

# **Creación de aplicaciones con Android**



# **Creación de aplicaciones con Android**

*Valeriano Moreno*





Creación de aplicaciones con Android

© Valeriano Moreno

© De la edición: Ra-Ma 2021

**MARCAS COMERCIALES.** Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas son ficticios a no ser que se especifique lo contrario.

RA-MA es marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente, ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la ley vigente, que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:

RA-MA Editorial

Calle Jarama, 3A, Polígono Industrial Igarsa  
28860 PARACUELLOS DE JARAMA, Madrid

Teléfono: 91 658 42 80

Fax: 91 662 81 39

Correo electrónico: [editorial@ra-ma.com](mailto:editorial@ra-ma.com)

Internet: [www.ra-ma.es](http://www.ra-ma.es) y [www.ra-ma.com](http://www.ra-ma.com)

ISBN: 978-84-1855-184-0

Depósito legal: M-20021-2021

Maquetación: Antonio García Tomé

Diseño de portada: Antonio García Tomé

Filmación e impresión: Safekat

Impreso en España en julio de 2021

*A mis padres,  
a la Universidad Carlos III de Madrid,  
a Google y a todos los que crean,  
divulgan y comparten conocimiento.*



---

# ÍNDICE

<b>SOBRE EL AUTOR.....</b>	<b>13</b>
<b>NOTA DEL AUTOR .....</b>	<b>15</b>
<b>PREFACIO.....</b>	<b>17</b>
<b>CAPÍTULO 1. INTRODUCCIÓN .....</b>	<b>33</b>
1.1    TAREAS PRÁCTICAS DEL TEMA 1.....	33
1.2    SDK.....	33
1.3    CONCEPTO CLAVE DE API .....	34
1.4    ENTORNO DE DESARROLLO.....	35
1.5    UN PROYECTO ANDROID .....	37
1.6    MANIFEST.....	40
1.7    EXTENSIONES .....	42
1.8    ESTRUCTURA DE UN PROYECTO .....	42
1.9    LA CLASE R .....	44
1.10    DEPURACIÓN.....	45
1.11    EL REGISTRO O LOG .....	45
1.12    TEST TEMA 1 .....	46
<b>CAPÍTULO 2. ACTIVIDADES I.....</b>	<b>49</b>
2.1    TAREAS PRÁCTICAS DEL TEMA 2.....	49
2.2    ACTIVIDADES.....	49
2.3    CONCEPTO CLAVE CLASE .....	50
2.4    CREACIÓN DE UNA ACTIVIDAD .....	50
2.5    MÉTODOS IMPORTANTES EN UNA ACTIVIDAD .....	51
2.5.1    onCreate (Bundle bundle) .....	51
2.5.2    findViewById (int id) .....	52
2.5.3    setContent View(int id) .....	52
2.5.4    finish() .....	52

---

2.6 VIEWS Y VIEWGROUP .....	52
2.7 ATRIBUTOS COMUNES DE UNA VISTA .....	53
2.8 UNIDADES DE MEDIDA .....	54
2.9 TEST TEMA 2 .....	55
<b>CAPÍTULO 3. VISTAS BÁSICAS.....</b>	<b>57</b>
3.1 TAREAS PRÁCTICAS DEL TEMA 3.....	57
3.2 LAYOUTS BÁSICOS .....	57
3.2.1 LinearLayout .....	58
3.2.2 ScrollView .....	58
3.2.3 FrameLayout .....	58
3.2.4 RelativeLayout .....	58
3.2.5 ConstraintLayout.....	58
3.3 BOTONES .....	59
3.4 CAJAS DE TEXTO .....	59
3.5 IMÁGENES.....	60
3.6 VÍDEO .....	60
3.7 SELECTORES.....	61
3.8 PÁGINAS WEB .....	61
3.9 CALLBACK Y LISTENER .....	62
3.10 VISIBILIDAD: VISIBLE, INVISIBLE Y GONE.....	63
3.11 TEST TEMA 3 .....	63
<b>CAPÍTULO 4. ICONOS, ESTILOS Y TEMAS.....</b>	<b>65</b>
4.1 TAREAS PRÁCTICAS DEL TEMA 4.....	65
4.2 ICONOS.....	65
4.2.1 Iconos de la Aplicación y Menús .....	66
4.2.2 Iconos Vectoriales.....	67
4.2.3 Iconos Material.....	69
4.2.4 Fuentes Iconográficas .....	69
4.3 ESTILOS Y TEMAS .....	70
4.3.1 Temas.....	71
4.4 TEST TEMA 4 .....	72
<b>CAPÍTULO 5. ACTIVIDADES II .....</b>	<b>75</b>
5.1 TAREAS PRÁCTICAS DEL TEMA 5.....	75
5.2 CICLO DE VIDA DE UNA ACTIVIDAD.....	75
5.3 ESTADOS DE UNA ACTIVIDAD .....	76
5.4 GIRANDO EL DISPOSITIVO.....	77
5.5 SALVAR EL ESTADO DE UNA ACTIVIDAD .....	77
5.6 BOTÓN DE IR HACIA ATRÁS.....	78
5.7 EL MÉTODO SETTAG () DE LA CLASE VIEW.....	78

