SOBRE LOS AUTORES

Miguel Ángel Lozano Ortega

Ingeniero en Informática y doctor por la Universidad de Alicante. Profesor Contratado Doctor del departamento de Ciencia de la Computación e Inteligencia Artificial. Profesor en el Grado de Ingeniería Multimedia en la asignatura "Videojuegos II". Coordinador del título "Experto en Diseño y Creación de Videojuegos" y de "Experto en Desarrollo de Aplicaciones para Dispositivos Móviles (iOS y Android)". Director del Máster Universitario "Desarrollo de Software para Dispositivos Móviles". Autor del libro Programación de dispositivos móviles con J2ME y de diferentes materiales online sobre iOS, Android y videojuegos. Desarrollador de aplicaciones para Android e iOS y de videojuegos con Unity y Cocos2d-x.

Antonio Javier Gallego Sánchez

Ingeniero en Informática y doctor en Informática Avanzada por la Universidad de Alicante. Profesor asociado del departamento de Lenguajes y Sistemas Informáticos. Imparte clases en el Máster Universitario "Desarrollo de Software para Dispositivos Móviles", en varias asignaturas del Grado en Ingeniería Informática, en cursos Ceclec y en cursos de Experto y Especialista en Desarrollo de Aplicaciones para Dispositivos Móviles. Ha trabajado en empresa privada en el desarrollo de aplicaciones Android, aplicaciones híbridas y sitios web. Es autor de varios capítulos de libro y de diferentes materiales online relacionados con la programación de aplicaciones Android, aplicaciones híbridas y sitios web.

1

INTRODUCCIÓN A LAS APLICACIONES ANDROID

En este primer capítulo comenzaremos con una breve introducción al sistema operativo Android y a sus principales características. A continuación también hablaremos sobre la anatomía de las aplicaciones Android, estudiando los diferentes tipos de elementos que contienen y la forma en la que se organizan.

1.1 HISTORIA DE ANDROID

Antiguamente los dispositivos electrónicos solo se podían programar a bajo nivel, por lo que los programadores necesitaban entender completamente el *hardware* para el que estaban trabajando. Esto fue evolucionando poco a poco y en la actualidad los sistemas operativos abstraen al programador del *hardware*. Un ejemplo clásico de sistema operativo para móviles es Symbian, que se incluyó en gran parte de los primeros teléfonos móviles que contaban con la posibilidad de instalar aplicaciones. Pero este tipo de plataformas todavía requerían que el programador escribiera código C/C++ relativamente complejo y que hiciera uso de librerías propietarias de bajo nivel. Además, la programación se podía llegar a complicar bastante cuando se trataba con el acceso a determinados componentes del dispositivo, como *hardware* específico, GPS, *trackballs*, pantallas táctiles, etc.

Otro problema importante para la programación de estos primeros dispositivos era la amplia variedad de sistemas operativos diferentes que podíamos encontrar, lo que unido a la dificultad del lenguaje a bajo nivel y del uso de las librerías propietarias, hacía que un programador solo se pudiera especializar en un tipo de terminales. Para paliar estos problemas se incorporó el soporte para aplicaciones Java ME a

los sistemas operativos, permitiendo así desarrollar aplicaciones multiplataforma mediante un lenguaje de alto nivel. Java ME abstrae completamente al programador del *hardware*, pero tiene como inconveniente las limitaciones impuestas por la máquina virtual, que restringen mucho la libertad a la hora de acceder al *hardware* del dispositivo.

Esta situación motivó la aparición de Android, cuya primera versión oficial (la 1.1) se publicó en febrero de 2009, coincidiendo con la proliferación de *smartphones* con pantallas táctiles. Desde entonces han ido apareciendo versiones nuevas del sistema operativo, desde la 1.5 llamada *Cupcake* y que se basaba en el núcleo de Linux 2.6.27, hasta las versiones actuales que están orientada a *tablets*, teléfonos móviles y otros dispositivos, como por ejemplo aplicaciones de TV. Cada versión del sistema operativo tiene un nombre inspirado en la repostería, siguiendo un orden alfabético con respecto al resto de versiones de Android: *Apple Pie*, *Banana Bread*, *Cupcake*, *Donut*, *Eclair*, *Froyo*, *Gingerbread*, *Honeycomb*, *Ice Cream Sandwich*, *Jelly Bean*, *Kit Kat*, *Lollipop* y *Marshmallow*.

