



CAPÍTULO 7.

Programación en código nativo

Por MIGUEL GARCÍA PINEDA

Todo el código utilizado hasta ahora en el SDK ha sido desarrollado en Java. El desarrollo en Java presenta grandes ventajas, siendo la más importante que el código obtenido tras compilar el programa (conocido como *bytecode*) puede ejecutarse en cualquier equipo, independientemente del tipo de procesador. Sin embargo, también presenta inconvenientes. El más importante es la velocidad de ejecución. Los *bytecode* no pueden ser ejecutados directamente por el procesador, sino que han sido pensados para ser ejecutados en una máquina virtual (JVM). Por lo tanto, estas instrucciones tendrán que ser interpretadas por *software*, lo que provocará un retardo mayor a diferencia de si estas instrucciones fueran ejecutadas directamente por el *hardware*.

El desarrollo de aplicaciones en Android no está limitado a Java. También vamos a poder crear aplicaciones formadas por código máquina de un procesador específico. A este tipo de código se le conoce como código nativo. Utilizar esta opción limita nuestra aplicación, dado que solo se podrá ejecutar en un tipo concreto de procesador. No obstante, prácticamente todos los teléfonos móviles y tabletas que se distribuyen con Android en la actualidad tienen el procesador ARM, por lo que esta limitación no es importante. En un futuro es posible que esto cambie. Por ejemplo, algunos *smartTV* permiten ejecutar aplicaciones Android. Sin embargo, estos dispositivos se basan en otro tipo de procesador y, por lo tanto, no son compatibles con el mismo código nativo para ARM, sino que deberían ser compilados para otro tipo de procesador. En conclusión, si nuestra aplicación tiene código nativo, solo funcionará con un tipo de procesador. Si por el contrario está escrita íntegramente en Java, funcionará en todos los terminales independientemente del procesador.

Aunque utilizar código nativo en Android tiene sus inconvenientes, en ciertos casos puede resultar adecuado. Si nuestro *software* tiene que ejecutar algoritmos de cómputo masivo, su ejecución en la máquina virtual Java puede ser demasiado lenta y, por lo tanto, será más conveniente compilarlo en el código nativo del procesador.

Para poder desarrollar aplicaciones en código nativo debemos utilizar una herramienta conocida como Android NDK (*Native Development Kit*). Es un conjunto de herramientas que permite incluir en nuestra aplicación código escrito en C o C++ que será compilado a código nativo. NDK se distribuye separadamente del SDK. A lo largo de este capítulo se describirá cómo instalar esta herramienta y cómo utilizarla mediante ejemplos.

La organización que vamos a seguir en esta unidad es la siguiente: en primer lugar, haremos una descripción de las características que nos aporta el entorno de desarrollo de Android NDK, cuándo utilizar código nativo y cuál es el contenido de Android NDK. Seguidamente, se mostrará cómo instalar el entorno de desarrollo de código nativo para Android. Después, será explicado el funcionamiento y la estructura básica de Android NDK. A continuación, veremos la interfaz JNI para poder hacer llamadas desde código Java a código en C/C++ y viceversa; este aspecto será explicado mediante un ejemplo. Por último, presentamos dos ejercicios paso a paso más complejos, para observar el rendimiento de la programación nativa y el procesado de imágenes.

**Objetivos:**

- Describir las características básicas del kit de desarrollo de Android.
- Conocer cuándo debe utilizarse código nativo en lugar de código Java.
- Analizar el contenido de Android NDK. Conocer las herramientas de desarrollo que aporta al programador, la documentación existente y ejecutar algunas aplicaciones de ejemplo.
- Instalar el kit de desarrollo de código nativo (Android NDK).
- Describir los elementos necesarios para el desarrollo de una aplicación nativa en Android.
- Desarrollar varias aplicaciones para aprender el funcionamiento de la interfaz nativa de Java (JNI).
- Manejar el Android NDK para el procesamiento de imágenes.

7.1. Android NDK

Android NDK es un conjunto de herramientas que permite incorporar los componentes que hacen uso de código nativo en las aplicaciones de Android.

Las aplicaciones de Android se suelen ejecutar en la máquina virtual Dalvik. El NDK permite implementar parte de sus aplicaciones utilizando código nativo a partir de lenguajes de programación como son C y C++. Esto puede proporcionar beneficios para ciertas clases de aplicaciones, ya sea por la reutilización de código existente y/o por el aumento de velocidad en algunas aplicaciones. Android NDK es:

- Un conjunto de herramientas para crear archivos que se utilizan para generar bibliotecas de código nativo en C y C++.
- Una manera de integrar las bibliotecas nativas en un archivo de paquete de aplicaciones (.apk) que posteriormente puede ser ejecutado en dispositivos Android.
- Un conjunto de cabeceras y bibliotecas nativas del sistema que se apoyarán en todas las futuras versiones de la plataforma Android, a partir de Android 1.5.

La última versión de la NDK es compatible con los conjuntos de instrucciones ARM, x86, MIPS, etc.

Lo más frecuente hoy en día es compilar para todas las arquitecturas soportadas por Android NDK. De esta forma estaremos haciendo nuestra aplicación más compatible con más dispositivos, pero a su vez un poco más pesada debido a que existirá una biblioteca para cada arquitectura de procesador. Al realizar un programa en código nativo, el desarrollador puede crear una aplicación para uno o varios conjuntos de instrucciones (realizando un cambio en el build.gradle o incluyendo dicha información en el fichero Application.mk). La construcción para las diferentes arquitecturas se realizará al mismo tiempo y todo será almacenado en la aplicación *final.apk*.

Además Android NDK ofrece encabezados estables para *libc* (la biblioteca de C), *libm* (la biblioteca matemática), OpenGL ES (biblioteca de gráficos 3D), la interfaz JNI y otras bibliotecas.

7.1.1. Cuándo utilizar código nativo

El desarrollo de aplicaciones con Android NDK no aporta beneficios en la mayoría de los casos. Como desarrollador, necesitamos encontrar el equilibrio entre beneficios e inconvenientes, ya que el uso de código nativo no se traduce en un aumento de rendimiento automático debido a la mejora en la potencia de los procesadores y la buena integración del API de Android; pero siempre aumenta la complejidad de la aplicación. En general, solo se debe utilizar código nativo si es esencial para su aplicación, no solo porque se prefiera programar en C o C++.

Ejemplos típicos y buenos candidatos para el desarrollo mediante Android NDK son aplicaciones con un uso intensivo de CPU, como el procesamiento de la señal, la simulación de la física, etc. El simple hecho de



recodificar un método para ejecutarlo en C no siempre da lugar a un gran aumento de rendimiento. Para saber si debemos desarrollar una aplicación en código nativo o no, lo primero que tenemos que hacer es evaluar las necesidades y comprobar si la API de Android puede proporcionarnos la funcionalidad que necesitamos. Si es así, lo mejor será seguir la API; en caso contrario, podemos ir pensando en desarrollar parte de nuestra aplicación en código nativo. Android NDK, sin embargo, puede ser una forma eficaz de volver a utilizar un gran cuerpo de código existente en C/C++.

Existen dos maneras de utilizar el código nativo en Android:

- Desarrollar nuestra aplicación utilizando el entorno y SDK de Android, y utilizar la interfaz nativa de Java (JNI) para acceder a las API proporcionadas por Android NDK. Esta técnica es la más utilizada ya que proporciona la comodidad de utilizar el entorno de Android; pero además presenta la opción de escribir código nativo cuando sea necesario.
- Desarrollar una aplicación totalmente nativa. Si trabajamos de esta forma, las devoluciones de llamada (*callback*) de ciclo de vida tendrán que ser en código nativo. El SDK de Android proporciona la clase `NativeActivity`, que es una clase de conveniencia que notifica el código nativo de cualquier actividad de las devoluciones de llamada de ciclo de vida (`onCreate()`, `onPause()`, `onResume()`, etc). Se pueden implementar los *callbacks* en el código nativo para controlar cuándo se producen estos eventos.

No se puede acceder a las funciones como los servicios y proveedores de contenido de forma nativa, por lo que si desea utilizar las API o cualquier otra utilidad de Android, se deberá usar el JNI.



Enlaces de interés:



Información sobre la clase `NativeActivity`:

<http://developer.android.com/reference/android/app/NativeActivity.html>

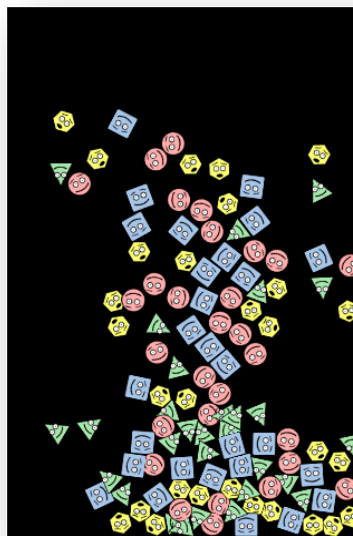
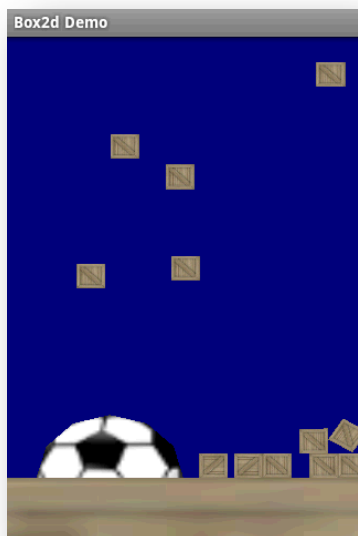


Ejercicio: *Diferenciación entre programas desarrollados en código nativo y/o Java.*

Para comparar las diferencias de rendimiento entre desarrollar ciertos algoritmos en Java y en código nativo te recomendamos que realices el siguiente ejercicio. Vamos a comparar dos aplicaciones en Android que utilizan la biblioteca Box2D. Box2D es una biblioteca que permite emular interacciones de objetos físicos en un espacio 2D. Esta magnífica biblioteca es de código abierto y ha sido desarrollada por Erin Catto.

1. Descarga de Android Market la aplicación *Box2d Demo* . Esta aplicación está basada en la biblioteca JBox2D, una transcripción a Java de la biblioteca de Erin Catto. Pulsa sobre la pantalla para introducir nuevos objetos y observa cómo estos caen atraídos por la gravedad y chocan entre ellos.
2. Descarga ahora del Market la aplicación *AndEngineExamples* . Selecciona la opción *Physics* y luego *Using Physics*. Igual que antes, pulsa sobre la pantalla para introducir nuevos objetos. Esta aplicación está basada en el motor de Juegos *AndEngine* que incorpora la biblioteca Box2D; pero esta vez compilada en código nativo.
3. Compara la velocidad de ejecución de ambas aplicaciones. Para ello introduce un número elevado de objetos en la primera aplicación. Observa como a partir de 10 objetos empieza a moverlos con lentitud. Utiliza ahora la aplicación *AndEngineExamples* y observa como la aplicación no pierde prestaciones aunque se introduzcan más de 100 objetos.

Nota: Es posible que si utilizas un dispositivo muy rápido no aprecies esta diferencia de velocidad. En dispositivos antiguos la diferencia es abrumadora.



7.1.2. Contenido de Android NDK

El Android NDK contiene las API, documentación y aplicaciones de ejemplo que nos ayudarán a desarrollar aplicaciones en código nativo.

7.1.2.1. Las herramientas de desarrollo

Android NDK incluye un conjunto de herramientas cruzadas (compiladores, enlazadores, etc.) que permiten generar los binarios nativos en las plataformas Linux, OS X y Windows.

Esta herramienta de desarrollo proporciona un conjunto de cabeceras del sistema para las API estables nativas que garantizan soporte en todas las versiones posteriores de la plataforma.

El Android NDK también proporciona un sistema de construcción que permite trabajar eficientemente con nuestras fuentes, sin tener que manejar los detalles de *toolchain/platform/CPU/ABI*. Podemos crear archivos muy cortos que describan las fuentes para compilar y qué aplicación Android las utiliza; el sistema de compilación compila las fuentes y los lugares de las bibliotecas compartidas directamente en el proyecto.

A partir de la versión 2.2 de Android Studio 2.2 o posteriores, se puede agregar código C y C++ a nuestra app al realizar una compilación en una biblioteca nativa que Gradle puede empaquetar con nuestra APK haciendo uso de los complementos NDK de Android para la versión 2.2.0 de Gradle o superior. Nuestro código Java podrá llamar a funciones en nuestra biblioteca nativa a través de la interfaz nativa de Java (JNI).

A día de hoy, la herramienta de compilación predeterminada de Android Studio para bibliotecas nativas es CMake. Android Studio también es compatible con ndk-build debido al gran número de proyectos existentes que usan el paquete de herramientas de compilación para compilar su código nativo. Es por ello que en este capítulo estudiaremos la dos formas de trabajar.

7.1.2.2. Documentación

El paquete Android NDK incluye un conjunto de documentación que describe las capacidades del NDK y cómo utilizarlo al crear bibliotecas compartidas para las aplicaciones de Android. En la última versión, la documentación se proporciona en la web <https://developer.android.com/ndk/guides/index.html>. A continuación se detallan los archivos más comunes (lista parcial):

- **Getting Started with the NDK:** describe cómo instalar y configurar el Android NDK.



- **NDK Programmer's Guide:** proporciona una visión general de las capacidades y del uso del Android NDK.
 - Concepts: muestra conceptos básicos de como utilizar NDK para desarrollar aplicaciones nativas en Android y describe cómo implementar las actividades nativas.
 - Samples: muestra ejemplos de uso de código nativo.
 - Building: describe el uso de la herramienta ndk-build, el uso del archivo *Android.mk* y del archivo *Application.mk*.
 - Architectures and CPUs: describe las arquitecturas de CPU soportadas y cómo dirigirse a ellas.
 - Libraries: Describe las APIs estables soportadas, el soporte a C++ e informa acerca de cómo funcionan las bibliotecas predefinidas compartidas y estáticas.
 - Debugging: describe cómo utilizar el depurador de código nativo.

Además de los archivos expuestos en este capítulo existe mucha más documentación en el mismo directorio que puede ser de ayuda a los desarrolladores en caso que quieran desarrollar una aplicación con algunas características más específicas.

7.1.2.3. Aplicaciones de ejemplo

En este punto vamos a explicar brevemente algunas de las aplicaciones ejemplo que aporta Android NDK. A través de estas aplicaciones podemos ver cómo utilizar el código nativo en aplicaciones Android. Como se ha expuesto en el punto 7.1.2.1 existen dos modos de trabajar con el código nativo en Android. A través de CMake, cuyos aplicaciones ejemplo se encuentran en: <https://github.com/google-samples/android-ndk> y haciendo uso de ndk-build cuyas aplicaciones ejemplo se encuentran en <https://github.com/google-samples/android-ndk/tree/android-mk>. Algunas de las aplicaciones que encontramos al instalar el Android NDK son:

- *hello-jni*: aplicación sencilla que carga una cadena de un método nativo implementado en una biblioteca compartida y luego lo muestra en la interfaz de usuario de la aplicación.
- *two-libs*: aplicación sencilla que carga una biblioteca compartida dinámicamente y llama a un método nativo proporcionado por la biblioteca. En este caso, el método se implementa en una biblioteca estática importado por la biblioteca compartida.
- *san-angeles*: aplicación sencilla que renderiza gráficos 3D a través de la API nativa de OpenGL ES, mientras que la gestión de la actividad del ciclo de vida es realizada por un objeto *GLSurfaceView*.
- *hello-gl2*: sencilla aplicación que hace un triángulo con OpenGL ES 2.0.
- *hello-neon*: sencilla aplicación que muestra cómo utilizar la biblioteca *cpufeatures* para comprobar las capacidades de la CPU en tiempo de ejecución, utilizando intrínsecos NEON si son compatibles con la CPU.
- *bitmap-plasma*: aplicación sencilla que muestra cómo obtener acceso a los *buffers* de píxeles de los objetos Android *Bitmap* desde el código nativo.
- *native-activity*: sencilla aplicación que muestra cómo utilizar la biblioteca estática *native-app-glue* para crear una actividad nativa.
- *native-plasma*: versión de *bitmap-plasma* implementado con una actividad nativa.

Para cada muestra, la NDK incluye el correspondiente código fuente en C/C++ y el archivo CMake-Lists.txt en el caso de usar CMake, y los archivos *Android.mk* y *Application.mk* en el caso de utilizar ndk-build. Todos estos archivos se encuentran en bajo la carpeta *cpp* en el caso de utilizar CMake o bajo la carpeta *jni* en el caso de los ejemplos con ndk-build.



Preguntas de repaso: [Android NDK](#).



7.2. Instalación de Android NDK

A raíz de la versión 2.2 de Android Studio ya es posible el desarrollo de aplicaciones nativas de manera integrada con el mismo IDE. Hasta la versión 1.4 de Android Studio el desarrollo de aplicaciones nativas no estaba soportado de manera integrada en este IDE y la propia web de desarrolladores de Android recomendaba utilizar Eclipse como IDE para el desarrollo de aplicaciones nativas en Android. Aún así era posible realizar dicho desarrollo con Android Studio.

En este punto se van a plasmar los dos modos de instalación de Android NDK. En el punto 7.2.1 se abordará el proceso de instalación en Android Studio 2.2 o superior, ya que se trata de la última forma que recomienda Android para implementar el desarrollo de aplicaciones nativas.

También se explicará el proceso en versiones inferiores a Android Studio en el punto 7.2.3, aunque dicho proceso funciona se recomienda el método presentado en el punto 7.2.1.

7.2.1. Instalación Android NDK en Android Studio 2.2 o superior

Para poder compilar y depurar el código nativo que exista en nuestra app, se necesitan los siguientes componentes:

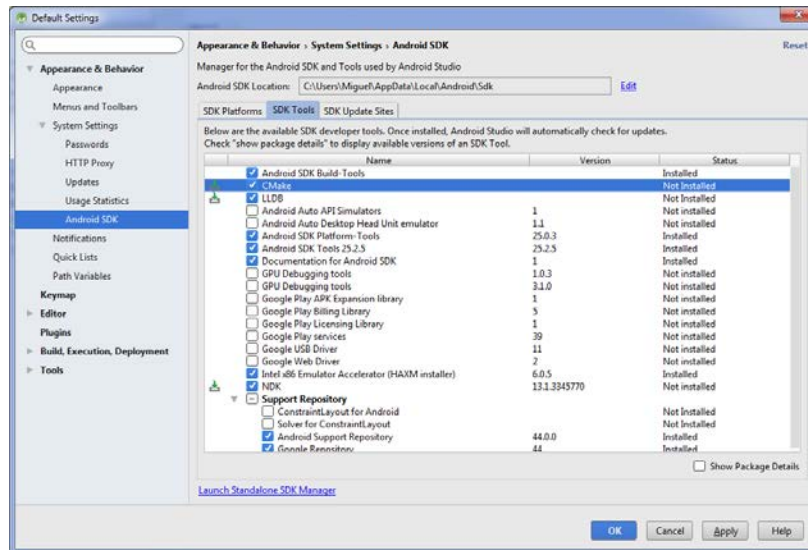
- El kit de desarrollo nativo (NDK): conjunto de herramientas que permiten utilizar código C y C++ con Android y proporciona las bibliotecas de cada plataforma que permiten manejar actividades nativas y acceder a componentes de dispositivos físicos, como sensores y entrada táctil.
- CMake: herramienta de compilación externa que funciona junto con Gradle para compilar la biblioteca nativa. No se necesita este componente si solo se planea utilizar ndk-build.
- LLDB: depurador que Android Studio utiliza para depurar código nativo.