5.8	INTERNACIONALIZACIÓN O I18N .....	79
5.9	INFLAR .....	80
5.10	TEST TEMA 5 .....	82
<b>CAPÍTULO 6. INTENTS.....</b>		<b>85</b>
6.1	TAREAS PRÁCTICAS DEL TEMA 6.....	85
6.2	INTENTOS EXPLÍCITOS .....	86
6.3	INTENTOS IMPLÍCITOS.....	87
6.4	INTENT FILTER .....	88
6.5	EXPORTED .....	90
6.6	BUNDLE EXTRAS.....	91
6.7	PARCELABLE Y SERIALIZABLE .....	92
6.8	INTENTS COMUNES .....	92
6.9	SUBACTIVIDADES .....	93
6.10	TEST TEMA 6 .....	95
<b>CAPÍTULO 7. MENÚS Y DIÁLOGOS.....</b>		<b>99</b>
7.1	TAREAS PRÁCTICAS DEL TEMA 7.....	99
7.2	MENÚS TEXTUALES .....	100
7.2.1	Definiendo el menú .....	100
7.2.2	Dibujando las acciones del menú .....	102
7.2.3	Escuchando las acciones sobre el menú .....	102
7.2.4	Eliminando la barra del menú .....	103
7.2.5	Botón de ir hacia atrás .....	104
7.3	MENÚS CONTEXTUALES .....	105
7.4	MENÚS POP O EMERGENTES .....	105
7.5	TOAST .....	106
7.6	ALERT DIALOG.....	107
7.7	DIALOG .....	108
7.8	SELECTOR DE HORA Y FECHA .....	108
7.9	TEST TEMA 7 .....	109
<b>CAPÍTULO 8. PERSISTENCIA.....</b>		<b>111</b>
8.1	TAREAS PRÁCTICAS DEL TEMA 8.....	111
8.2	VECTORES TIPADOS .....	112
8.3	MEMORIA INTERNA Y MEMORIA EXTERNA.....	113
8.4	ARCHIVOS DE PREFERENCIAS.....	114
8.5	API JAVA IO .....	116
8.6	BASES DE DATOS RELACIONALES CON SQLITE .....	119
8.7	TEST TEMA 8 .....	121

---

<b>CAPÍTULO 9. VISTAS AVANZADAS .....</b>	<b>123</b>
9.1 TAREAS PRÁCTICAS DEL TEMA 9.....	123
9.2 LISTAS DE ELEMENTOS .....	123
9.2.1 RecyclerView .....	124
9.2.2 Adapter .....	125
9.2.3 ViewHolder .....	126
9.2.4 LayoutManager .....	127
9.2.5 Actualizando la colección .....	128
9.3 FRAGMENTOS .....	128
9.4 VISTAS DESLIZANTES .....	130
9.4.1 ViewPager y PagerAdapter.....	130
9.5 PESTAÑAS.....	131
9.6 MENU LATERAL DESPLEGABLE .....	133
9.7 FORMULARIOS ANIMADOS.....	138
9.8 BOTÓN FLOTANTE .....	139
9.9 BARRA EMERGENTE.....	141
9.10 CAJA DE BÚSQUEDA.....	142
9.11 TARJETAS.....	143
9.12 VISTAS PERSONALIZADAS.....	144
9.13 TEST TEMA 9 .....	147
<b>CAPÍTULO 10. HTTP DESDE ANDROID.....</b>	<b>149</b>
10.1 TAREAS PRÁCTICAS DEL TEMA 10.....	149
10.2 HTTP.....	149
10.3 ASYNCTASK.....	150
10.4 ATRIBUTOS IMPORTANTES DE HTTP .....	152
10.4.1 URL .....	152
10.4.2 Contenido .....	153
10.4.3 Status .....	153
10.4.4 Método .....	153
10.5 JSON .....	153
10.6 TEST TEMA 10 .....	154
<b>CAPÍTULO 11. CLASES PRINCIPALES .....</b>	<b>157</b>
11.1 TAREAS PRÁCTICAS DEL TEMA 11.....	157
11.2 CONNECTIVITYMANAGER.....	157
11.3 MEDIAPLAYER .....	158
11.4 DOWNLOADMANAGER.....	159
11.5 BROADCASTRECEIVER.....	161
11.6 ALARMMANAGER.....	163
11.7 PENDINGINTENT.....	164
11.8 CONTENTPROVIDER.....	165