Android es un sistema operativo de código abierto para dispositivos móviles, se programa principalmente en Java y su núcleo está basado en Linux. Tanto el sistema operativo como la plataforma de desarrollo están liberados bajo la licencia de Apache. Esta licencia permite a los fabricantes añadir sus propias extensiones propietarias sin tener que ponerlas en manos de la comunidad de *software* libre. Además, el hecho de ser *open source* conlleva una serie de ventajas añadidas para Android:

- Una gran comunidad de desarrollo, gracias a sus completas API y documentación ofrecida.
- Desarrollo desde cualquier plataforma (Linux, Mac, Windows, etc.).
- Su uso en cualquier tipo de dispositivo móvil.
- Que cualquier fabricante pueda diseñar un dispositivo que trabaje con Android, incluso adaptando o extendiendo el sistema para satisfacer las necesidades de su dispositivo concreto.
- Los fabricantes de dispositivos se ahorran el coste de desarrollar un sistema operativo completo desde cero.
- Los desarrolladores se ahorran tener que programar API, entornos gráficos, aprender a acceder a dispositivos *hardware* particulares, etc.

El sistema operativo de Android está formado por los siguientes componentes:

- Núcleo basado en el de Linux para el manejo de memoria, procesos y *hardware* (se trata de una rama independiente de la rama principal, de manera que las mejoras introducidas no se incorporan en el desarrollo del núcleo de GNU/Linux).
- Bibliotecas *open source* para el desarrollo de aplicaciones, incluyendo SQLite, WebKit, OpenGL y el manejador de medios.
- Entorno de ejecución para las aplicaciones Android. La máquina virtual Dalvik (y su nueva versión llamada ART) y las bibliotecas específicas dan a las aplicaciones acceso a todas las funcionalidades de Android.
- Un *framework* de desarrollo que pone a disposición de las aplicaciones los servicios del sistema, como el manejador de ventanas, la localización, los proveedores de contenidos, los sensores, etc.
- Un SDK (kit de desarrollo de *software*) que incluye el entorno Android Studio, otras herramientas, emuladores, ejemplos y documentación.
- Interfaz de usuario útil para pantallas táctiles y otros tipos de dispositivos de entrada, como por ejemplo, teclado y *trackball*.
- Aplicaciones preinstaladas que hacen que el sistema operativo sea útil para el usuario desde el primer momento.
- Muy importante es la existencia de Google Play, y más todavía la presencia de una comunidad de desarrolladores que publican allí sus aplicaciones, tanto de pago como gratuitas. De cara al usuario, el verdadero valor del sistema operativo está en las aplicaciones que se puede instalar.

El principal responsable del desarrollo de Android es la *Open Handset Alliance*, un consorcio de varias compañías que tratan de definir y establecer una serie de estándares abiertos para dispositivos móviles. El consorcio cuenta con decenas de miembros de distintos tipos de empresas: operadores de telefonía móvil, fabricantes de dispositivos, fabricantes de procesadores y microelectrónica, compañías de *software* y compañías de comercialización. Por lo tanto, Android no es de Google como se suele decir, aunque Google es una de las empresas con mayor participación en el proyecto.

Por último concluimos esta sección considerando algunas cuestiones éticas. Uno de los aspectos más positivos de Android es su carácter de **código abierto**. Gracias a él, tanto fabricantes como usuarios se ven beneficiados, y tanto el proceso de programación de dispositivos móviles como su fabricación se acelera. Todos salen ganando. Otra consecuencia de que sea de código abierto es la **mantenibilidad**. Si aparece algún problema debido al sistema operativo (no nos referimos a que el usuario lo estropee, por supuesto) el fabricante podría abrir el código fuente, descubrir el problema y solucionarlo. Esto es una garantía de éxito muy importante. Por otro la **seguridad** informática también se ve beneficiada, tal y como ha demostrado la experiencia con otros sistemas operativos abiertos frente a los propietarios. Hoy en día los dispositivos móviles cuentan con *hardware* que recoge información de nuestro entorno: cámara, GPS, brújula y acelerómetros, además de una conexión a Internet continua, a través de la cuál circulan nuestros datos más personales. El carácter abierto del sistema operativo nos ofrece una transparencia con respecto al uso que se hace de esa información. Por ejemplo, si hubiera la más mínima sospecha de que el sistema operativo captura fotos sin preguntarnos y las envía, a los pocos días ya sería noticia.

Esto no concierne a las aplicaciones que nos instalamos, las cuales no tienen por qué ser código abierto (de hecho la mayoría no lo son). Por este motivo el sistema operativo las analiza antes de instalarlas y nos avisa de los permisos requeridos. Si los aceptamos nos hacemos responsables de lo que la aplicación haga. Es necesario aclarar que esto no es un problema de seguridad, ya que los problemas de seguridad realmente surgen si se hace algo sin el consentimiento ni conocimiento del usuario.