La instalación de estos componentes se realizará a través de SDK Manager, como se observa en el siguiente ejercicio.

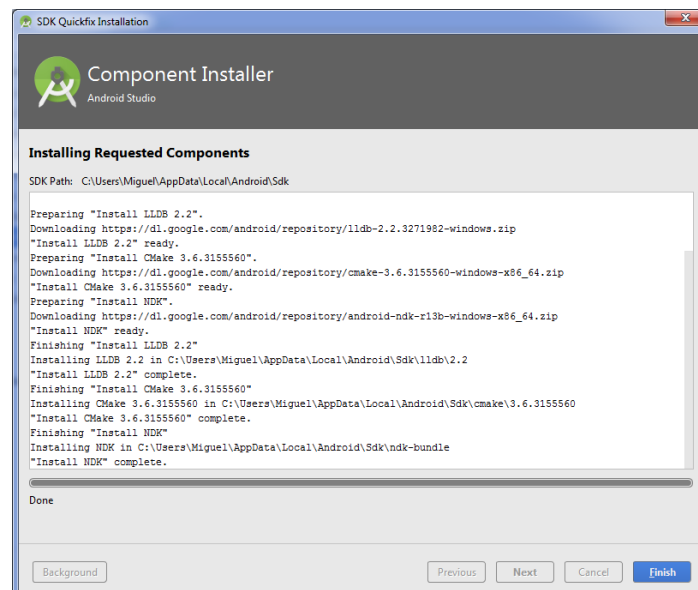


Ejercicio: *Instalación de Android NDK para trabajar con Android Studio 2.2.*

1. Desde la ventana de inicio, seleccionamos Configure → SDK Manager o desde un proyecto abierto, seleccionamos desde la barra de herramientas SDK Manager.
2. Hacemos clic en la pestaña SDK Tools.
3. Seleccionamos las casillas LLDB, CMake y NDK tal y como se muestra en la siguiente figura.



4. Hacemos clic en Apply y después en OK.
5. Empezará la descarga y la instalación de las herramientas, tal y como se observa en la siguiente figura.



6. Cuando se complemente la instalación, haremos clic en Finish y OK.

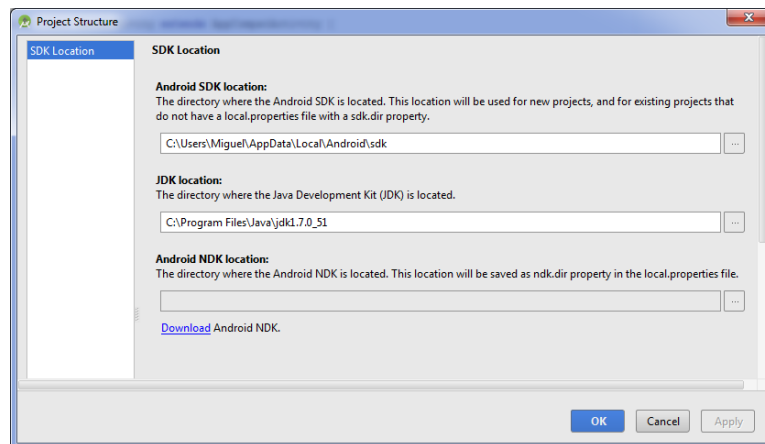
7.2.2. Instalación Android NDK en Android Studio 2.1 o inferior

Debemos recordar que este proceso de instalación no es el recomendado por Android. Una vez tenemos el IDE Android Studio 2.1 o inferior instalado y funcionando de forma correcta. Lo primero que debemos realizar es comprobar que el soporte de Android NDK esta habilitado en el IDE. Para ello vamos a *File* → *Settings* → *Plugins*. Si no esta habilitado el plugin “Android NDK Support” habilitarlo y en caso de que no aparezca actualice el Android Studio a la última versión.

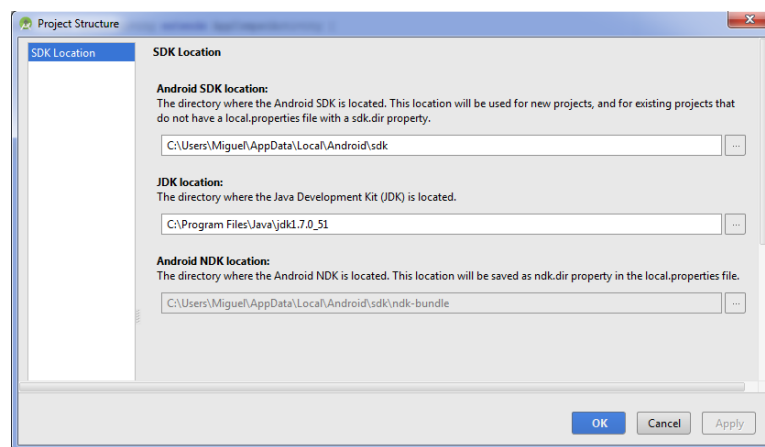


Ejercicio: Instalación de Android NDK en Android Studio 1.5

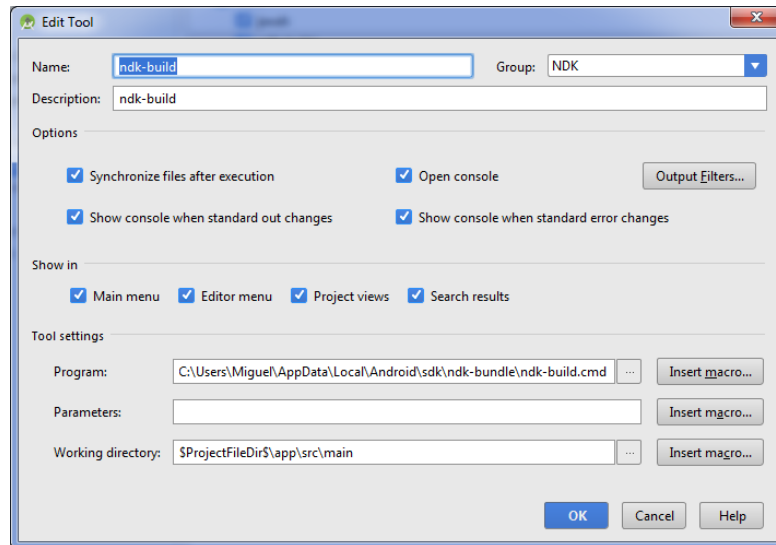
1. El primer será descargar el Android NDK y guardar el directorio en la variable `ndk.dir` del fichero `local.properties`. Esto lo realiza Android Studio de forma automática. Para ello vamos a **File → Other Settings → Default Project Structure** y pulsamos sobre **Download Android NDK** (la descarga y la posterior instalación durará unos minutos).



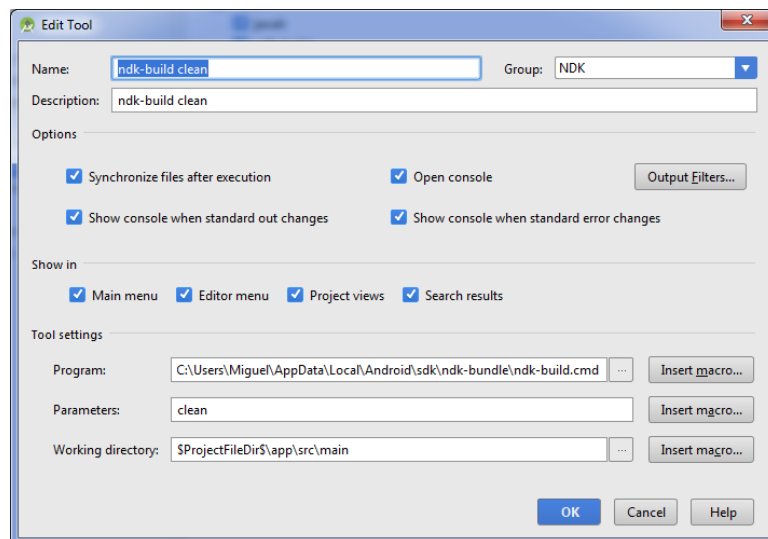
2. Una vez instalado si volvemos a acceder a esta pantalla obtendremos donde esta localizado el Android NDK.



3. Una vez instalado vamos a preparar las herramientas `ndk-build` para realizar la compilación del código nativo de forma automática desde Android Studio. Para ello iremos a **File → Settings → Tools → External Tools**. Pusaremos sobre el +.
4. Se abrirá una ventana e introduciremos la siguiente información. Name: `ndk-build`, Group: `NDK`, Description: `ndk-build`, habilitaremos todas las opciones y en Tool settings indicaremos donde esta la herramienta `ndk-build.cmd` en el caso de Windows (`ndk-build` en el caso Linux/Unix), y en el working directory: `$ProjectFileDir$/app/src/main` (el directorio donde están los archivos en código nativo).



5. A continuación pulsamos sobre OK y en la pantalla del punto 3 volvemos a crear otra herramienta. Esta será un ndk-build clean, es decir, llamaremos a ndk-build pero con el parámetro clean para realizar el borrado de los binarios creados anteriormente. La configuración será la misma que en el punto 4 pero indicando como parámetro clean.



7.2.3. Un primer ejemplo con Android NDK

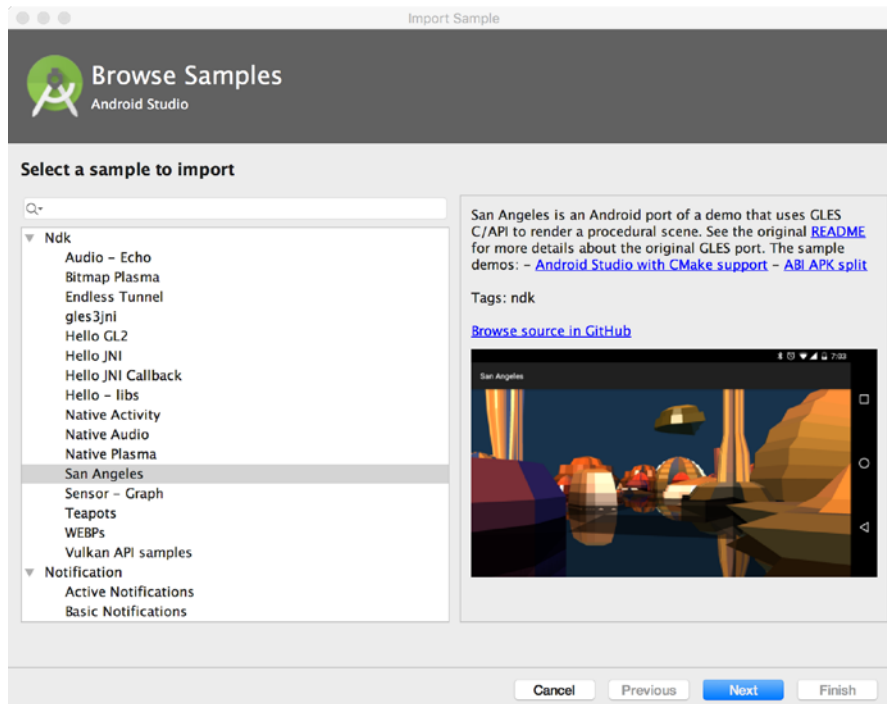
En este punto vamos a ver mediante un ejercicio paso a paso la instalación de Android NDK y la ejecución de un ejemplo del propio NDK.



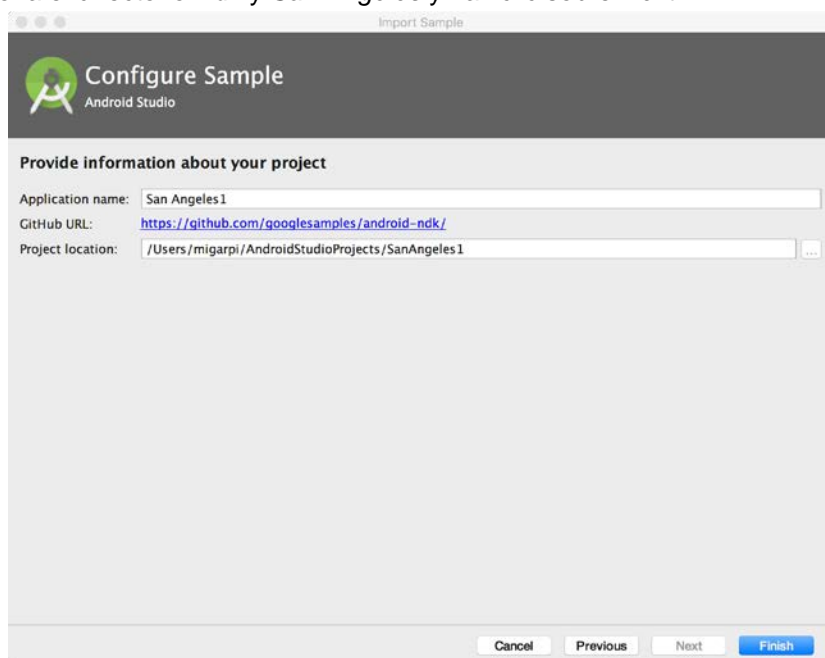
Ejercicio: *Compilación y ejecución de un ejemplo del Android NDK con Android Studio 2.2. o superior.*

Para compilar un ejemplo del Android NDK, lo primero que debemos hacer es descargar e importar el proyecto correspondiente a Android Studio. Para ello realizaremos las siguientes tareas:

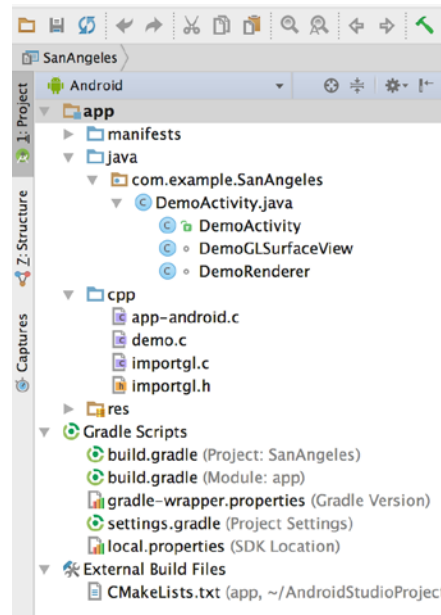
1. Abre nuestro Android Studio. Selecciona *File → New → Import Sample...*



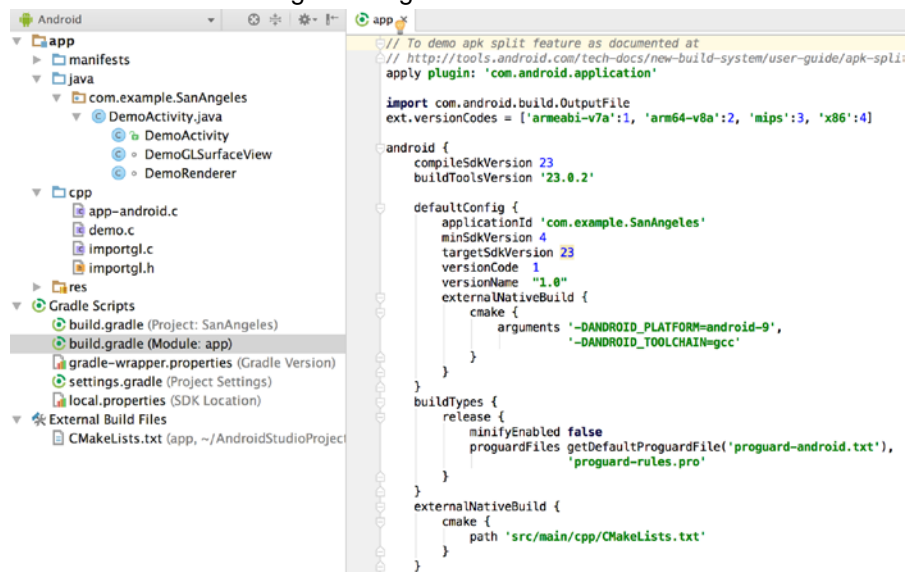
2. Después selecciona el directorio *Ndk* y *San Angeles* y haz clic sobre *Next*.



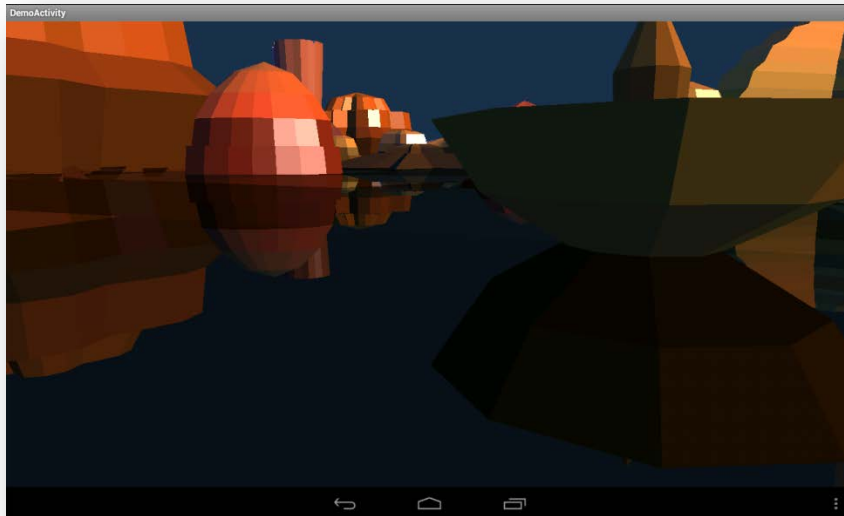
3. A continuación, introducimos el nombre de la aplicación y el lugar donde guardar el proyecto.
4. Finalmente pinchamos sobre *Finish*.
5. Al finalizar, podremos comprobar que dicho ejemplo se ha copiado en nuestro directorio de trabajo. Cuando Android Studio termine de crear el proyecto, al abrir el subpanel Project del lado izquierdo del IDE y seleccionar la vista de Android, observamos que aparece el directorio `cpp` y `External Build Files`.