11.9 FILEPROVIDER .....	168
11.10 TEST TEMA 11 .....	170
<b>CAPÍTULO 12. NOTIFICACIONES Y SERVICIOS.....</b>	<b>173</b>
12.1 TAREAS PRÁCTICAS DEL TEMA 12.....	173
12.2 NOTIFICACIONES .....	173
12.3 SERVICIOS .....	175
12.4 SERVICIOS DEL SISTEMA.....	175
12.5 SERVICIOS PROPIOS.....	176
12.5.1 Servicios Iniciados .....	177
12.5.2 Servicios en Primer Plano .....	179
12.5.3 Servicios Enlazados.....	180
12.5.4 IntentService.....	180
12.6 TEST TEMA 12 .....	182
<b>CAPÍTULO 13. PERIFÉRICOS Y APIs DE GOOGLE .....</b>	<b>185</b>
13.1 TAREAS PRÁCTICAS DEL TEMA 13.....	185
13.2 CÁMARA .....	185
13.3 UBICACIÓN .....	187
13.4 MAPAS DE GOOGLE .....	191
13.5 APIs DE GOOGLE .....	191
13.6 TEST TEMA 13 .....	195
<b>APÉNDICE A. IMPORTAR LIBRERÍAS Y PROYECTOS.....</b>	<b>197</b>
A.1 IMPORTACIÓN DE LIBRERÍAS DE TERCEROS .....	197
A.2 IMPORTACIÓN DE PROYECTOS DESDE GIT.....	198
<b>APÉNDICE B. EL CONTEXTO .....</b>	<b>201</b>
B.1 LA CLASE CONTEXT .....	201
B.2 NUMERO DE CONTEXTOS.....	202
B.3 ACCESO AL CONTEXTO .....	202
<b>APÉNDICE C. GESTIÓN DE PERMISOS .....</b>	<b>205</b>
<b>APÉNDICE D. UML.....</b>	<b>209</b>
D.1 LA NECESIDAD DE MODELAR .....	209
D.2 DIAGRAMAS UML.....	211
D.3 DIAGRAMA DE CASOS DE USO.....	211
D.4 DIAGRAMA DE ACTIVIDAD .....	213
D.5 DIAGRAMA DE NAVEGACIÓN DE PANTALLAS.....	214
<b>APÉNDICE E. PUBLICACIÓN DE UNA APP .....</b>	<b>217</b>
<b>APÉNDICE F. LÍNEAS FUTURAS.....</b>	<b>219</b>
<b>MATERIAL ADICIONAL .....</b>	<b>221</b>



---

## SOBRE EL AUTOR

Valeriano Moreno (Madrid, 1983), es titulado en Ingeniería Informática por la Universidad Carlos III de Madrid así como Licenciado en Periodismo por la misma institución. Culminó sus estudios técnicos presentando el Proyecto Fin de Carrera <<Un compilador para ABNF basado en el RFC 4234>>.

Al finalizar la carrera, se incorporó a la mayor multinacional española del sector tecnológico, donde adquirió una valiosa experiencia profesional en la gestión y desarrollo de Sistemas de Información con millones de usuarios. Durante su estadía se inició en la programación en Java, lenguaje usado en la capa del servidor dentro de la Arquitectura v10, patrón de relativo éxito y exportado a países como Estados Unidos. Ocupó cargos de jefatura de proyecto y responsabilidades de investigación varias dentro del equipo de arquitectura antes de su marcha voluntaria.

Actualmente, vuelca sus esfuerzos profesionales en la investigación y divulgación de conocimiento técnico experto sobre programación. Sus servicios como formador han sido demandados por numerosas empresas e instituciones, entre las que destacan Accenture, BBVA o la prestigiosa sede de NASA en Madrid (MDSCC).

Combina esta labor docente con el desarrollo y ejercicio de consultoría en proyectos profesionales propios, sintiendo predilección por el diseño y modelado de Sistemas con UML, el ecosistema Java y últimamente, también Angular y aplicaciones híbridas.

Puede seguir su perfil profesional en:  
<https://es.linkedin.com/in/valerianomoreno>



---

## NOTA DEL AUTOR

No dejar de estudiar me convierte de facto en un eterno aprendiz. Gracias a esta actitud, me enfrento al punto de partida psicológico y motivacional que afrontan los alumnos ante la asimilación de un nuevo reto, concepto o técnica. Desde este prisma, elaboro mis manuales y el desarrollo de mis clases y ha sido desde este enfoque precisamente, donde he echado de menos la existencia de un manual de Android en español, que guíe al alumno de principio a fin y lo capacite de pleno para al desarrollo de aplicaciones en esta tecnología.