1.2 APLICACIONES ANDROID

Las aplicaciones Android están compuestas por un conjunto heterogéneo de componentes enlazados mediante un archivo llamado `AndroidManifest.xml` que los describe e indica cómo interactúan. Este archivo también contiene metainformación acerca de la aplicación, como por ejemplo, los requerimientos que debe cumplir la plataforma sobre la que se ejecuta.

Una aplicación Android estará compuesta por los siguientes componentes (no necesariamente todos ellos):

- **Actividades**. Las actividades son la capa de presentación de la aplicación. Cada pantalla a mostrar en la aplicación será una subclase de la clase `Activity`. Las actividades hacen uso de componentes de tipo `View` para mostrar elementos de la interfaz gráfica que permitan mostrar datos y reaccionar ante la entrada del usuario.

- **Servicios**. Los servicios son componentes que se ejecutan en el segundo plano (*background*) de la aplicación, ya sea actualizando fuentes de información, atendiendo a diversos eventos o activando la visualización de notificaciones. Se utilizan para llevar a cabo procesamiento que debe ser realizado de manera regular, incluso en el caso en el que nuestras actividades no sean visibles o ni siquiera estén activas.
- **Proveedores de contenidos**. Permiten almacenar y compartir datos entre aplicaciones. Los dispositivos Android incluyen de serie un conjunto de proveedores de contenidos nativos que permiten acceder a datos del terminal, como por ejemplo los contactos, calendario o contenido multimedia.
- **Intents**. Los *intents* constituyen una plataforma para la ejecución de acciones y el paso de mensajes dentro de una misma aplicación o incluso entre distintas aplicaciones. Al emitir un *intent* se declara la intención de que se lleve a cabo una determinada acción, por ejemplo: cambiar de actividad, abrir la cámara, visualizar un documento, etc.
- **Receptores** (*broadcast receivers*). Permiten a tu aplicación hacerse cargo de determinadas acciones solicitadas mediante *intents*. Estos receptores se iniciarán automáticamente cuando se lance un *intent* al que estaban escuchando, por lo que son ideales para la creación de aplicaciones guiadas por eventos.
- **Widgets**. Se trata de componentes visuales que pueden ser añadidos a la ventana principal (*home*) de Android.
- **Notificaciones.** Las notificaciones permiten comunicarse con el usuario sin necesidad de robar el foco de la aplicación activa actualmente. Por ejemplo, cuando un dispositivo recibe un mensaje de texto, avisa al usuario mediante luces, sonidos o mostrando algún icono.

