FAST TRACK

Android

Guía básica de programación en Android

Si quieres comenzar a **programar en Android** y no sabes cómo, aquí te presentamos nuestra guía básica de programación para Android. Un manual muy didáctico donde podrás **descubrir paso a paso** todo lo que necesitas para comenzar a hacer tus primeros programas y juegos, desde que escribes las primeras líneas de código hasta que lo **publicas en Google Play**.

Índice de la guía

Instalación y descarga de Recursos

En internet encontraremos muchos tutoriales dedicados a este tema, pero la mayoría tiene un problema: se han quedado obsoletos. Con el paso del tiempo, Google ha ido haciendo cambios en las herramientas para desarrollar Android. No siempre han sido del agrado de los usuarios porque se veían obligados a cambiar su forma de trabajar, o incluso se encontraban con que los manuales y las guías que usaban dejaban de estar al día.

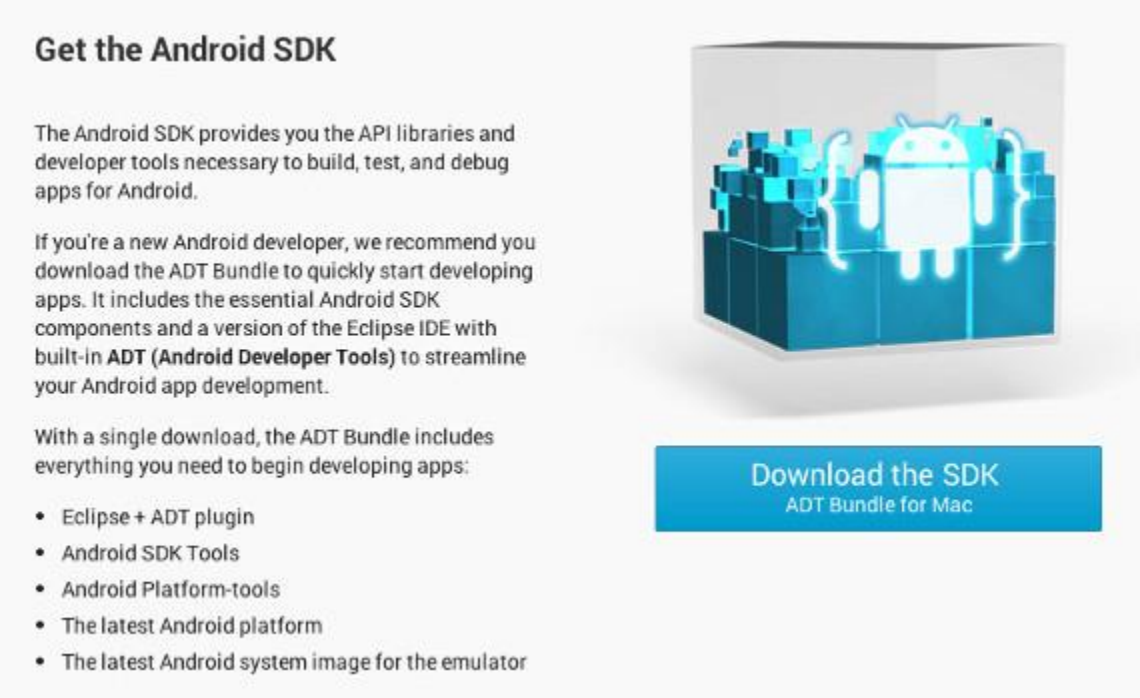
Afortunadamente, el proceso de instalación del entorno de desarrollo de Android es más fácil que nunca, y aquí lo vamos a ver en detalle. Partiremos del supuesto de no tener nada instalado, y nuestro objetivo es desarrollar nuestra primera aplicación en Android, el famoso Hola Mundo.

Descarga del SDK

En este enlace descargaremos en un solo paso todo el entorno de desarrollo. El paquete incluye casi todo lo que necesitaremos:

- Eclipse + plugin ADT
- Las tools del Android SDK
- Las herramientas de plataforma Android
- La plataforma Android más reciente
- Emuladores más recientes

Si estás más familiarizado con Android, puedes descargarte solamente el SDK. La web te mostrará un único enlace en función de tu sistema operativo, pero también puedes descargar las otras versiones. Existen para Windows de 32 y 64 bits, Linux de 32 y 64 bits, y Mac. En caso de duda, ve directamente a la descarga sugerida, Mac para este ejemplo:



Instalación

Para las 3 plataformas tendremos que tomar nota del directorio donde se instala el SDK, lo necesitaremos más adelante.

Mac

Descargaremos un zip que se descomprimirá en una carpeta con un nombre del tipo adt-bundle-mac-x86_64-xxxx . La moveremos a un directorio que conozcamos, típicamente ~/Development.

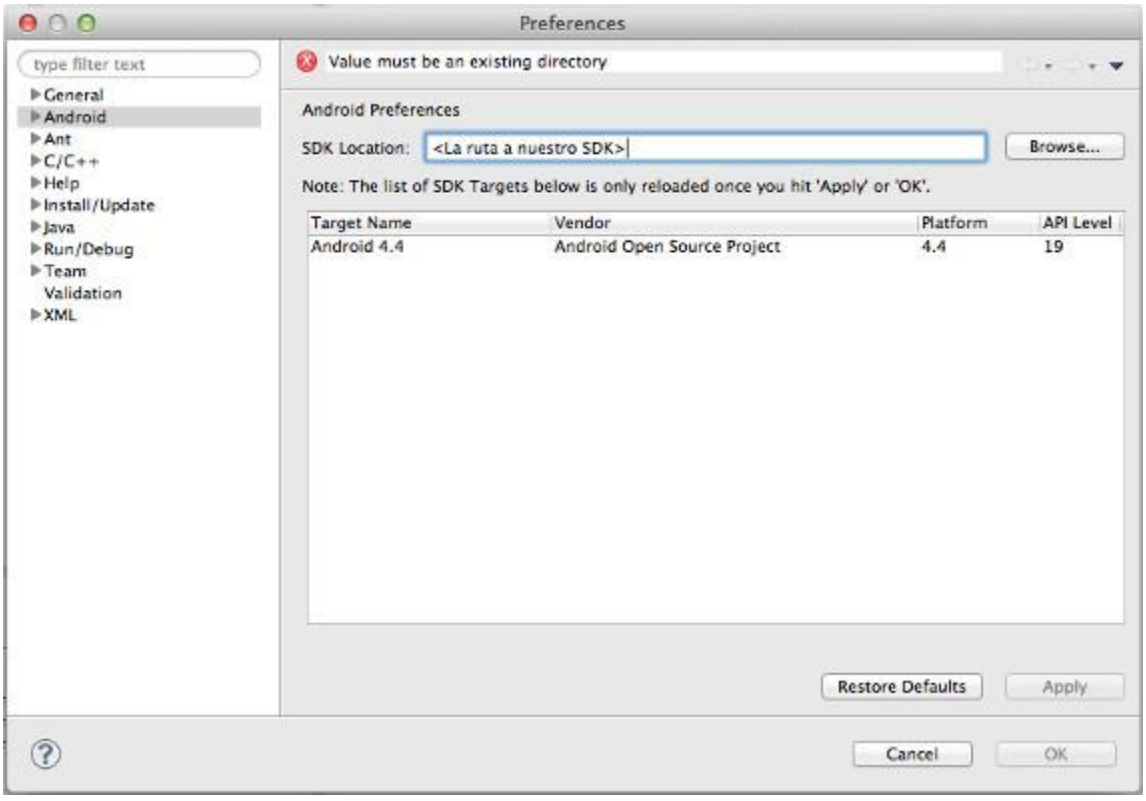
Linux

Funcionará de forma muy parecida a Mac, seguiremos los mismos pasos.

Windows

Descargaremos un .exe que nos guiará durante todo el proceso, descargando el Java JDK si es necesario.