6. En el directorio `cpp` podemos encontrar todos los archivos de origen nativos, encabezados y bibliotecas compiladas previamente que forman parte de tu proyecto. En `External Build Files` puedes encontrar las secuencias de comandos de compilación para CMake o ndk-build. Así como los archivos `build.gradle` que indican a Gradle la manera de compilar tu app, CMake o ndk-build. En este ejemplo se utiliza CMake, tal y como se observa en la siguiente figura.



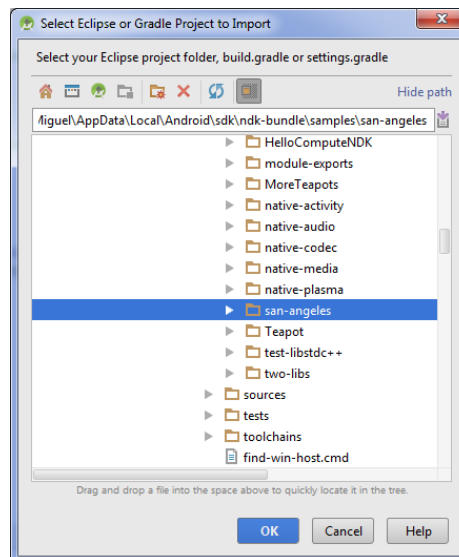
7. Ahora queda compilar y construir la aplicación. Para realizar esta tarea seleccionamos en el menú superior `Project` → `Build Project`. Si todo funciona correctamente se creará la aplicación `San Angeles.apk`.
8. Por último lanzaremos la aplicación con el menú `Run` → `run`. Esta aplicación será lanzada al emulador que tenemos configurado o la enviará al terminal móvil. La aplicación final tendrá la siguiente apariencia:



Ejercicio: *Compilación y ejecución de un ejemplo del Android NDK en Android Studio 2.1. o inferior.*

Para compilar un ejemplo del Android NDK, lo primero que debemos hacer es importar el proyecto correspondiente a Android Studio. Para ello realizaremos las siguientes tareas:

1. Abre nuestro Android Studio. Selecciona File → New → Import Project... Indicamos el directorio donde tengamos el ndk instalado, carpeta *samples* y seleccionamos *san-angeles*.



2. Se realizará la importación y una vez importado lo primero que debemos realizar es comprobar que el fichero build.gradle posea el nombre del modulo ndk y la información de los ficheros fuente. En caso de que no insertarlo.

```
defaultConfig {  
    applicationId "com.example.SanAngeles"  
    minSdkVersion 4  
}
```

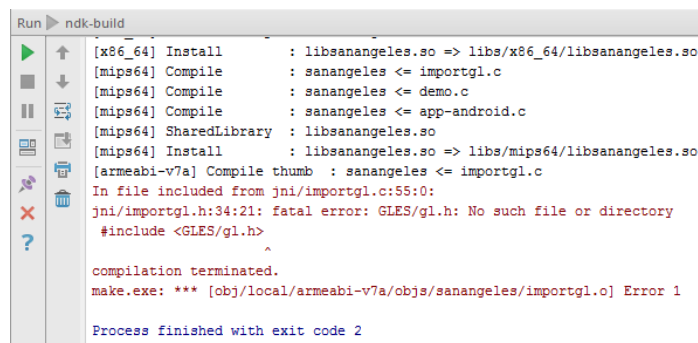


```
targetSdkVersion 4

ndk {
    moduleName "sanangeles"
}

sourceSets.main {
    jni.srcDirs = []
    jniLibs.srcDir "src/main/libs"
}
}
```

- Lo siguiente será construir el proyecto, para ello pulsaremos Build → Make Project... Aparecerá el siguiente error: *"Error:(12, 0) Error: NDK integration is deprecated in the current plugin. Consider trying the new experimental plugin. For details, see <http://tools.android.com/tech-docs/new-build-system/gradle-experimental>. Set "android.useDeprecatedNdk=true" in gradle.properties to continue using the current NDK integration"*.
- Para solucionarlo introduciremos *"android.useDeprecatedNdk=true"* en el fichero gradle.properties. En caso que dicho fichero no exista crearlo.
- Ahora vamos a compilar el código nativo para que se creen las bibliotecas correspondientes para ello seleccionaremos la vista Project, sobre la carpeta *jni* pulsaremos botón derecho → NDK → ndk-build. Aparecerá el siguiente error.



```
Run ▶ ndk-build
[x86_64] Install      : libsanangeles.so => libs/x86_64/libsanangeles.so
[mips64] Compile     : sanangeles <= importgl.c
[mips64] Compile     : sanangeles <= demo.c
[mips64] Compile     : sanangeles <= app-android.c
[mips64] SharedLibrary : libsanangeles.so
[mips64] Install     : libsanangeles.so => libs/mips64/libsanangeles.so
[armeabi-v7a] Compile thumb : sanangeles <= importgl.c
In file included from jni/importgl.c:55:0:
jni/importgl.h:34:21: fatal error: GLES/gl.h: No such file or directory
#include <GLES/gl.h>
^
compilation terminated.
make.exe: *** [obj/local/armeabi-v7a/objs/sanangeles/importgl.o] Error 1
Process finished with exit code 2
```

- Esto es debido a que el ejemplo trabaja con instancias a "EGL/egl.h" que a partir de la API 15 de Android ya no se encuentra. Para solucionarlo accede al fichero Application.mk del directorio jni e inserta la siguiente línea *"APP_PLATFORM := android-14"*.
- Guarda y vuelve a realizar el paso 5.
- Por último lanzaremos la aplicación con el menú *Run*. Esta aplicación será lanzada al emulador que tenemos configurado o la enviará al terminal móvil.

7.3. Funcionamiento y estructura de Android NDK

Como hemos indicado anteriormente, Android NDK es un conjunto de herramientas que permite a los desarrolladores de aplicaciones Android incrustar código máquina nativo compilado en C y/o C++ a los archivos de código fuente en sus paquetes de aplicaciones.

Actualmente Android permite la utilización de dos herramientas para el desarrollo de este tipo de aplicaciones. Por un parte CMake y otra ndk-build.

- CMake: es una herramienta multiplataforma de generación o automatización de código. El nombre es una abreviatura para "cross platform make" (make multiplataforma). CMake es una familia de herramientas diseñada para construir, probar y empaquetar software. CMake se utiliza para controlar el proceso de compilación del software usando ficheros de configuración sencillos e independientes de la plataforma. Cmake genera makefiles nativos y espacios de trabajo que pueden usarse en el entorno de desarrollo deseado. El proceso de construcción



se controla creando uno o más ficheros CMakeLists.txt, que contienen diversos comandos para su ejecución.

- Ndk-build: es una secuencia de comandos de shell introducida en Android NDK r4. Cuyo propósito es invocar el script de construcción NDK de forma correcta. Si estamos trabajando con Windows el comando se denomina ndk-build.cmd, mientras que en Linux o MacOS podemos invocarlos como ndk-build desde el directorio de instalación del NDK. Al igual que el comando anterior ndk-build requiere del archivo Android.mk y Application.mk (es último opcional) para la ejecución correcta del mismo.

La máquina virtual de Android permite que el código fuente de la aplicación pueda llamar a métodos implementando en código nativo a través de la interfaz JNI. En pocas palabras, esto significa que:

- El código fuente de su aplicación debe ser declarado a través de uno o más métodos con la palabra clave `native`, para indicar que se implementan a través de código nativo. Por ejemplo:
`native byte[] cargarFichero(String rutaFichero);`
- Además, se deberá proporcionar una biblioteca compartida nativa que contenga la aplicación de estos métodos, que se empaquetará en la aplicación. apk. Esta biblioteca debe tener el nombre de acuerdo a la norma que veremos en el punto 5, un ejemplo podría ser:
`libFichero.so`
- La aplicación deberá cargar explícitamente la biblioteca. Esta debe ser cargada en el inicio de la aplicación. Para realizar esta acción solo debemos añadir el siguiente código fuente, donde *Fichero* es el nombre de la biblioteca:

```
static {  
    System.loadLibrary ("Fichero");  
}
```

7.3.1. Desarrollo práctico de Android NDK con CMake

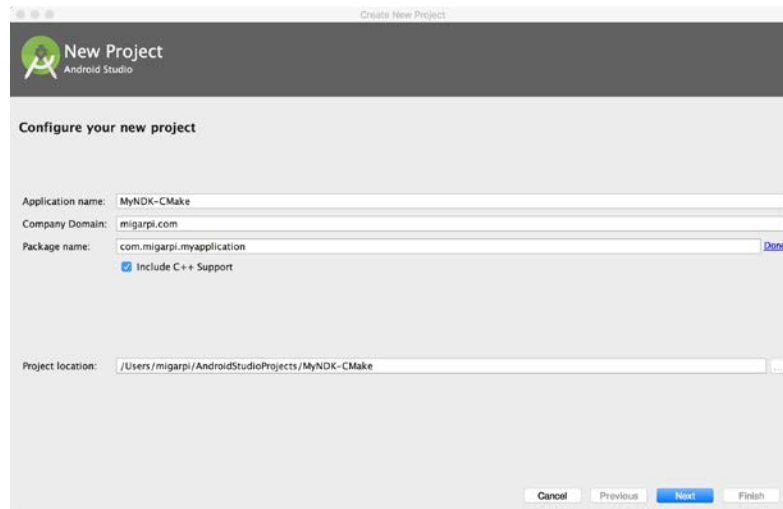
El desarrollo de una aplicación mediante Android NDK con CMake puede realizarse fácilmente a partir de la información expuesta anteriormente y través de los siguientes pasos:

1. Crear un proyecto compatible con C/C++.
2. Seleccionar los parámetros de compilación nativa e inclusión del código nativo en su correspondiente directorio.
3. Generar la llamada a la secuencia de comandos de compilación externa (CMakeLists.txt).
4. Revisar el build.gradle para comprobar la llamada a CMake y CMakeLists.txt.
5. Compilar y ejecutar la aplicación a través de los medios habituales para poder tener la aplicación final.



Ejercicio: Creación de una app nativa simple a través del Wizard y con CMake.

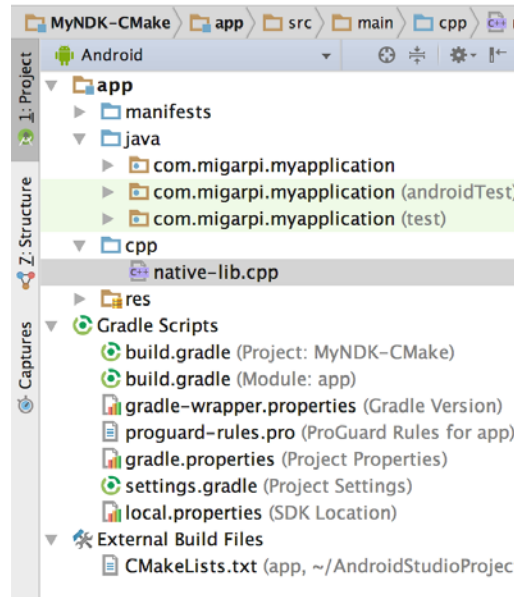
1. Crea un proyecto nuevo con el nombre MyNDK-CMake.
2. Selecciona la casilla Include C++ Support.



3. Haz clic en Next.
4. Completa los otros campos y las siguientes secciones del asistente como de costumbre.
5. En la sección Customize C++ Support del asistente, puedes personalizar tu proyecto con las siguientes opciones:
 - a. C++ Standard: usa la lista desplegable para seleccionar la estandarización de C++ que desees usar. Al seleccionar Toolchain Default, se usará la configuración predeterminada de CMake.
 - b. Exceptions Support: marca esta casilla y así habilitaremos la compatibilidad con el manejo de excepciones de C++. Al habilitarlo, Android Studio agrega la marca `-fexceptions` a `cppFlags` en tu archivo `build.gradle`, que lo que significa que Gradle le pasará esa información a CMake.
 - c. Runtime Type Information Support: marca esta casilla si deseas admitir RTIs. Si se habilita, Android Studio agrega el indicador `-frtti` a `cppFlags` en tu de archivo `build.gradle`, el cual se lo pasará a CMake.
6. Haz clic en Finish.

7.3.1.1. Situación del código nativo

Cuando Android Studio termina de crear el proyecto nuevo, si abres el subpanel Project del lado izquierdo de IDE y seleccionas la vista de Android. Se puede observar que Android Studio ha agrega los directorios `cpp` y `External Build Files`. En el grupo `cpp` puedes encontrar todos los archivos de origen nativos, encabezados y bibliotecas compiladas previamente que forman parte de tu proyecto. Para proyectos nuevos, Android Studio crea un ejemplo de archivo de origen de C++, `native-lib.cpp` y lo ubica en el directorio `src/main/cpp/` de tu módulo de app. Este código de ejemplo proporciona una función simple de C++, `stringFromJNI()`, que muestra la string "Hello from C++".



Debemos recordar que esta vista no refleja la jerarquía actual de archivos en el disco, pero en ella se agrupan archivos similares para simplificar la navegación de tu proyecto.

7.3.1.2. La herramienta CMake

CMake es una herramienta multiplataforma de generación o automatización de código. CMake es una familia de herramientas diseñada para construir, probar y empaquetar software. Todo el proceso de construcción se controla creando uno o más ficheros CMakeLists.txt, que contienen diversos comandos para su ejecución.

Para entender y poder depurar los posibles problemas de compilación de CMake, es útil conocer los argumentos de compilación específicos que usa Android Studio cuando se compila. Android Studio guarda los argumentos de construcción que utiliza para ejecutar una compilación de CMake, en un archivo denominado cmake_build_command.txt. Android Studio crea una copia para cada ABI con la configuración específica. Para poder mostrar este ejemplo, lo mejor es desde Android Studio, pulsar 2 veces shift e introducir cmake_build_command.txt. A continuación se observa un ejemplo:

```
Executable : /Users/migarpi/Library/Android/sdk/cmake/3.6.3155560/bin/cmake
arguments :
-H/Users/migarpi/AndroidStudioProjects/MyNDK-ndk-build/app
-B/Users/migarpi/AndroidStudioProjects/MyNDK-ndk-build/app/.externalNativeBuild/cmake/debug/arm64-v8a
-GAndroid Gradle - Ninja
-DANDROID_ABI=arm64-v8a
-DANDROID_NDK=/Users/migarpi/Library/Android/sdk/ndk-bundle
-DCMAKE_LIBRARY_OUTPUT_DIRECTORY=/Users/migarpi/AndroidStudioProjects/MyNDK-ndk-build/app/build/intermediates/cmake/debug/obj/arm64-v8a
-DCMAKE_BUILD_TYPE=Debug
-DCMAKE_MAKE_PROGRAM=/Users/migarpi/Library/Android/sdk/cmake/3.6.3155560/bin/ninja
-DCMAKE_TOOLCHAIN_FILE=/Users/migarpi/Library/Android/sdk/ndk-bundle/build/cmake/android.toolchain.cmake
-DANDROID_NATIVE_API_LEVEL=21
-DCMAKE_CXX_FLAGS=-std=c++11 -frtti -fexceptions
jvmArgs :
```

A continuación se muestran los principales argumentos de compilación de CMake para Android:

- -G <build-system>: Tipo de archivos de compilación generados por CMake. Para proyectos en Android Studio con código nativo, <build-system> será Android Gradle – Ninja.



- -DANDROID_ABI <abi>: Es la ABI objetivo. Se introduce el nombre de la ABI que queramos, siempre que este soportada por Android NDK.
- -DANDROID_NDK <path>: Ruta absoluta al directorio raíz de la instalación de NDK.
- -DCMAKE_LIBRARY_OUTPUT_DIRECTORY <path>: Ubicación donde CMake coloca los archivos de destino LIBRARY cuando se construye.
- -DCMAKE_BUILD_TYPE <type>: Similar a los tipos de compilación para la herramienta ndk-build . Los valores válidos son Release y Debug.
- -DCMAKE_BUILD_TYPE <type>: Nivel de API de Android que CMake compila.
- -DANDROID_TOOLCHAIN <type>: La cadena de herramientas del compilador que utiliza CMake. Los valores válidos son clang (predeterminado) y gcc (obsoletos).

También es posible hacer uso de variables para realizar llamadas específicas de CMake a través del archivo build.gradle. Estas variables se introducen dentro del parámetro cmake {}, tal y como se verá en el punto 7.3.1.4. A continuación se muestran las variables más utilizadas:

- ANDROID_TOOLCHAIN: Especifica la cadena de herramientas del compilador que CMake debería utilizar. Sus posibles argumentos son: gcc o clang.
- ANDROID_PLATFORM: Especifica el nombre de la plataforma Android destino. Por ejemplo, android-18.
- ANDROID_CPP_FEATURES: Especifica ciertas características de C++ que CMake debe utilizar al compilar su biblioteca nativa. Por ejemplo, rtti (indica que nuestro código utiliza RTTI) y exceptions (indica que nuestro código utiliza excepciones de C++).

Se puede encontrar más información en: <https://developer.android.com/ndk/guides/cmake.html>

7.3.1.3. Fichero CMakeLists.txt

En el apartado External Build Files puedes encontrar secuencias de comandos de compilación para CMake en el archivo CMakeLists.txt. CMake, al igual que ndk-build, requiere de una secuencia de comandos de compilación para conocer la manera de compilar la biblioteca nativa. Para los proyectos nuevos, Android Studio crea una secuencia de comandos de CMake denominada CMakeLists.txt y la sitúa en el directorio raíz del módulo.

En esta sección se explican algunos de los comandos básicos que debes incluir en tu secuencia de comandos para indicar a CMake las fuentes que debe usar cuando se cree la biblioteca nativa. Para ello nos fijaremos en el fichero CMakeLists.txt creado, aunque incluremos más conceptos sobre este fichero para el conocimiento del lector.