1.2.1 El archivo *Manifest*

Cada proyecto Android contiene un archivo principal de configuración, llamado `AndroidManifest.xml`, donde se establecen una serie de metadatos (que veremos más adelante), junto con la estructura, componentes y requisitos de la aplicación. Este archivo está definido en formato XML e incluye un nodo por cada uno de los componentes de la aplicación (actividades, servicios, proveedores de contenidos, etc.). También se utilizan atributos para especificar la metainformación asociada a la aplicación, como su icono, etiqueta, etc. Veamos como ejemplo el archivo `AndroidManifest` de un proyecto sencillo:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="es.ua.eps.android">
   <application android:icon="@drawable/icon"
                android:label="@string/app_name">
      <activity android:name=".NombreProyectoActivity"
                android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
</manifest>
```

Como elemento raíz del XML se utiliza la etiqueta `<manifest>`, en la cual utilizamos el atributo `package` para indicar el nombre del paquete del proyecto, que nos servirá para **identificar nuestra aplicación**. Esta etiqueta tiene que contener **un único nodo** `<application>` para establecer la metainformación de la aplicación: nombre, icono, etc. En el ejemplo anterior, el valor del atributo `icon` como el del atributo `label` hacen referencia respectivamente al icono y al nombre de la aplicación, que se encuentran dentro de los recursos de la aplicación (de los que hablaremos más adelante).

El elemento `<application>` deberá contener una etiqueta de tipo `<activity>` por cada una de las actividades presentes en nuestra aplicación. El atributo `name` de cada actividad indica el nombre de la clase Java asociada a la actividad. Es importante declarar todas las actividades, ya que si intentamos iniciar una actividad que no esté listada en este fichero se producirá un error en tiempo de ejecución. Cada elemento `<activity>` podrá contener a su vez un elemento de tipo `<intent-filter>` para especificar los *intents* a los que puede responder. En el ejemplo utilizamos este campo para indicar que nuestra única actividad es además la actividad principal, y por lo tanto, la que se deberá mostrar al iniciar la aplicación.

Conforme avancemos en este y otros capítulos iremos viendo más elementos del archivo `AndroidManifest.xml`.

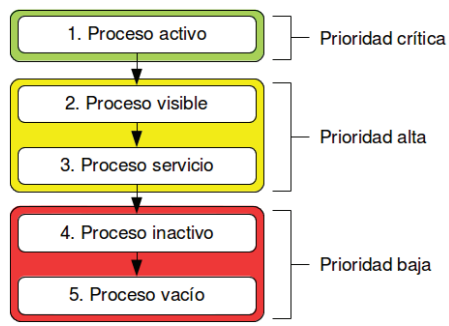
1.2.2 El ciclo de ejecución de una aplicación

Al contrario que en otros entornos, las aplicaciones Android tienen muy poco control sobre su propio ciclo de ejecución. Los componentes de una aplicación Android deben estar atentos a los cambios producidos en el estado de la misma y

reaccionar como corresponda, estando especialmente preparados para el caso de una finalización repentina de la ejecución de la aplicación.

Por defecto, cada aplicación Android se ejecutará en su propio proceso, cada uno con su propia instancia asociada de Dalvik o ART (la máquina virtual de Android). Android administra sus recursos de manera agresiva, haciendo todo lo posible para que el dispositivo siempre responda a la interacción del usuario. Esto puede conllevar que aplicaciones o procesos abiertos en segundo plano dejen de ejecutarse de manera repentina, incluso sin un aviso previo, con el objetivo de liberar recursos para aplicaciones de mayor prioridad. Estas aplicaciones de mayor prioridad suelen ser normalmente aquellas que están interactuando con el usuario en ese preciso instante.

El orden en el que los procesos de las aplicaciones son detenidos viene determinado por su prioridad, la cual se establece a partir del estado en el que se encuentre su componente de mayor prioridad. Los estados en los que se puede encontrar una aplicación se resume en la siguiente figura y se detalla a continuación:



Ciclo de ejecución de los procesos en Android

- **Procesos activos:** son aquellos que se encuentran interactuando con el usuario en ese preciso instante. Android liberará recursos para intentar que estos procesos activos siempre respondan sin latencia. Los procesos activos solo serán detenidos en última instancia.

- **Procesos visibles**: procesos visibles pero inactivos, ya sea porque su interfaz se está mostrando detrás de otras o porque no están respondiendo a ninguna entrada del usuario. Esto sucede cuando una actividad se encuentra parcialmente oculta por otra actividad, cuando aparece un diálogo por encima o cuando no ocupa toda la pantalla. Estos procesos son detenidos tan solo bajo condiciones extremas.

- **Procesos asociados a servicios en ejecución**: los servicios permiten que exista procesamiento sin necesidad de una interfaz de usuario visible. Debido a que estos servicios no interactúan directamente con el usuario, reciben una prioridad ligeramente inferior a la de los procesos visibles. Sin embargo se siguen considerando procesos activos y no serán detenidos a menos que sea estrictamente necesario.
- **Procesos inactivos**: se trata de procesos que albergan actividades que ni son visibles ni se encuentran realizando un procesamiento en este momento, y que además no están ejecutando ningún servicio. El orden en el que se detendrán estos procesos vendrá determinado por el tiempo que éstos llevan inactivos desde la última vez que fueron visibles, de mayor a menor.
- **Procesos vacíos**: son el resultado del intento de Android de retener aplicaciones en memoria a modo de caché una vez que éstas han terminado. Con esto se consigue que al lanzar de nuevo la aplicación se requiera menos tiempo.

1.2.3 Recursos

Se suele considerar una buena práctica de programación mantener todos los recursos de la aplicación que no sean código fuente separados del propio código, como imágenes, cadenas de texto, etc. Android permite externalizar recursos de diversos tipos, no solo los comentados anteriormente, sino recursos más complejos como los `layouts`, o lo que es lo mismo, las vistas o especificación de la interfaz gráfica de las diferentes actividades.

Una ventaja adicional de externalizar los recursos es que podemos proporcionar valores diferentes dependiendo del *hardware*, del idioma del usuario o de otras características. Si lo hacemos todo de manera correcta, será Android el encargado de, al iniciar una actividad, seleccionar de forma automática la variante de los recursos adecuada para nuestra configuración.

1.2.3.1 TIPOS DE RECURSOS

Todos los recursos de la aplicación se almacenan dentro de la carpeta `res` del proyecto. En esta carpeta encontramos diferentes subcarpetas para los distintos tipos de recursos, como por ejemplo:

- `values`: cadenas de texto, listas y valores simples
- `drawables` y `mipmaps`: imágenes y otros recursos gráficos
- `layouts`: interfaces para las actividades
- `menus`: definición de menús de opciones
- `raw`: recursos *crudos*
- etc.

Al compilar nuestra aplicación estos recursos serán incluidos en el paquete *APK* que será instalado en el dispositivo. La mayoría de estos recursos se definen en XML, pero al empaquetarlos en la aplicación son binarizados y optimizados de forma automática (excepto los contenidos en `raw`, que se incluyen tal como son originalmente).

Durante el proceso de compilación se generará también de forma automática una clase Java llamada `R` que contendrá referencias a cada uno de los recursos. Esto nos permitirá referenciar los recursos desde nuestro código fuente. Por otra parte, a lo largo del libro iremos haciendo uso de algunos de estos recursos que se han comentado anteriormente. Iremos aprendiendo su uso conforme los necesitemos.

1.2.3.2 RECURSOS PARA DIFERENTES CONFIGURACIONES

En Android es posible preparar una aplicación para que utilice distintas versiones de los recursos según la configuración del dispositivo. Para ello definiremos archivos de recursos alternativos, específicos para cada configuración. Android escogerá en tiempo de ejecución el archivo o archivos de recursos adecuados. Para conseguirlo definimos una estructura paralela de directorios dentro de la carpeta `res`, haciendo uso del guión `-` para indicar las diferentes alternativas de recursos que se están proporcionando (lo que se conoce como *especificadores* o *qualifiers*). En el siguiente ejemplo se hace uso de una estructura de carpetas que permite tener valores por defecto para las cadenas, así como cadenas para el idioma francés y el francés de Canadá:

```
res/
    values/
        strings.xml
    values-fr/
        strings.xml
    values-fr-rCA/
        strings.xml
```

De esta forma, al ejecutar la aplicación, se utilizará un fichero de recursos u otro dependiendo del idioma que tenga configurado el usuario. Aparte de poder definir recursos para diferentes lenguajes utilizando especificadores como los que acabamos de ver (`en`, `en-rUS`, `es`, etc.), también veremos más adelante algunos otros referidos al *hardware* o los sensores, como por ejemplo el tamaño o densidad de la pantalla, si es de noche o de día, etc.

> **NOTA**
> Si no se encuentra un directorio de recursos que se corresponda con la configuración del dispositivo en el que se está ejecutando la aplicación, se lanzará una excepción al intentar acceder al recurso no encontrado. Para evitar esto se debería incluir una carpeta por defecto para cada tipo de recurso, sin ninguna especificación de idioma, configuración de la pantalla, etc.

1.2.4 Actividades

Las actividades se podrían interpretar como cada una de las "pantallas" de nuestra aplicación. Cada actividad contendrá objetos de la clase `View` que permitirán mostrar los diferentes elementos gráficos de la interfaz así como añadir interactividad. Deberemos añadir una nueva actividad por cada pantalla que queramos mostrar en nuestra aplicación. Esto incluye la pantalla principal de nuestra aplicación, la primera que se mostrará al iniciar nuestro programa, y desde la cual podremos acceder a todas las demás. Para movernos entre pantallas comenzaremos una nueva actividad (o volveremos a una anterior desde otra previamente ejecutada). La mayoría de las actividades están diseñadas para ocupar toda la pantalla, pero es posible crear actividades "flotantes" o semitransparentes.

1.2.4.1 CREANDO ACTIVIDADES

Para crear una nueva actividad añadimos a nuestra aplicación una subclase de `Activity`, donde se deberá definir la interfaz gráfica de la actividad junto a su funcionalidad. A continuación se muestra el esqueleto básico de esta clase:

```
package es.ua.eps.android;

import android.app.Activity;
import android.os.Bundle;

public class MiActividad extends Activity {
   /** Método invocado al crearse la actividad **/
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
   }
}
```

Para añadir la actividad a la aplicación no basta con crear la clase Java correspondiente, sino que además deberemos registrarla en el *Manifest*. Para ello añadiremos un nuevo nodo `<activity>` dentro del elemento `<application>`. Los atributos de `<activity>` permiten incluir información sobre su icono, los permisos que necesita, los temas o estilos que utiliza, etc. A continuación tenemos un ejemplo de este tipo de elemento:

```
<activity android:label="@string/app_name"
          android:name=".MiActividad">
</activity>
```

> **ⓘ NOTA**
>
> En el ejemplo anterior la notación `@string/app_name` hace referencia a un recurso de tipo *cadena* (`string`) cuyo identificador es `app_name`. Más adelante veremos en detalle la notación de recursos.

Como parte del contenido del elemento `<activity>` podemos incluir el nodo `<intent-filter>` para indicar los `Intent` a los que escuchará nuestra actividad y por lo tanto a los que reaccionará. Los `Intent` serán tratados más adelante, pero es necesario destacar que para que una actividad sea marcada como actividad principal (la primera actividad que se ejecutará al iniciar la aplicación) debe incluir el elemento `<intent-filter>` tal cual se muestra en el siguiente ejemplo:

```
<activity android:label="@string/app_name"
          android:name=".MiActividad">
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

> **ⓘ NOTA**
>
> Aunque es importante conocer los elementos que debe incluir el proyecto al crear una actividad, normalmente todo lo anterior lo realizará automáticamente el entorno de desarrollo integrado (IDE) que utilicemos, tal como veremos más adelante.