Como ferviente hispanista además, siempre he añorado la presencia de nuestra lengua materna en obras de divulgación sobre programación. Diferentes manuales son meras traducciones del inglés, lo que en muchas ocasiones nos aleja de la semántica original y dificulta enormemente la asimilación completa de una idea o método. Esta carencia, se traduce en una motivación personal extra para llevar a cabo esta obra, donde quiero plasmar mis conocimientos y expresar además mi compromiso con nuestra lengua y nuestra cultura, empleando el español de forma genuina en todo momento y/o traduciéndolo simultáneamente del modo más fiel.



# PREFACIO

En este tema, meramente introductorio, adquirimos una visión general de los objetivos del curso, la capacitación requerida para el acceso, los recursos que vamos a ir usando así como una puesta a punto del entorno práctico.

## TAREAS PRÁCTICAS DEL PREFACIO

- ▶ Crearse una cuenta en Github
- ▶ Descargar un cliente Git
- ▶ Crearse una cuenta en stackoverflow
- ▶ Hacer una carpeta de favoritos en el navegador para el curso
- ▶ Descargar jdk 8 (requiere registro gratuito)
- ▶ Descargar android studio

## PÚBLICO OBJETIVO

El libro va dirigido a cualquier estudiante de programación que quiera instruirse en el universo Android. Aunque el desarrollo del temario será gradual y partirá de los conocimientos básicos, la obra no está enfocada a personas neófitas en la programación, ya que Android, representa en sí mismo una especialidad dentro del mundo del desarrollo de aplicaciones.

Si bien un médico puede especializarse en cirugía o cardiología, antes debió cursar estudios de medicina general. De forma análoga sucede lo mismo aquí, por lo que al menos se recomienda el dominio un lenguaje imperativo o estructurado (PHP, C, Delphi, Cobol, Pascal, ShellScript, etc.) para afrontar con mayor garantía de éxito el curso.

De manera ideal o preferente, poseer conocimientos en Java, sería el requisito básico de acceso. El test de repaso de Java, antes del test del tema 0, es una prueba para determinar la aptitud inicial del estudiante. Deberías obtener al menos responder 8 preguntas completamente correctas para considerar idónea tu candidatura. Si es menos, no desesperes.

Dicho esto, por mi experiencia docente, hace más el estudiante que quiere que el que puede, por lo que sí sólo sabes un lenguaje declarativo como HTML, pero te mueres de ganas de hacer tus aplicaciones Android, es muy probable que saques un gran provecho al libro.

## METODOLOGÍA Y RECURSOS

De acorde a los tiempos que corren, no quiero producir una obra de carácter enciclopédico como única fuente de consulta. También discrepo del modelo exclusivo del videotutorial, pues al final, ves cómo lo hace otro, pero no lo haces tú. Y si bien la práctica plasma la realización del conocimiento adquirido, no hay praxis posible sin noción teórica previa. Por estas razones, los materiales que componen esta obra son:

1. Sección teórica escrita. Cada capítulo se desglosará en varios apartados que explicarán con detalle y la mayor claridad posible cada concepto de alcance, así como ideas subyacentes o relacionadas que ayuden a la comprensión de la temática en curso.
2. Videos prácticos demostrativos. Cada apartado anterior, irá acompañado de uno o más vídeos que pongan en práctica la teoría asociada. Generalmente, se construirán aplicaciones desde pequeños enunciados donde se ilustren las capacidades de un subconjunto de la plataforma Android.
3. Repositorios con el código fuente. El resultante del código producido en los vídeos quedará almacenado y listo para su descarga desde cualquier equipo con acceso a Internet y su importación a Android Studio.
4. Exámenes autoevaluables. Por cada capítulo, se adjuntará un enunciado tipo test que servirán para contrastar cuánto se ha asimilado, así como para repasar, aclarar y afianzar el temario visto. Las soluciones razonadas a estas pruebas serán resueltas en respectivos videos.

El orden de los temas está diseñado de menor a mayor complejidad, enlazado de forma gradual, por lo que se recomienda seguir la secuencia para su estudio completo; aunque también se pueden consultar temas saltados para estudiantes más avanzados.

## ¿KOTLIN, C++ O JAVA?

---

A decir verdad, una aplicación Android puede desarrollarse combinando el uso de hasta tres lenguajes: C/C++, Java y más recientemente Kotlin.

Por desgracia, a veces en el mundo de la programación, sucede lo mismo que en el mercado de la moda, y como si de una pasarela se tratase, ese ávido deseo por lo nuevo, acaba produciendo nuevos lenguajes que ganan adeptos entre departamentos y desarrolladores, como si de una tendencia triunfal se tratase.

A continuación, se explican las peculiaridades y usos de cada lenguaje, justificando nuestra elección por Java:

### C/C++

El uso de estos lenguajes se haría usando el NDK (*Native Development Kit*). Básicamente consiste en importar librerías hechas en C/C++ en nuestro proyecto para que, desde que el código Java, puedan ser invocadas, usando para ello la *Java Native Interface* (JNI [https://es.wikipedia.org/wiki/Java\\_Native\\_Interface](https://es.wikipedia.org/wiki/Java_Native_Interface)). Esta JNI hace de puente entre la ejecución de código Java en la máquina virtual y el código C o nativo.