Para indicar a CMake que cree una biblioteca nativa desde el código fuente nativo, debemos agregar los comandos cmake_minimum_required() y add_library() en nuestra secuencia de comandos de compilación. Al primer comando debemos indicarle la versión mínima del compilador CMake y al comando add_library() primero especificaremos el nombre de la biblioteca, si queremos que sea compartida o estática y donde se encuentran los ficheros con código en C/C++.

```
# Sets the minimum version of CMake required to build your native library.
# This ensures that a certain set of CMake features is available to
# your build.

cmake_minimum_required(VERSION 3.4.1)

# Specifies a library name, specifies whether the library is STATIC or
# SHARED, and provides relative paths to the source code. You can
# define multiple libraries by adding multiple add_library() commands,
# and CMake builds them for you. When you build your app, Gradle
# automatically packages shared libraries with your APK.
```



```
add_library( # Specifies the name of the library.
            native-lib

# Sets the library as a shared library.
            SHARED

# Provides a relative path to your source file(s).
            src/main/cpp/native-lib.cpp )
```

Cuando agregamos un archivo o una biblioteca de origen a nuestra secuencia de comandos de compilación de CMake usando `add_library()`, Android Studio también muestra los archivos de cabecera asociados en la vista Project una vez que sincronizas tu proyecto. Sin embargo, para que CMake ubique tus archivos de cabecera durante el tiempo de compilación, debes agregar el comando `include_directories()` al archivo `CMakeLists.txt` y especificar la ruta de acceso para tus encabezados:

```
add_library(...)

# Specifies a path to native header files.
include_directories(src/main/cpp/include/)
```

Por convención, el CMake nombra el archivo de nuestra biblioteca de la siguiente manera, siempre y cuando sea una biblioteca compartida:

`liblibrary-name.so`

Por ejemplo, si especificamos “native-lib” como nombre para nuestra biblioteca compartida en la secuencia de comandos de compilación, CMake crea un archivo llamado `libnative-lib.so`. Sin embargo, cuando se cargue esta biblioteca en nuestro código Java, usaremos el nombre que especificamos en la secuencia de comandos de compilación de CMake:

```
static {
    System.loadLibrary("native-lib");
}
```

Nota: Si volvemos a nombrar o eliminar una biblioteca en el archivo `CMakeLists.txt` de CMake, debes limpiar tu proyecto para que Gradle aplique los cambios o elimine la versión anterior de la biblioteca de nuestra APK. Para limpiar nuestro proyecto, seleccionamos Build → Clean Project en la barra de menú.

Uso de bibliotecas de la API de NDK

El NDK de Android proporciona un conjunto de APIs y bibliotecas nativas que pueden resultar muy útiles. Puedes usar cualquiera de estas APIs incluyendo las bibliotecas del NDK en el archivo `CMakeLists.txt` de tu proyecto. Puedes consultar las bibliotecas nativas de Android NDK en: https://developer.android.com/ndk/guides/stable_apis.html

Las bibliotecas del NDK previamente compiladas ya existen en la plataforma de Android, por lo cual no necesitamos compilarlas ni empaquetarlas en tu APK. Debido a que las bibliotecas del NDK ya forman parte de la ruta de búsqueda de CMake, solo debemos proporcionar a CMake el nombre de la biblioteca que deseamos utilizar y vincularla con nuestra propia biblioteca nativa.

Para ello, agregamos el comando `find_library()` a nuestra secuencia de comandos de compilación de CMake para disponer una biblioteca del NDK y almacenar su ruta de acceso como una variable. Esta variable se usa para hacer referencia a la biblioteca del NDK en otras partes de la secuencia de comandos de compilación. En el siguiente ejemplo se busca la biblioteca `log` y se almacena su ruta en `log-lib`.

```
find_library( # Defines the name of the path variable that stores the
            # location of the NDK library.
            log-lib

            # Specifies the name of the NDK library that
            # CMake needs to locate.
            log )
```



Android, en su web denomina a todas las bibliotecas con una `l` delante de su nombre, por ejemplo la biblioteca `log` la denomina `llog`, pero nosotros debemos indicar solo el nombre sin la primera `ele`. Por ejemplo si quisiéramos utilizar la biblioteca `OpenGL ES 3.0`, lo haríamos de la siguiente manera.

```
find_library( # Defines the name of the path variable that stores the
              # location of the NDK library.
              GLESv3-lib

              # Specifies the name of the NDK library that
              # CMake needs to locate.
              GLESv3 )
```

Para que tu biblioteca nativa llame a funciones de la biblioteca `log`, debemos vincular las bibliotecas usando el comando `target_link_libraries()`, tal que así:

```
find_library(...)

# Links your native library against one or more other native libraries.
target_link_libraries( # Specifies the target library.
                      native-lib

                      # Links the log library to the target library.
                      ${log-lib} )
```

El NDK también incluye algunas bibliotecas como código fuente. Las cuales debemos compilar y vincular a nuestra biblioteca nativa. Podemos compilar el código fuente de una biblioteca nativa usando el comando `add_library()`, pero deberemos de proporcionar una ruta dicho código fuente. El siguiente comando indica a CMake que compile `android_native_app_glue.c`, que administra eventos de ciclo de vida de `NativeActivity` y la entrada táctil en una biblioteca estática y por último lo vincule a `native-lib`:

```
add_library( app-glue
            STATIC
            ${ANDROID_NDK}/sources/android/native_app_glue/android_native_app_glue.c )

# You need to link static libraries against your shared native library.
target_link_libraries( native-lib app-glue ${log-lib} )
```

Agregar otras bibliotecas compiladas previamente

El proceso de agregar una biblioteca compilada previamente es similar al de especificar otra biblioteca nativa para que CMake realice la compilación. Sin embargo, debido a que la biblioteca ya está compilada, debemos hacer uso de la palabra clave `IMPORTED` para indicar a CMake que solo deseamos importar la biblioteca a nuestro proyecto:

```
add_library( imported-lib
            SHARED
            IMPORTED )
```

Luego debes especificar la ruta de acceso a la biblioteca con el comando `set_target_properties()`, como se muestra a continuación:

Algunas bibliotecas proporcionan paquetes separados para arquitecturas de CPU específicas, o interfaces binarias de aplicación (ABI), y las organiza en directorios separados. Este enfoque permite que las bibliotecas aprovechen determinadas arquitecturas de CPU y, al mismo tiempo, dan la opción de utilizar solo las versiones de bibliotecas que deseemos. Para agregar varias versiones de ABI de una biblioteca a nuestro `CMakeLists.txt` sin necesidad de escribir varios comandos para cada versión de la biblioteca, puedes usar la variable de ruta de acceso `ANDROID_ABI`. Esta variable usa una lista de las ABI predeterminadas que el NDK admite, o una lista filtrada de ABI que podemos configurar manualmente para que Gradle la use, aspecto que se verá más adelante.

```
add_library(...)
set_target_properties( # Specifies the target library.
                      imported-lib
```



```
# Specifies the parameter you want to define.
PROPERTIES IMPORTED_LOCATION

# Provides the path to the library you want to import.
imported-lib/src/${ANDROID_ABI}/libimported-lib.so )
```

Al igual que habíamos comentado anteriormente, si queremos ubicar los archivos de cabeceras durante el tiempo de compilación, debes usar el comando `include_directories()` e incluir la ruta de acceso a tus archivos de cabecera:

```
include_directories( imported-lib/include/ )
```

Nota: Si deseamos empaquetar una biblioteca compilada previamente que no sea una dependencia de tiempo de compilación (por ejemplo, al agregar una biblioteca compilada previamente que sea una dependencia de `imported-lib`), no es necesario que apliques las siguientes instrucciones para vincular la biblioteca.

Para vincular la biblioteca compilada previamente, debemos de incluirla en el comando `target_link_libraries()`, tal que así.

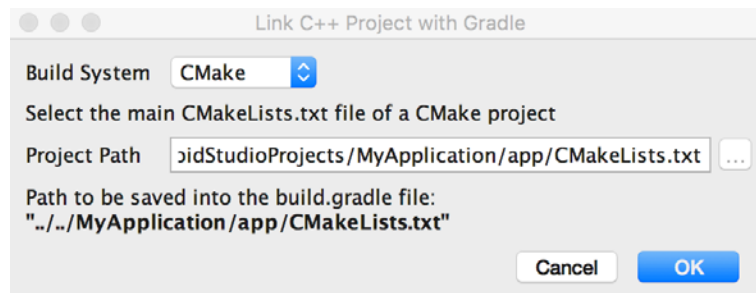
```
target_link_libraries( native-lib imported-lib app-glue ${log-lib} )
```

7.3.1.4. Fichero build.gradle

Los archivos `build.gradle` indican a Gradle la manera de compilar tu app, ya sea con CMake o `ndk-build`.

Para el caso de CMake, que es el estudiado en este punto, se puede establecer que Gradle se vincule con nuestra biblioteca nativa haciendo usando la IU de Android Studio, para ello:

9. Abrimos el subpanel Project del lado izquierdo de IDE y seleccionamos la vista de Android.
10. Hacemos clic con el botón secundario en el módulo que desees vincular con nuestra biblioteca nativa (por ejemplo, el módulo de app) y selecciona Link C++ Project with Gradle en el menú.
11. Seleccionamos CMake, en el campo junto a Project Path debemos indicar el archivo de secuencia de comandos `CMakeLists.txt` de tu proyecto de CMake externo.





```
// Provides a relative path to your CMake build script.  
path "CMakeLists.txt"  
}  
}
```

Especificar las ABI

De forma predeterminada, Gradle compila nuestra biblioteca nativa en archivos separados .so para las ABI que el NDK admite y las empaqueta en nuestra APK. Si deseamos que Gradle compile y empaquete solo determinadas configuraciones de ABI de nuestras bibliotecas nativas, podemos especificarlas con el indicador `ndk.abiFilters` en nuestro archivo `build.gradle`, tal y como se muestra a continuación:

```
android {  
    ...  
    defaultConfig {  
        ...  
        externalNativeBuild {  
            cmake {...}  
            // or ndkBuild {...}  
        }  
  
        ndk {  
            // Specifies the ABI configurations of your native  
            // libraries Gradle should build and package with your APK.  
            abiFilters 'x86', 'x86_64', 'armeabi', 'armeabi-v7a',  
                'arm64-v8a'  
        }  
    }  
    buildTypes {...}  
    externalNativeBuild {...}  
}
```

También podemos especificar argumentos e indicadores opcionales para CMake configurando otro bloque `externalNativeBuild {}` dentro del bloque `defaultConfig {}` de nuestro `build.gradle`. Como en el caso de otras propiedades del bloque `defaultConfig {}`, puedes anular estas propiedades para cada clase de producto de tu configuración de compilación. Así como incluir algunos argumentos del comando CMake (ver <https://developer.android.com/ndk/guides/cmake.html>).

Por ejemplo, si nuestro proyecto CMake define varias bibliotecas nativas, podemos utilizar la propiedad `targets` para compilar y empaquetar solo un subconjunto de esas bibliotecas para una clase de producto dado. En el siguiente ejemplo de código se describe alguna de las propiedades que se pueden configurar:

```
android {  
    ...  
    defaultConfig {  
        ...  
        // This block is different from the one you use to link Gradle  
        // to your CMake or ndk-build script.  
        externalNativeBuild {  
  
            // For ndk-build, instead use ndkBuild {}  
            cmake {  
  
                // Passes optional arguments to CMake.  
                arguments "-DANDROID_ARM_NEON=TRUE", "-DANDROID_TOOLCHAIN=clang"  
  
                // Sets optional flags for the C compiler.  
                cFlags "-D_EXAMPLE_C_FLAG1", "-D_EXAMPLE_C_FLAG2"  
  
                // Sets a flag to enable format macro constants for the C++ compiler.  
                cppFlags "-D__STDC_FORMAT_MACROS"  
            }  
        }  
    }  
}
```



```
    }  
  }  
}  
  
buildTypes {...}  
productFlavors {  
    ...  
    demo {  
        ...  
        externalNativeBuild {  
            cmake {  
                ...  
                // Specifies which native libraries to build and package for this  
                // product flavor. If you don't configure this property, Gradle  
                // builds and packages all shared object libraries that you define  
                // in your CMake or ndk-build project.  
                targets "native-lib-demo"  
            }  
        }  
    }  
}  
  
paid {  
    ...  
    externalNativeBuild {  
        cmake {  
            ...  
            targets "native-lib-paid"  
        }  
    }  
}  
}  
  
// Use this block to link Gradle to your CMake or ndk-build script.  
externalNativeBuild {  
    cmake {...}  
    // or ndkBuild {...}  
}  
}
```

7.3.1.5. Compilar y ejecutar una app con código nativa

En este apartado vamos a analizar el proceso que realiza Android Studio al compilar y ejecutar una aplicación con código nativo.

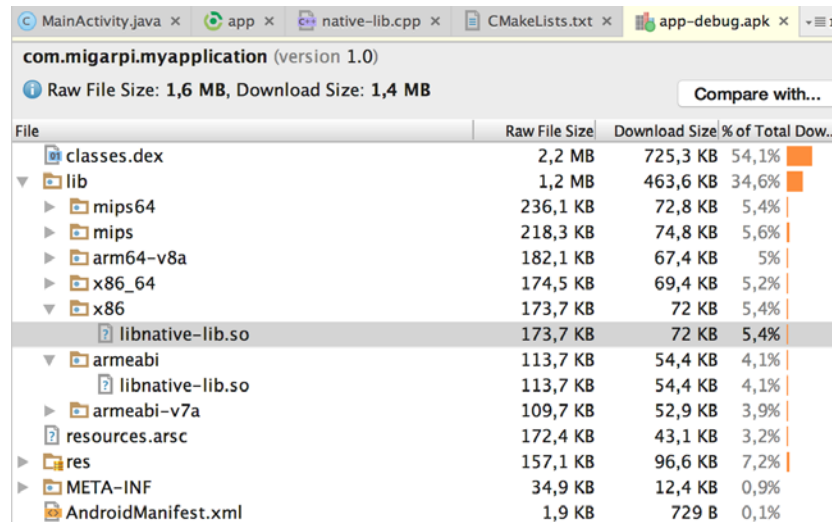
Cuando hacemos clic en Run, Android Studio crea y lanza una app que muestra el texto “Hello from C++” en tu dispositivo o emulador de Android. Los eventos que tienen lugar para compilar y ejecutar la app con código nativo son los siguientes:

1. Gradle llama a la secuencia de comandos de compilación externa, en este caso CMakeLists.txt.
2. CMake sigue los comandos en la secuencia de comandos de compilación para compilar un archivo de origen de C++ denominado native-lib.cpp, en una biblioteca de objetos compartidos y la crea la biblioteca compartida libnative-lib.so. Luego, Gradle la empaqueta en el APK.
3. Durante la ejecución, la MainActivity de la app carga la biblioteca nativa usando System.loadLibrary(). La función nativa de la biblioteca, stringFromJNI(), quedará disponible para la app.
4. MainActivity.onCreate() llama a stringFromJNI(), que muestra “Hello from C++”, y la usa para actualizar el TextView.



El Instant Run no es compatible con proyectos que usan código nativo, es por ello que Android Studio inhabilita la característica de manera automática.

Para comprobar si el Gradle ha empaquetado las bibliotecas nativas en el APK vamos a utilizar el analizador de APK. Para ello, seleccionamos Build → Analyze APK y después seleccionamos la APK del directorio app/build/outputs/apk/ y hacemos clic en OK. Como podemos observar existe una biblioteca para cada una de las arquitecturas de los procesadores que soportan el desarrollo en NDK de Android.



File	Raw File Size	Download Size	% of Total Dow...
classes.dex	2,2 MB	725,3 KB	54,1%
lib	1,2 MB	463,6 KB	34,6%
mips64	236,1 KB	72,8 KB	5,4%
mips	218,3 KB	74,8 KB	5,6%
arm64-v8a	182,1 KB	67,4 KB	5%
x86_64	174,5 KB	69,4 KB	5,2%
x86	173,7 KB	72 KB	5,4%
libnative-lib.so	173,7 KB	72 KB	5,4%
armeabi	113,7 KB	54,4 KB	4,1%
libnative-lib.so	113,7 KB	54,4 KB	4,1%
armeabi-v7a	109,7 KB	52,9 KB	3,9%
resources.arsc	172,4 KB	43,1 KB	3,2%
res	157,1 KB	96,6 KB	7,2%
META-INF	34,9 KB	12,4 KB	0,9%
AndroidManifest.xml	1,9 KB	729 B	0,1%

7.3.2. Desarrollo práctico de Android NDK con ndk-build

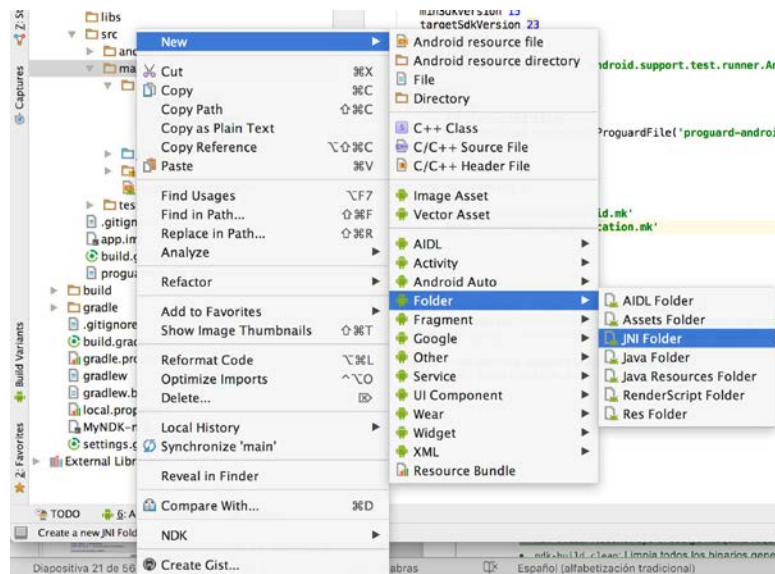
El desarrollo de una aplicación mediante Android NDK con ndk-build puede realizarse a través de cuatro pasos básicos:

1. Situar el código nativo.
2. Describir las fuentes para la construcción NDK (*Android.mk*).
3. Describir opciones acerca del compilar, tipo de CPU, etc. a través de *Application.mk* (opcional).
4. Revisar/Modificar el build.gradle para realizar la llamada a ndk-build y *Android.mk* y *Application.mk*.
5. Compilar y ejecutar la aplicación a través de los medios habituales para poder tener la aplicación final.

7.3.2.1. Situación del código fuente nativo

El código nativo escrito en C o en C++ y todo lo relacionado con el desarrollo de aplicaciones nativas deben estar situados en el directorio *jni* de nuestro proyecto. Este directorio no existe por defecto; por tanto, debe ser creado por el desarrollador.

Este directorio cuelga directamente del proyecto. Así, la situación será *\$PROYECTO/jni/* donde *\$PROYECTO* y corresponde a la ruta del proyecto Android. Para crear el directorio JNI. Simplemente desde la vista "Project" y dentro del directorio "main", botón secundario New → Folder → JNI Folder.



Dentro del directorio *jni* se puede organizar la información como quiera el desarrollador, ya que los nombres de directorios y archivos no van a influir en el paquete final de la aplicación. Lo único a tener en cuenta es que no pueden utilizar pseudónimos como *com.<miempresa>.<miproyecto>* ya que dicho nombre sí interfiere en el APK.