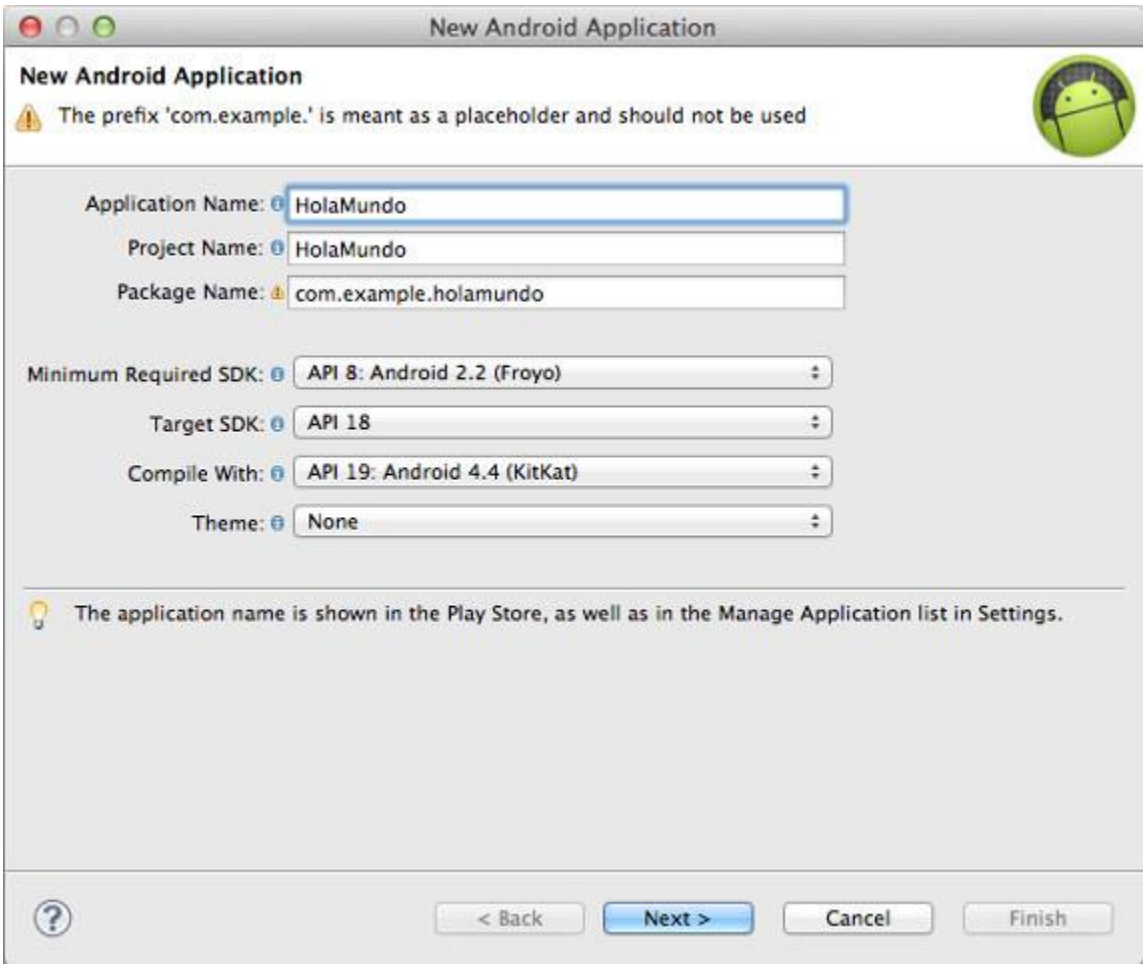
Por último arrancaremos Eclipse, y nos iremos a Preferences → Android, y pondremos la ruta al SDK que nos corresponda.



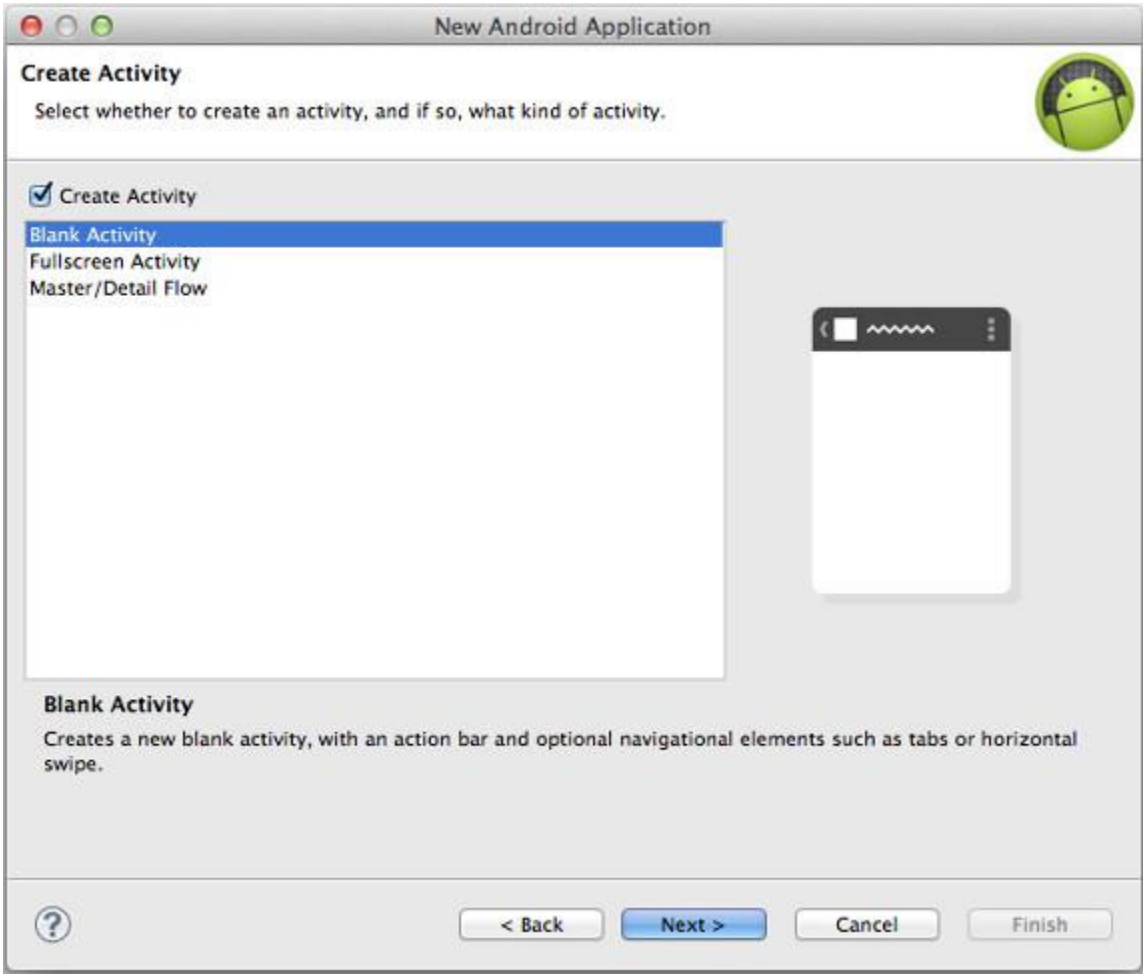
Con estos sencillos pasos, ya estamos preparados para hacer nuestro primer proyecto y arrancarlo.

Primer proyecto

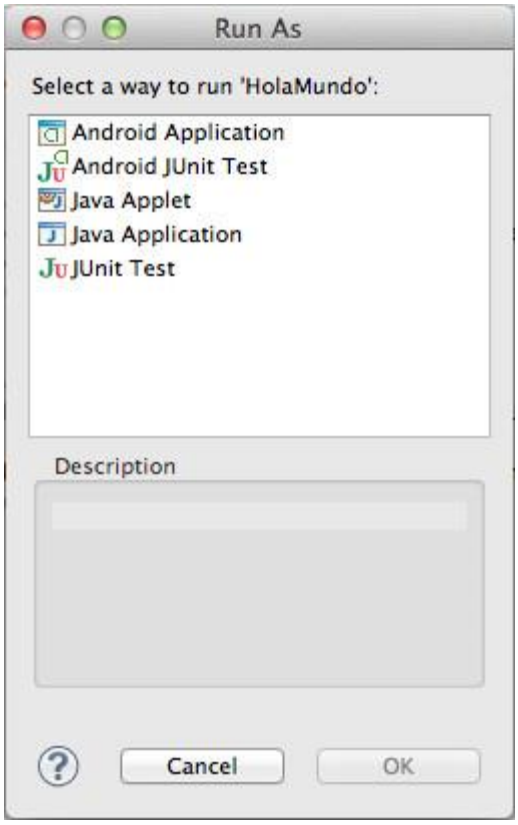
Haremos File → New → Android Application Project, y pondremos “Hola Mundo” como nombre de la aplicación. Pondremos el tema a “none”.



