Práctica 2.

Android y KMP.

# Resultados de aprendizaje y criterios de evaluación.

CE1b. Se han identificado las tecnologías de desarrollo de aplicaciones para dispositivos móviles.

CE1c. Se han instalado, configurado y utilizado entornos de trabajo para el desarrollo de aplicaciones para dispositivos móviles.

CE1f. Se ha analizado la estructura de aplicaciones existentes para dispositivos móviles identificando las clases utilizadas.

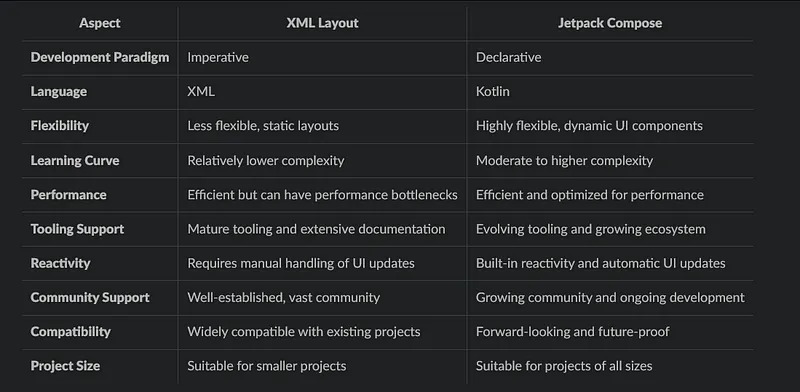
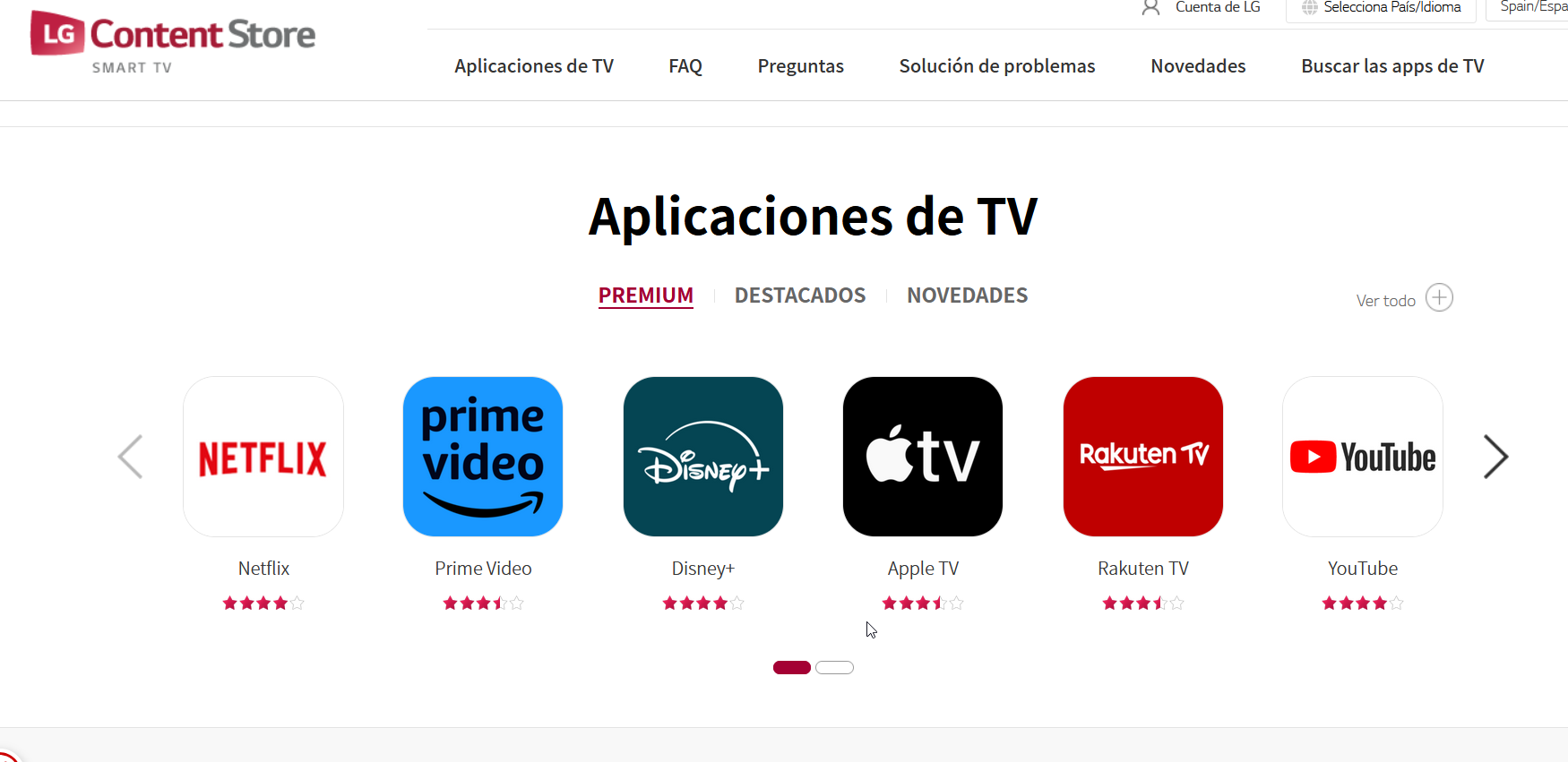
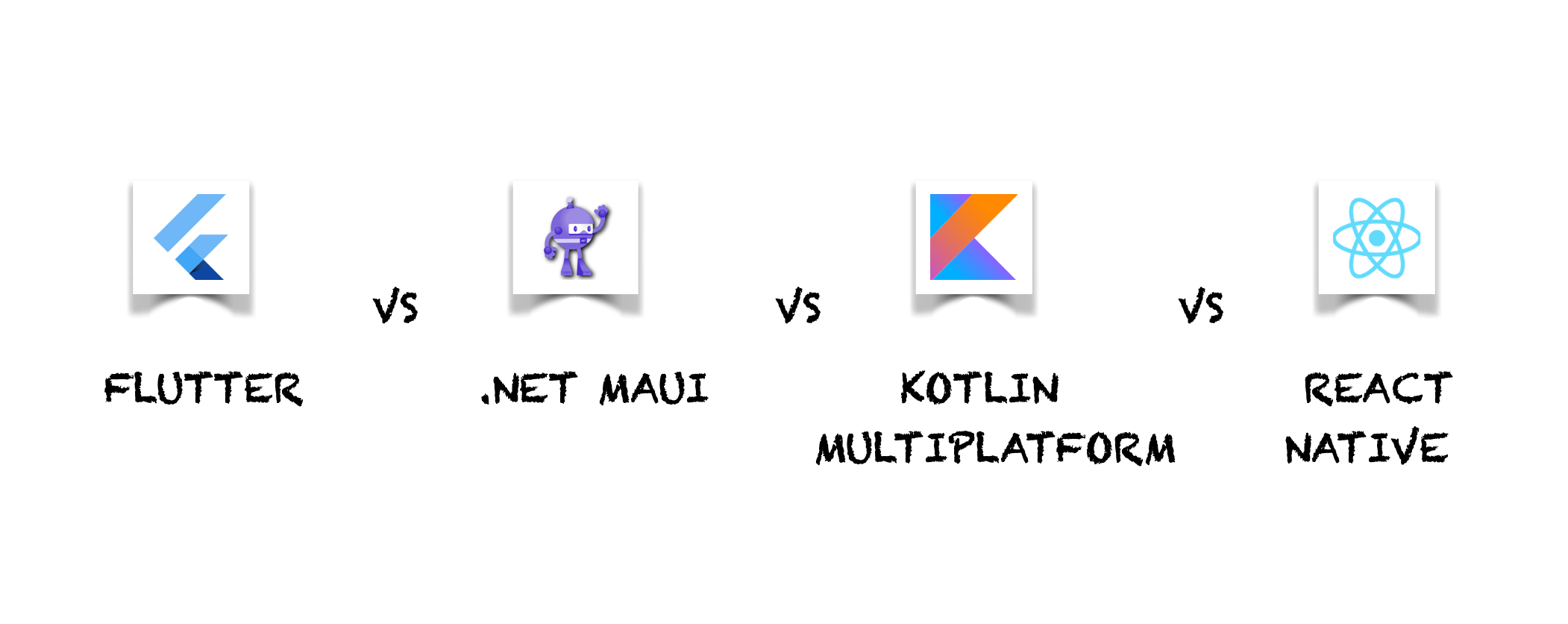
CE1g. Se han realizado modificaciones sobre aplicaciones existentes.

CE1h. Se han utilizado emuladores para comprobar el funcionamiento de las aplicaciones.