Su uso quedaría justificado en aplicaciones donde el rendimiento sea crítico (videojuegos, sistemas en tiempo real) o donde se quieran emplear librerías que aporten una funcionalidad muy concreta en nuestra aplicación. Programar en este entorno no es en absoluto trivial, y puede tener graves consecuencias al tocar cosas de bajo nivel en la máquina virtual.

### Kotlin

Recuerdo en una entrevista de trabajo reciente para una universidad, a un responsable argentino con la corbata ahogándole el pescuezo, cómo aseveraba: <<todo es Kotlin, ya todo Android es Kotlin>> con ese tono de vendedor que tanto conviene a veces para tapar inseguridades o desconocimiento, tan propio de las tendencias de pasarela que comentábamos antes.

---

Este lenguaje ha surgido como una alternativa a Java y se puede combinar en el mismo fuente sin la dificultad que presenta el NDK, debido a que el código Kotlin, será traducido a *bytecodes* e interpretado en la máquina virtual como si de un mismo código Java se tratase.

Y bien, cabe preguntarse entonces, ¿qué aporta Kotlin como novedoso? Como lenguaje nuevo, Kotlin define una sintaxis propia, que en algunos casos, resulta más económica que la de Java cuando se domina—al principio más engorrosa—. Ofrece además algunas características alternativas a dicho lenguaje y sigue una tendencia con guiños a la programación funcional al estilo del lenguaje Swift (usado para aplicaciones IOs).

Desde un punto de vista más crítico y serio, Kotlin surge impulsado por Google como lenguaje alternativo a Java, de modo que se logre una independencia de Oracle, propietario de Java, en un gesto muy al estilo de Google, que quiere ser propietario de todo el código y las tecnologías que se empleen en su plataforma. Más allá de poder mejorar desde el punto de vista técnico a Java en algunos aspectos, es para mí la clave de propiedad la que justifica el nacimiento de Kotlin, para evitar así depender de la voluntad de evolución de los dueños de Java.

Desde Kotlin recomiendan o consideran sencillo el salto desde Java, siendo el paso inverso menos natural y desaconsejado, por lo que no abordaremos este lenguaje en la obra.

## Java

La característica multiplataforma concebida en la génesis de Java, le convierte en el lenguaje por excelencia en el desarrollo de aplicaciones Android. Gracias a que existe una máquina virtual detrás de cada proceso Java, una aplicación puede ser ejecutada con independencia de la capa hardware subyacente. Es decir, mi código puede correr en un dispositivo Samsung, Bq, Huawei o cualquier otro modelo físico, siempre que el fabricante aporte una máquina Java adaptada a dicho dispositivo. Ésa es la gran baza de Android. De modo que al contrario que la competencia, mi aplicación pueda acabar ejecutándose en miles y miles de dispositivos distintos. No de modo totalmente uniforme, también hay que decirlo.

Además de emplearse en la capa de presentación en Android, Java es el lenguaje rey indiscutible en la industria en los procesos de servidor. De modo que el conocimiento y la práctica adquiridos en Android, pueden extrapolarse a habilidades relacionadas en otras capas del desarrollo.

No sólo la extensión, sino sobre todo la madurez de Java es la característica definitiva para elegirlo como lenguaje de referencia en el desarrollo del libro. No

es una moda y no tiene lo volátil o perecedero de otras tecnologías en ciernes. Su instauración y su posición de dominancia es un hecho en la realidad empresarial, a pesar de algún argentino esnob y los deseos de otros cazadores de tendencias.

## EL SOLDADO ANDROID

---

Programar una aplicación es como construir un edificio. A ojos de cualquier persona ajena este mundo, no se me ocurre mejor analogía. Una aplicación deberá ser analizada, diseñada, codificada, testada, desplegada, mantenida y gestionada en todo su ciclo vital. No es por tanto, por norma general, una labor abordable por una única persona.

Si un edificio necesita un arquitecto, unos planos, un presupuesto, una cuadrilla de especialistas en luz, fontanería o albañilería; una aplicación, dependiendo de su tamaño, necesitará de un arquitecto o responsable técnico, un grupo de desarrolladores, maquetadores, probadores, analistas, diseñadores y gestores.

La presente obra capacita al estudiante a desarrollar sus propias aplicaciones Android de principio a fin. Pero debemos ser conscientes, que la complejidad que implica la construcción y puesta en marcha de una aplicación comercial es inasumible por una única persona, por muy super programador que se sea.

Hay una incultura promovida por la industria entorno al rol del <<chico para todo>> referido con el anglicismo *fullstack developer*, que viene a ser un perfil de programador sabelotodo, multiusos, adaptable y de bajo coste. Es la utopía en la contabilidad de una empresa; pero es una irreabilidad y una agresión a los principios de una profesión sostenible y productiva para la sociedad.