7.3.2.2. La herramienta *ndk-build*

El Android NDK r4 introdujo un nuevo *shell script*, llamado *ndk-build*, para simplificar la construcción del código máquina. Este *script* se encuentra en el directorio de nivel superior del NDK, y deberá llamarse desde la línea de comandos en el directorio principal del proyecto de nuestra aplicación. Por ejemplo:

```
cd $PROYECTO
ndk-build
```

La herramienta *ndk-build* posee varias opciones que vamos a analizar a continuación:

- **ndk-build**: Reconstruye el código máquina requerido.
- **ndk-build clean**: Limpia todos los binarios generados.
- **ndk-build NDK_DEBUG=1**: Genera código nativo depurable, mediante la generación de binarios de depuración.
- **ndk-build NDK_DEBUG=0**: Genera código nativo depurable, mediante la liberación de binarios.
- **ndk-build V=1**: Lanza el *build* mostrando comandos integrados.
- **ndk-build -B**: Fuerza una reconstrucción completa.
- **ndk-build -B V=1**: Fuerza una reconstrucción completa y muestra los comandos.
- **ndk-build NDK_LOG=1**: Muestra un *display* interno para los mensajes de *log* de Android NDK.
- **ndk-build NDK_HOST_32BIT=1**: Utiliza siempre herramientas de 32 bits.
- **ndk-build NDK_APPLICATION_MK = <archivo>**: Reconstruye mediante una *Application.mk* específica apuntada por la variable *NDK_APPLICATION_MK*.
- **ndk-build -C <proyecto>**: Construye el código nativo para el proyecto situado en el *path* <proyecto>. Esta opción es útil si no se quiere utilizar el comando *cd* en el terminal.

7.3.2.3. Fichero *Android.mk*

El archivo *Android.mk* describe las fuentes de compilación al sistema. Más concretamente se trata de:

- Un pequeño fragmento del *makefile* de GNU que es analizado una o más veces por el sistema de compilación. Se debe intentar minimizar las variables declaradas en este fichero y dar por válida cualquier suposición que no haya sido definida durante el análisis.



- La sintaxis de los archivos se ha diseñado para permitir que su grupo de fuentes sean módulos. Un módulo es considerado como una:

- Biblioteca estática.
- Biblioteca compartida.

Solo las bibliotecas compartidas se instalarán/copiarán en el paquete de la aplicación (.apk). Las bibliotecas estáticas, sin embargo, se pueden utilizar para generar bibliotecas compartidas.

Podemos definir uno o más módulos en cada archivo *Android.mk*, y se puede utilizar el mismo archivo de origen en varios módulos.

El sistema de construcción se encarga de todos los detalles por nosotros. Por ejemplo, no es necesario enumerar los archivos de cabecera o dependencias explícitas entre los archivos dentro del archivo *Android.mk*. El sistema de construcción de Android NDK es el que se encarga de realizar estas acciones automáticamente.

A continuación se muestra un ejemplo de un fichero *Android.mk*. Sobre el cual se explicarán las funcionalidades comando.

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := sanangeles

LOCAL_CFLAGS := -DANDROID_NDK \
                -DDISABLE_IMPORTGL

LOCAL_SRC_FILES := \
    importgl.c \
    demo.c \
    app-android.c \

LOCAL_LDLIBS := -lGLESv1_CM -ldl -llog

include $(BUILD_SHARED_LIBRARY)
```

Ahora, vamos a explicar estas líneas. La primera de ellas es:

```
LOCAL_PATH := $(call my-dir)
```

Un archivo *Android.mk* debe comenzar con la definición de la variable **LOCAL_PATH**. Se utiliza para localizar los archivos de origen en el árbol de directorios de desarrollo. En este ejemplo, la función macro **my-dir**, proporcionada por el sistema de construcción, se utiliza para devolver la ruta del directorio actual (es decir, el directorio que contiene el propio archivo *Android.mk*).

La siguiente línea es:

```
include $(CLEAR_VARS)
```

La variable **CLEAR_VARS** es proporcionada por el sistema de construcción y apunta a un *makefile* especial de GNU, que liberará distintas variables del tipo **LOCAL_XXX** (**LOCAL_MODULE**, **LOCAL_SRC_FILES**, **LOCAL_STATIC_LIBRARIES**, etc.), con la excepción de la variable **LOCAL_PATH**. Esto es necesario porque todos los archivos de control de compilación se analizan en un solo contexto de ejecución *make* GNU donde todas las variables son globales.

Después encontramos:

```
LOCAL_MODULE := sanangeles
```

La variable **LOCAL_MODULE** debe definirse para identificar cada módulo descrito en el archivo *Android.mk*. El nombre debe ser único y no contener ningún espacio. Además, hay que tener en cuenta que el sistema de construcción agregará automáticamente el prefijo y el sufijo adecuado. Es decir, el módulo de biblioteca compartida denominada *sanangeles* generará «libsanangeles.so».



Si el nombre de nuestro módulo es *libsanangeles*, el sistema de construcción no añadirá otro prefijo «lib» y generará «libsanangeles.so».

A continuación tenemos:

```
LOCAL_CFLAGS := -DANDROID_NDK \  
                -DDISABLE_IMPORTGL
```

La variable `LOCAL_CFLAGS` es un conjunto opcional de parámetros del compilador que se utilizará en la construcción de los archivos de código fuente C y/o C++. Esto puede ser útil para especificar las definiciones de macros adicionales u opciones de compilación. En nuestro caso utilizaremos el Android NDK para deshabilitar la importación de la biblioteca de gráficos.

La siguiente variable que presenta este ejemplo es:

```
LOCAL_SRC_FILES := \  
    importgl.c \  
    demo.c \  
    app-android.c \  
    app-android.c \
```

La variable `LOCAL_SRC_FILES` debe contener una lista de los archivos de código C y/o C++ que van a ser compilados y montados en este módulo. Cabe tener en cuenta que aquí no debemos incluir los ficheros cabecera del módulo. El propio sistema de construcción será el encargado de calcular las dependencias automáticamente por nosotros; solo pasaremos una lista de los archivos de código fuente al compilador.

Tenemos que tener en cuenta que la extensión por defecto para los archivos de código fuente de C++ es `.cpp`. Sin embargo, es posible especificar una extensión diferente si modificamos la variable `LOCAL_CPP_EXTENSION`.

Luego tenemos la variable:

```
LOCAL_LDLIBS := -lGLESv1_CM -ldl -llog
```

La variable `LOCAL_LDLIBS` indica una lista de enlazadores adicionales que se utilizarán en la construcción del módulo. Esto es útil para pasar el nombre de las bibliotecas del sistema específicas con el prefijo «-l».

Por último, tenemos:

```
include $(BUILD_SHARED_LIBRARY)
```

Esta variable es proporcionada por el sistema de construcción que apunta a un *script makefile* de GNU que es el encargado de recoger toda la información que ha definido en las variables `LOCAL_XXX` desde el último `include $(CLEAR_VARS)` y de determinar qué construir y cómo hacerlo exactamente. También hay la llamada `BUILD_STATIC_LIBRARY` para generar una biblioteca estática.

Por último, debemos indicar que existen más variables que pueden ser definidas en el archivo *Android.mk*. Creemos que las expuestas en este punto son las más relevantes; pero si se quiere averiguar más sobre las variables que pueden ser incluidas en este archivo podemos ir a la documentación de Android NDK.

7.3.2.4. Fichero *Application.mk* (opcional)

El propósito del fichero *Application.mk* es describir los módulos nativos que son requeridos en nuestra aplicación. Este fichero está situado dentro del directorio *jni* de nuestro proyecto.

El archivo *Application.mk* es realmente un fragmento diminuto GNU *makefile* que debe definir algunas variables, como:

- `APP_PROJECT_PATH`: Esta variable debería dar la * ruta absoluta * a nuestro directorio raíz del proyecto de la aplicación. Esto se utiliza para copiar/instalar versiones livianas de las bibliotecas compartidas JNI para concretar la ubicación específica de las herramientas de generación de nuestra APK.
- `APP_MODULES`: Esta variable es opcional. Si es definida por el desarrollador, el compilador de NDK seleccionará por defecto todos los declarados en nuestro archivo *Android.mk*. Si no es definida por el desarrollador, el compilador de NDK seleccionará todos los módulos por defecto declarados en nuestro archivo *Android.mk*.



- **APP_OPTIM**: Esta variable opcional se puede definir como *release* o *debug*. Se utiliza para modificar el nivel de optimización cuando se están compilando los módulos de la aplicación. El modo *release* es el modo predeterminado; por tanto, si no está definido, este será el modo utilizado.
- **APP_CFLAGS**: Son un conjunto de indicadores del compilador C al compilar cualquier código fuente C o C++ de cualquiera de los módulos. Se puede utilizar para cambiar la construcción de un módulo dependiendo de la aplicación que lo necesite, en lugar de modificar el propio archivo *Android.mk*.
- **APP_CPPFLAGS**: Posee las mismas características que **APP_CFLAGS**; pero en este caso orientado a C++.
- **APP_BUILD_SCRIPT**: Por defecto, el sistema de construcción NDK buscará un archivo llamado *Android.mk* situado en `$(APP_PROJECT_PATH)/jni`. Si desea cambiar este comportamiento, podemos definir el **APP_BUILD_SCRIPT** para apuntar a un *script* de construcción alternativo.
- **APP_ABI**: Por defecto, el sistema de construcción NDK generará código máquina para el ABI 'armeabi'. Esto corresponde a una ARMv5TE basado en una CPU con soporte de operaciones en punto flotante. Podemos utilizar **APP_ABI** para seleccionar una arquitectura diferente. Por ejemplo, para el soporte de dispositivos basados en ARMv7, podríamos indicar: **APP_ABI := armeabi-v7a**. Si quisiéramos soporte para todas las arquitecturas indicaríamos **APP_ABI := armeabi armeabi-v7a x86 mips** o **APP_ABI := all**.
- **APP_PLATFORM**: Indica el nombre de la plataforma Android de destino. Por ejemplo, **android-3** correspondería a la API 3 de Android (v1.5).
- **APP_SHORT_COMMANDS**: Es el equivalente de **LOCAL_SHORT_COMMANDS** para todo el proyecto.
- **APP_PIE**: A través de esta opción se hace más difícil explotar los errores de corrupción de memoria en la ubicación aleatoria del código.

A continuación se muestra un ejemplo de un fichero *Application.mk*.

```
# The ARMv7 is significantly faster due to the use of the hardware FPU
APP_ABI := armeabi armeabi-v7a
APP_PLATFORM := android-8
```

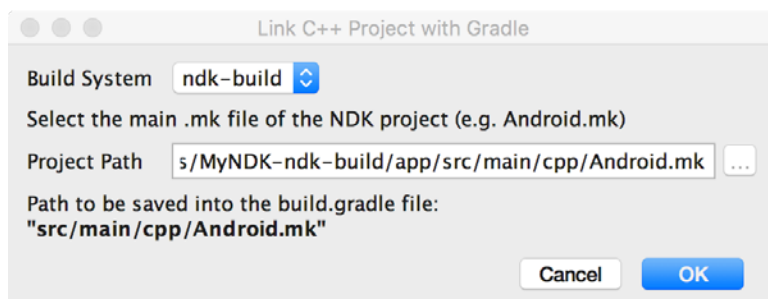
Existen más variables que pueden ser definidas en el archivo *Application.mk*. Creemos que las expuestas en este punto son las más relevantes; pero si se quiere averiguar más sobre las variables que pueden ser incluidas en este archivo podemos ir a la documentación de Android NDK.

7.3.2.5. Fichero build.gradle

Los archivos *build.gradle* indican a Gradle la manera de compilar tu app, ya sea con CMake o *ndk-build*.

Para el caso de *ndk-build*, se puede establecer que Gradle se vincule con nuestra biblioteca nativa haciendo usando la IU de Android Studio, para ello:

1. Abrimos el subpanel Project del lado izquierdo de IDE y seleccionamos la vista de Android.
2. Hacemos clic con el botón secundario en el módulo que desees vincular con nuestra biblioteca nativa (por ejemplo, el módulo de app) y selecciona Link C++ Project with Gradle en el menú.
3. Seleccionamos *ndk-build*, y utilizamos el campo junto a Project Path para especificar el archivo *Android.mk* del proyecto *ndk-build* externo. Android Studio también incluye el archivo *Application.mk* si se encuentra en el mismo directorio que tu archivo *Android.mk*.





También es posible realizar esta vinculación de forma manual. Para ellos debemos de agregar el bloque `externalNativeBuild {}` al archivo `build.gradle` y configurarlo con `ndkBuild {}` e incluir el path de los archivos `Android.mk` y `Application.mk` creados con anterioridad.

```
defaultConfig {
    ...
}
buildTypes {
    ...
}

externalNativeBuild {
    ndkBuild {
        path 'src/main/cpp/Android.mk'
    }
}
```

Al igual que vimos con CMake, de forma predeterminada, Gradle compila nuestra biblioteca nativa en archivos separados `.so` para las ABI que el NDK admite y las empaqueta en nuestra APK. Si deseamos que Gradle compile y empaquete solo determinadas configuraciones de ABI de nuestras bibliotecas nativas, podemos especificarlas con el indicador `ndk.abiFilters` en nuestro archivo `build.gradle`, tal y como se muestra a continuación:

```
android {
    ...
    defaultConfig {
        ...
        externalNativeBuild {
            ndkBuild {...}
        }

        ndk {
            // Specifies the ABI configurations of your native
            // libraries Gradle should build and package with your APK.
            abiFilters 'x86', 'x86_64', 'armeabi', 'armeabi-v7a',
                'arm64-v8a'
        }
    }
}
```

7.3.2.6. Compilar y ejecutar una app con código nativa

En este apartado vamos a analizar el proceso que realiza Android Studio al compilar y ejecutar una aplicación con código nativo con la herramienta `ndk-build`.

Cuando hacemos clic en Run, Android Studio crea y lanza la app desarrollada. Los eventos que tienen lugar para compilar y ejecutar la app con código nativo son los siguientes:

1. Gradle llama a la secuencia de comandos de compilación externa, en este caso `Android.mk` y `Application.mk`.
2. `ndk-build` sigue los comandos en la secuencia de comandos para compilar un archivo de origen de C/C++, en una biblioteca de objetos compartidos y crea tantas bibliotecas como ABI hayamos incluido. Luego, Gradle la empaqueta en el APK.
3. Durante la ejecución, la `MainActivity` de la app carga la biblioteca nativa usando `System.loadLibrary()`. La función nativa de la biblioteca quedará disponible para la app y cuando se llame al método nativo se ejecutará la acción.

El Instant Run no es compatible con proyectos que usan código nativo, es por ello que Android Studio inhabilita la característica de manera automática.



Para comprobar si el Gradle ha empaquetado las bibliotecas nativas en el APK vamos a utilizar el analizador de APK. Para ello, seleccionamos Build → Analyze APK y después seleccionamos la APK del directorio app/build/outputs/apk/ y hacemos clic en OK. Como podemos observar existe una biblioteca para cada una de las arquitecturas de los procesadores que soportan el desarrollo en NDK de Android.



Ejercicio: Ejercicio paso a paso con CMake.

Vamos a desarrollar una aplicación muy sencilla denominada SumaNDK para ver el funcionamiento de la programación NDK con CMake.

1. Crea un proyecto con con las siguientes definiciones:

Application name: SumaNDK
Project name: SumaNDK
Package name: com.sumandk
Activity name: MainActivity
Layout name: activity_main
Resto de parámetros por defecto.

2. Modifica el fichero layout *activity_main.xml* para que posea el siguiente aspecto.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:paddingLeft="@dimen/activity_horizontal_margin"
    android:orientation="vertical"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin" tools:context=".MainActivity">

    <EditText
        android:layout_width="300dp"
        android:layout_height="wrap_content"
        android:id="@+id/editText1"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <EditText
        android:layout_width="300dp"
        android:layout_height="wrap_content"
        android:id="@+id/editText2"
        android:layout_below="@+id/editText1"
        android:layout_centerHorizontal="true" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="SUMAR"
        android:id="@+id/btnSumar" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="Resultado es: "
        android:id="@+id/textViewResultado" />
</LinearLayout>
```

3. Crea el directorio cpp dentro del directorio app/src/main/ de la aplicación.
4. El siguiente paso será modificar el archivo *MainActivity.java* para introducir nuestro método nativo *calcularSuma()*, la carga de la biblioteca «libsuma.so» e implementar la actividad.



```
public class MainActivity extends AppCompatActivity {

    static {
        System.loadLibrary("suma");
    }

    EditText et1, et2;

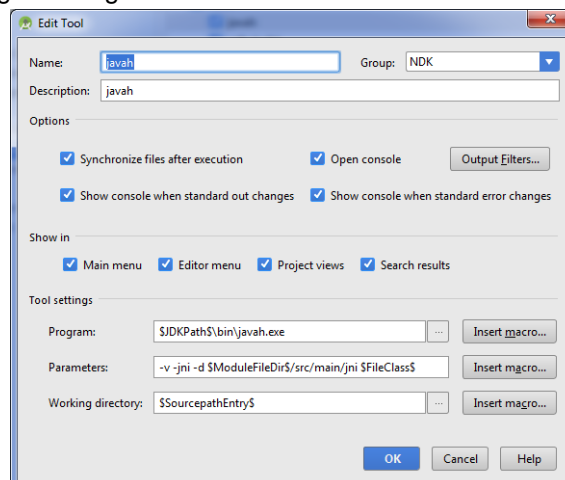
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btnSumar = (Button)findViewById(R.id.btnSumar);
        btnSumar.setOnClickListener(OkListener);
    }

    private OnClickListener OkListener = new OnClickListener(){

        public void onClick(View view){
            et1 = (EditText)findViewById(R.id.editText1);
            et2 = (EditText)findViewById(R.id.editText2);
            int aux1 = Integer.valueOf(et1.getText().toString());
            int aux2 = Integer.valueOf(et2.getText().toString());
            int resultado = calcularSuma(aux1, aux2);
            final TextView tvResultado = (TextView)findViewById(R.id.textViewResultado);
            tvResultado.setText(""+resultado);
        }
    };

    public native int calcularSuma(int aux1, int aux2);
}
```

5. Vamos a preparar la herramienta javah para crear las caberas de forma automática desde Android Studio a partir de los métodos Java. Para ello iremos a *File → Settings → Tools → External Tools*. Pusaremos sobre el +.
6. Se abrirá una ventana e introduciremos la siguiente información. Name: javah, Group: NDK, Description: javah, habilitaremos todas las opciones y en Tool settings indicaremos donde esta la herramienta javah.exe en el caso de Windows (javah en el caso Linux/Unix), en Parameters: `-v -jni -d $ModuleFileDir$/src/main/cpp $FileClass$` y en el Working directory: `$SourcepathEntry$`. Quedando como se muestra en la siguiente figura.