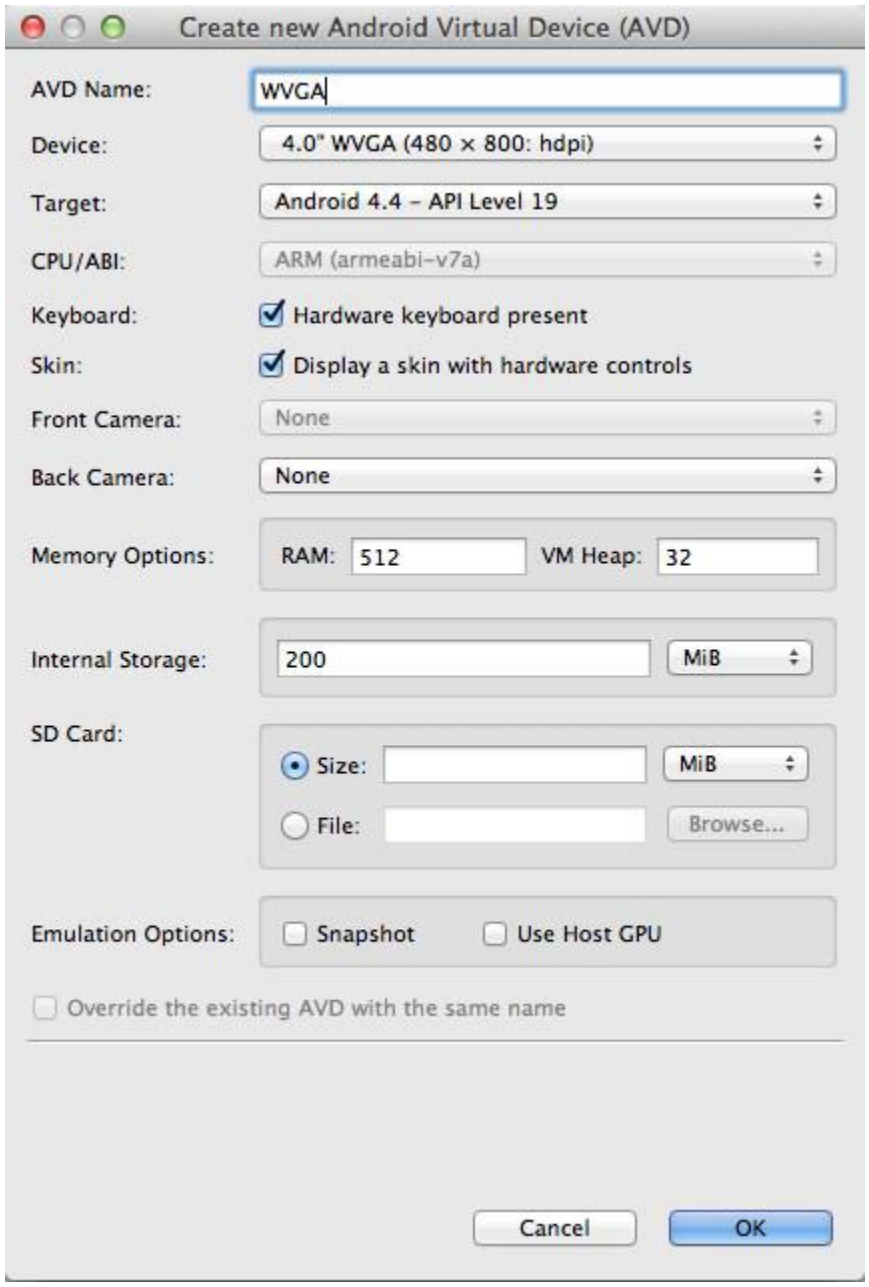
Aceptamos todo, y cuando lleguemos a la pantalla de crear actividad, la dejaremos vacía, y continuamos hasta finalizar el asistente.



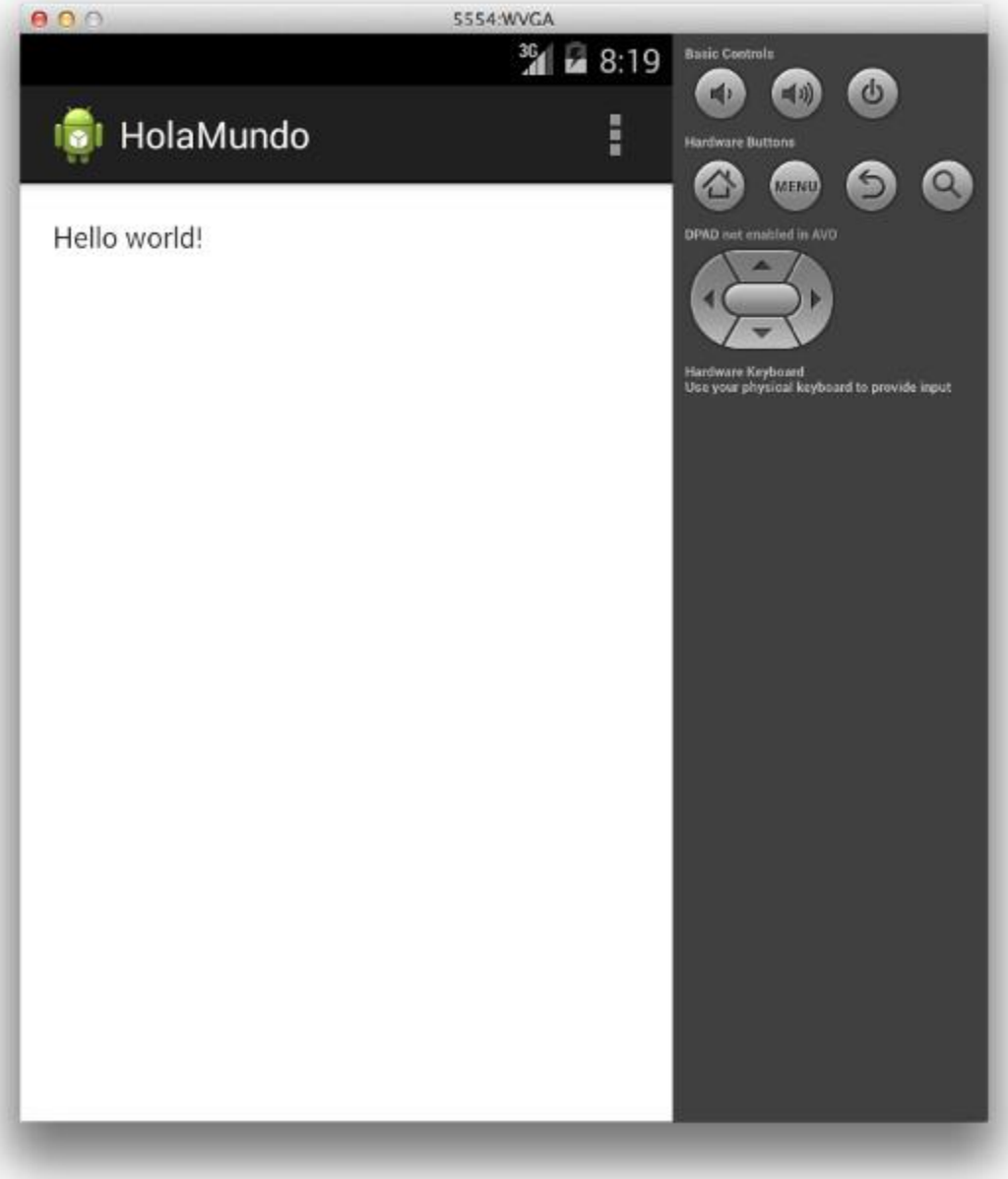
Pulsaremos sobre la flecha verde en la barra de herramientas superior, y ejecutaremos nuestro proyecto como una aplicación de Android:



Como aún no tenemos ningún emulador virtual (AVD) creado, tendremos un mensaje de error al que contestaremos que sí. Cuando tengamos abierto el Android Virtual Device Manager, crearemos uno nuevo de este modo:



Aceptamos, continuamos, y si todo ha salido bien, tras unos instantes de carga, tendremos nuestro primer proyecto andando. ¡Felicidades!

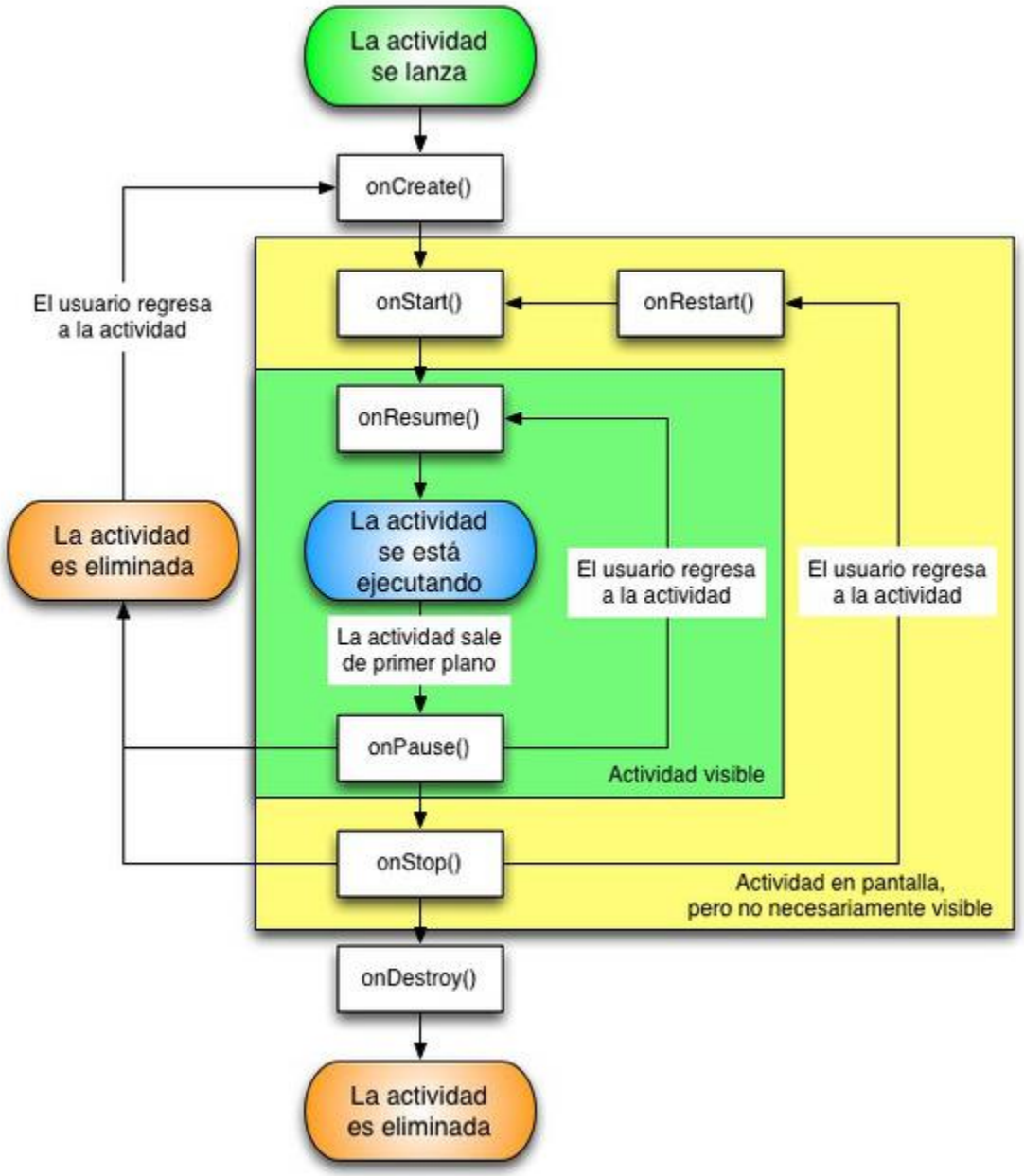

5554:WVGA
8:19
HolaMundo
Hello world!
Basic Controls
Hardware Buttons
MENU
DPAD not enabled in AVD
Hardware Keyboard
Use your physical keyboard to provide input

Ciclo de vida de Android

Cuando se empieza a programar en un lenguaje como C++ o Java, lo primero que se enseña es el método main, el punto al que llamará el sistema operativo cuando vayamos a arrancar nuestra aplicación.

En Android no existe un método **main** como tal, pero sí existen varios métodos de nuestra actividad que serán llamados por el SSOO cuando ocurran **eventos importantes**. En este capítulo estudiaremos a fondo cuáles son esos eventos, y cómo funciona **el ciclo completo de una actividad** de Android. La documentación oficial ofrece una explicación extensa de este tema, aquí estudiaremos los elementos más importantes junto con algunos errores comunes a la hora de manejarlos.

El ciclo de vida de Android sigue este esquema:



Los eventos del ciclo de vida

1. onCreate(Bundle)
    - Representa el momento en el que **la actividad se crea**. Este método normalmente lo generará el asistente al crear una nueva actividad en Android, y es donde crearemos todo lo que vaya a necesitar la actividad. Si antes hemos salvado los datos de la actividad en un objeto Bundle, podremos utilizarlo para regenerarla. Normalmente no lo usaremos.
2. onStart()
    - La actividad va a pasar a **estar en pantalla**, aunque no necesariamente visible. Si venimos de una parada, pasaremos antes por onRestart().
3. onRestart()
    - Anterior a onStart() cuando procedemos de una llamada a onStop().
4. onResume()
    - La actividad va a empezar a **responder a la interacción** del usuario.
5. onPause()
    - La actividad va a **dejar de responder** a la interacción del usuario.
6. onStop()
    - La actividad ha pasado completamente a **segundo plano**.
7. onDestroy()
    - La actividad **va a ser destruida** y sus recursos liberados.

Cuando necesitemos implementar uno de estos métodos, lo haremos añadiendo a nuestra actividad con estos perfiles:

```
public class MiActividad extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
    }
    protected void onStart() {
        super.onStart();
        ...
    }
    protected void onRestart() {
        super.onRestart();
        ...
    }
    protected void onResume() {
        super.onResume();
        ...
    }
    protected void onPause() {
        ...
        super.onPause();
    }
    protected void onStop() {
        ...
        onStop();
    }
    protected void onDestroy() {
        ...
        super.onDestroy();
    }
}
```

Es importante mantener la **llamada al método de la superclase** para no encontrarnos con sorpresas. Las tareas de cada evento que están por encima de nuestra actividad deben mantenerse. Esta llamada irá al principio de los eventos de entrada, y al final de los de salida. De esa forma nos evitaremos sorpresas, ya que los elementos de la actividad que necesitemos pero que no estarán bajo nuestro control estarán creados antes de usarlos, y se destruirán después.

No tenemos que añadir todos los eventos, los que no necesitemos usarán la implementación por defecto. Los métodos que a menudo utilizaremos -y no se recomienda tocar los demás- son onCreate, onPause, y onRestart.

El significado de onCreate es claro: es el lugar donde cargaremos los recursos que necesitemos, las vistas, y cualquier otra cosa que necesitemos. Para la salida, el único método en el que nos centraremos es onPause. La razón de evitar onStop y onDestroy es que no tenemos ningún control sobre ellas. onPause se ejecutará siempre que la aplicación salga de primer plano, mientras que los otros dos los ejecutará el sistema operativo en función de sus necesidades. **¡Puede que nunca lleguen a ejecutarse!** Esto se hace para evitar el coste que tiene crear la actividad una y otra vez si pasamos de la actividad al escritorio y viceversa, y el sistema operativo sólo liberará los recursos utilizados si los necesita, y no tiene para atender todos los procesos abiertos.

Eso implica que asumiremos que la aplicación morirá después de ejecutar onPause, y que es nuestra **última oportunidad de salvar los datos** que necesitemos salvar, y de parar servicios que estemos utilizando, como la geolocalización. Si hemos parado servicios, el sitio adecuado para reiniciarlos es onRestart.

Los demás métodos no necesitaremos usarlos mucho. Un caso habitual en el que sí nos hará falta será cuando integremos librerías de terceros, como Facebook o Flurry. En estos casos, se nos pedirá que unamos los métodos de nuestra actividad a su código. Por ejemplo, para registrar una sesión de Flurry se nos pedirá que la sesión comience en el método onStart.

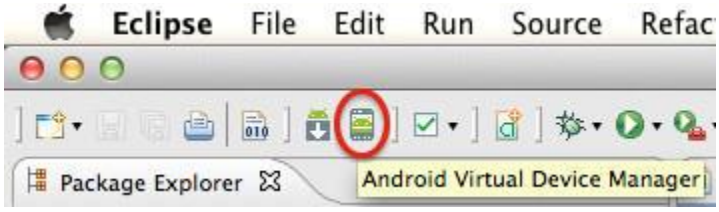
Algunas ideas útiles

- **Recuerda para qué sirve cada evento**. Siempre necesitarás onCreate, y bastante a menudo necesitarás onPause y onResume para parar y relanzar los servicios que más consuman.
- **No toques los demás eventos** si no lo necesitas expresamente.
- No confíes en onStop y onDestroy, **podrían no llamarse nunca**. Salva todo lo que necesites en onPause.
- **Evita usar variables estáticas no finales**. La aplicación puede seguir cargada cuando regreses, y conservarán los valores con los que se quedaron. Si no tienes más remedio que usarlas, asegúrate de reiniciar sus valores al volver al primer plano.

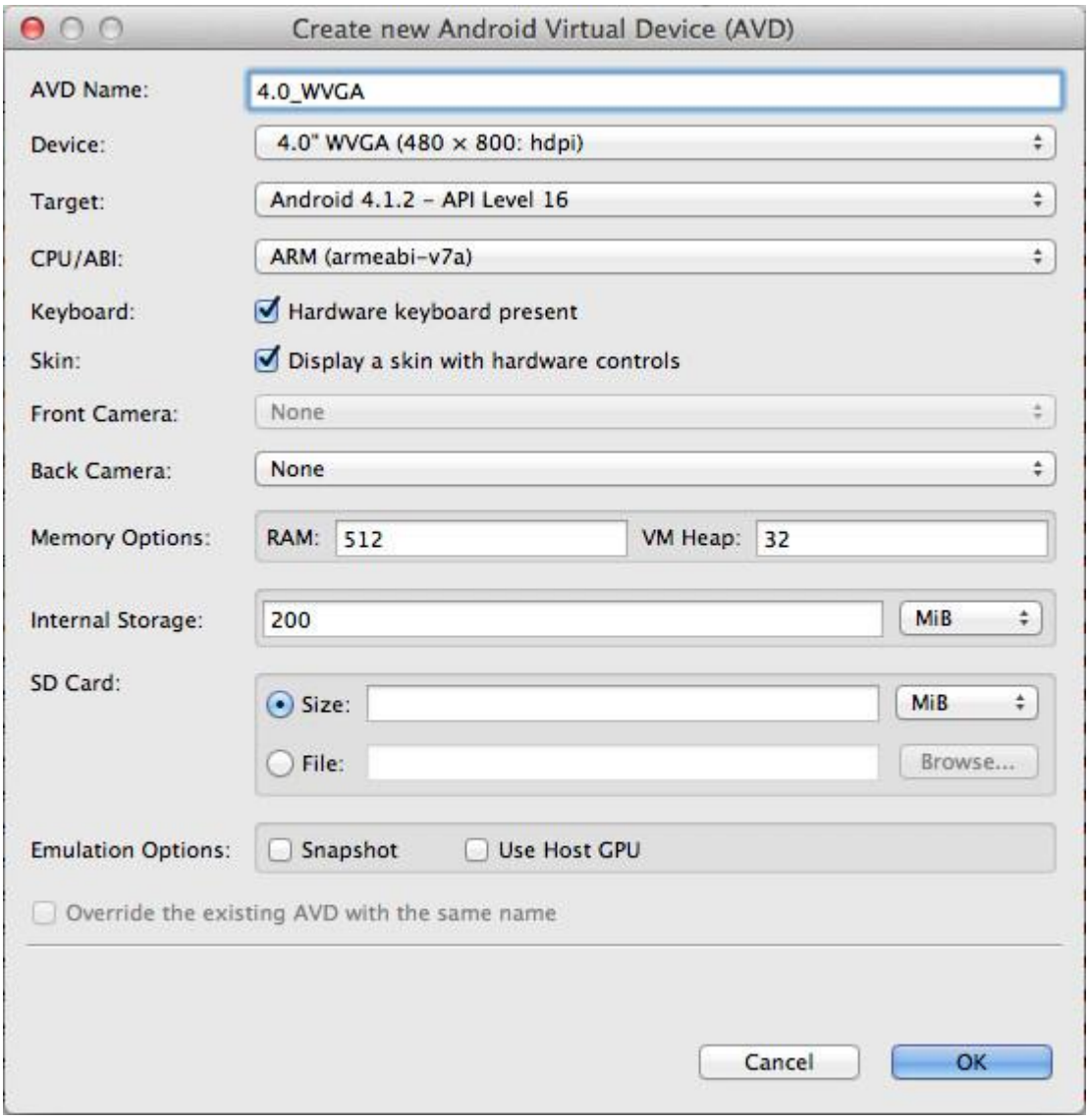
No siempre tendremos un dispositivo Android a mano. Incluso teniéndolo, necesitamos **verificar** de alguna manera que nuestra aplicación funcionará bien en **tantos modelos Android diferentes como sea posible**. Aquí entran en escena los emuladores. Con ellos podremos evitar caer en algunos errores frecuentes en el desarrollo. Pero, como veremos más adelante, para tener una mayor seguridad lo mejor es echar mano de dispositivos reales.

Creación y gestión de emuladores

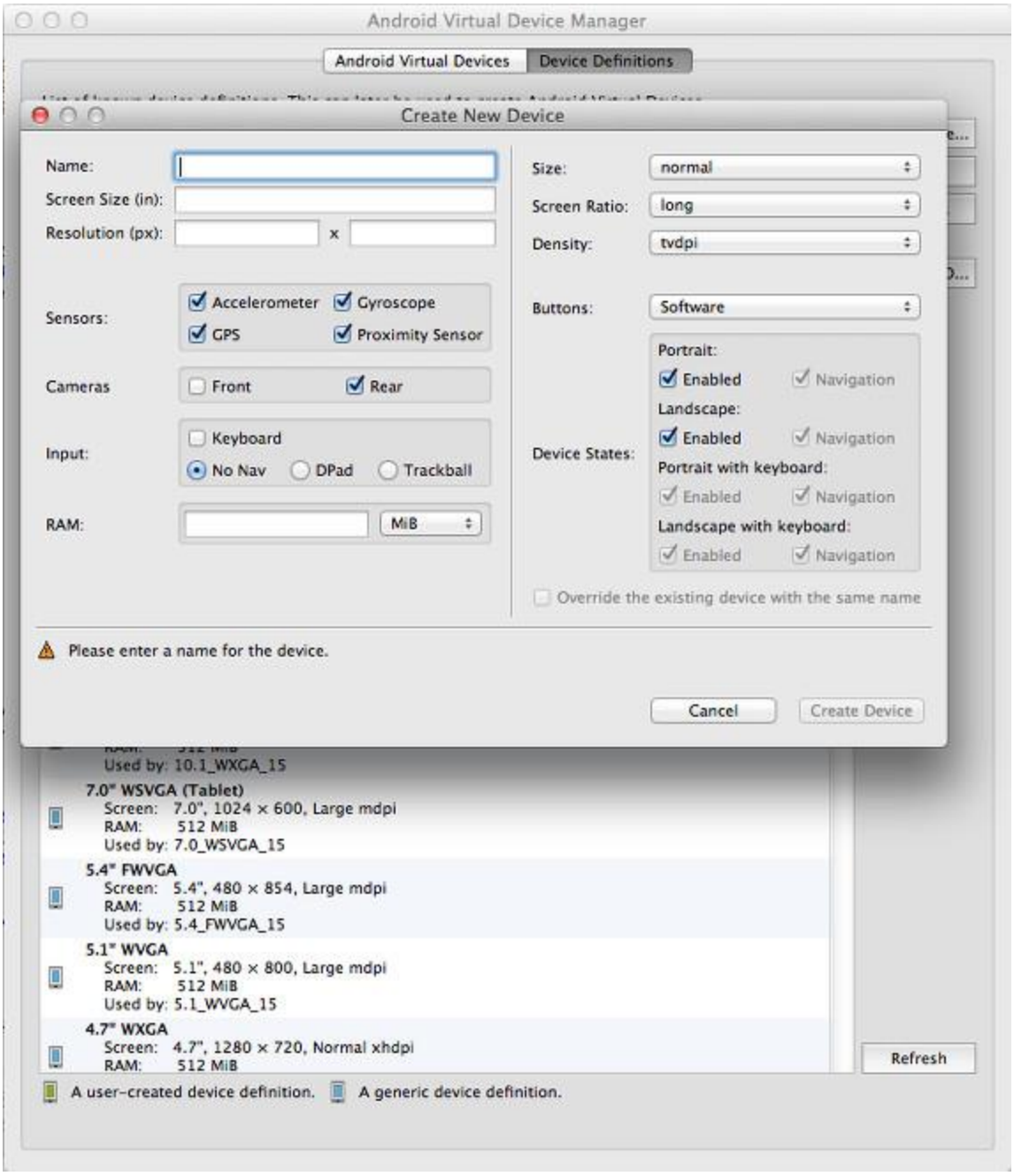
Accederemos al gestor de emuladores (Android Virtual Device Manager) a través de este botón:



Veremos una ventana donde podremos acceder a dos listas: la de los dispositivos virtuales que tenemos creados, y de las definiciones de dispositivos. Para crear un **dispositivo virtual (AVD)** nuevo, pulsaremos en New, y lo rellenaremos de la siguiente manera:



Con esto, habremos creado nuestro primer emulador. Cuando necesitemos editar nuestros AVDs, lo seleccionaremos en la lista y pulsaremos "Edit", para volver a ver esta pantalla. En ella podemos editar detalles como la cámara frontal, la trasera, las opciones de memoria y almacenamiento, etc. Pero para los primeros desarrollos, estas opciones las dejaremos sin tocar, nos interesan solamente **"Device"** y **"Target"**. Con "Device" **elegiremos un dispositivo** de entre las definiciones que tenemos. Todos los modelos de uso frecuente ya están predefinidos, por lo que en principio no necesitaremos crear nuevas definiciones. Con "Target" podremos decidir **la versión del sistema operativo** que tendrá nuestro emulador. A día de hoy el valor más adecuado es el API 16 (v 4.1.2), que cubre la mayoría de los terminales. Si necesitamos crear una nueva definición de dispositivo, lo haremos a través de esta pantalla:

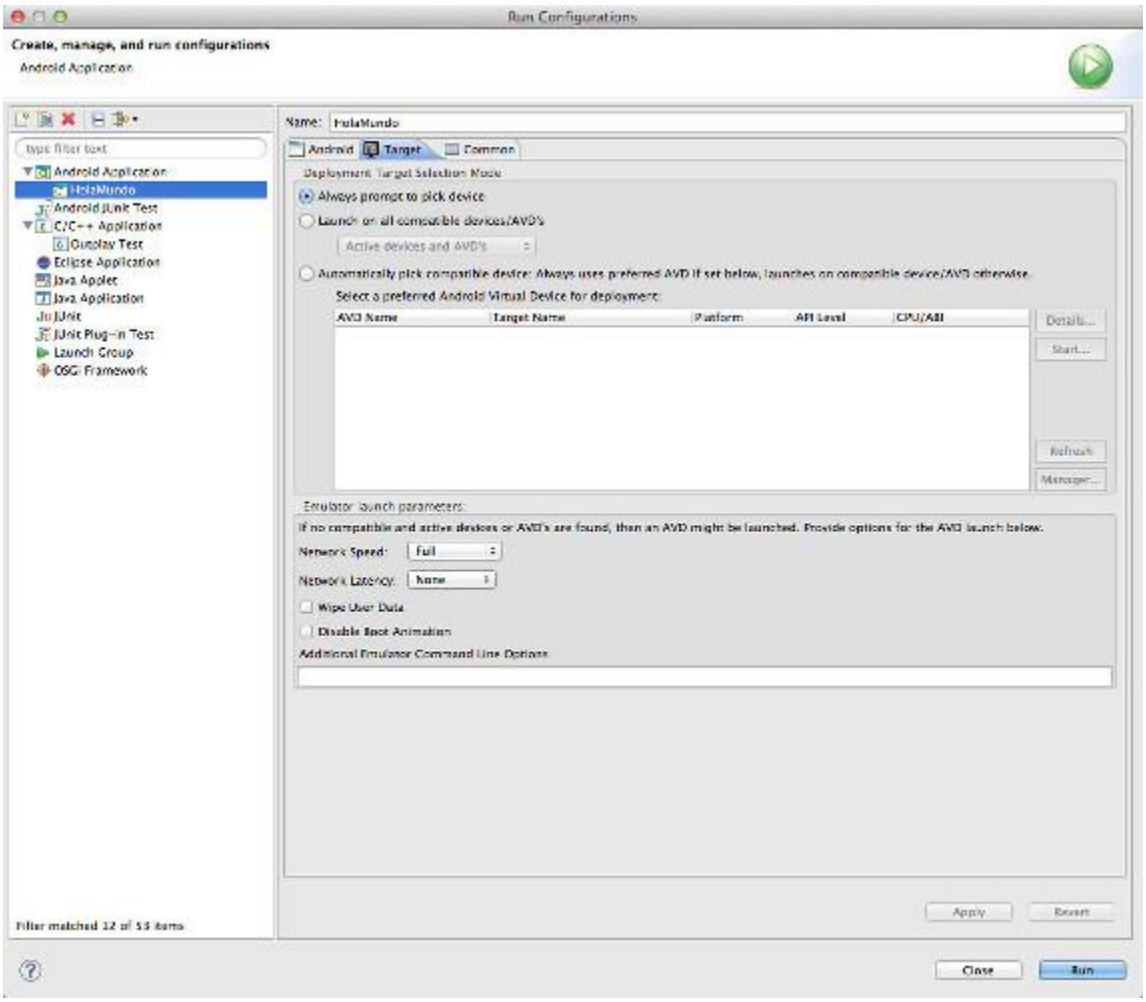


Normalmente no lo necesitaremos, ya que las definiciones predefinidas incluyen la mayoría de los modelos que existen. Si necesitamos crear un nuevo modelo de dispositivo -normalmente será porque no tenemos ninguno con una determinada resolución de pantalla-, indicaremos su nombre, **tamaño de pantalla en pulgadas, y resolución en píxeles**. Los valores "size", "screen ratio", y "density" se autocalcularán y lo más probable es que no necesitemos tocarlos. También podemos decidir qué elementos hardware queremos añadir, como el acelerómetro o el GPS, el teclado físico, etc.

Uso de emuladores

Si aún estamos empezando a manejar Android, veremos que al ejecutar nuestro proyecto se elige automáticamente el dispositivo (o AVD) en el que se ejecuta nuestra aplicación. Si necesitamos que se ejecute en un dispositivo (real o emulado) concreto, tendremos que cambiar un ajuste primero. Dentro de Run → Run

Configurations, elegiremos la configuración de nuestro proyecto, y en la pestaña Target activaremos elegir siempre el dispositivo:



Con esto, ya estaremos preparados para usar tantos emuladores como necesitemos o queramos.

Ventajas y limitaciones de los emuladores

Un emulador es una aproximación no del todo perfecta hacia un dispositivo real. **Hay utilidades que no tendremos disponibles**, normalmente relacionadas con características avanzadas. Por ejemplo, no se puede emular Bluetooth, ni Google Maps. Si se diese el caso de estar diseñando juegos que utilicen OpenGL, el emulador nos servirá más bien poco. Entonces, ¿por qué usar emuladores si tienen limitaciones y si tenemos un dispositivo real? Hay una razón muy importante, y es lidiar con los **diferentes tamaños de pantalla que existen**.

Si sólo trabajamos con el dispositivo que tenemos, podemos encontrarnos con la desagradable sorpresa de que para los demás tamaños no hemos diseñado bien las pantallas. Una de las formas que hay de evitar esto es **trabajar siempre con proporciones y medidas relativas** y escalables, nunca absolutas. Con el emulador tendremos más fácil el poder comprobar otros tamaños de pantalla sin tener que comprar varios dispositivos.

Este problema es muy importante porque **hoy día no hay control sobre los tamaños de pantalla existentes**. En la práctica podemos considerar que hay infinitos, y que no podemos hacer una solución expresa para cada uno. Porque el aspect ratio, o relación entre el ancho y el alto, también es variable. En cualquier momento un

fabricante puede diseñar un nuevo tamaño y dejar nuestra aplicación obsoleta. Por eso hay que asumir que no podremos comprobarlos todos, porque no podremos comprar todos los modelos existentes.

La mejor solución consiste en trabajar con valores relativos, y comprobar nuestra aplicación con tamaños variados de pantalla. No tendremos el 100% de seguridad, pero si nuestra aplicación funciona bien en una buena variedad de tamaños, tendremos más posibilidades de ir por el buen camino. Por lo tanto, ésta es la razón más importante para combinar nuestro dispositivo real con emuladores: poder probar diferentes tamaños de pantalla sin tener que comprar todos los dispositivos que existen.

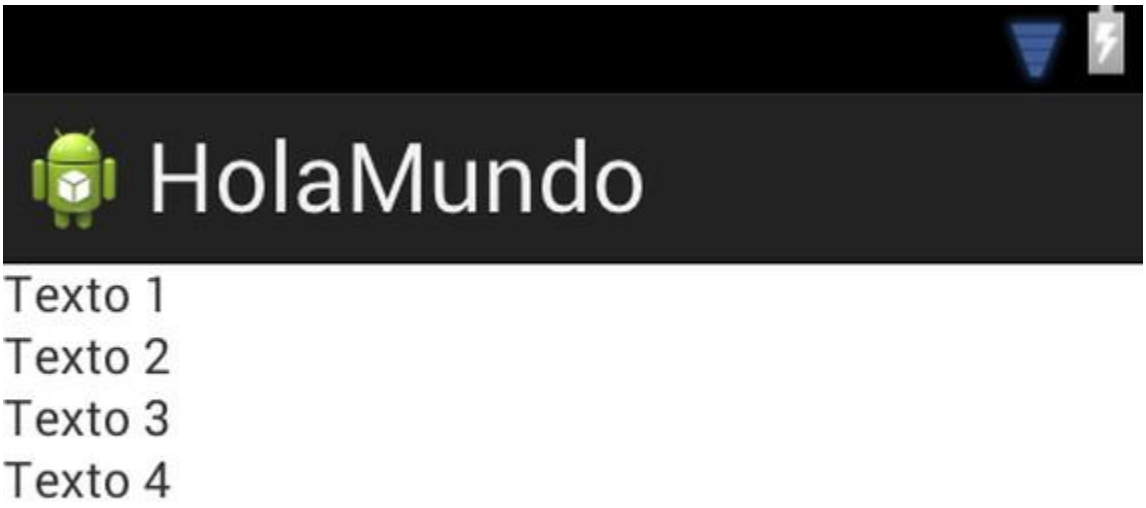
Todos los elementos que aparecen en la pantalla de una aplicación de Android son vistas. Desde los elementos sueltos como textos o botones, a los contenedores como los grupos de vistas. Este tema es bastante complejo por la cantidad de detalles que intervienen, y para explorar las posibilidades disponibles más a fondo, es muy recomendable acudir a la documentación oficial. En este tutorial analizaremos los elementos más socorridos, incluyendo grupos de vistas y algunos elementos básicos.

Grupos de vistas

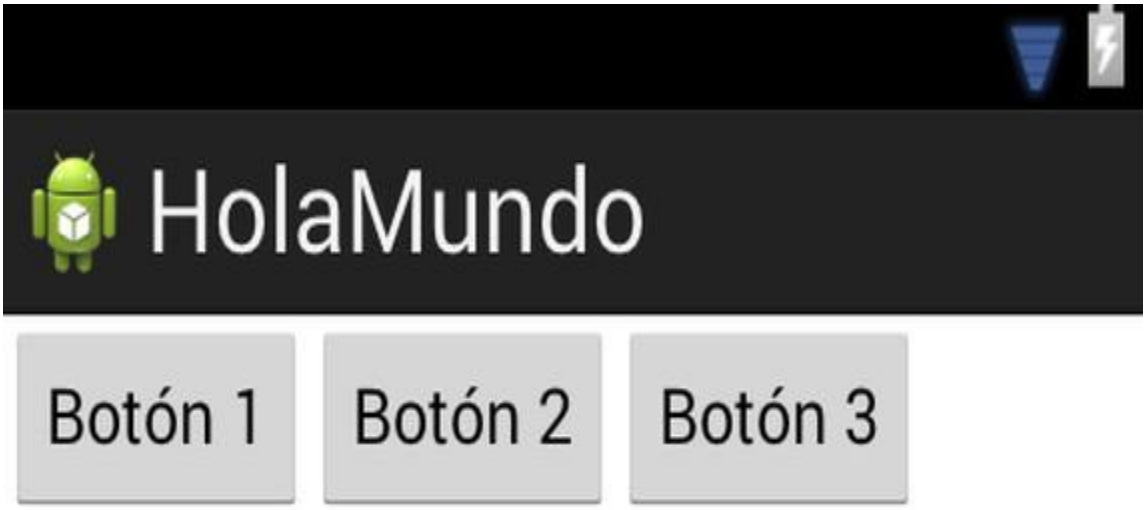
- **LinearLayout**
  - Agrupa los elementos en una sola línea, que puede ser vertical u horizontal.
- **RelativeLayout**
  - Los elementos se disponen en relación entre ellos y los márgenes. Es la más flexible, y la más utilizada.
- **ScrollView**
  - Se utiliza para vistas que no caben en pantalla. Sólo puede contener una vista o grupo de vistas, y añade automáticamente las barras de desplazamiento.
- **TableLayout**
  - Agrupa los elementos en filas y columnas. Contiene elementos TableRow, que a su vez contienen los elementos de cada celda.
- **FrameLayout**
  - Está pensada para contener una sola vista. Si se añaden más, todas se alinean en la esquina superior izquierda, solapándose.
- **AbsoluteLayout**
  - Está desaprobado desde la versión 1.5 de Android. En este contenedor, los elementos se referencian con coordenadas absolutas partiendo de la esquina superior izquierda. Se ha desaprobado porque no se adapta a pantallas de diferentes tamaños, que se popularizaron justamente a partir de Android 1.5.

Para una aplicación sencilla, los grupos más interesantes para ver en detalle son LinearLayout, RelativeLayout, y ScrollView. Por ejemplo, podemos hacer una **LinearLayout** vertical conteniendo textos, y otra horizontal con botones:

```
1
2
3  <?xml version="1.0" encoding="utf-8"?>
4  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     android:orientation="vertical">
8     <TextView
9        android:id="@+id/textView1"
10       android:layout_width="wrap_content"
11       android:layout_height="wrap_content"
12       android:text="Texto 1" />
13    <TextView
14       android:id="@+id/textView2"
15       android:layout_width="wrap_content"
16       android:layout_height="wrap_content"
17       android:text="Texto 2" />
18    <TextView
19       android:id="@+id/textView3"
20       android:layout_width="wrap_content"
21       android:layout_height="wrap_content"
22       android:text="Texto 3" />
23    <TextView
24       android:id="@+id/textView4"
25       android:layout_width="wrap_content"
26       android:layout_height="wrap_content"
         android:text="Texto 4" />
    </LinearLayout>
```



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="horizontal">
   <Button
      android:id="@+id/button1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Botón 1" />
   <Button
      android:id="@+id/button2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Botón 2" />
   <Button
      android:id="@+id/button3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Botón 3" />
</LinearLayout>
```



Una **ScrollView** se puede componer de forma muy fácil, sólo tiene que envolver al contenedor que debe escrolar:

```
<ScrollView
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   xmlns:android="http://schemas.android.com/apk/res/android"
   <...>
</ScrollView>
```

Por último, una RelativeLayout es el contenedor más versátil y poderoso, aunque también uno de los más complicados de manejar. No tocaremos aquí todas las opciones porque son demasiadas. Para una referencia más detallada acudiremos a la documentación oficial. Un ejemplo sencillo sería el siguiente: dos botones en la misma horizontal, uno con el tamaño del texto que contiene, y el otro se expande para ocupar el resto de la fila:

```
1
2
3  <?xml version="1.0" encoding="utf-8"?>
4  <RelativeLayout
5    xmlns:android="http://schemas.android.com/apk/res/android"
6    android:layout_width="match_parent"
7    android:layout_height="match_parent" >
8    <Button
9      android:id="@+id/button1"
10     android:layout_width="wrap_content"
11     android:layout_height="wrap_content"
12     android:layout_alignParentLeft="true"
13     android:layout_alignParentTop="true"
14     android:text="Botón 1" />
15   <Button
16     android:id="@+id/button2"
17     android:layout_width="wrap_content"
18     android:layout_height="wrap_content"
19     android:layout_alignParentRight="true"
20     android:layout_alignParentTop="true"
21     android:layout_toRightOf="@+id/button1"
22     android:text="Botón 2" />
23 </RelativeLayout>
```

En este ejemplo, el primer botón se alinea con el margen izquierdo y superior del contenedor, y el botón 2 se alinea con el margen superior, el derecho, y a la derecha del botón 1.

Vistas

- **TextView**
  - Muestra un texto fijo.
- **EditText**
  - Contiene un texto editable.
- **Button**
  - Botón simple.
- **ImageButton**
  - Este botón permite mostrar una imagen en vez de un texto
- **ToggleButton**
  - Botón que puede mantener su estado de pulsado hasta que se pulsa otra vez.
- **CheckBox**
  - Botón similar a ToggleButton que funciona como una caja de verificación.

Ya hemos visto antes cómo funciona un TextView, ya que sólo debe incluir un texto. En la documentación oficial podemos encontrar opciones más avanzadas como variar la fuente, el tamaño del texto, el color, y muchas más.

Más interés tienen los botones, ya que de alguna forma tenemos que asociarles una acción. Vamos a ver dos formas. En una, asociamos directamente la acción en el código de nuestra actividad:

```
Button button = (Button) findViewById(R.id.button1);
button.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    DisplayToast("Has pulsado el botón");
  }
});
```

Aquí la clave es el id que le hemos dado al elemento en el fichero XML, lo necesitamos para localizarlo en el código. Con él, podemos asociarle la acción que necesitemos. Otra alternativa sería incluir en el XML del botón el elemento "android:onClick="btnClicked" , y luego añadir directamente en el código de la actividad el método con el nombre indicado:

```
public void onClick(View v) {
  DisplayToast("Has pulsado el botón");
}
```

Para un CheckBox o un ToggleButton podemos hacer algo similar al primer método. Obtenemos la referencia a través de findViewById, y aplicamos el siguiente fragmento:

```
button.setOnCheckedChangeListener(new OnCheckedChangeListener() {
  @Override public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
    if (isChecked) DisplayToast("Has activado el botón");
    else DisplayToast("Has desactivado el botón");
  }
});
```

Los **recursos** son una parte fundamental en una aplicación Android. Siempre que sea posible **los separaremos del código**, de ese modo conseguiremos una aplicación más mantenible. Si los organizamos bien, Android también se encargará de elegir por nosotros el elemento adecuado dependiendo de la configuración -típicamente, del tamaño de pantalla y del idioma del usuario-.

Recursos básicos: imágenes y textos

Los **textos** siempre han de separarse del código. Si no lo hacemos, nos costará más mantener el código: habría que revisar todas las ocurrencias y se nos podría olvidar alguna cada vez que tengamos que hacer cambios. Lo peor de todo es que entonces nuestra aplicación no será internacionalizable.

Por eso, en el código no se recomienda poner textos. La única excepción podrían ser mensajes no destinados al usuario final, como mensajes de salida de consola. Pero en general, todos nuestros textos tendrán que ir a un **fichero de recursos separado**. Ese fichero lo llamaremos **strings.xml**, y su ruta completa dentro del proyecto será **res/values/strings.xml**.

Un fichero strings.xml tendrá un contenido similar a éste:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="string_1">text_1</string>
   <string name="string_2">text_2</string>
</resources>
```

Ahora, en nuestro código podremos recuperar estos textos con sentencias como getResources().getString(R.string.string_1), o dentro de nuestros layouts como @string/string_1 . De este modo hemos conseguido lo que queríamos: el código pasa a ser independiente de los recursos.

El otro recurso que utilizaremos habitualmente serán las **imágenes**. Para ello tenemos dentro del proyecto varias carpetas, como res/drawable, res/drawable-hdpi, res/drawable-xhdpi, y similares.

Los sufijos que siguen a “drawable” se refieren a la **densidad de la pantalla**. Cada dispositivo Android tiene una densidad de píxeles por pantalla que se agrupa en varios grupos:

- ldpi (pequeña, x0,75)
- mdpi (media x1)
- hdpi (grande x1,5)
- xhdpi (extra grande x2)
- xxhdpi (extra extra grande x3)

De este modo, la misma imagen puede adaptarse a **diferentes densidades de pantalla** sin que aparezcan deformidades. Se recomienda que en cada carpeta haya una copia de la imagen, respetando las proporciones (tomando mdpi como la base e igual a 1), aunque también se puede utilizar solamente una de ellas y entonces Android hará automáticamente una operación de reescalado para las demás resoluciones. Sin embargo, se recomienda cubrir todas las carpetas para ahorrar al sistema este esfuerzo extra.

La carpeta “drawable” a secas no contendrá imágenes, la utilizaremos solamente para definiciones xml de elementos avanzados, como los selectores.

Otros recursos

Dentro del directorio /res del proyecto pueden ir muchos más recursos. Sonidos, definiciones de colores usables de una forma similar a los textos, dimensiones para nuestra aplicación, estilos para nuestras vistas, y mucho más.

Para quien esté interesado en aprender más sobre estos recursos, puede consultar la documentación oficial de Android.

Internacionalización en Android

Con las imágenes ya hemos hecho una introducción al **sistema de recursos alternativos en Android**: si tenemos una pantalla de densidad xhdpi (la de un Galaxy S3, por ejemplo), Android preferirá las imágenes del directorio drawable-xhdpi sobre las demás.

Ese sistema se puede utilizar para distinguir muchas otras configuraciones: orientación normal o apaisada, modo en conducción o normal, modo nocturno o normal… De nuevo, os recomendamos la documentación oficial de Android para conocer todos los detalles. Ya hemos hablado de cómo proporcionar imágenes según la densidad de la pantalla, ahora nos centraremos en otro uso muy importante para nuestra aplicación: la **internacionalización**.

Si queremos que nuestros textos estén disponibles en varios idiomas, tendremos que añadir una carpeta diferente para cada copia del fichero strings.xml. De este modo, si queremos portar nuestra aplicación a inglés y francés, tendríamos estos tres directorios:

- res/values (valores por defecto)
- res/values-en
- res/values-fr

Es decir, a la carpeta se le añade un sufijo con el código ISO del idioma concreto, y dejaremos los textos por defecto en la carpeta sin sufijo. Esto último es muy importante, porque salvo con las densidades de la pantalla, siempre tenemos que dar un **valor por defecto** para todos los recursos. Cuando requerimos un recurso del tipo que sea, Android buscará el que mejor encaje, y si no encuentra ninguno podemos encontrarnos con cierres inesperados de la aplicación. Por eso siempre debe haber un idioma que elijamos por defecto, y que todos los recursos tengan su valor por defecto.

De este modo, volvemos a conseguir lo que queríamos: además de haber separado los textos del código, ahora podemos tener la **aplicación en diferentes idiomas** sin tocar para nada nuestro código.

El sistema de generar recursos alternativos es, por supuesto, combinable. Siguiendo el enlace que hemos indicado antes, veremos en qué orden tienen que ir los sufijos. Si seguimos el ejemplo sencillo que estamos indicando en este tutorial, podríamos pensar en tener imágenes que sólo sean para la versión en inglés. Esto se consigue creando una carpeta res/drawable-en-xhdpi, o cualquier otra densidad de pantalla que necesitemos. De ese modo podemos proveer una gran variedad de recursos y no tener que preocuparnos por elegir el correcto. Eso sí: sin olvidar que la densidad de pantalla es el único modificador que no necesita valores por defecto. Todos los demás necesitan un valor por defecto o nos arriesgamos a que la aplicación se nos pare.

La fragmentación de Android es la mayor dificultad con la que se encuentran los desarrolladores para lanzar una aplicación al mercado. Android está muy lejos de ser una plataforma unificada, con unos pocos dispositivos tal como sucede con iOS.

Algunos números sobre la fragmentación

Para hacernos una idea de cómo está Android de dividido, podemos ver un caso real de uso. Existen varias empresas que publican aplicaciones muy utilizadas, y recogen más adelante los datos de uso. Una de ellas es OpenSignal, que recientemente ha publicado su último estudio.

Los números son demoledores:

- 18.796 dispositivos Android diferentes vistos este año, frente a 11.868 el año pasado (un 58% más).
- Samsung es el fabricante líder destacado con un 43% de los dispositivos. El resto se lo reparten más de 80 fabricantes diferentes.
- Hay activas 6 versiones diferentes del sistema operativo con un número de usuarios lo bastante importante como para ser ignorado.
- También hay un número muy elevado de diferentes resoluciones y tamaños de pantalla. Y por supuesto, con diferentes ratios entre el alto y el ancho.

A estos datos hay que añadir diferentes elementos de hardware, como un conjunto de sensores que puede variar de un dispositivo a otro, o un procesador gráfico diferente que hace que los desarrolladores de juegos en OpenGL tengan que cubrirlos todos.

En fin, una pesadilla, que si no controlamos adecuadamente nos puede costar más de un disgusto. No es raro encontrar proyectos en Android en los que tras acabar la primera versión se acaba gastando más tiempo en portar para los diferentes modelos que en la propia primera versión. Puede llegar a ser muy frustrante.

Enfrentándose a la fragmentación

Aunque es una tarea complicada, si seguimos una cierta disciplina en el desarrollo podremos conseguir un buen resultado en un tiempo razonable. Para eso, empezaremos con un par de consideraciones previas.

Trabajar con la fragmentación desde el principio

Crear primero una versión específica para un móvil en concreto y luego portar es un error frecuente. Es habitual caer en la comodidad que supone fijarnos sólo en el dispositivo que tenemos a mano, pero si vamos a sacar nuestra aplicación para un mercado amplio, dejar la fragmentación para el final nos obligará a costosos cambios en nuestro proyecto. Tardaremos más tiempo y cometeremos más errores. Por ejemplo, si no diseñamos nuestras vistas para ser flexibles y adaptarse a varios tamaños de pantalla, las tendremos que rehacer luego. Algo parecido a lo que pasaba con la localización de recursos.

En este sentido, hay una serie de preguntas que podemos hacernos antes de empezar, y que nos ayudarán a tener un mapa del camino.

- ¿Qué versión del sistema operativo quiero soportar? ¿Sólo móviles recientes, o quiero que mi aplicación funcione para los modelos más antiguos?
- ¿Quiero soportar sólo móviles, sólo tabletas, o ambos?

- ¿En qué países quiero publicar mi aplicación? ¿Qué idiomas quiero soportar?

Con la primera pregunta podremos plantearnos qué funcionalidad queremos incluir en nuestra aplicación. Si soportamos versiones antiguas tendremos que elegir entre sacrificar funcionalidad de las nuevas versiones de Android, o sacar diferentes versiones de nuestra aplicación. Mi recomendación personal es la primera opción, a menos que se tengan medios y desarrolladores suficientes para trabajar con dos versiones diferentes del mismo producto.

Con la segunda, tendremos claro cómo tendremos que desarrollar nuestras vistas, sin perder de vista las diferentes versiones de nuestros recursos gráficos. Por último, aparte de la localización de los textos, hay que tener en cuenta que dependiendo del país donde publiquemos nuestra aplicación, habrá móviles más antiguos o más modernos.

Asumir que no se pueden cubrir todos los móviles

Con tanta fragmentación siempre quedarán casos "raros" que no nos merecerá la pena cubrir. Siempre habrá algún modelo que tenga algún problema grabando o reproduciendo el sonido, o ejecutando un determinado formato de vídeo… o cualquier otra posibilidad. El hecho de que Android sea un sistema libre permite que cada fabricante implemente el sistema operativo a su gusto hasta cierto punto, lo que inevitablemente causará que tendremos modelos difíciles de cubrir.

Aquí es imprescindible un buen pragmatismo. Cubrir unos pocos dispositivos que utiliza un número muy reducido de usuarios no es viable, nos llevará más tiempo que cubrir dispositivos comunes. La mejor estrategia consiste en asegurar los dispositivos con más presencia en el mercado en ese momento, lo que a su vez nos ayudará a que gran parte de los demás funcione también. Luego seguiremos refinando nuestra aplicación hasta conseguir una cobertura razonablemente buena -una aplicación bien desarrollada supera fácilmente una cobertura del 80%- .

Con todo esto ya podemos empezar a trabajar. Aunque ya hemos citado alguna técnica útil, las repasaremos ahora en detalle.

- Nuestras vistas siempre serán flexibles. Nunca utilizaremos valores absolutos para los tamaños en píxeles, y mucho menos un AbsoluteLayout. Todas nuestras medidas serán en píxeles dependientes o dp, y usaremos proporciones y medidas relativas siempre que sea posible.
- Testearemos nuestras vistas en diferentes tamaños de pantalla. Para no tener que probarlos todos, una buena aproximación consiste en probar un dispositivo de los más grandes, otro de los más pequeños, y uno intermedio.
- Nos aseguraremos de tener todos los recursos gráficos disponibles para todas las densidades de pantalla, lo que nos facilitará el tener vistas 100% flexibles.
- Nos aseguraremos de tener los textos separados del código para dar soporte a la internacionalización.
- Elegiremos la versión del sistema operativo más baja con la que trabajaremos y desarrollaremos sólo con ella si es posible. Si no lo fuese, crearemos diferentes versiones para los diferentes sistemas operativos, aunque cuantos menos sean mejor. A veces encontraremos librerías de terceros que implementan funcionalidades de versiones recientes sin tener que usarlas directamente, es una alternativa interesante a considerar.
- Inevitablemente testearemos. En el mercado existen empresas dedicadas únicamente a testeo, y con precios bastante razonables podremos conseguir un testeo automático para una amplia gama de dispositivos.
- Y por último no descartaremos los reportes de error de los usuarios, que inevitablemente nos llegarán. Con ellos seguro que descubriremos detalles que se nos pasaron por alto.