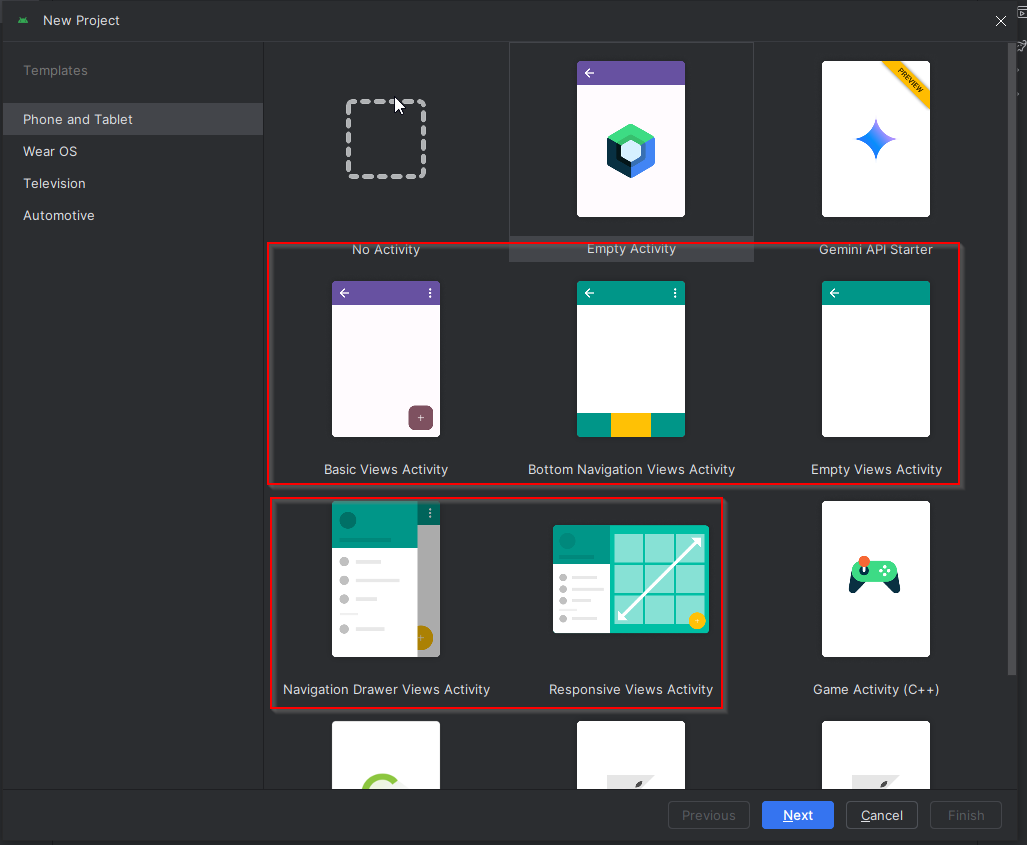
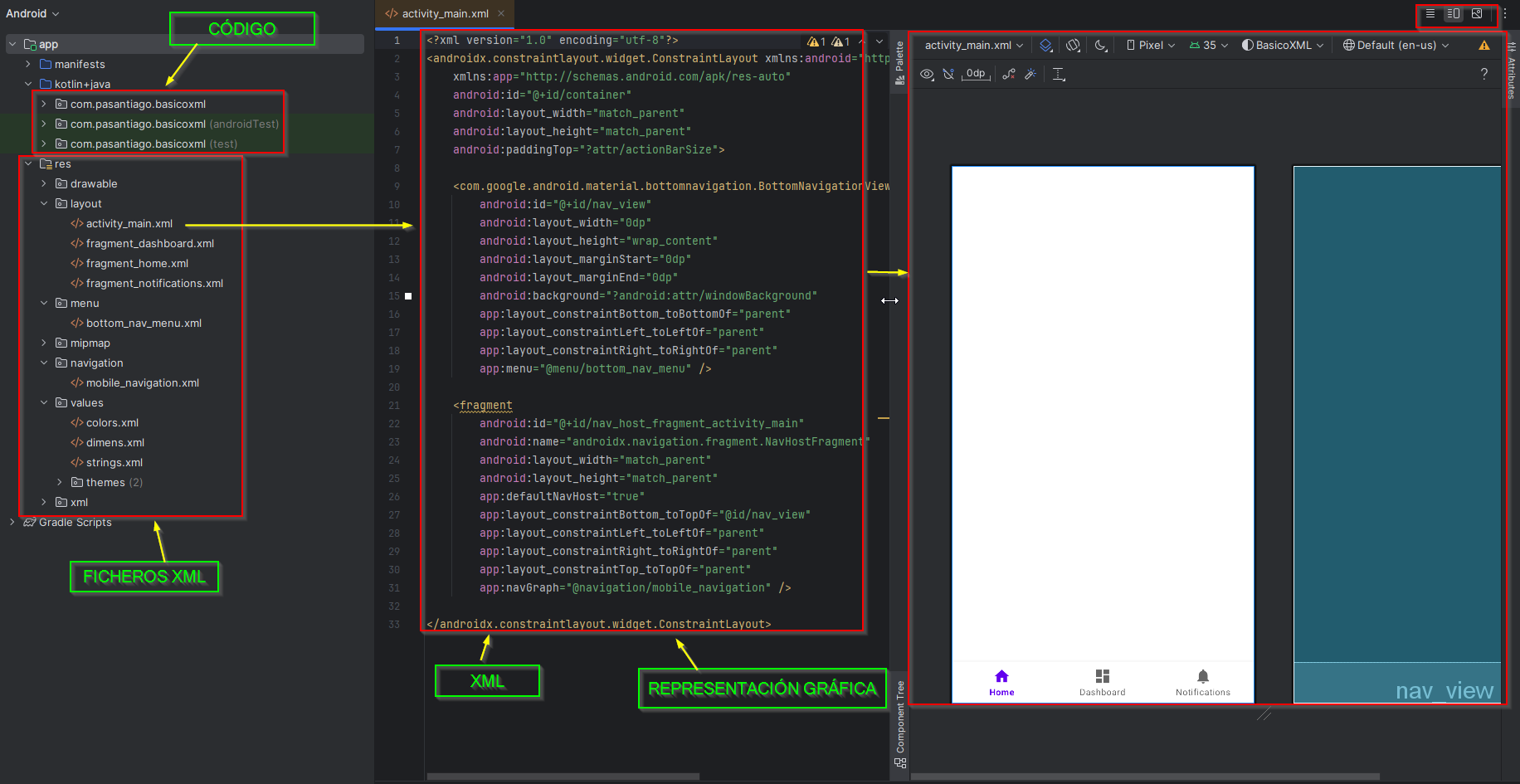
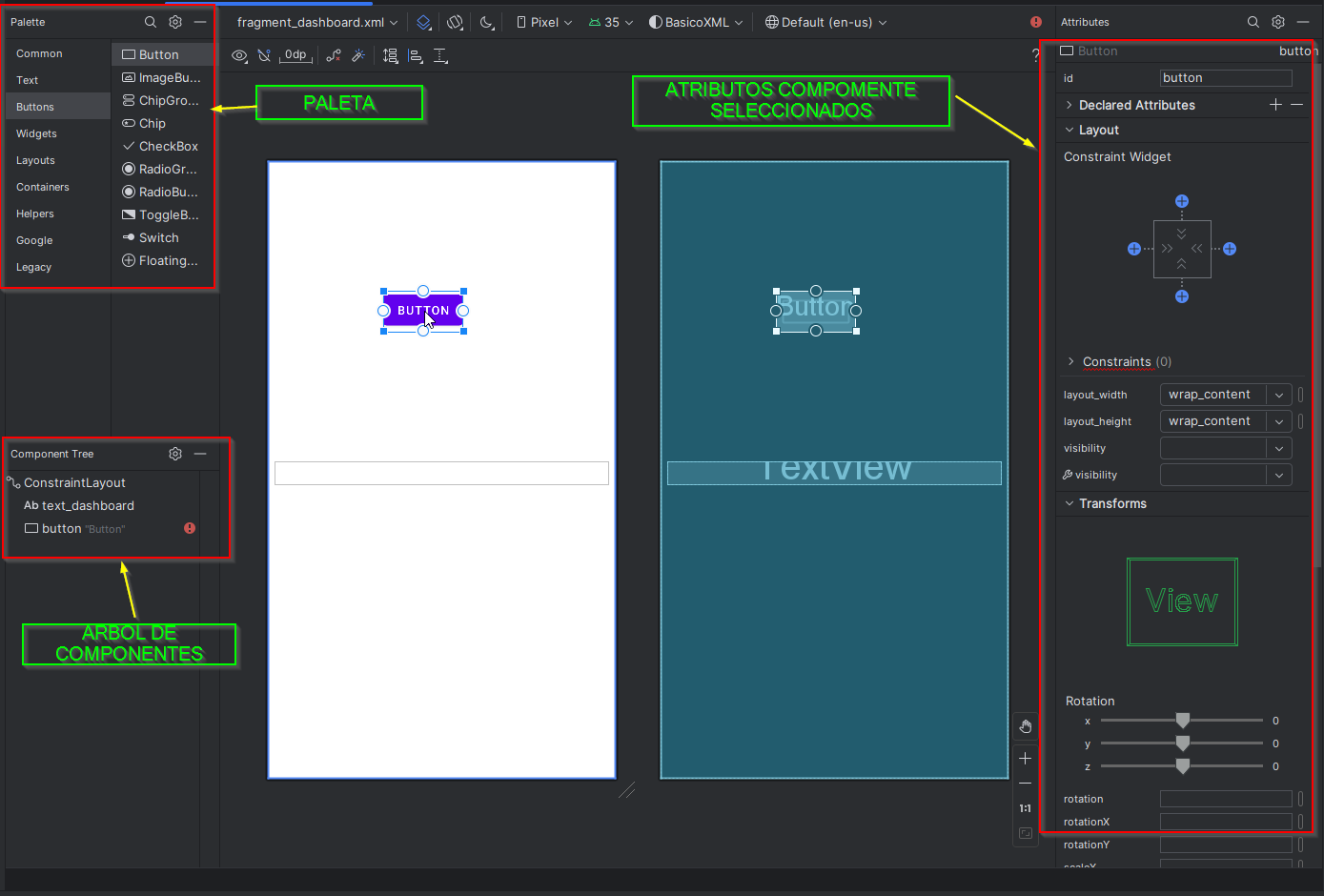
# Introducción.