7. Para ejecutar esta herramienta nos situaremos sobre la carpeta *jni* pulsaremos botón derecho → NDK → javah



8. Una vez creado el fichero .h en el directorio cpp. El siguiente paso será implementar el método nativo en código C. Para ello crearemos un fichero denominado `com_sumandk_MainActivity.c`. La implementación de la función nativa debe tener el mismo prototipo que en la cabecera. Un método nativo siempre tiene al menos dos parámetros `JNIEnv` y `jobject`. El parámetro `env` apunta a una tabla de punteros a funciones, que son las funciones que usaremos para acceder a los datos Java de JNI. El segundo argumento varía dependiendo de si es un método de instancia o un método de clase (estático). Si es un método de instancia se trata de un `jobject` que actúa como un puntero `this` al objeto Java. Si es un método de clase, se trata de una referencia `jclass` a un objeto que representa la clase en la cual están definidos los métodos estáticos.

```
#include "com_sumandk_MainActivity.h"

jint Java_com_sumandk_MainActivity_calcularSuma (JNIEnv * env, jobject thiz, jint aux1, jint aux2){
    jint total= (aux1+aux2);
    return total;
}
```

9. Como nuestras fuentes nativas aún no tienen una secuencia de comandos de compilación de CMake, debemos crear una e incluir los comandos correspondientes de CMake.
- Abrimos el subpanel Project del lado izquierdo de IDE y seleccionamos la vista Project del menú desplegable.
 - Hacemos clic con el botón secundario en el directorio raíz de *nuestro módulo* y seleccionamos *New → File*.
 - Ingresamos "CMakeLists.txt" como nombre de archivo y haz clic en OK.
10. Abrimos el fichero CMakeLists.txt e introduciremos los comandos `cmake_minimum_required()` y `add_library()`. Que son los comandos mínimos necesarios para que nuestra aplicación nativa funcione.

```
# Sets the minimum version of CMake required to build your native library.
# This ensures that a certain set of CMake features is available to
# your build.

cmake_minimum_required(VERSION 3.4.1)

# Specifies a library name, specifies whether the library is STATIC or
# SHARED, and provides relative paths to the source code. You can
# define multiple libraries by adding multiple add_library() commands,
# and CMake builds them for you. When you build your app, Gradle
# automatically packages shared libraries with your APK.

add_library( # Specifies the name of the library.
            suma

            # Sets the library as a shared library.
            SHARED

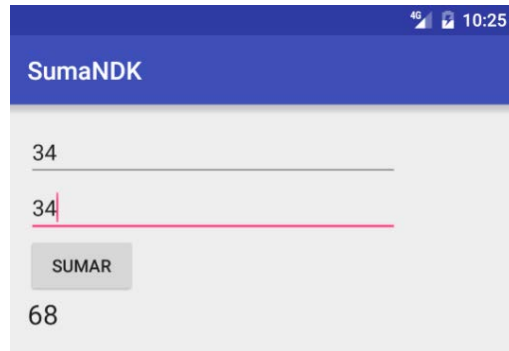
            # Provides a relative path to your source file(s).
            src/main/cpp/com_sumandk_MainActivity.c )
```

11. El penúltimo paso será vincular el Gradle con nuestra biblioteca nativa. Para ello:
- Abrimos el subpanel Project del lado izquierdo de IDE y seleccionamos la vista de Android.
 - Hacemos clic con el botón secundario en el módulo que desees vincular con nuestra biblioteca nativa (por ejemplo, el módulo de app) y selecciona Link C++ Project with Gradle en el menú.
 - Seleccionamos CMake, en el campo junto a Project Path debemos indicar el archivo de secuencia de comandos CMakeLists.txt que acabamos de crear.
12. Finalmente para limitaremos la compilación a un tipo de ABI, por ejemplo x86. Para realizar esto debemos de abrir el archivo `build.gradle` e introducimos la siguiente información dentro del `defaultConfig{}`.



```
ndk{
    abiFilters("x86")
}
```

13. Por último, vamos a lanzar nuestra aplicación. El resultado deberá ser algo similar a la siguiente captura.



Preguntas de **repaso:** Funcionamiento y estructura de Android NDK.

7.4. Interfaz entre JAVA y C/C++ (JNI)

JNI es un mecanismo que nos permite ejecutar código nativo desde Java y viceversa. El código nativo son funciones compiladas en código máquina del procesador para un sistema operativo donde se está ejecutando la máquina virtual. Las bibliotecas nativas se suelen escribir en C o C++ y se compilan dando lugar a códigos binarios que pueden ejecutar directamente el procesador.

JNI tiene una interfaz bidireccional que permite a las aplicaciones Java llamar a código nativo y viceversa, donde se permite llamar a funciones implementadas en código nativo desde Java y también se permite incrustar una máquina virtual en una aplicación nativa.

7.4.1. Bibliotecas de enlace estático y dinámico

Las bibliotecas de enlace estático son ficheros destinados a almacenar funciones, clases y variables globales y tradicionalmente se crean a partir de varios ficheros de código objeto .o (UNIX) o .obj (Windows). En UNIX las bibliotecas de enlace estático suelen tener la extensión .a y en Windows la extensión .lib.

Las bibliotecas de enlace dinámico son ficheros cuyas funciones no se incrustan en el ejecutable durante la fase de enlazado, sino que se hace en tiempo de ejecución. El programa busca el fichero, carga su contenido en memoria y enlaza su contenido según va siendo necesario; es decir, según vamos llamando a las funciones. Esto tiene la ventaja de que varios programas pueden compartir las mismas bibliotecas, lo cual reduce el consumo de recursos en los dispositivos. La extensión de estas bibliotecas es .so para sistemas operativos UNIX y .dll para Windows.

7.4.2. Tipos fundamentales, referencias y arrays

En Java existen principalmente dos tipos de datos:

- Tipos fundamentales, como: `int`, `float` o `double`. Su correspondencia con tipos C es directa, ya que en el fichero `jni.h` encontramos definiciones de tipos C equivalentes.
- Referencias. Apuntan a `arrays` y objetos. JNI pasa estos datos a los métodos nativos como punteros a la posición de memoria donde se almacenan. Desde JNI no se informa de la estructura interna de estos datos. Por lo tanto, el programador ha de ser informado de cuál es esta estructura para poder manipular los objetos o `arrays`.



En la siguiente tabla resumen se exponen los tipos de datos que podemos utilizar en la programación nativa mediante la interfaz JNI:

tipo Java	tipo nativo	tipo array nativo	código tipo	código tipo array
boolean	jboolean	jbooleanArray	Z	[Z
Byte	jbyte	jbyteArray	B	[B
Char	jchar	jcharArray	C	[C
Doublé	jdouble	jdoubleArray	D	[D
Float	jfloat	jfloatArray	F	[F
Int	Jint	jintArray	I	[I
Long	jlong	jlongArray	J	[J
Short	jshort	jshortArray	S	[S
Object	jobject	jobjectArray	L	[L
String	jstring	No disponible	L	[L
Class	jclass	No disponible	L	[L
Throwable	jthrowable	No disponible	L	[L
Void	Void	No disponible	V	No disponible

Tabla 7. Tipos de datos en la interfaz JNI.

String es una clase que está representada por el tipo C `jstring`. Para acceder al contenido de este objeto existen funciones que convierten un String Java en cadenas C. Estas funciones nos permiten convertir tanto a Unicode como a UTF-8. Para obtener el texto UTF-8 correspondiente a un String Java tenemos la función:

```
const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean*
                               isCopy);
```

Que nos devuelve un puntero a un array de caracteres UTF-8 del String. Luego para llamar a esta función haríamos:

```
const jbyte* str = (*env)->GetStringUTFChars(env,text,NULL);
if (str==NULL)
    return NULL;
```

Como veremos más tarde, el lanzamiento de excepciones en JNI es distinto al lanzamiento de excepciones en Java. Cuando se lanza una excepción en JNI, no se retrocede por la pila de llamadas hasta encontrar el `catch`, sino que la excepción queda pendiente, y en cuanto salimos del método nativo es cuando se lanza la excepción. El *buffer* obtenido por esta llamada no se libera hasta que lo liberamos explícitamente usando la función:

```
void ReleaseStringUTFChars(JNIEnv* env, jstring string, const char*
                           utf_buffer);
```

Además de las funciones `GetStringUTFChars()` y `ReleaseStringUTFChars()` tenemos las funciones:

```
const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean*
                             isCopy);
void ReleaseStringChars(JNIEnv* env, jstring string, const jchar*
                        chars_buffer);
```

También podemos crear nuevas instancias de un `java.lang.String` desde un método nativo usando las funciones JNI:



```
jstring NewStringUTF(JNIEnv* env, const char* bytes);
```

Esta función recibe un *array* de caracteres UTF-8 y crea un objeto *jstring*:

```
jstring NewString(JNIEnv* env, const jchar* ubuffer, jsize length);
```

Esta otra función recibe un *array* de caracteres Unicode y crea un *jstring*. Ambas funciones, si fallan, producen una excepción *OutOfMemoryError*, en cuyo caso además retornan *NULL*, con lo que siempre hay que comprobar el retorno de la función.

JNI permite trabajar con dos tipos de *arrays*:

- *Arrays* de tipos de datos fundamentales.
- *Arrays* de referencias.

Para acceder a *arrays* de tipos fundamentales existen funciones JNI que nos devuelven el *array* en una variable de tipo *jarray* o derivada. Una vez tengamos una variable nativa de tipo *jarray* o derivada podemos obtener la longitud del *array* con:

```
jsize GetArrayLength(JNIEnv* env, jarray array);
```

Después podemos acceder a los datos del *array* usando funciones JNI. Para cada tipo fundamental existe su correspondiente función JNI. A continuación incluimos un ejemplo de la llamada, donde habría que sustituir *type* por el tipo de dato correspondiente, ya sea *boolean*, *int*, *double*, etc.

```
jtype* GetTypeArrayElements(JNIEnv* env, jtypeArray array, jtype*  
isCopy);
```

Donde *isCopy* es un parámetro de salida que nos dice si el puntero devuelto es el *array* original, o si se ha hecho una copia de este.

Al igual que pasaba con *String*, las funciones procuran enviar el *array* original si es posible, con lo que el código nativo no debe bloquearse o llamar a otras funciones JNI entre las llamadas a estas funciones. Por último; también existen funciones para crear *arrays* Java desde el código nativo.

```
jtypeArray NewTypeArray(JNIEnv* env, jsize length);
```

7.4.3. Desarrollo paso a paso de un programa mediante JNI (I)

En este punto vamos a desarrollar un programa con código nativo desde cero utilizando la interfaz JNI.



Ejercicio: Desarrollo de la aplicación nativa *HolaMundoNDK* mediante JNI (I y *ndk-build*).

14. Crea un proyecto con con las siguientes definiciones:

```
Application name: HolaMundoNDK  
Project name: HolaMundoNDK  
Package name: com.holamundondk  
Activity name: HolaMundoNDK  
Layout name: activity_hola_mundo_ndk  
Resto de parámetros por defecto.
```

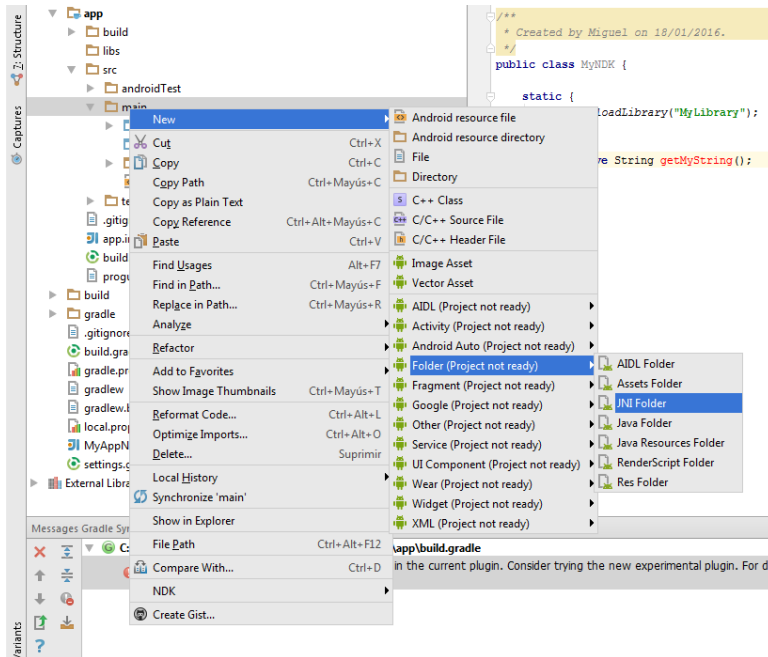
15. Modifica el fichero *layout activity_hola_mundo_ndk.xml* para que posea el siguiente aspecto. Modifica el *string hello_world* para que nuestra aplicación muestre el mensaje «*Hola Mundo NDK!!!*»:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:paddingBottom="@dimen/activity_vertical_margin"  
    android:paddingLeft="@dimen/activity_horizontal_margin"  
    android:paddingRight="@dimen/activity_horizontal_margin"  
    android:paddingTop="@dimen/activity_vertical_margin"
```



```
tools:context=".HolaMundoNDK" >
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
</RelativeLayout>
```

16. La siguiente tarea será crear la carpeta *jni*. Seleccionamos nuestro proyecto, pulsamos el botón derecho, *New/Folder* e indicamos el nombre *jni*.



7.4.3.1. Declaración del método nativo y creación del archivo *Android.mk*

Los métodos nativos llevan el modificador *native* y están sin implementar, ya que su implementación estará en una biblioteca nativa.

El método para realizar esta declaración es el siguiente:

```
static {
    System.loadLibrary("biblioteca");
}
public native String nombreMetodo();
```

Como podemos observar, primero recibe el nombre de una biblioteca de enlace dinámico y la carga. La biblioteca debe cargarse antes de llamar a cualquier método nativo.



Ejercicio: Desarrollo de la aplicación nativa *HolaMundoNDK* mediante *JNI* (I) - continuación.

1. El siguiente paso será modificar el archivo *HolaMundoNDK.java* para introducir nuestro método nativo *dameDatos()* y la carga de la biblioteca «*libholamundondk.so*». Para ello modificaremos el código y el resultado debe quedar así:

```
public class HolaMundoNDK extends Activity {
    static {
        System.loadLibrary ("holamundondk");
    }
}
```



```
public native String dameDatos();

@Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_hola_mundo_ndk);
    setTitle(dameDatos());
}
}
```

2. Dentro del directorio *jni* debemos crear el archivo *Android.mk* que contendrá los siguientes parámetros:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := holamundondk
LOCAL_SRC_FILES := com_holamundondk_HolaMundoNDK.c

include $(BUILD_SHARED_LIBRARY)
```

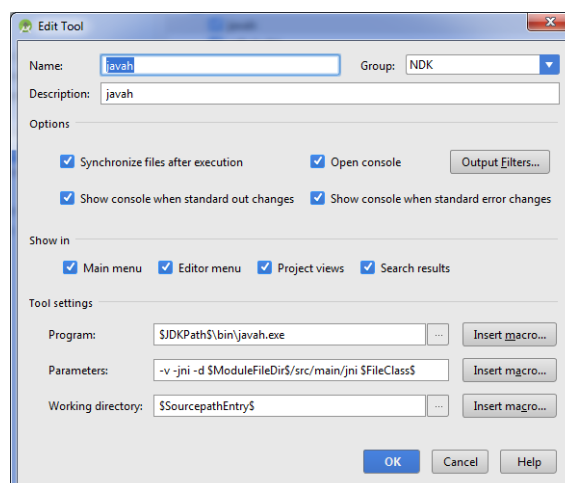
7.4.3.2. Creación del fichero de cabecera nativo

Cuando trabajamos en C o C++ almacenaremos el código fuente en ficheros *.c* o *.cpp*. Pero además también se requieren ficheros *.h* donde se indican las cabeceras de los métodos implementados. Para ahorrarnos trabajo podemos utilizar la herramienta *javah* para generar de forma automática estos ficheros. A partir de ficheros Java anteriores esta herramienta generará los *.h* que serán utilizados en nuestro código nativo.



Ejercicio: Desarrollo de la aplicación nativa *HolaMundoNDK mediante JNI (I) – continuación (Android Studio).*

1. Vamos a preparar la herramienta *javah* para crear las cabeceras de forma automática desde Android Studio a partir de los métodos Java. Para ello iremos a *File* → *Settings* → *Tools* → *External Tools*. Pusaremos sobre el +.
2. Se abrirá una ventana e introduciremos la siguiente información. Name: *javah*, Group: *NDK*, Description: *javah*, habilitaremos todas las opciones y en Tool settings indicaremos donde esta la herramienta *javah.exe* en el caso de Windows (*javah* en el caso Linux/Unix), en Parameters: *-v -jni -d \$ModuleFileDir\$/src/main/jni \$FileClass\$* y en el Working directory: *\$SourcepathEntry\$*. Quedando como se muestra en la siguiente figura.



3. Para ejecutar esta herramienta nos situaremos sobre la carpeta *jni* pulsaremos botón derecho → *NDK* → *javah*



7.4.3.3. Implementación del método nativo

La implementación de la función nativa debe tener el mismo prototipo que en la cabecera. Un método nativo siempre tiene al menos dos parámetros `JNIEnv` y `jobject`. El parámetro `env` apunta a una tabla de punteros a funciones, que son las funciones que usaremos para acceder a los datos Java de JNI. El segundo argumento varía dependiendo de si es un método de instancia o un método de clase (estático). Si es un método de instancia se trata de un `jobject` que actúa como un puntero `this` al objeto Java. Si es un método de clase, se trata de una referencia `jclass` a un objeto que representa la clase en la cual están definidos los métodos estáticos.



Ejercicio: Desarrollo de la aplicación nativa *HolaMundoNDK mediante JNI (I) – continuación.*

1. Lo primero que debemos hacer es crear un archivo nuevo en el directorio `jni` de nuestro proyecto ayudándonos del fichero de cabeceras creado en el punto anterior. El nombre del fichero en C de este ejemplo será `com_holamundondk_HolaMundoNDK.c`.
2. El contenido será el siguiente:

```
#include "com_holamundondk_HolaMundoNDK.h"
JNIEXPORT jstring Java_com_holamundondk_HolaMundoNDK_dameDatos(JNIEnv * env, jobject this) {
    return (*env)->NewStringUTF(env, "App nativa");
}
```

Donde llamamos a la biblioteca creada en el paso anterior y creamos un `string` App nativa en C, que será utilizado en nuestro proyecto al llamar al método nativo `dameDatos()`.