Por eso, como soldado Android, todos los conocimientos complementarios que ya traigas de otros entornos o lenguajes serán un plus, una ventaja para ti. Pero si de verdad quieras cursar una especialización y ser productivo en esta tecnología, te recomiendo te ciñas a ella, profundices y hagas por estar al día. No te frustres por no poder acaparar todo lo que rodea a una aplicación desde su concepción a su puesta en funcionamiento. Simplemente son más años de estudio y dedicación lo que ello conllevaría.

## LA POLISEMIA DE ANDROID

---

Cuando hablamos de Android, podemos estar refiriéndonos a varias de sus acepciones.

---

La primera de ellas es como sistema operativo. Android es una versión derivada de los sistemas operativos tipo Linux/Unix, competencia histórica de Windows. Y decimos que un dispositivo es Android si su sistema operativo lo es.

Si decimos que estamos haciendo una aplicación Android, el resultado de nuestro trabajo será un ejecutable para dispositivos con dicho sistema operativo. Y también, estaremos diciendo que hemos hecho la aplicación usando Android como marco de trabajo o *framework*.

Y bien, ¿qué es un marco de trabajo? Intentaremos explicarlo. Cuando aprendes a programar en Java, siempre defines un punto de acceso a tu aplicación por donde comienza su ejecución, la famosa función *main*. Pero cuando programas en Android, ese *main* ya está hecho. Al lanzar una aplicación Android, la plataforma, comienza su ejecución (en esa función *main*, definida de antemano), hará un montón de cosas, cargará archivos de configuración, instanciará una serie de clases y en un determinado momento, el flujo de la ejecución engarzará con nuestro código, siendo entonces invocadas las instrucciones programadas por nosotros.

Es decir, hacer una aplicación en Android, será completar un mecanismo ya dado, incorporar mi código a un sistema con un funcionamiento y diseño previos, para terminar produciendo nuestra *app*. De ahí el nombre de marco de trabajo o *framework*.

Hago este inciso, pues como programador, la primera vez que usé uno fue Struts 2 (empleado en la construcción de aplicaciones web de Java) y debo admitir mi confusión y desorientación por yo estar acostumbrado a hacer todas mis aplicaciones desde cero. De repente, al arrancar mi aplicación, esta hacía un montón de cosas que yo no había programado y me costaba entender cómo construir mi lógica sobre ese monstruo, sin saber de dónde venían las llamadas a mi código y a dónde iba la ejecución después de pasar por mis líneas.

Por tanto y como hemos mencionado, será obligado incluir nuestro código de acuerdo a una serie de normas predefinidas (declarar elementos xml en un determinado fichero, hacer que nuestra clase herede de determinada clase padre, etc.) para que el motor de Android sea capaz de integrar e interpretar el código fuente que producimos. Ese motor de Android es ese programa <>hecho a medias<>, es la aplicación cuyo funcionamiento vamos a estudiar y a la que podemos referirnos como entorno, marco de trabajo o simplemente Android.

Hay mucho debate en cuanto al uso de marcos de trabajo, su conveniencia o no para el aprendizaje o su uso e implantación en ámbitos profesionales. Personalmente, considero que si un marco ha alcanzado un nivel de madurez aceptable, es una ventaja para la empresa y para el programador; pues aporta unas tareas comunes ya

implementadas de modo robusto, que agilizan la creación de la aplicación. Y ése es el caso de Android.

Ahora bien, de cara a adquirir buenos fundamentos en programación, lo desaconsejo rotundamente, debido a la cantidad de detalles técnicos omitidos (ya dados por el entorno) que se hacen transparentes al programador; lo que le convierte en un mero usuario que no crea ya su propio código, sino que se limita a usar el que le viene dado. Motivo este último entre otros expresados, por lo cual no recomiendo este libro a alguien novel en las lides de la programación.

## TECNOLOGÍAS Y RECURSOS TRANSVERSALES

---

Si bien es cierto que con el libro vamos a aprender todo lo necesario, en este mundo en constante evolución, hay herramientas de consulta que se han convertido en esenciales y cuyo uso recomiendo encarecidamente como complemento de esta obra.

1. *stackoverflow.com* Es a día de hoy el foro de programadores más empleado en el mundo. En él se plantean a diario cientos de dudas sobre problemas o errores que aparecen al programar. En la mayoría de los casos se obtienen soluciones aportadas por otros usuarios de modo altruista. Considero indispensable su consulta ante cualquier fallo cuya causa no identifiquemos de primeras. No es la biblia, ojo, pero sí un lugar de consulta con una fiabilidad notable. ¡Hazte un usuario y contribuye! Además, está en español, así que no tienes excusa.
2. *github.com* Recientemente adquirido por Microsoft, miles de programadores a diario vuelcan allí su código. La propia casa de Android, publica allí ejemplos con aplicaciones de muestra. Aprender viendo código ya resuelto y ejecutándolo paso a paso, es otra forma de adquirir conocimiento nada desdeñable. De paso, puedes alojar allí tus ejemplos, que valgan para otros programadores y como copia de seguridad propia. Aprender Git es una tecnología transversal que recomiendo encarecidamente. Pero ello, sí que cae el margen del propósito de esta obra.
3. *developer.android.com* El dominio oficial de divulgación de Android, donde nos contarán todas las novedades de la plataforma y se publica toda la documentación. No es todo lo clara, completa o actualizada que desearíamos en ocasiones, pero sí es un lugar donde profundizar, consultar y encontrar pequeños ejemplos listos para consumir.

## ANDROID EN LAS ARQUITECTURAS DE APLICACIONES MODERNAS

¿Qué se puede hacer con Android y qué no? De dónde surge su necesidad, su uso y alcance, es lo que vamos a disertar en este punto.

Android se concibe para ser usado en la capa de presentación, como el cliente, la vista o el *front-end*, término guiri este último. Es decir, como norma general se empleará para:

1. Representar la interfaz de usuario
2. Ofrecer al usuario las operaciones que este puede realizar
3. Realizar operaciones sencillas de cálculo y almacenamiento
4. Obtener y representar datos de un servidor con el que intercambiamos información

Normalmente, una aplicación moderna (WhatsApp, Facebook, Metro de Madrid, etc.) está distribuida generalmente en tres capas: cliente, servidor y base de datos, presentando esta arquitectura tipo:



De izquierda a derecha: el dispositivo Android de turno, el servidor de aplicaciones y la base de datos. Todos estos elementos juntos en acción constituyen el cuerpo y la estructura habitual de una aplicación moderna. Cabe destacar que como clientes alternativos a Android, se emplea IOS, páginas web y aplicaciones híbridas.

Iremos viendo paso a paso cómo realizar estas tareas, pero quiero dar a entender desde ya, que por lo general, nuestra aplicación de Android será sólo una pieza más dentro de una aplicación de índole superior, de un ecosistema más

complejo. A veces un mero consumidor, cuya mayor responsabilidad se limita a cargar y representar los datos obtenidos de un servidor.

Aunque en Android podemos tener bases de datos locales, en los sistemas <<gordos>>, la base de datos siempre debe residir en una capa independiente, para garantizar su disponibilidad e integridad, además de por motivos de seguridad.

La capa del servidor y la base de datos, tienen sus respectivas capas tecnológicas que quedan al margen de este libro, aunque sí dedicaremos un capítulo a estudiar con detalle cómo comunicarnos con ellas desde nuestro dispositivo.

## La conversación remota tipo

No siempre es necesario que una aplicación Android tenga parte remota. Es decir, que para cubrir su funcionalidad, necesite de un servidor externo. Por ejemplo la Galería, que nos muestra fotos almacenadas localmente, o la Calculadora, carecen de ella. Pero en caso de que fuera necesario, la conversación tipo sería así:

1. Nuestra *app* enviará datos a un servidor (generalmente mediante HTTP), bien para su almacenamiento remoto, bien para conseguir la comunicación con otro dispositivo (usando al servidor como intermediario) o para obtener algún tipo de Información. Cabe destacar que será siempre el terminal Android, el que inicie las conversaciones con el servidor.
2. El servidor, recibido el mensaje, llevará a cabo la acción requerida (consultar correo, buscar restaurante, confirmar contraseña, etc.), leerá o escribirá en la base de datos, en caso de ser necesario, y procurará una respuesta al dispositivo.
3. Esa respuesta, a veces tendrá su representación gráfica en nuestra aplicación, en forma de listado, notificación o aviso.

## CERTIFICACIONES ANDROID

---

Actualmente, existen dos compañías que acreditan los conocimientos y destreza en esta tecnología:

1. ATC y su certificación <<Android Certified Application Developer>>
2. Google y su certificación <<Associated Android Developer>>

---

Aunque cada una tiene su propio temario y procedimiento, el libro abarca unos contenidos y estructura equiparables, por lo que superar los objetivos y evaluaciones marcados por esta obra puede considerarse suficiente para afrontar la obtención de estos títulos con éxito.

Yo mismo obtuve la certificación ATC en la categoría de formador, por solicitud de una empresa con las que colaboré.

Personalmente, no considero que la posesión de un certificado demuestre la capacitación de un alumno. Valoraría mucho más ejemplos del código un autor en un repositorio público tipo Github, o aplicaciones publicadas por él en la tienda de Google.

Eso sí, si buscas ser contratado por alguna empresa, es posible que te interese obtenerlas, por pura imagen. Sabe Dios cómo operan los departamentos de Recursos Humanos y esa cadena interminable de buscavidas e intermediarios, ignorantes de la técnica.