1.2.4.2 EL CICLO DE EJECUCIÓN DE UNA ACTIVIDAD

Conviene incidir de nuevo en la administración que lleva a cabo Android de sus diferentes elementos ejecutables. Hemos hablado anteriormente sobre la ejecución de aplicaciones; veamos ahora cómo se administra la ejecución de las diferentes actividades dentro de una aplicación.

Conforme se produce la ejecución de una determinada aplicación irá modificándose el estado de sus correspondientes actividades. El estado de una actividad servirá para determinar su prioridad en el contexto de su aplicación padre. Esto es importante porque hemos de recordar que la prioridad de una aplicación, y por lo tanto, la probabilidad de que dicha aplicación sea detenida en el caso de que sea necesario liberar recursos, dependerá de su actividad de mayor prioridad.

El estado de cada actividad viene determinado por su posición en la **pila de actividades**, una estructura de tipo *last-in-first-out* que contiene todas las actividades de la aplicación actualmente en ejecución. Cuando comienza una nueva actividad, aquella que se encontrara mostrándose en ese momento se mueve al tope de la pila. Si el usuario pulsa el botón que permite volver a la actividad anterior o cierra la actividad que se esté mostrando en ese determinado momento, la actividad que se encuentre en el tope de la pila saldrá de ella y pasará a ser la actividad activa.

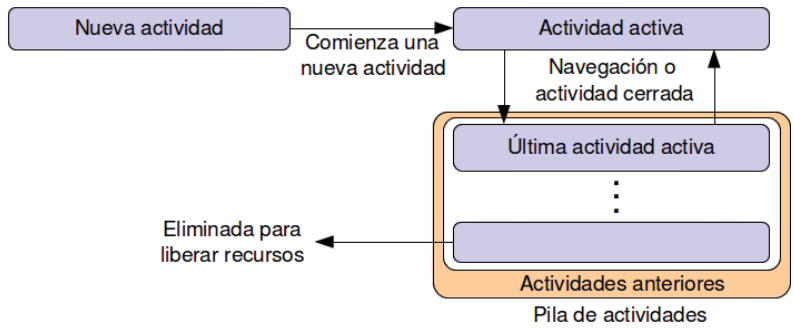


Figura 1.1. Diagrama de la pila de actividades

Conforme las actividades se crean o destruyen, van entrando o saliendo de la pila. Al hacerlo pueden ir transitando entre cuatro diferentes estados:

- **Activa**: se trata de la actividad que se está ejecutando en ese momento; es visible, tiene el foco de la aplicación y es capaz de recibir datos por parte del usuario. Android tratará por todos los medios de mantener esta actividad en ejecución, deteniendo cualquier otra actividad en la pila siempre que sea necesario.

- **En pausa**: se trata del estado en el que se encuentra una actividad cuando ésta está activa pero no dispone del foco. Este estado se puede alcanzar por ejemplo cuando se encuentra situada por debajo de otra transparente o que no ocupe toda la pantalla. Una actividad en pausa recibe el mismo tratamiento que una actividad activa, con la única diferencia de que no recibe eventos relacionados con la entrada de datos.

- **Detenida**: este es el estado en el que se encuentra una actividad que no es visible en ese momento. La actividad permanece en memoria, manteniendo toda su información asociada. Sin embargo, ahora podría ser escogida para ser eliminada de la memoria en el caso en el que se requieran recursos. Por eso es importante almacenar los datos de la actividad y el estado de su interfaz de usuario cuando ésta pasa a estar detenida.

- **Inactiva:** una actividad estará inactiva si se ha terminado su ejecución o si todavía no se ha iniciado. Las actividades inactivas han sido extraídas de la pila de actividades y deben ser reiniciadas para poder ser mostradas y utilizadas.