7.4.3.4. Compilación del fichero nativo

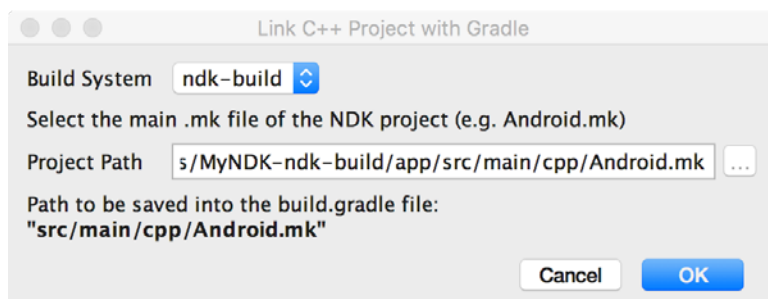
El siguiente paso es compilar todo lo realizado hasta ahora con la herramienta de construcción `ndk-build`.



Ejercicio: Desarrollo de la aplicación nativa *HolaMundoNDK mediante JNI (I) – continuación (Android Studio).*

Los archivos `build.gradle` indican a Gradle la manera de compilar nuestra app nativa, ya sea con CMake o `ndk-build`. Para el caso de `ndk-build`, se puede establecer que Gradle se vincule con nuestra biblioteca nativa haciendo usando la IU de Android Studio, para ello:

1. Abrimos el subpanel Project del lado izquierdo de IDE y seleccionamos la vista de Android.
2. Hacemos clic con el botón secundario en el módulo que desees vincular con nuestra biblioteca nativa (por ejemplo, el módulo de app) y selecciona Link C++ Project with Gradle en el menú.
3. Seleccionamos `ndk-build`, y utilizamos el campo junto a Project Path para especificar el archivo `Android.mk` del proyecto `ndk-build` externo. Android Studio también incluye el archivo `Application.mk` si se encuentra en el mismo directorio que tu archivo `Android.mk`.



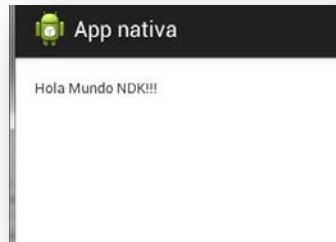


7.4.3.5. Ejecución de la aplicación

Por último, vamos a lanzar nuestra aplicación. Para ello simplemente:

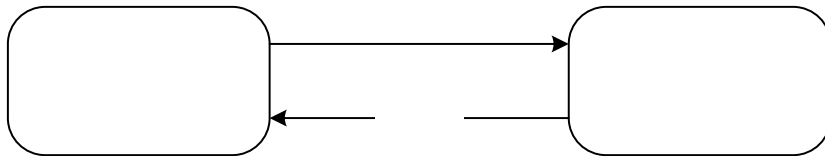
1. Accedemos al menú *Run* → *Run*.
2. Si no sale un cuadro de diálogo, le indicamos que queremos una aplicación Android y automáticamente se creará el .apk y este será lanzado en el emulador y/o dispositivo móvil.

La apariencia será la siguiente:



7.4.4. Acceso a métodos Java desde código nativo (JNI *callback*)

Además de poder acceder a los atributos de un objeto Java desde un método nativo, también podemos acceder desde un método nativo a sus métodos Java. A este acceso muchas veces se le llama acceso *callback*, porque primero Java ejecutó un método nativo y luego el método nativo vuelve a ejecutar un método Java.



Las llamadas *callback* se hacen de forma distinta dependiendo del tipo del método Java: métodos de instancia, métodos de clase y constructores.

7.4.4.1. Métodos de instancia

Para ejecutar un método de instancia de un objeto tenemos que hacer dos cosas:

1. Obtener el *method ID* del método usando:

```
jmethodID GetMethodID(JNIEnv* env, jclass class, const char* name,
                      const char* signature);
```

2. Ejecutar el método usando una de las siguientes funciones:

```
void CallVoidMethod(JNIEnv* env, jobject object, jmethodID methodID, ...);
jboolean CallBooleanMethod(JNIEnv* env, jobject object,
                          jmethodID methodID, ...);
jbyte CallByteMethod(JNIEnv* env, jobject object, jmethodID methodID,
                    ...);
jshort CallShortMethod(JNIEnv* env, jobject object, jmethodID methodID,
                     ...);
jchar CallCharMethod(JNIEnv* env, jobject object, jmethodID methodID,
                    ...);
jint CallIntMethod(JNIEnv* env, jobject object, jmethodID methodID,
                  ...);
jlong CallLongMethod(JNIEnv* env, jobject object, jmethodID methodID,
                    ...);
```




```
jfloat CallFloatMethod(JNIEnv* env, jobject object, jmethodID methodID,
                        ...);
jdouble CallDoubleMethod(JNIEnv* env, jobject object, jmethodID methodID,
                          ...);
jobject CallObjectMethod(JNIEnv* env, jobject object, jmethodID methodID,
                          ...);
```

Debemos usar una función u otra según el tipo de retorno que tenga el método. Los parámetros se pasan en la lista de parámetros variables que tiene al final la función (...). `CallObjectMethod()` se usa tanto para los métodos que devuelven objetos, como para los que devuelven *arrays*; los métodos que devuelven tipos fundamentales debemos ejecutarlos usando la función que corresponda de las que se enumeran arriba.

7.4.4.2. Métodos de clase

El proceso de ejecutar métodos de clase es similar al de ejecutar métodos de instancia, solo que ahora usamos otras funciones. En concreto los pasos son:

1. Para obtener el *method ID* usamos:

```
jmethodID GetStaticMethodID(JNIEnv* env, jclass class, const char* name,
                             const char* signature);
```

2. Para ejecutar los métodos usamos las funciones:

```
void CallStaticVoidMethod(JNIEnv* env, jclass class, jmethodID methodID,
                           ...);
jbyte CallStaticByteMethod(JNIEnv* env, jclass class, jmethodID methodID,
                            ...);
jshort CallStaticShortMethod(JNIEnv* env, jclass class, jmethodID
                              methodID, ...);
jchar CallStaticCharMethod(JNIEnv* env, jclass class, jmethodID methodID,
                            ...);
jint CallStaticIntMethod(JNIEnv* env, jclass class, jmethodID methodID,
                          ...);
jlong CallStaticLongMethod(JNIEnv* env, jclass class, jmethodID methodID,
                            ...);
jfloat CallStaticFloatMethod(JNIEnv* env, jclass class, methodID methodID,
                              ...);
jdouble CallStaticDoubleMethod(JNIEnv* env, jclass class, methodID
                                methodID, ...);
jobject CallStaticObjectMethod(JNIEnv* env, jclass class, methodID
                                methodID, ...);
```

Estas son idénticas a las de ejecutar métodos de instancia, solo que ahora reciben como parámetro un *jclass* en vez de un *jobject*.

7.4.4.3. Invocar constructores

Para llamar a un constructor se llama igual que cualquier otro método de instancia, solo que al ir a acceder al constructor usamos `<init>` como nombre del método. Una vez tengamos el *methodID* podemos pasárselo a la función:

```
jobject NewObject(JNIEnv, jclass class, jmethodID constructorID,...);
```



Ejercicio: Desarrollo de la aplicación nativa
HolaMundoNDK mediante JNI (II).



A partir del ejemplo anterior vamos a incluir unos botones para hacer una llamada desde el código JAVA a un método nativo, y viceversa; es decir, desde el código nativo llamar a un método desarrollado en Java.

1. Lo primero será modificar el *layout* que teníamos anteriormente. Para ello modificaremos el fichero *activity_hola_mundo_ndk.xml* introduciendo:

```
...
<Button
    android:id="@+id/button1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/textView1"
    android:layout_below="@+id/textView1"
    android:layout_marginTop="88dp"
    android:onClick="button1"
    android:text="@string/button1" />
<Button
    android:id="@+id/button0"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignBottom="@+id/button1"
    android:layout_alignLeft="@+id/button1"
    android:layout_marginBottom="42dp"
    android:onClick="button0"
    android:text="@string/button0" />
<TextView
    android:id="@+id/output"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_above="@+id/button0"
    android:layout_alignLeft="@+id/button0"
    android:gravity="center" />
...
```

2. El siguiente paso es modificar el archivo *HolaMundoNDK.java* (ver código siguiente). En él hemos incluido dos métodos nativos: *funcion1()* y *funcion2()*. También existe un método denominado *funcion3Callback()* que será llamado desde el código nativo, como veremos más adelante. Como podemos observar son métodos sencillos donde se les pasa una cadena *string* y devuelven otra cadena *string*:

```
public class MainActivity extends AppCompatActivity {

    private TextView salida;
    public native String dameDatos();
    public native String funcion1(String message);
    public native void funcion2();

    static {
        System.loadLibrary ("holamundondk");
    }

    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        setTitle(dameDatos());
        salida = (TextView)super.findViewById(R.id.output);
    }

    public void button0(View v){
        salida.setText(funcion1("testString"));
    }
    public void button1(View v){
```



```

        funcion2();
    }
    public void funcion3Callback(){
        String message = "funcion3Callback llamada por la funcion2 nativa";
        salida.setText(message);
    }
    public void run(){}
}

```

3. El siguiente paso es modificar el archivo *Android.mk*. En él vamos a introducir las `LOCAL_LDLIBS := -llog` y `LOCAL_CFLAGS := -Werror`. Esto implica que vamos a cargar las bibliotecas de *log* y vamos a activar los *flags* de error en C. De esta forma podremos extraer información a través del *log* y detectar errores en la ejecución del código nativo:

```

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := holamundondk
LOCAL_SRC_FILES := com_holamundondk_HolaMundoNDK.c
LOCAL_LDLIBS := -llog
LOCAL_CFLAGS := -Werror

include $(BUILD_SHARED_LIBRARY)

```

4. El siguiente paso es actualizar el fichero cabecera. Para ello volveremos a ejecutar nuestra herramienta externa *javah*. Al volver a ejecutar esta herramienta nuestro fichero *com_holamundondk_HolaMundoNDK.h* se actualizará y contendrá las dos nuevas declaraciones de los métodos nativos:

```

/*
 * Class:     com_holamundondk_HolaMundoNDK
 * Method:    funcion1
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_holamundondk_HolaMundoNDK_funcion1
    (JNIEnv *, jobject, jstring);
/*
 * Class:     com_holamundondk_HolaMundoNDK
 * Method:    funcion2
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_holamundondk_HolaMundoNDK_funcion2
    (JNIEnv *, jobject);

```

5. Ahora vamos a modificar el fichero *com_holamundondk_HolaMundoNDK.c* para implementar los dos nuevos métodos. Primero hemos definido los parámetros del *log*, como son la etiqueta (`LOG_TAG`), el *log* de información (`LOGI`) y el *log* de error (`LOGE`).

En la *funcion1()* hemos decido pasarle una cadena como entrada para tener la oportunidad de mostrar cómo el desarrollador debe proceder cuando tiene que manejar los tipos de variables que no son los primarios. En este caso, tenemos que convertir explícitamente la cadena de *jstring* a un elemento nativo *char **. Si no lo hacemos, a veces, el error no aparece a simple vista, pero mientras la aplicación se ejecuta este error puede llevarnos a comportamientos no previstos, y la depuración se convertirá en una tarea muy difícil. Por lo tanto, hay que prestar especial atención en los datos. Hemos declarado una cadena nativa, lo que permitirá guardar el resultado de la función mediante `GetStringUTFChars` que obtiene los caracteres de la cadena representados en el formato Unicode. El tercer parámetro indica si queremos una copia o el puntero a la cadena real *java*; en este último caso, el desarrollador debe prestar atención a no modificar el contenido de la cadena devuelta. Este parámetro es típico tenerlo a `NULL`.

El método *funcion2()* representa un ejemplo muy simple de una devolución de llamada (*callback*) JNI. Normalmente, cada vez que tenemos que devolver la llamada a un método Java implementado en la clase que hizo la llamada, vamos a tener que realizar los siguientes tres pasos:



- Obtener la clase (`jclass`), empezando desde el objeto (`jobject`) que hizo la llamada, a través de la `GetObjectClass`.
- Obtener el identificador del método, a través de `GetMethodID`, que lleva a cabo una búsqueda para el método en la clase dada. La búsqueda se basa en el nombre y el descriptor del tipo de método. Si el método no existe, `GetMethodID` devuelve `NULL`.
- Llamar al método, en el objeto de llamadas, pasando el `methodID`.

El código dentro de la `funcion2()` es fácil de entender, a pesar del detalle que vamos a analizar en la función `GetMethodID`, que toma como parámetros de entrada:

- `JNIEnv *`, el puntero al entorno Java.
- `jobject`, el objeto que llama, que pasamos a la función nativa como parámetro de entrada.
- `const char *`, la cadena que identifica el nombre del método para devolver la llamada.
- `const char *`, la firma del método, llamado descriptor de método: a pesar de que pueda parecer extraño, es muy fácil de entender. La primera parte, entre paréntesis, representa los parámetros de entrada, y sus tipos. El último identificador, después del paréntesis, representa el tipo de cambio. En nuestro caso, «`()V`» significa `void funcion3Callback()`. Si nuestra función hubiera sido, `float funcion3Callback (int i)`, el descriptor sería «`(I)F`». Y si tuviéramos `float funcion3Callback (int i, float f, int j)` escribiríamos «`(IFI)F`».

6. A través de esa función llamamos a un método Java desde el código nativo:

```
#include "com_holamundondk_HolaMundoNDK.h"
#include <android/log.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LOG_TAG    "HolaMundoNDK"
#define LOGI(...)  __android_log_print(ANDROID_LOG_INFO,LOG_TAG,__VA_ARGS__)
#define LOGE(...)  __android_log_print(ANDROID_LOG_ERROR,LOG_TAG,__VA_ARGS__)

JNIEXPORT jstring Java_com_holamundondk_HolaMundoNDK_dameDatos (JNIEnv *
                                                                env, jobject thiz) {
    return (*env)->NewStringUTF(env,"App nativa");
}

jstring Java_com_holamundondk_HolaMundoNDK_funcion1(JNIEnv* env,
                                                    jobject thiz, jstring message) {
    const char *nativeString = (*env)->GetStringUTFChars(env, message, 0);
    LOGI("funcion1 llamada! Parametro entrante: %s", nativeString);
    (*env)->ReleaseStringUTFChars(env, message, nativeString);
    return (*env)->NewStringUTF(env, "Llamada nativa JNI realizada!");
}

void Java_com_holamundondk_HolaMundoNDK_funcion2(JNIEnv* env,
                                                    jobject thiz) {
    LOGI("funcion2 llamada!");
    jclass clazz = (*env)->GetObjectClass(env, thiz);
    if (!clazz) {
        LOGE("callback_handler: FALLO object Class");
        goto failure;
    }
    jmethodID method = (*env)->GetMethodID(env, clazz, "funcion3Callback",
                                           "()V");

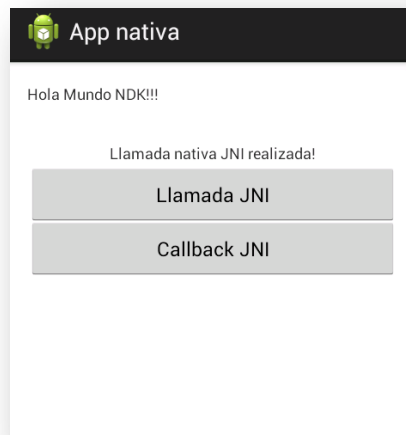
    if (!method) {
        LOGE("callback_handler: FALLO metodo ID");
        goto failure;
    }
    (*env)->CallVoidMethod(env, thiz, method);

failure: return;
```

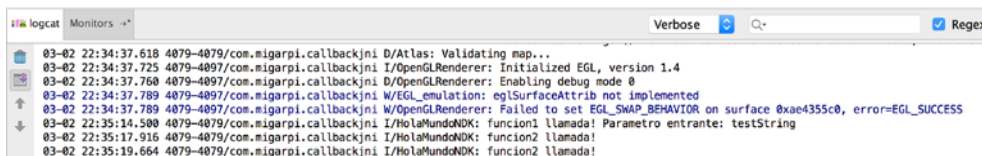


```
}
```

7. En el caso de Android Studio incluiremos el nombre del modulo ndk y la información de los ficheros fuente en el fichero build.gradle e introduciremos `"android.useDeprecatedNdk=true"` en el fichero gradle.properties.
8. Ejecutamos el comando `ndk-build`.
9. Si no existe ningún error ya podremos realizar el siguiente paso: la creación del .apk. En caso contrario deberemos analizar los errores y subsanarlos hasta que no exista ninguno de ellos.
10. Accedemos al menú `Run/Run`.
11. Si no sale un cuadro de diálogo, le indicamos que queremos una aplicación Android y automáticamente se creará el .apk y este será lanzado en el emulador y/o dispositivo móvil. La aplicación tendrá la siguiente apariencia:



12. Observa el Android Monitor. Podemos ver que cada vez que pulsamos sobre un botón, en nuestra aplicación aparece una entrada en el log, lo cual nos puede ayudar a la hora de comprobar el correcto funcionamiento de nuestra aplicación.



7.4.5. Excepciones

Cuando se produce una excepción (bien sea en un método JNI o bien en un método *callback* de Java) no se interrumpe el flujo normal de ejecución del programa (como pasa en Java), sino que en el hilo se activa un *flag* indicando que se ha lanzado la excepción. Existe un único *flag* por cada hilo y este está almacenado en la estructura *env*, con lo que el hecho de que se produzca una excepción en un hilo no afecta a los demás hilos. Podemos comprobar si en nuestro hilo se ha producido una excepción llamando a:

```
jboolean ExceptionCheck(JNIEnv* env);
```

O alternativamente podemos usar:

```
jthrowable ExceptionOccurred(JNIEnv* env);
```

Esta última función devuelve una referencia a la excepción ocurrida, o NULL si no ha habido excepción. Aunque parezca tedioso es necesario que una función nativa bien construida compruebe si se ha producido



una excepción cada vez que llama a una función JNI que puede producir una excepción o a un método *callback* de Java.

```
jfieldID fieldID = (*env)->GetFieldID(env,clase,"atributo","I");
if (fieldID==NULL) {
    // Se ha producido una excepcion y la tratamos
}
```

En cualquier caso, si nuestro método nativo no se molesta en comprobar las posibles excepciones de cada llamada pueden producirse resultados inesperados al seguir llamando a más funciones de JNI.