1. Las tecnologías de desarrollo de aplicacione
2. s para dispositivos móviles se encuentran en constante evolución desde sus inicios, desde C/C++ corriendo directamente en el hardware, o en sistemas operativos primitivos, pasando por el desarrollo en Java con versiones concretas Java MicroEdition, hasta la estandarización de plataformas y sistemas operativos concretos como Java/Kotlin y Android o Swift para iOS.
3. 

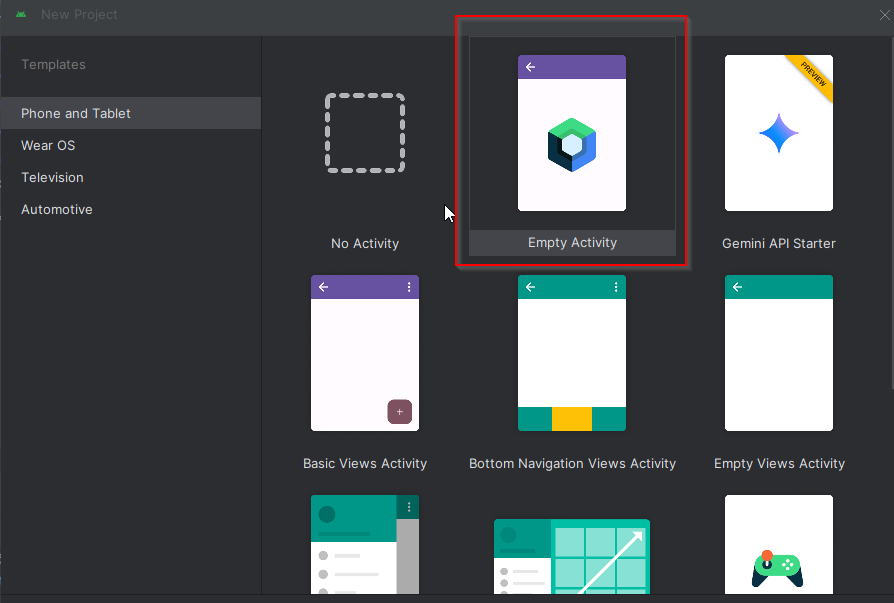
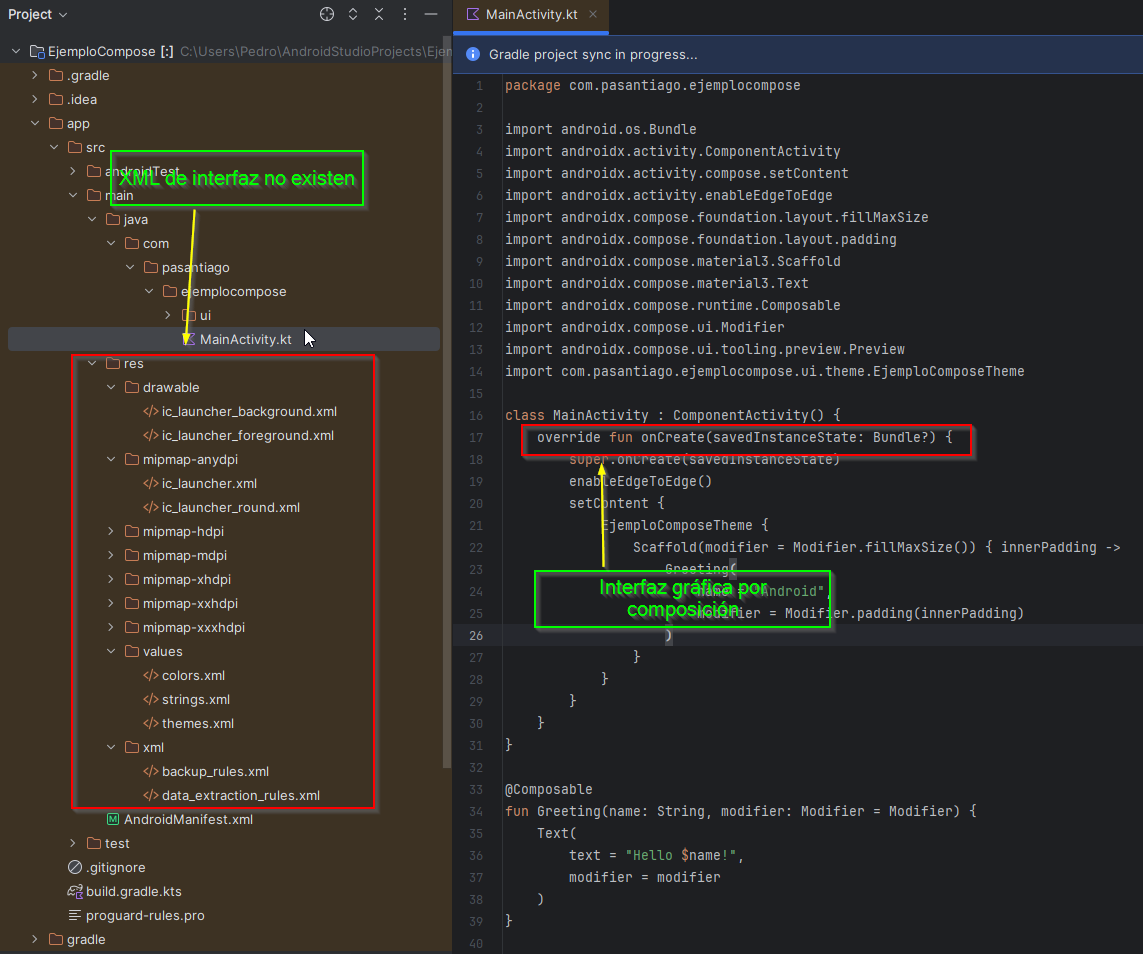
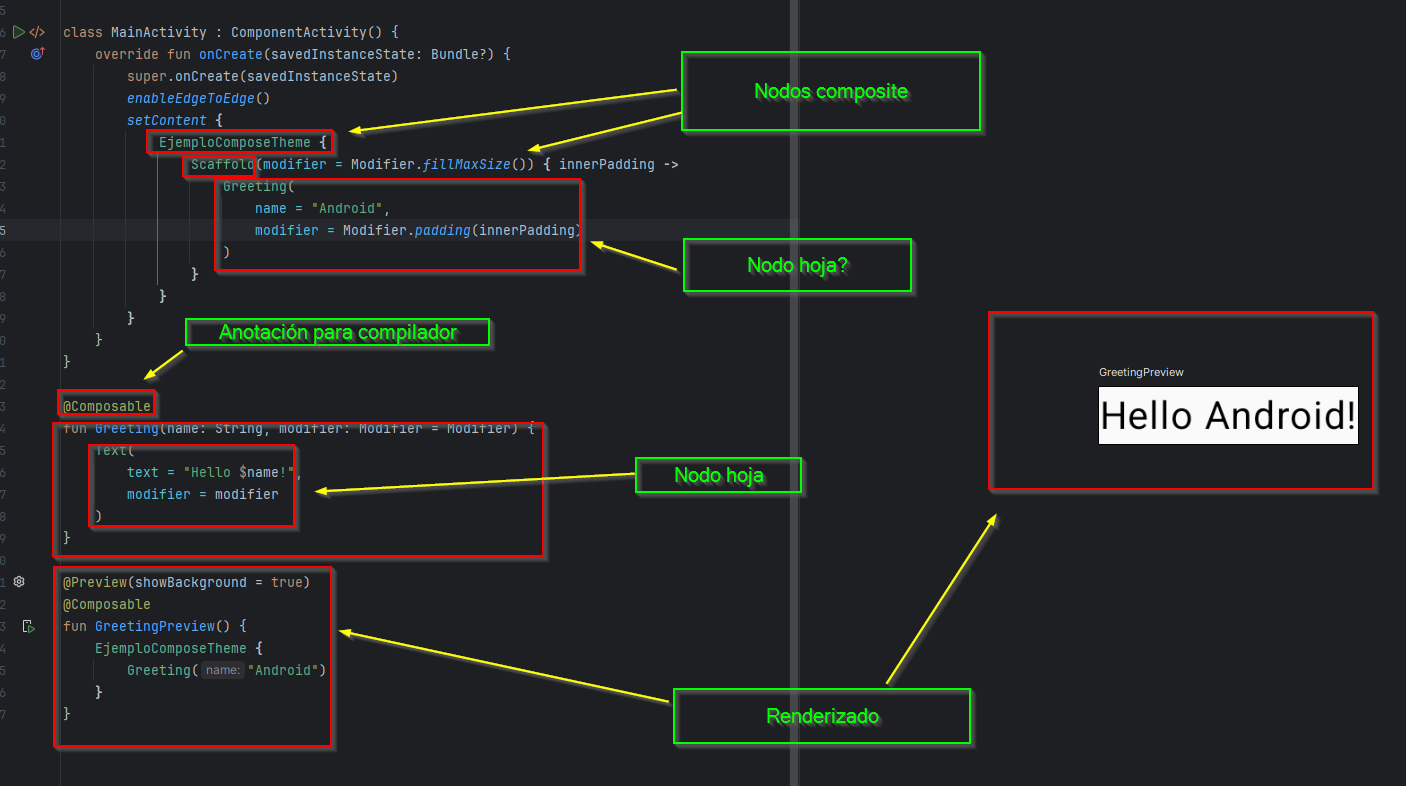
Antiguas versiones de Java en función de los dispositivos.

1. En el caso de **Android** se ha producido una evolución, comenzando con una tecnología similar a JavaFX en cuanto a la interfaz gráfica e implementado una arquitectura MVC con diferentes componentes. Actualmente esta forma de desarrollo convive con JeckPack, un conjunto de librerías, componentes y herramientas que facilitan el desarrollo de aplicaciones, que simplifica el desarrollo, aunque se pueden seguir usando en aplicaciones clásicas.
2. Una de las mayores diferencias es la definición de la interfaz gráfica, que cambia los ficheros XML por definición de interfaces de forma declarativa y reactiva (actualización automática en función de otros componentes, similar al comportamiento de las hojas de cálculo y las fórmulas con referencias) (tomando la idea de React).
3. 
4. El problema de las tecnologías para sistemas operativos concretos es la necesidad de conocer en profundidad la plataforma para realizar aplicaciones con un mínimo de calidad. Si se quiere ofrecer una aplicación en las plataformas más utilizadas las empresas han de tener personas especializadas en cada uno de los sistemas operativos, lenguaje, plataformas...con lo que el coste se multiplica, tanto en el desarrollo como en el mantenimiento. **Este problema hace que algunos sistemas operativos con un uso pequeño se encuentran que no se implementan ni tan siquiera aplicaciones ampliamente utilizadas como NextFlix o Whatshap,** un ejemplo es “Tizen” de “Samgsum” o “webOs” de LG.
5. 
6. Una solución adoptada hace ya algunos años ha sido las aplicaciones basadas en simular aplicaciones nativas sobre el navegador, pero adolecen de problemas como la velocidad y el aspecto.
7. Por último, han aparecido plataformas que permiten desarrollar en un lenguaje y generar aplicaciones para las plataformas más usadas, pudiendo seleccionar el nivel de código nativo y común en función de las necesidades, un ejemplo es **KMP** en el caso de Kotlin o MAUI de Microsoft con el lenguaje C#.
8. 

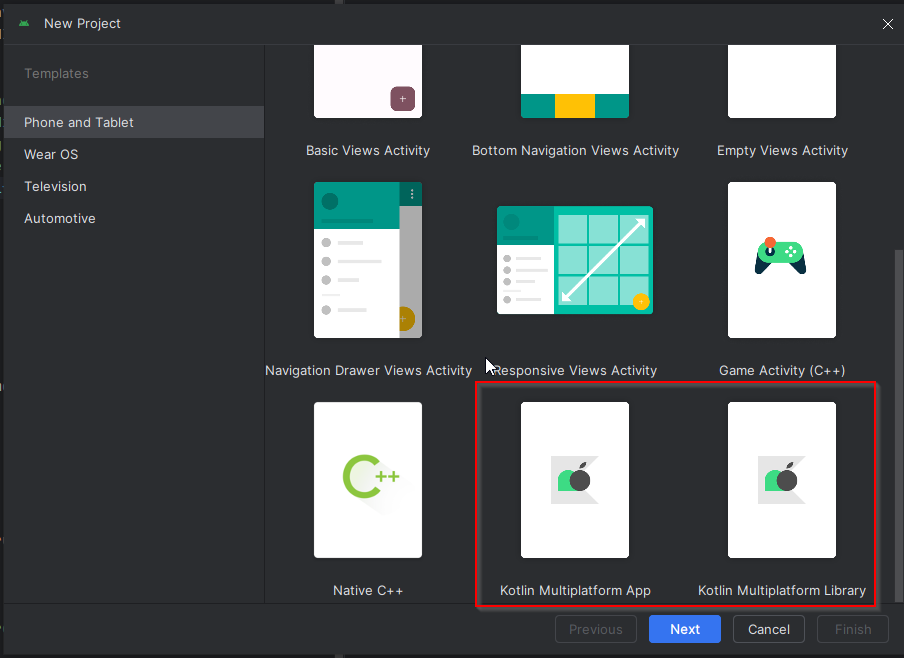
# Android clásico.

1. Android Studio posee varias plantillas para crear aplicaciones clásicas, en las que se define la interfaz con XML. En concreto las que tienen la palabra Views en el nombre:
2. 
3. Al crear un proyecto, la interfaz gráfica, la navegación, la configuración u otros valores se encuentra en XML.
4. 
5. Observar la carpeta “resource” en la que se encuentran multitud de ficheros XML.
6. Al abrir un fichero de definición de interfaz gráfica (carpeta “layout”) se tienen 3 vistas: Código, mixto y “renderizado”, además se pueden añadir nuevos elementos usando la paleta de componentes:
7. 

# Android con JeckPack Compose.

1. JeckPack facilita el desarrollo de aplicaciones, en especial la navegación y la reactividad de las interfaces gráficas, eliminando gran parte de los ficheros XML. Se basa en el uso de la composición (https://devexpert.io/herencia-vs-composicion/)
2. En Android Studio, para crear un proyecto con Compose, seleccionar la opción **“Empty Activity”**.
3. 
4. Observar en la siguiente imagen como **ya no se dispone de los ficheros XML para definir la interfaz gráfica**, ahora se en el fichero Main.tk, usando la función onCreate:
5. 
6. El diseño de la interfaz gráfica utiliza el **patrón “Composite”** y el **patrón “Observer”**. Este patrón permite crear componentes complejos a partir de componentes más sencillos.
7. Es una alternativa a la herencia en la POO, creando una estructura en forma de árbol, pero que se puede crear en tiempo de ejecución y cambiar a medida que evoluciona el programa. Si se compara con la herencia es más flexible, ya que la herencia se define en el diseño y no cambia en la ejecución.
8. Un muy buen artículo sobre el patrón “Composer”: <https://refactoring.guru/es/design-patterns/composite>. Tiene 3 elementos, la clase “Component”, de la que heredan la clase “Leaf” (hoja) que no tiene hijos, y la clase “Composite”, que posee un conjunto de "Component” (pueden ser “Leaf” u otros elementos “Composite”), existiendo por tanto recursividad y creando una estructura en árbol.
9. 

# KMP.

1. Google y JetBrean han decidido utilizar la tecnología desarrollada para Android con JeckPack para crear aplicaciones multiplataforma, aunque no todas las librerías se han portado en la actualidad.
2. Desde Android Studio se puede crear un proyecto para KMP **(instalando el plugin KMP)**, pero no dispone de todas las opciones como la generación de aplicaciones para Web o Escritorio.
3. 
4. JetBrains (al igual que Spring) ha creado una página web en la que se puede configurar un proyecto con las opciones básicas, descargarlo y abrirlo con cualquier IDE. <https://kmp.jetbrains.com/>
5. 
6. Actualmente se pueden crear aplicaciones multiplataforma para:

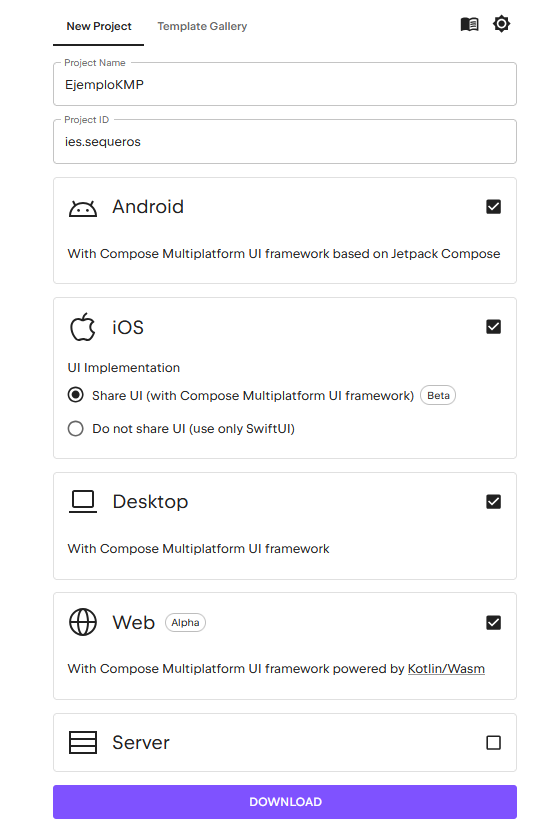
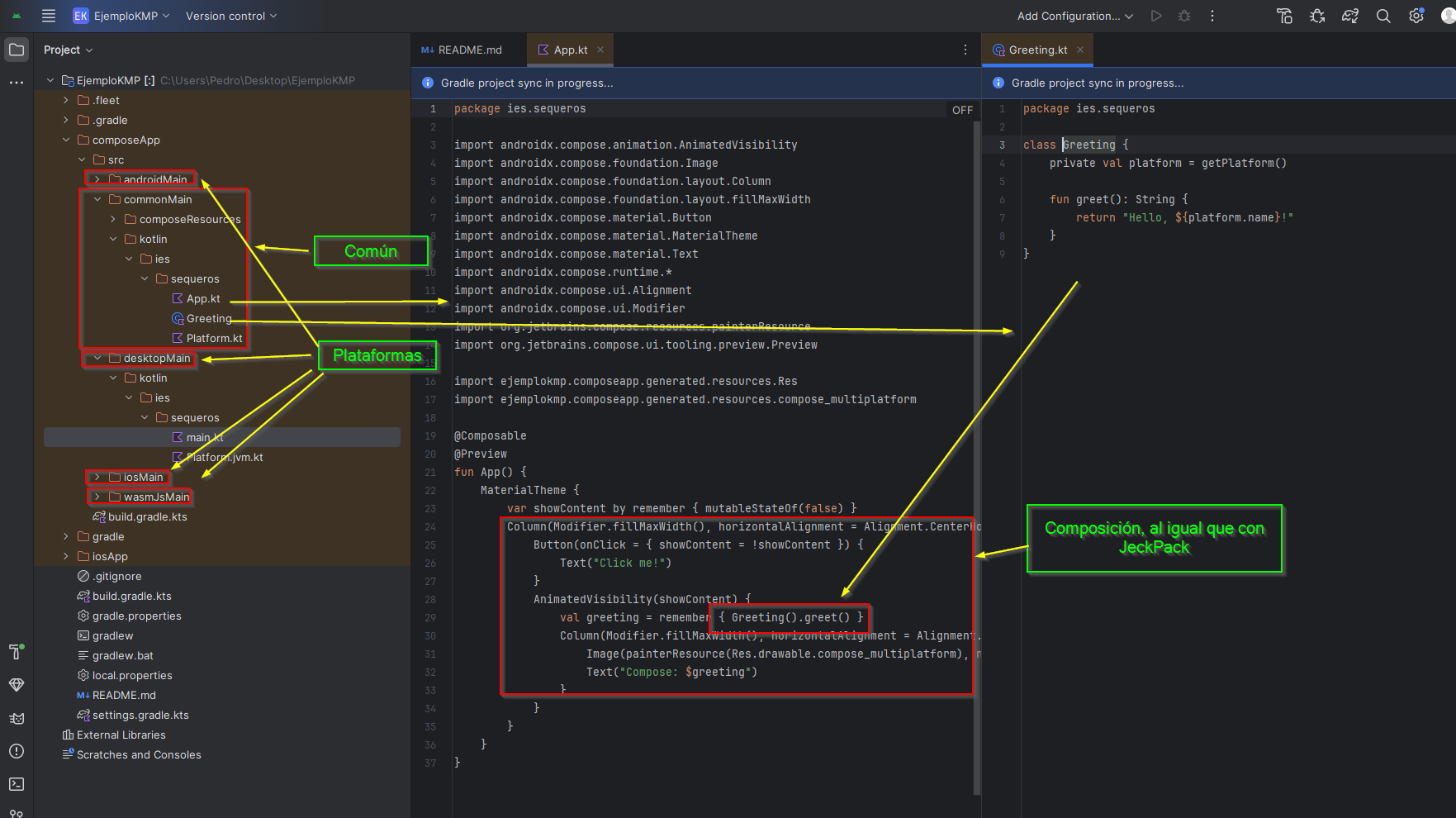
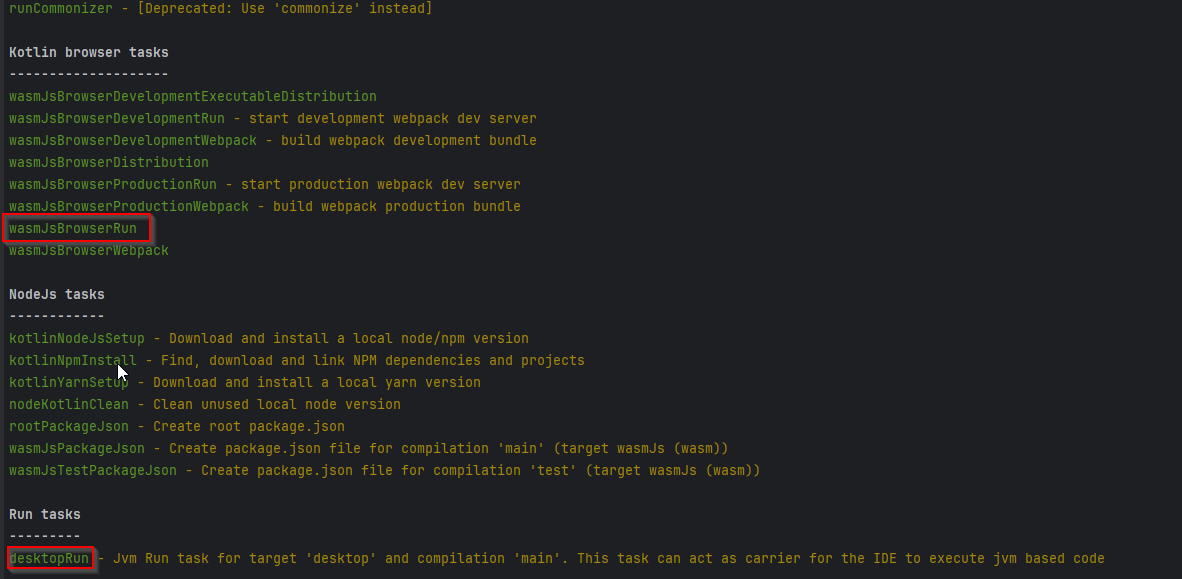
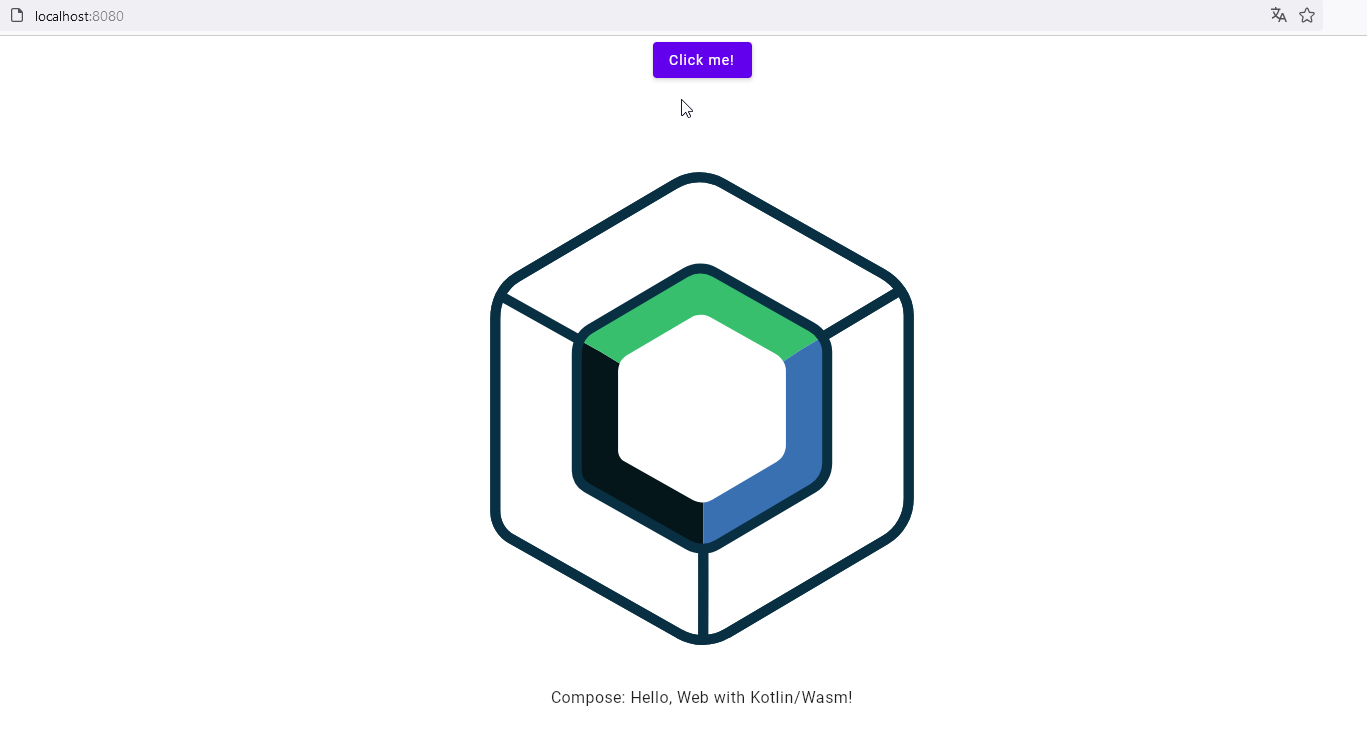
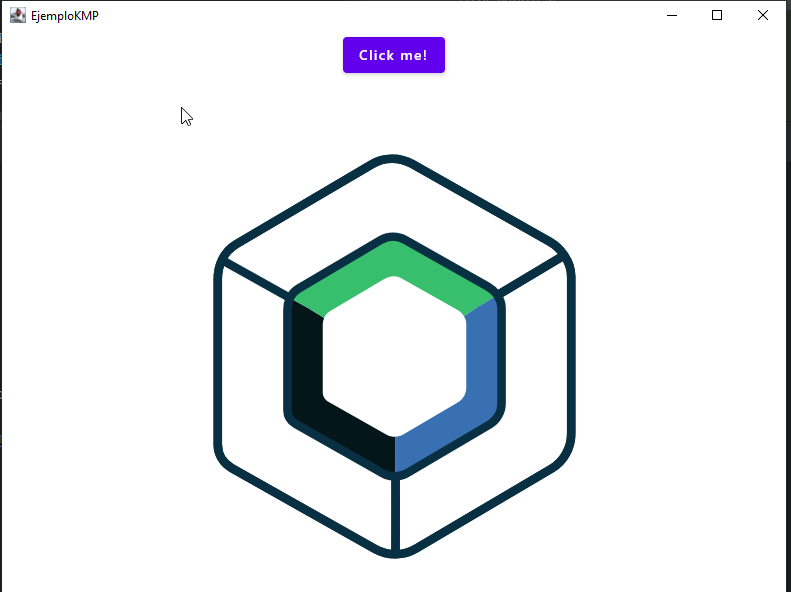
Android.

IOS.

Escritorio.

Web (Se encuentra en beta).

Servidor.

1. 
2. Al abrir el proyecto con Android Studio, se puede comprobar que se ha usado la herramienta Gradle, existiendo algunas diferencias con respecto a un proyecto de JeckPack en Android, en especial se dispone en src de carpetas individuales para cada plataforma y **una común llamada commonMain**.
3. 
4. Al listar las tareas del proyecto Gradle, se ven las diferentes tareas para ejecutar en cada una de las plataformas, generar instalables, jar, pruebas...:
5. 
6. Por ejemplo, al ejecutar wasmJsBrowserRun, se compila para wasm y se abre el navegador con la aplicación (utiliza un servidor de tipo Nodejs):
7. 
8. Si se ejecuta composeApp:run o simplemente run, se inicia la aplicación de escritorio:
9. 

# Actividades.

1. Se quiere diseñar una aplicación sencilla para calcular la cantidad de producto para tener el ph de una piscina correcta.
2. El usuario introduce el nivel de Ph medido y la dosis recomendada (en gramos) por metro cúbico del fabricante que eleva o disminuye en 0,1 unidades el Ph medido. El programa devuelve la cantidad de productos (en gramos o Kg, si son más de 1000 gramos, se pasa a Kg). Un ejemplo:
3. Se mide el Ph de una piscina de 43 metros cúbicos y el resultado es 7.0, el fabricante indica que para subir 0,1 el Ph en un metro cúbico se han de añadir 5 gramos de producto:
4. 43\*5 gramos, eleva 0,1 el PH, se ha de elevar 0,4, por tanto 43\*5\*4=860 gramos de producto.

# Actividad 1.

1. Crear un proyecto “clásico” usando la plantilla **“Empty Views Activity”**. El nombre del proyecto será **PoolTUSINICIALES**.
2. Abrir el editor de layout y crear un pequeño formulario en el que se ingresa:

m^3 de la piscina.

Lectura del Ph.

Gramos necesarios para subir 0,1 el Ph en un m^3

Una etiqueta con el resultado.

Dos botones: Calcular y borrar.

1. Ayudarse de los “layout” disponibles, en especial el Horizontal y el Vertical.
2. 
3. Crear en la actividad “MainActivity” el código necesario para realizar los cálculos. Para obtener un componente usar el método de la actividad “findViewById”, al que se le pasa el identificador indexado en el objeto estático “R.layout” con el identificador que se le ha dado en el XML, en el siguiente ejemplo, se obtiene un botón identificado por “buttonBorrar”.

var bCalcular=this.findViewById<Button>(R.id.*buttonBorrar*)

1. A partir de ese momento se pueden obtener atributos del componente o modificar su comportamiento, para añadir código al pulsarlo:

bCalcular.setOnClickListener**{** Log.d("Piscina","Se a pulsado el botón de calcular")**}**

1. Si se desea obtener el valor de un campo de texto:

var metroscubicos=this.findViewById<EditText>(R.id.*editTextMetros*)

# Actividad 2.

1. Crear ahora un proyecto con “JeckPack Compose”, seleccionando **“Empty Activity”.** El nombre de la aplicación es **PoolComposeTUSINICIALES**. Implementar la misma aplicación del cálculo del Ph.
2. Hay que recordar que se utiliza la composición de componentes, siendo un componente una función con la anotación “@Composable”. Crear un componente usando una función anotada con “@Composable”, en la que se crean variables “escuchables”, de forma que la interfaz cambia de forma automática ante los cambios:

class MainActivity : ComponentActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 *enableEdgeToEdge*()  
 *setContent* **{**  
PoolComposeTheme **{**  
Scaffold(modifier = Modifier.*fillMaxSize*()) **{** innerPadding **->**  
Formulario(  
 Modifier  
 .*padding*(innerPadding)  
 .*fillMaxSize*())  
 **}**  
 **}**  
 **}**  
}  
}  
@Composable  
fun Formulario(modifier: Modifier = Modifier) {  
 var metrosCubicos by remember **{** *mutableStateOf*("") **}**  
var ph by remember **{** *mutableStateOf*("") **}**  
var gramos by remember **{** *mutableStateOf*("") **}**  
Column(modifier= Modifier.*fillMaxSize*().*padding*(64.*dp*)) **{**  
Row(Modifier.*fillMaxWidth*(1.0f)) **{**  
TextField(  
 modifier = Modifier.*fillMaxWidth*(),  
 value = metrosCubicos,  
 label = **{** Text("Metros cúbicos") **}**,  
 onValueChange = **{**  
metrosCubicos = **it**  
 **}**,  
 keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number)  
 )  
 **}**  
 **}**  
}

1. No es necesario pulsar un botón para cambiar el resultado, ya que las variables son de tipo mutableStateOf (patrón Observer), sino que al cambiar el valor de alguna de ella se actualiza. Un ejemplo, que asocia el valor de la variable “metrosCubicos” al texto de un cuadro de texto.

Row() **{**  
Text(if(metrosCubicos.*isNotEmpty*())metrosCubicos else "VACIO", modifier = modifier.*background*(Color.Red))  
**}**

# Actividad 3.

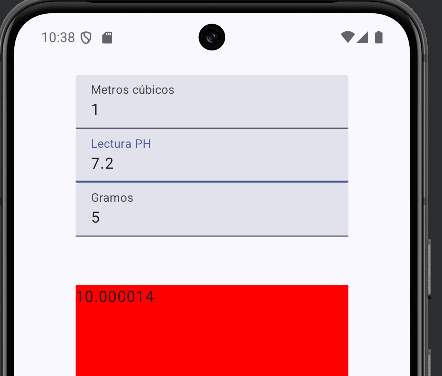
1. Comparar las dos formas de crear aplicaciones, diseñando con XML y añadiendo código a los botones, o usando “Compose” de forma declarativa, aspectos como, por ejemplo:

Cantidad de código.

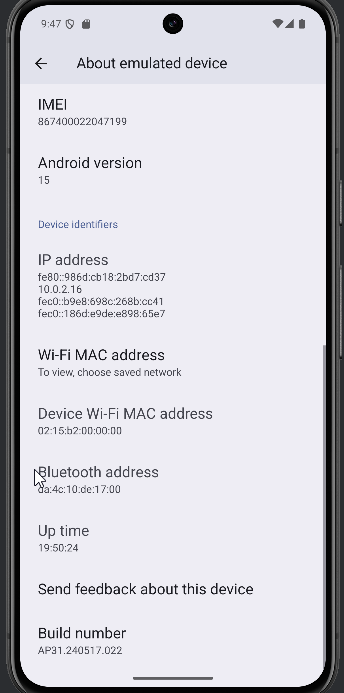
Tiempo de diseño.

Dificultad técnica.

Expresividad.

1. 

# Actividad 4.

1. Las pruebas de las aplicaciones no solo se limitan a las pruebas unitarias, existen más tipos de pruebas, entre ellas pruebas de las interfaces gráficas. Hace algunos años este tipo de pruebas se realizaba de forma manual, pero en la actualidad y siguiendo la tendencia, se han automatizado, por ejemplo, en las aplicaciones web la librería Selenium es ampliamente utilizada.
2. En esta actividad se utiliza la librería UI Automator, aunque existen otras como Espresso, Robolectric o integradas en JeckPack Compose. La documentación oficial de Android para trabajar con UI Automator es: <https://developer.android.com/training/testing/instrumented-tests/ui-tests?hl=es-419>
3. En primer lugar, incluir la dependencia:
4. androidTestImplementation(“androidx.test.uiautomator:uiautomator:2.3.0-alpha03”)
5. Activar las opciones de depuración en el emulador. Settings" > "About emulated device" y tocando varias veces en "Build number" hasta que aparezca el mensaje que indica que las opciones de desarrollador están habilitadas:
6. 
7. Crear el caso de prueba en la carpeta AndroidTest, por ejemplo, el siguiente código comprueba si existe un TextField a partir de su Id:

package com.pasantiago.poolpass  
import android.content.Intent  
import androidx.test.platform.app.InstrumentationRegistry  
import org.junit.Before  
import androidx.test.uiautomator.UiDevice  
import androidx.test.uiautomator.By  
import androidx.test.uiautomator.UiObjectNotFoundException  
import androidx.test.uiautomator.UiSelector  
import androidx.test.uiautomator.Until  
import org.junit.Assert.assertEquals  
import org.junit.Test  
import org.junit.runner.RunWith  
  
@RunWith(androidx.test.ext.junit.runners.AndroidJUnit4::class)  
class EjemploTest { private val APP\_PACKAGE ="com.pasantiago.poolpass" // Paquete de la aplicacion  
 private val LAUNCH\_TIMEOUT = 5000L  
 private lateinit var mDevice: UiDevice  
  
 @Before  
 fun setUp() {  
 // Inicializa el UiDevice para comunicarse con el dispositivo/emulador.  
 mDevice = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation())  
  
 // Presiona el botón de inicio para asegurarse de estar en la pantalla principal.  
 mDevice.pressHome()  
  
 // Inicia la aplicación.  
 val context = InstrumentationRegistry.getInstrumentation().*targetContext*  
val intent = context.*packageManager*.getLaunchIntentForPackage(APP\_PACKAGE)  
 intent?.addFlags(Intent.*FLAG\_ACTIVITY\_CLEAR\_TASK*) // Limpia tareas anteriores.  
 context.startActivity(intent)  
  
 // Espera a que la aplicación se abra.  
 mDevice.wait(Until.hasObject(By.pkg(APP\_PACKAGE).depth(0)), LAUNCH\_TIMEOUT)  
 }  
  
 @Test  
 @Throws(UiObjectNotFoundException::class)  
 fun validarExisteTextField() {  
 var textField = mDevice.findObject(UiSelector().resourceId("$APP\_PACKAGE:id/editTextMetros"))  
  
 // Verifica si el cuadro de texto existe.  
 val actualText = textField.*text*  
// Verifica si el cuadro de texto tiene el valor esperado.  
 assertEquals("El cuadro de texto no está presente", true, textField.exists())  
   
 }  
}

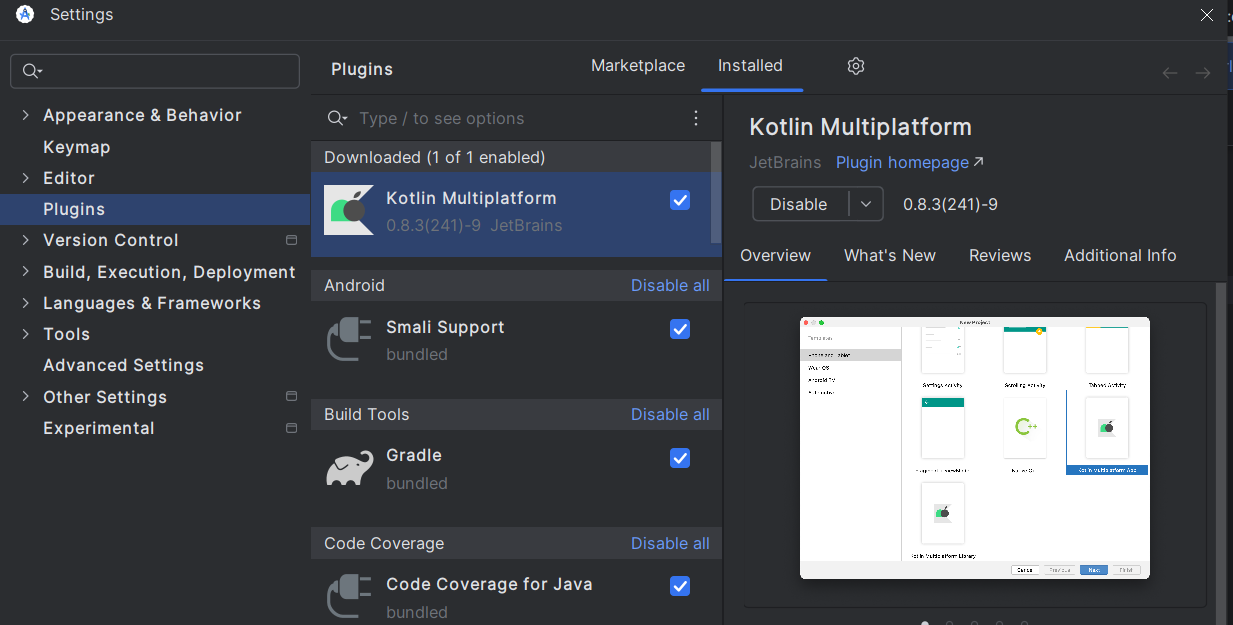
1. Crear 3 casos de prueba para:

Comprobar que los campos de texto existen.

Comprobar de darle valores positivos a los 3 campos y ver que el resultado es correcto.

Comprobar que si el Ph es mayor de 7.4 el resultado es negativo y correcto.

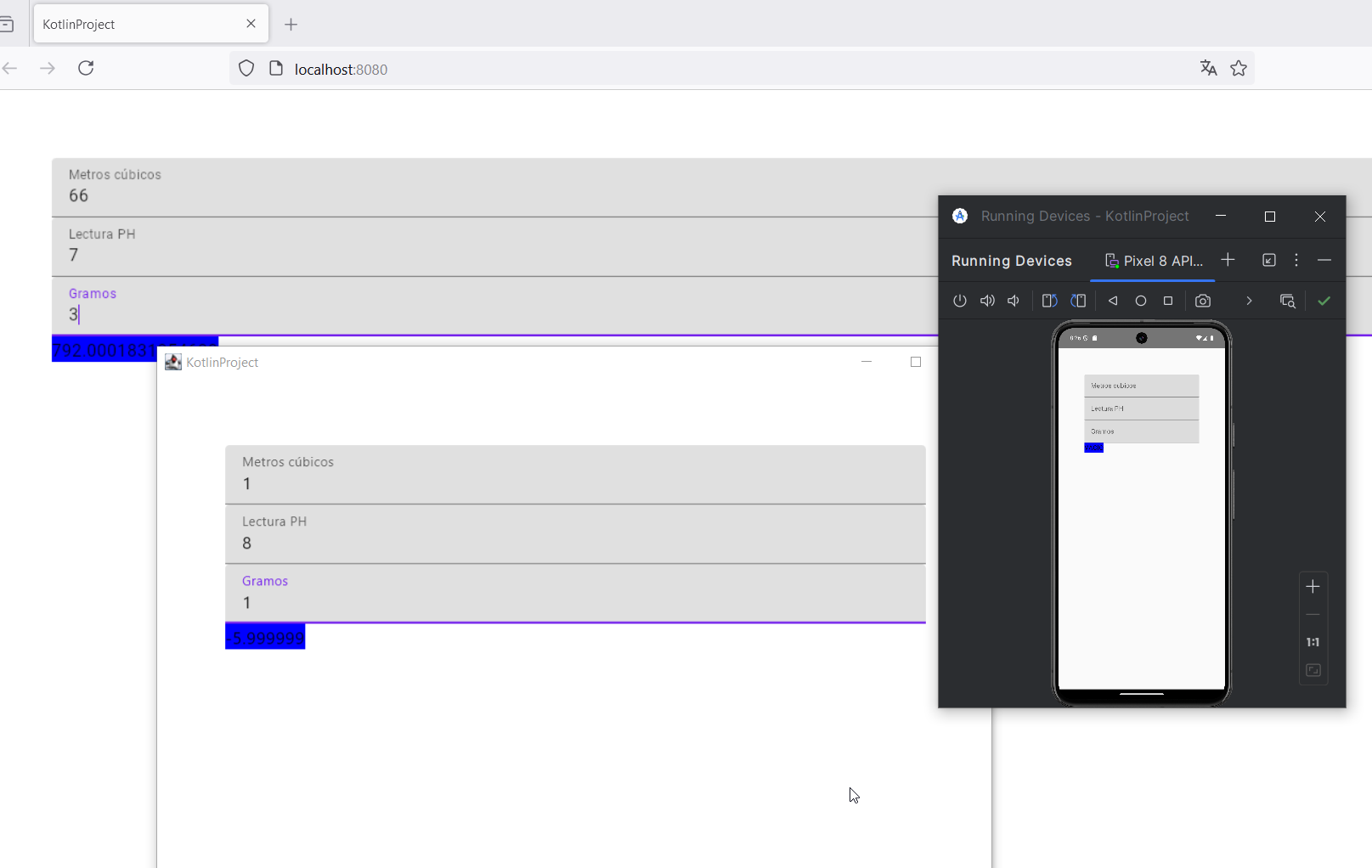
# Actividad 5.

1. Crear una nueva aplicación, pero ahora con un proyecto de KMP para Android, Escritorio y Web. El proyecto se llama **PoolKMPTusIniciales**.
2. Instalar el plugin KMP para Android Studio:
3. 
4. Crear o importar el componente creado en el punto 5.3.
5. Insertarlo en la función App del fichero App.kt de commondMain
6. Generar y realizar capturas de pantalla de la aplicación funcionando en las 3 plataformas.

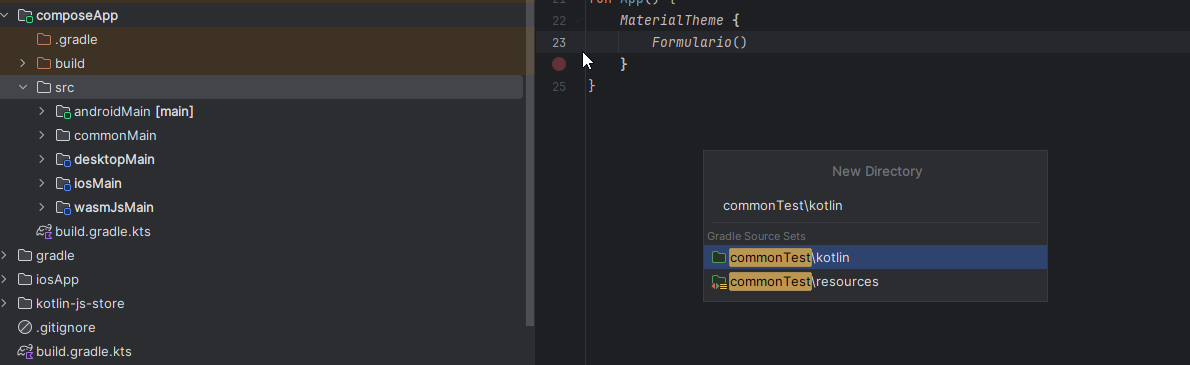
Para la versión de Android usar Adroid Studio.

Para la versión web usar la tarea de Gradle wasmJsBrowserRun.

Para la versión de escritorio usar la tarea de Gradle run.

1. 

# Actividad 6.

1. Las pruebas en JeckPack Compose son algo diferentes de las de la programación clásica, integrándose en la librería la tecnología necesaria. A definirse componentes individuales, las pruebas se realizan instanciando estos componentes sin necesidad de iniciar toda la aplicación (aunque es posible iniciar actividades). La documentación oficial para realizar pruebas con “Compose”: <https://developer.android.com/develop/ui/compose/testing?hl=es-419>
2. Para KMP se pueden realizar pruebas en cada una de las plataformas de forma individual, pero se han de utilizar librerías concretas para cada una de ellas. Actualmente (finales 2024) la API se encuentra en fase experimental. En el siguiente enlace se explica como configurar las pruebas. <https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-test.html#-faexsk_56>
3. Dado que el proceso es muy similar, se ejemplifica con KMP, siguiendo los pasos indicados en la web anterior.
4. **Configuración de proyecto KMP.**
5. Crear el directorio commonTest/kotlin en src.
6. 
7. Modificar el fichero “buid.gradle.kts” para incluir las librerías necesarias:

*sourceSets* **{**  
val desktopMain by *getting*  
val desktopTest by *getting*  
 *commonTest*.*dependencies* **{**  
implementation(kotlin("test"))  
  
 @OptIn(org.jetbrains.compose.ExperimentalComposeLibrary::class)  
 implementation(*compose*.uiTest)  
 **}**  
desktopTest.dependencies **{**  
implementation(*compose*.desktop.currentOs)  
 **}**

1. Para poder ejecutarlo en el emulador, se han de realizar algunas modificaciones más:

androidTarget **{**  
@OptIn(ExperimentalKotlinGradlePluginApi::class)  
 compilerOptions **{**  
jvmTarget.set(JvmTarget.*JVM\_11*)  
 **}**  
@OptIn(ExperimentalKotlinGradlePluginApi::class)  
 instrumentedTestVariant.sourceSetTree.set(KotlinSourceSetTree.test)  
  
**}**

1. \*Dar un poco de tiempo a que encuentre kotlinsourceSetTree.test
2. Añadir el siguiente código a android->defaultConfig:

defaultConfig **{**  
applicationId = "ies.sequeros"  
 minSdk = *libs*.*versions*.*android*.*minSdk*.get().*toInt*()  
 targetSdk = *libs*.*versions*.*android*.*targetSdk*.get().*toInt*()  
 versionCode = 1  
 versionName = "1.0"  
 testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"  
  
**}**

1. Por último, añadir las siguientes dependencias a nivel global, dentro del mismo fichero:

*dependencies* **{**  
*androidTestImplementation*("androidx.compose.ui:ui-test-junit4-android:1.6.8")  
 *debugImplementation*("androidx.compose.ui:ui-test-manifest:1.6.8")  
**}**

1. Con esta configuración ya es posible realizar pruebas de interfaces gráficas en KMP
2. **Creación de pruebas.**
3. Se crea un fichero, por ejemplo, llamado “PruebaEjempo,kt” en src/commonTest/kotlin. Para facilitar la escritura se puede copiar el ejemplo de la documentación y modificarlo:

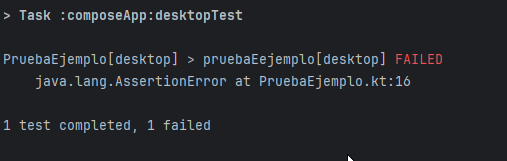
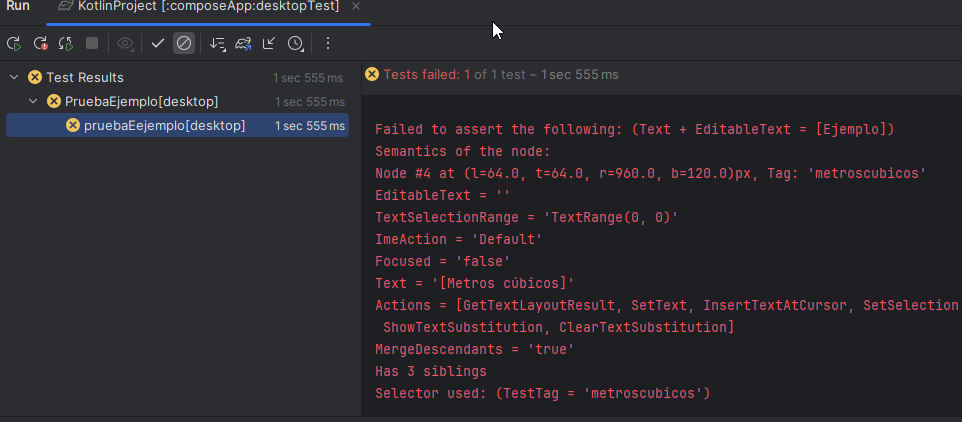
import androidx.compose.ui.test.\*  
import ies.sequeros.Formulario  
import kotlin.test.Test  
class PruebaEjemplo {  
 @OptIn(ExperimentalTestApi::class)  
 @Test  
 fun pruebaEejemplo() = *runComposeUiTest* **{**  
  
setContent **{**  
*Formulario*()  
 **}**  
//se busca el campo de texto por tag y se comprueba que tenga el texto "Ejemplo", tiene que dar fallo.  
 *onNodeWithTag*("metroscubicos").*assertTextEquals*("Ejemplo")  
  
 **}**  
}

1. Además, se modifica el componente para poder obtenerlo por la etiqueta:

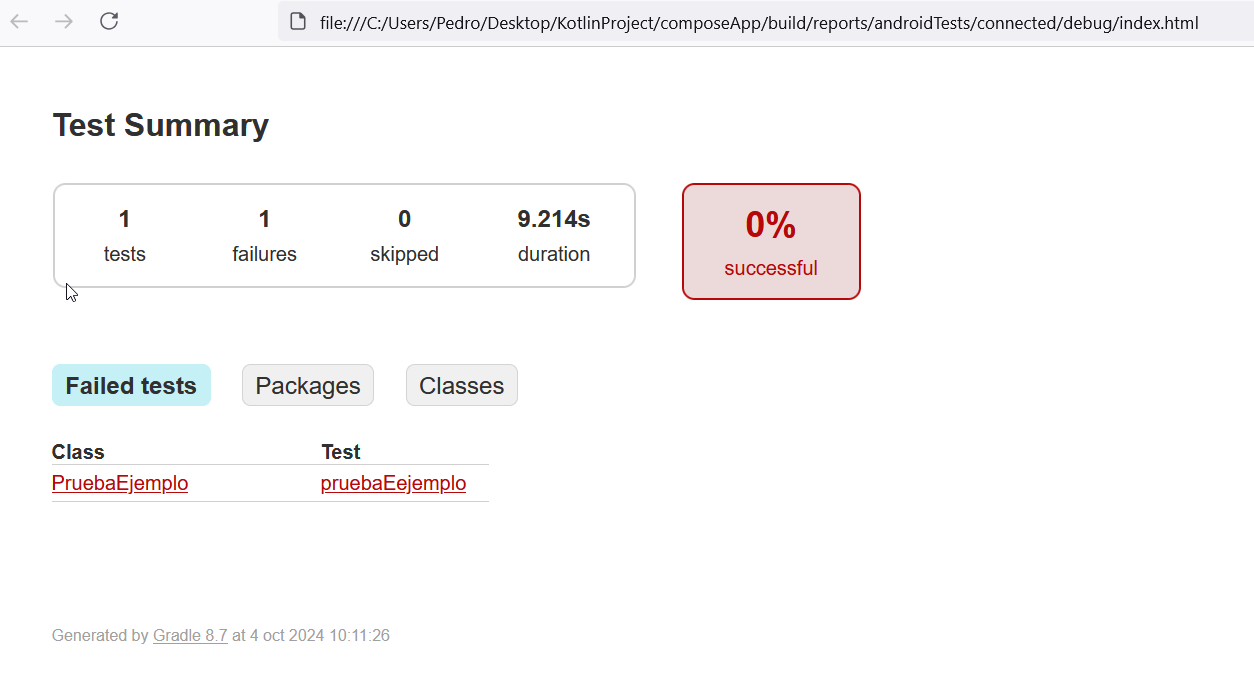
*TextField*(  
 modifier = Modifier.*fillMaxWidth*().*testTag*("metroscubicos"),  
 value = metrosCubicos,  
 label = **{** *Text*("Metros cúbicos") **}**,  
  
 onValueChange = **{**  
metrosCubicos = **it**  
 **}**,  
 keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number)  
)

1. **Ejecutar las pruebas.**
2. Para la ejecución de las pruebas se utilizan tareas de gradle:

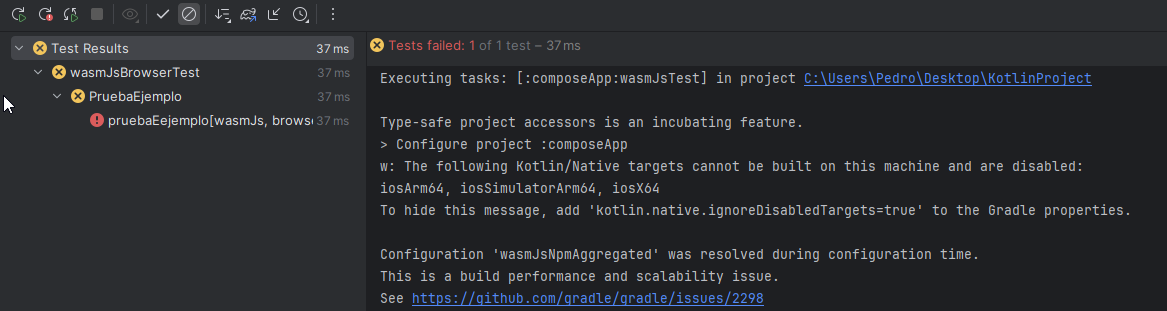
**Escritorio**: ./gradlew :composeApp:desktopTest

1. 
2. También se puede ejecutar desde Android Studio y ver de forma más amigable los resultados:
3. 
4. 

**Emulador Android**: ./gradlew :composeApp:connectedAndroidTest

1. Genera un informe en HTML con los resultados (aparece la URL en el terminal):
2. 

**Wasm (Web)**: ./gradlew :composeApp:wasmJsTest

1. 

**IOs:** /gradlew :composeApp:iosSimulatorArm64Test

1. **Destacar que estas pruebas son de la parte común, pudiendo definir pruebas concretas para plataformas concretas y código exclusivo, por ejemplo, la funcionalidad de ciertos botones de iOS, o la lectura de huellas digitales.**
2. Repetir el ejercicio 4 pero con Compose y KMP.

# Entrega.

1. Se realizará en Aules, con un único fichero comprimido.
2. Se incluye:
3. Proyectos comprimidos y si compilar y limpios.
4. Memoria de la práctica comentada en formato PDF.