Todo este proceso debe ser transparente al usuario. No debería haber ninguna diferencia entre actividades pasando a un estado activo desde cualquiera de los otros estados. Android nos facilita una serie de funciones en la clase `Activity` que podemos sobrecargar para gestionar de forma sencilla estos eventos. De esta forma podremos realizar las tareas pertinentes según el estado de la actividad, como por ejemplo almacenar los datos cuando la actividad pasa a estar detenida o inactiva y volver a leerlos cuando ésta pasa a estar activa. Para manejar estos diferentes eventos podemos sobrecargar las siguientes funciones:

- `onCreate`: es llamada cuando la actividad se crea. En ella deberemos introducir el código para inicializar la actividad y su interfaz.

```
public void onCreate(Bundle savedInstaceState) { ... }
```

- `onStart`: es llamada cuando la actividad pasa a estado visible.

```
public void onStart() { ... }
```

- `onResume`: es llamada cuando la actividad pasa a estar activa. Esta función es útil, por ejemplo, para poner en marcha los hilos de ejecución que realicen las tareas de actualización necesarias. Por ejemplo, en un videojuego podríamos poner en marcha el movimiento y las animaciones de los personajes.

```
public void onResume() { ... }
```

- `onPause`: es llamada cuando la actividad pasa a estar pausada. Aquí deberemos hacer lo contrario que en `onResume`. Si en `onResume` estábamos poniendo en marcha hilos de ejecución, aquí deberemos detenerlos. Todo lo que se haga en `onResume` deberá deshacerse en `onPause`, ya que las llamadas a ambos métodos siempre estarán equilibradas (tras una llamada a `onPause` siempre tendrá que pasar por `onResume` para volver a poner en marcha la actividad).

```
public void onPause() { ... }
```

- `onStop`: es llamada cuando la actividad deja de estar visible. En este caso se trata de la acción complementaria a `onStart`. Todo lo que se haga en `onStart` deberá deshacerse en `onStop`, ya que al igual que ocurría con `onPause` y `onResume`, las llamadas a estos métodos siempre estarán equilibradas.

```
public void onStop() { ... }
```

- `onRestart`: cuando una actividad vuelve a estar visible después de haber pasado a estado no visible, se llamará a este método justo antes de llamar a `onStart`. Esto nos puede servir para tareas que necesitemos hacer solo cuando la actividad vuelva a este estado tras haber pasado a estar no visible.

```
public void onRestart() { ... }
```

- `onDestroy`: se ejecuta cuando se va a destruir la actividad. Aquí deberemos asegurarnos de deshacer todo lo que se haya hecho en `onCreate`.

```
public void onDestroy() { ... }
```

En el siguiente diagrama se puede ver de forma gráfica el ciclo de vida que seguiría una actividad desde que se crea hasta que se destruye:

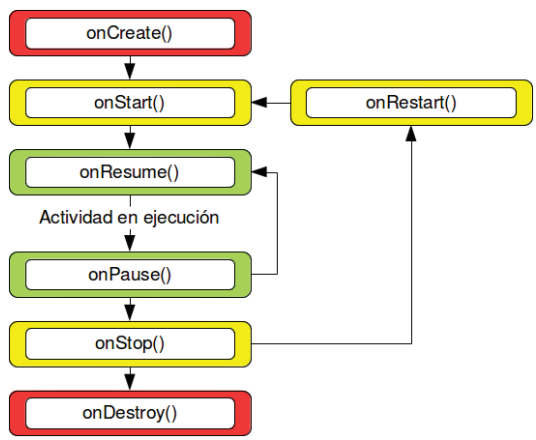


Figura 1.2. Ciclo de vida de una actividad

La clase `Activity` también dispone de funciones que nos permiten guardar el estado de la interfaz para poderla recuperar en caso de que la actividad sea destruida. Esto se puede hacer con el método `onSaveInstanceState`, que es ejecutado siempre inmediatamente antes de llamar a `onPause`:

```
public void onSaveInstanceState(Bundle savedInstanceState)
```

Este método nos permite utilizar el objeto `savedInstanceState` que recibimos como parámetro para guardar el estado en el que se encuentra la interfaz, de forma que si la actividad fuese destruida antes de volver a estar activa podríamos recuperar posteriormente el estado en el que se encontraba.

Para recuperar el estado podemos optar por dos alternativas:

- Hemos visto que el método `onCreate` toma como parámetro también un objeto `savedInstanceState`. En caso de que se hubiese guardado el estado previamente, este parámetro recibirá los datos que introdujimos y podremos utilizarlo para recuperar el estado. En caso de que no hubiera estado guardado, este objeto será `null`.

  ```
  public void onCreate(Bundle savedInstaceState)
  ```

- En algunos casos podríamos necesitar que la recuperación del estado se realice cuando la actividad ha sido ya inicializada. En este caso podríamos utilizar el método `onRestoreInstanceState`, que se ejecutará tras la llamada a `onStart` en caso de haber un estado previo guardado.

  ```
  public void onRestoreInstanceState(Bundle savedInstanceState)
  ```