En realidad, cuando se produce una excepción hay una pequeña lista de funciones JNI que son las únicas que se pueden ejecutar de forma segura antes de tratar la excepción que es la siguiente. Estas funciones nos permiten tratar la excepción o liberar recursos antes de retornar a Java:

DeleteLocalRef()	PushLocalFrame()
DeleteGlobalRef()	PopLocalFrame()
ExceptionOccurred()	ReleaseStringChars()
ExceptionDescribe()	ReleaseStringUTFChars()
ExceptionClear()	ReleaseStringCritical()
ExceptionCheck()	Release<Primitive>ArrayElements()
MonitorExit()	ReleasePrimitiveArrayCritical()

Una función nativa puede lanzar una excepción usando la función JNI:

```
jint ThrowNew(JNIEnv* env, jclass class, const char* message);
```

Esto activa el *flag* de la excepción y deja la excepción pendiente hasta que la tratemos. La función retorna 0 si la excepción se lanza o un valor distinto de 0 si no se pudo lanzar la excepción.



Preguntas de repaso: [Interfaz entre Java y C/C++ \(JNI\).](#)

7.5. Rendimiento de aplicaciones con código nativo

En este punto del tema vamos a comprobar el rendimiento de las aplicaciones con código nativo con respecto a las aplicaciones desarrolladas íntegramente con Java. Para ello vamos a desarrollar un ejemplo que mostrará el valor que ocupa la posición X en la serie de Fibonacci; es decir, si la serie de Fibonacci es 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144... el valor correspondiente a la posición 6 será 8. Esta acción será implementada en dos métodos de programación y veremos cuál es más rápido.



Ejercicio: *Desarrollo de una aplicación para comprobar el rendimiento del código nativo con ndk-build.*



1. Crea un proyecto con con las siguientes definiciones:

Application name: FibonacciNDK
Project name: FibonacciNDK
Package name: com.fibonaccindk
Resto de parámetros por defecto.
Activity name: FibonacciNDK
Layout name: main

2. Modifica el fichero *main.xml* para que posea el siguiente aspecto:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center_horizontal">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Dalvik vs. Nativo"
        android:gravity="center"
        android:textSize="40sp"
        android:layout_margin="10dp" />
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        <EditText
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/ValorEntrante"
            android:hint="Valor"
            android:textSize="30sp"
            android:inputType="number"
            android:layout_margin="10dp">
        </EditText>
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/botonLanzar"
            android:text="Lanzar"
            android:textSize="30sp"
            android:layout_margin="10sp">
        </Button>
    </LinearLayout>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/Resultado"
        android:text="RESULTADO"
        android:textSize="20sp">
    </TextView>
</LinearLayout>
```

3. La siguiente tarea será crear la carpeta *jni*. Seleccionamos nuestro proyecto, pulsamos el botón derecho, *New/Folder* e indicamos el nombre *jni*.
4. Ahora añadimos el código Java en el archivo *FibonacciNDK.java*. Este fichero debe poseer dos métodos que ejecuten el objetivo del ejercicio tanto de forma recursiva como iterativa. En nuestro caso *fibonacciDalvikR()* y *fibonacciDalvikI()*. Después tenemos la llamada a la biblioteca «libfibonacci.so» y los métodos nativos *fibonacciNativoR()* y *fibonacciNativoI()*.
5. A través del método *onClick()* extraeremos los tiempos utilizados tanto en las implementaciones nativas como las virtualizadas en la máquina Dalvik:



```
public class FibonacciNDK extends Activity implements OnClickListener {
    TextView Resultado;
    Button botonLanzar;
    EditText ValorEntrante;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ValorEntrante = (EditText) findViewById(R.id.ValorEntrante);
        Resultado = (TextView) findViewById(R.id.Resultado);
        botonLanzar = (Button) findViewById(R.id.botonLanzar);
        botonLanzar.setOnClickListener(this);
    }

    public static long fibonacciDalvikR(long n) {
        if (n <= 0)
            return 0;
        if (n == 1)
            return 1;
        return fibonacciDalvikR(n - 1) + fibonacciDalvikR(n - 2);
    }

    public static long fibonacciDalvikI(long n) {
        long previous = -1;
        long result = 1;
        for (long i = 0; i <= n; i++) {
            long sum = result + previous;
            previous = result;
            result = sum;
        }
        return result;
    }

    static {
        System.loadLibrary("fibonacci");
    }

    public static native long fibonacciNativoR(int n);

    public static native long fibonacciNativoI(int n);

    public void onClick(View view) {
        int input = Integer.parseInt(ValorEntrante.getText().toString());
        long start1, start2, stop1, stop2;
        long result;
        String out = "";
        // Dalvik - Recursivo
        start1 = System.currentTimeMillis();
        result = fibonacciDalvikR(input);
        stop1 = System.currentTimeMillis();
        out += String.format("Dalvik recursiva - Valor: %d Tiempo: (%d msec)",
                             result, stop1 - start1);
        // Dalvik - Iterativo
        start2 = System.currentTimeMillis();
        result = fibonacciDalvikI(input);
        stop2 = System.currentTimeMillis();
        out += String.format("\nDalvik iterativa-Valor: %d Tiempo: (%d msec)",
                             result, stop2 - start2);
        // Nativo - Recursivo
        start1 = System.currentTimeMillis();
        result = fibonacciNativoR(input);
        stop1 = System.currentTimeMillis();
    }
}
```



```

        out += String.format("\nNativo recursivo-Valor: %d Tiempo: (%d msec)",
                                result, stop1 - start1);
// Nativo - Iterativo
start2 = System.currentTimeMillis();
result = fibonacciNativoI(input);
stop2 = System.currentTimeMillis();
out += String.format("\nNativo iterativo-Valor: %d Tiempo: (%d msec)",
                                result, stop2 - start2);

Resultado.setText(out);
    }
}

```

6. Dentro del directorio *jni* debemos crear el archivo *Android.mk* que contendrá.

```

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := fibonacci
LOCAL_SRC_FILES := fibonacci.c

include $(BUILD_SHARED_LIBRARY)

```

7. Crea el fichero *com_fibonaccindk_fibonacciNDK.h*. Para ello, nos situaremos sobre el directorio *jni* y pulsaremos botón secundario → NDK → javah y así ejecutaremos la herramienta javah . La información más importante de este archivo autogenerado son las siguientes llamadas:

```

...
/*
 * Class:      com_fibonaccindk_FibonacciNDK
 * Method:     fibonacciNativoR
 * Signature:  (I)J
 */
JNIEXPORT jlong JNICALL Java_com_fibonaccindk_FibonacciNDK_fibonacciNativoR
    (JNIEnv *, jclass, jint);

/*
 * Class:      com_fibonaccindk_FibonacciNDK
 * Method:     fibonacciNativoI
 * Signature:  (I)J
 */
JNIEXPORT jlong JNICALL Java_com_fibonaccindk_FibonacciNDK_fibonacciNativoI
    (JNIEnv *, jclass, jint);

#ifdef __cplusplus
}
#endif
#endif

```

8. Lo siguiente que debemos hacer es crear un archivo nuevo en el directorio *jni* de nuestro proyecto. El nombre será *fibonacci.c*. En él tendremos la implementación de los métodos *fibonacciNativoR()* y *fibonacciNativoI()* y sus respectivas llamadas:

```

#include "com_fibonaccindk_FibonacciNDK.h"
jint fibonacciNativoR(jint n) {
    if(n<=0) return 0;
    if(n==1) return 1;
    return fibonacciNativoR(n-1) + fibonacciNativoR(n-2);
}

jint fibonacciNativoI(jint n) {
    jint previous = -1;
    jint result = 1;
    jint i=0;
    jint sum=0;

```



```
for (i = 0; i <= n; i++) {
    sum = result + previous;
    previous = result;
    result = sum;
}
return result;
}

JNIEXPORT jlong JNICALL Java_com_fibonaccindk_FibonacciNDK_fibonacciNativoR
    (JNIEnv *env, jclass obj, jint n) {
    return fibonacciNativoR(n);
}

JNIEXPORT jlong JNICALL Java_com_fibonaccindk_FibonacciNDK_fibonacciNativoI
    (JNIEnv *env, jclass obj, jint n) {
    return fibonacciNativoI(n);
}
```

9. El siguiente paso es la vinculación de nuestro código nativo con el sistema de compilación Gradle. Para ello hacemos clic con el botón secundario en el módulo de app y selecciona Link C++ Project with Gradle en el menú.
10. Seleccionamos ndk-build, y utilizamos el campo junto a Project Path para especificar el archivo Android.mk del proyecto ndk-build externo.
11. Si no existe ningún error ya podremos realizar el siguiente paso que es la creación del .apk. En caso contrario deberemos analizar los errores y subsanarlos hasta que no exista ninguno de ellos.
12. Accedemos al menú Run/Run.
13. Si no sale un cuadro de diálogo le indicamos que queremos una aplicación Android y automáticamente se creará el .apk y este será lanzado en el emulador y/o dispositivo móvil. La aplicación tendrá la siguiente apariencia:



Práctica: Número primo en código nativo.

Para realizar otra comprobación del comportamiento del código nativo debes realizar un programa similar al anterior, pero que al introducir un número indique si es primo o no. Este algoritmo ya fue explicado en capítulos anteriores; por tanto la parte algorítmica ya está resuelta.

7.6. Procesado de imagen con código nativo

En este último punto del capítulo vamos a desarrollar una aplicación de procesamiento de imagen utilizando los conceptos vistos a lo largo de esta unidad.

**Ejercicio:** *Desarrollo de una aplicación de procesamiento de imagen con código nativo con CMake.*

1. Crea un proyecto con las siguientes definiciones:

Application name: ImgProcesadoNDK
Project name: ImgProcesadoNDK
Package name: com.imgprocesadondk
Resto de parámetros por defecto.
Activity name: ImgProcesadoNDK
Layout name: main

2. Modificar el *layout*, incluyendo el siguiente código XML:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:layout_gravity="center_vertical|center_horizontal"
        android:id="@+id/ivDisplay"/>
    <LinearLayout
        android:orientation="horizontal"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <Button
            android:id="@+id/btnReset"
            style="?android:attr/buttonStyleSmall"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Original"
            android:onClick="onResetImagen" />
        <Button
            android:id="@+id/btnConvert"
            style="?android:attr/buttonStyleSmall"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Grises"
            android:onClick="onConvertirGrises" />
    </LinearLayout>
</LinearLayout>
```

3. Crea la carpeta *cpp*. Para ello, selecciona en nuestro proyecto, pulsa con el botón derecho y selecciona *New/Folder*.
4. Modificaremos el fichero de la actividad:

```
public class ImgProcesadoNDK extends AppCompatActivity {

    private String tag = "ImgProcesadoNDK";
    private Bitmap bitmapOriginal = null;
    private Bitmap bitmapGrises = null;
    private ImageView ivDisplay = null;

    static {
        System.loadLibrary("imgprocesadondk");
    }
}
```



```
public native void convertirGrisen(Bitmap bitmapIn, Bitmap bitmapOut);

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_img_procesado_ndk);
    Log.i(tag, "Imagen antes de modificar");
    ivDisplay = (ImageView) findViewById(R.id.ivDisplay);
    BitmapFactory.Options options = new BitmapFactory.Options();
    // Asegurar que la imagen tiene 24 bits de color
    options.inPreferredConfig = Bitmap.Config.ARGB_8888;
    bitmapOriginal = BitmapFactory.decodeResource(this.getResources(),
        R.drawable.sampleImage, options);
    if (bitmapOriginal != null)
        ivDisplay.setImageBitmap(bitmapOriginal);
}

public void onResetImagen(View v) {
    Log.i(tag, "Resetear Imagen");
    ivDisplay.setImageBitmap(bitmapOriginal);
}

public void onConvertirGrisen(View v) {
    Log.i(tag, "Conversion a escala de grises");
    bitmapGrisen = Bitmap.createBitmap(bitmapOriginal.getWidth(),
        bitmapOriginal.getHeight(), Bitmap.Config.ARGB_8888);
    convertirGrisen(bitmapOriginal, bitmapGrisen);
    ivDisplay.setImageBitmap(bitmapGrisen);
}
}
```

5. Crea el fichero CMakeLists.txt:

```
# Sets the minimum version of CMake required to build the native
# library. You should either keep the default value or only pass a
# value of 3.4.0 or lower.

cmake_minimum_required(VERSION 3.4.1)

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds it for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
    imgprocesadondk

    # Sets the library as a shared library.
    SHARED

    # Provides a relative path to your source file(s).
    # Associated headers in the same location as their source
    # file are automatically included.
    src/main/cpp/com_migarpi_imgprocesadondk_ImgProcesadoNDK.c )

# Searches for a specified prebuilt library and stores the path as a
# variable. Because system libraries are included in the search path by
# default, you only need to specify the name of the public NDK library
# you want to add. CMake verifies that the library exists before
# completing its build.

find_library( # Sets the name of the path variable.
```




```

log-lib

# Specifies the name of the NDK library that
# you want CMake to locate.
log )

find_library( # Sets the name of the path variable.
jnigraphics-lib

# Specifies the name of the NDK library that
# you want CMake to locate.
jnigraphics )

# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in the
# build script, prebuilt third-party libraries, or system libraries.

target_link_libraries( # Specifies the target library.
imgprocesadondk

# Links the target library to the log library
# included in the NDK.
${log-lib} ${jnigraphics-lib})

```

6. Crea el fichero `com_imgprocesadondk_ImgProcesadoNDK.h` con la herramienta `javah`. Para ello modifica los parámetros para incluir la clase que da soporte a `Bitmap`. Desde el menú de External Tools, selecciona `javah` y modifícala introduciendo como parámetros: `-v -jni -d $ModuleFileDir$/src/main/cpp -classpath $ModuleSdkPath$/platforms/android-23/android.jar: $FileClass$`. Si estamos trabajando con Windows los “.” deberán cambiar por “.”.

7. Implementa el siguiente código nativo en el fichero `imgprocesadondk.c`:

```

#include "com_imgprocesadondk_ImgProcesadoNDK.h"
#include <android/log.h>
#include <android/bitmap.h>

#define LOG_TAG "libimgprocesadondk"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__)

typedef struct {
    uint8_t red;
    uint8_t green;
    uint8_t blue;
    uint8_t alpha;
} rgba;

/*Conversion a grises por pixel*/
JNIEXPORT void JNICALL Java_com_imgprocesadondk_ImgProcesadoNDK_convertirGrises
(JNIEnv *env, jobject obj, jobject bitmapcolor, jobject bitmapgris) {
    AndroidBitmapInfo infocolor;
    void *pixelscolor;
    AndroidBitmapInfo infogris;
    void *pixelsgris;
    int ret;
    int y;
    int x;
    LOGI("convertirGrises");
    if ((ret = AndroidBitmap_getInfo(env, bitmapcolor, &infocolor)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
    }
}

```



```

        return;
    }

    if ((ret = AndroidBitmap_getInfo(env, bitmapgris, &infogris)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }

    LOGI("imagen color :: ancho %d;alto %d;avance %d;formato %d;flags %d", infocolor.width,
        infocolor.height, infocolor.stride,
        infocolor.format, infocolor.flags);
    if (infocolor.format != ANDROID_BITMAP_FORMAT_RGBA_8888) {
        LOGE("Bitmap no es formato RGBA_8888 !");
        return;
    }

    LOGI("imagen color :: ancho %d;alto %d;avance %d;formato %d;flags %d",
        infogris.width, infogris.height, infogris.stride,
        infogris.format, infogris.flags);
    if (infogris.format != ANDROID_BITMAP_FORMAT_RGBA_8888) {
        LOGE("Bitmap no es formato RGBA_8888 !");
        return;
    }

    if ((ret = AndroidBitmap_lockPixels(env, bitmapcolor, &pixelscolor))
        < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }

    if ((ret = AndroidBitmap_lockPixels(env, bitmapgris, &pixelsgris)) < 0) {
        LOGE("AndroidBitmap_lockPixels() fallo ! error=%d", ret);
    }
    // modificacion pixeles en el algoritmo de escala grises
    for (y = 0; y < infocolor.height; y++) {
        rgba *line = (rgba *) pixelscolor;
        rgba *grisline = (rgba *) pixelsgris;
        for (x = 0; x < infocolor.width; x++) {
            float output = (line[x].red + line[x].green + line[x].blue) / 3;
            if (output > 255) output = 255;
            grisline[x].red = grisline[x].green = grisline[x].blue =
                (uint8_t) output;
            grisline[x].alpha = line[x].alpha;
        }
        pixelscolor = (char *) pixelscolor + infocolor.stride;
        pixelsgris = (char *) pixelsgris + infogris.stride;
    }

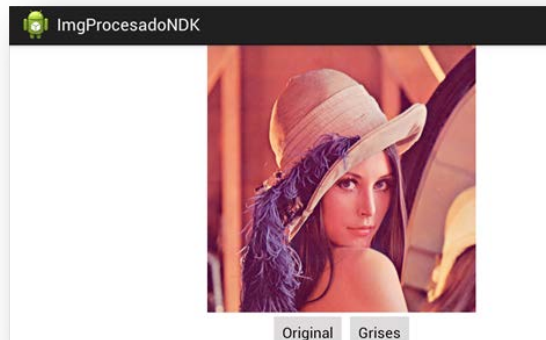
    LOGI("unlocking pixels");
    AndroidBitmap_unlockPixels(env, bitmapcolor);
    AndroidBitmap_unlockPixels(env, bitmapgris);
}

```

8. El siguiente paso es la vinculación de nuestro código nativo con el sistema de compilación Gradle. Hacemos clic con el botón secundario en el módulo de app y selecciona Link C++ Project with Gradle en el menú. Seleccionamos CMake, en el campo junto a Project Path debemos indicar el archivo de secuencia de comandos CMakeLists.txt de tu proyecto de CMake externo.
9. Si no existe ningún error, ya podrás realizar el siguiente paso que es la creación del .apk. En caso contrario, deberemos analizar los errores y subsanarlos.
10. Accedemos al menú *Run/Run*.



11. Si no sale un cuadro de diálogo le indicamos que queremos una aplicación Android y automáticamente se creará el .apk y este será lanzado. La aplicación tendrá la siguiente apariencia:



Práctica: Mi InstagramNDK.

Partiendo del ejercicio anterior, trata de realizar una aplicación que tome una imagen a partir de la cámara del dispositivo móvil (mediante el SDK de Android, es decir, mediante código Java). Modifica el programa realizado anteriormente para que la imagen sea la tomada por la cámara. Añade otro procesamiento de imagen en código nativo para que la imagen pase a tono sepia. A continuación, se indica el algoritmo que has de utilizar para cada píxel; recordad que debemos mantener la componente *alpha* de la imagen:

```
outputRed = (inputRed * .393) + (inputGreen * .769) + (inputBlue * .189)
outputGreen = (inputRed * .349) + (inputGreen * .686) + (inputBlue * .168)
outputBlue = (inputRed * .272) + (inputGreen * .534) + (inputBlue * .131)
```

El resultado será:

