que hemos hecho referencia en el manifest. Dicho fichero debe contener esto:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
   <external-path
        name="external_files"
        path="." />
   <files-path
        name="files"
        path="." />
</paths>
```

Por último, ya solo queda acceder al fichero desde nuestro código fuente. Tenemos dos formas: external cache y external file

```
// Ejemplo con external cache
capturedImage = File(externalCacheDir, "My_Captured_Photo.jpg")

// Esta parte es común a external cache y external file
mUri = if(Build.VERSION.SDK_INT >= 24){

FileProvider.getUriForFile(Objects.requireNonNull(getApplicationCont
ext()),
       BuildConfig.APPLICATION_ID + ".provider", capturedImage);

} else {
   Uri.fromFile(capturedImage)
}
```

```
// Ejemplo con external file
val dir = getExternalFilesDir(Environment.DIRECTORY_PICTURES)
capturedImage = File(dir, "My_Captured_Photo.jpg")

// Esta parte es común a external cache y external file
mUri = if(Build.VERSION.SDK_INT >= 24) {
  FileProvider.getUriForFile(
    Objects.requireNonNull(getApplicationContext()),
    BuildConfig.APPLICATION_ID + ".provider",
    capturedImage
  );
} else {
  Uri.fromFile(capturedImage)
}
```

6 Acceso a los sensores del teléfono

Hay dos formas de acceder a un sensor: directamente o a través de una aplicación de captura

Acceder directamente es difícil, porque debemos interactuar con el driver del sensor (alguna librería específica) y necesitamos darle permiso a la aplicación

Siempre que se pueda, es más recomendable (y fácil) usar una aplicación de captura. Para ello lanzaremos dicha aplicación como actividad y a su terminación recogeremos el resultado con la función **onActivityResult()**.

Vamos a ver un ejemplo con la cámara:

Si queremos acceder directamente a la cámara, primeramente necesitamos dar permisos (que el usuario aceptará o rechazará). Pero esto no es necesario si lo hacemos a través de una aplicación de captura. No obstante, si quisiéramos acceder directamente, las líneas a añadir en el Manifest dentro del tag <Manifest> serían estas:

```
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
```

A continuación vamos a ver el código para acceder a través de la aplicación de captura.

Podemos usar memoria interna o externa para la imagen pero como la aplicación de captura es **otra aplicación diferente de la nuestra**, tenemos que crear una uri que le pasaremos a la aplicación de captura para que deje ahí el resultado. La uri la conseguiremos a través de un provider (ver apartado de acceso a almacenamiento). En caso de usar memoria interna, el fichero asociado a la imagen quedará oculto a cualquier otra aplicación a menos que nuestra aplicación le pase la uri (como es este caso). Nadie puede pedirle una uri contra nuestro fichero al provider salvo nosotros pues es memoria interna.

```
// Creamos file en memoria interna
// -----------------------------------
capturedImage = File(getFilesDir() , "My_Captured_Photo.jpg")
if(capturedImage.exists()) {
   capturedImage.delete()
}
capturedImage.createNewFile()

// Esta parte es necesaria para conseguir la uri
// ------------------------------------------------
mUri = if(Build.VERSION.SDK_INT >= 24) {
  FileProvider.getUriForFile(
    Objects.requireNonNull(getApplicationContext()),
    BuildConfig.APPLICATION_ID + ".provider",
    capturedImage
  );
} else {
  Uri.fromFile(capturedImage)
}

// Ahora lanzamos la aplicacion de captura pasando nuestra uri
// ------------------------------------------------------------------
```

```
val intent = Intent("android.media.action.IMAGE_CAPTURE")
intent.putExtra(MediaStore.EXTRA_OUTPUT, mUri)
startActivityForResult(intent, OPERATION_CAPTURE_PHOTO)

// Cuando la actividad termine, invocará nuestra función "onActivityResult"
// en este ejemplo simplemente mostramos la imagen en el layout
// ----------------------------------------------------------------------
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
  super.onActivityResult(requestCode, resultCode, data)

  println ("imagen capturada")
  println(mUri)
  var visor: ImageView = ImageView (this)
  visor.setImageURI(mUri)

  val layout: ConstraintLayout = findViewById(R.id.milayout)
  layout.addView(visor)
}
```