1.3 VERSIONES DE ANDROID Y COMPATIBILIDAD

Tal como hemos visto anteriormente, existen diferentes versiones de la plataforma Android, y cada una de ellas tiene un **nombre en clave**, un **número de versión** y un **código**. A continuación mostramos algunos ejemplos de versiones:

Nombre en clave	Número versión	Código API
Cupcake	1.5	3
Froyo	2.2.x	8
KitKat	4.4	19
Lollipop	5.0	21

En nuestras aplicaciones siempre haremos referencia a las versiones de Android mediante el código de la API. Deberemos especificar las versiones de Android para las cuales está preparada nuestra aplicación mediante los siguientes atributos del proyecto:

- `minSdkVersion`. Versión mínima de Android para que nuestra aplicación funcione. La aplicación no podrá instalarse en dispositivos con versiones inferiores y tampoco aparecerá en la tienda para dichos dispositivos.
- `targetSdkVersion`. Se refiere a la versión de la plataforma Android en la que la aplicación ha sido probada, normalmente se establecerá por defecto a la versión del SDK con el que se ha compilado. Si la versión de la plataforma en la que se ejecuta una aplicación es mayor o inferior que la del `targetSdkVersion`, en muchas ocasiones se introducirán de forma automática funciones de compatibilidad para que nuestra aplicación se vea de forma correcta.

> **NOTA**
> En el próximo capítulo veremos cómo configurar estos atributos en nuestro proyecto utilizando el entorno Android Studio.

Estos atributos determinan las versiones de Android con las que nuestra aplicación es compatible. A la hora de desarrollar una aplicación es importante decidir para qué versiones estará destinada. Las últimas versiones nos dan muchas más facilidades para crear las aplicaciones, pero es importante dar soporte a versiones antiguas para abarcar a un mayor número de usuarios. Se recomienda que nuestras aplicaciones soporten al menos el 90% de los dispositivos que hay actualmente en uso (veremos que esta información nos la proporciona Android Studio al crear el proyecto).

El atributo `minSdkVersion` indica la versión mínima de Android necesaria para poder utilizar nuestra aplicación. Por debajo de dicha versión nuestra aplicación no funcionará. Sin embargo, este atributo no nos condiciona a utilizar solo las características compatibles con la versión mínima. Podemos utilizar características de versiones mayores. La versión para la cual hemos probado la aplicación se indica con `targetSdkVersion`, y en el código podremos utilizar cualquier característica que soporte dicha versión.

Pero, ¿qué ocurre si utilizamos una característica solo disponible a partir de `targetSdkVersion` y no soportada en `minSdkVersion`? En ese caso la aplicación fallará. Por lo tanto, es importante que antes de utilizar dichas características comprobemos la versión de Android en la que se está ejecutando:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    // Utilizar características de Android 3.0 (Honeycomb)
}
```

> **NOTA**
> En el caso de las propiedades de los ficheros de recursos XML, si utilizamos atributos que no estaban definidos en la versión mínima, cuando ejecutemos la aplicación en dicha versión simplemente serán ignorados.

Deberemos llevar cuidado de hacer siempre esta comprobación cuando estemos utilizando características no presentes en la versión mínima. Para asegurar el correcto funcionamiento deberemos probar la aplicación de forma exhaustiva con dispositivos que tengan tanto la versión mínima como la versión para la cual estamos desarrollando.

Sin embargo, existen algunas características que son fundamentales en la construcción de la aplicación, y no se pueden ignorar mediante la comprobación anterior. Una de estas características son por ejemplo los *fragments*, introducidos en Android 3.0 para facilitar la construcción de aplicaciones que se adapten a *tablets* y móviles. Para estas características importantes la plataforma Android nos proporciona una serie de librerías de compatibilidad que añaden soporte para versiones previas de Android, y que veremos en el próximo capítulo.

1.4 EJERCICIOS PROPUESTOS

1.4.1 Ejercicio 1. Aplicaciones y servicios

Vamos a obtener información sobre las actividades y servicios que tenemos funcionando en nuestro dispositivo Android. Para ello, puedes entrar en *Ajustes*, y dentro de ese apartado en *Aplicaciones* (los nombres podrían variar entre diferentes versiones del operativo). ¿Qué información nos da sobre las aplicaciones y servicios en ejecución?

1.4.2 Ejercicio 2. Versión de Android

También en la pantalla de *Ajustes* del dispositivo podremos encontrar *Información del teléfono*. ¿Qué información nos da? ¿Qué versión de Android está instalada? ¿Cuál es su nombre en clave?

1.4.3 Ejercicio 3. Guía de estilo

Entra en la página *developer.android.com* y localiza la guía de estilo (sección *Diseñar*). Es conveniente leer esta guía para empezar a pensar en el diseño de la interfaz de nuestras aplicaciones.