Hasta aquí la previa del libro, hecha para ayudaros a aterrizar en la tecnología Android. Desde ya, entrando en harina. Por cierto, no olvides realizar el test de este tema.

## TEST DE REPASO DE JAVA

---

La prueba a continuación es una piedra de toque que pretender servir al lector para calibrar su idoneidad a la hora de estudiar el libro. Es un test de respuesta múltiple por lo que puede haber más de una respuesta correcta a cada pregunta planteada.

Es un ejercicio meramente orientativo, pero en caso de obtener una puntuación inferior a 5 respuestas totalmente correctas, el lector debería plantearse mejorar sus habilidades previas en Java para retomar su aprendizaje con mayor garantía de éxito.

De la infinidad de material que puedes encontrar sobre Java en internet, me atrevo a recomendarte el curso básico de Aprende a programar en Java de Antonio Martín Sierra, experimentado formador y programador, además de colega.

1. ¿Cuáles de las siguientes afirmaciones son ciertas?
  - a) Java es un lenguaje que pertenece al paradigma de orientación a objetos
  - b) Un programa hecho en Java, puede ser ejecutado en cualquier ordenador

- c) El JDK incluye la API de Java SE
  - d) Java se caracteriza por ser un lenguaje rápido en su ejecución
2. Un objeto en Java, es:
- a) Una clase con el modificador private
  - b) Una instancia de una Clase
  - c) Un concepto importante en mi aplicación
  - d) Una variable global
3. Sobre el método constructor de una clase, podemos afirmar:
- a) Es un método implícitamente definido para cada clase
  - b) Podemos escribir varias versiones de él, realizando la sobrecarga del mismo
  - c) Al invocarlo, nos devuelve un objeto de esa clase
  - d) Es necesario para reservar un espacio de memoria al declarar un objeto
4. Documentar el código:
- a) Es una actividad carente de interés
  - b) En Java no se hace necesario, ya que el propio lenguaje es autoexplicativo
  - c) Se tiene que hacer una vez acabado el desarrollo de una aplicación
  - d) Aumenta el tamaño del código de forma innecesaria
  - e) Ninguna de las anteriores
5. ¿Qué salida produce el siguiente código?
- 
- ```
.....  
class Prueba {  
    public int calcula ()  
    {  
        return 1+1;  
    }  
    public static void main (String [] argumentos)  
    {  
        System.out.println (calcula());  
    }  
}
```
-

- a) Imprime un 2 en la consola
  - b) Imprime 1+1 en la consola
  - c) No compila, porque no puede devolver la suma de dos constantes
  - d) No compila, porque calcula no se invoca sobre una instancia de Prueba
  - e) Da un error de ejecución, por ser argumentos el nombre del parámetro del main
6. Una Clase, es:
- a) Un concepto importante dentro de mi aplicación
  - b) Un tipo de dato necesario para la compilación de un programa Java
  - c) El archivo .class de cada objeto
  - d) Ninguna de las anteriores
7. Los tipos primitivos, son:
- a) Tipos de datos usados en Java para instanciar variables simples
  - b) Algunos ejemplos son char, byte, int, float
  - c) Clases con sus atributos y métodos
  - d) Todas las anteriores son ciertas
8. La Herencia es un proceso en el cual:
- a) Una clase hija, hereda de otra padre, sus atributos y métodos
  - b) Una técnica para acelerar la ejecución de un proceso Java
  - c) Se emplea usando la palabra reservada extends
  - d) Se emplea usando la palabra reservada implements
9. El siguiente fragmento de código, produce como resultado

```
.....  
for (int a=0; (a%6)<6; a++)  
{  
    System.out.println (a);  
}  
.....
```

- a) No compila puesto que % no pertenece al lenguaje
- b) Imprime la serie de números 0, 1, 2, 3, 4, 5
- c) Imprime la serie de números 0, 1, 2, 3, 4, 5, 6
- d) Ninguna de las anteriores

10. El método main:
  - a) Debe estar definido en todas las clases que se utilizan en la aplicación
  - b) Devuelve un valor como resultado de la ejecución
  - c) Recibe como parámetros los argumentos que se pasan al invocar al programa
  - d) Lo contendrá la clase Main, necesaria para ejecutar la aplicación y significa el punto de entrada al programa
11. Respecto de una interfaz, podemos afirmar que:
  - a) Es una clase que sirve para instanciar objetos
  - b) Es un tipo de archivo que permite describir la funcionalidad
  - c) Si una clase implementa una Interfaz, deberá sobrescribir todos sus métodos
  - d) Su uso, acelerará la ejecución de nuestros programas Java
12. Indique cuáles son las formas correctas de declarar un array de caracteres:
  - a) char [] mi\_array;
  - b) char mi\_array;
  - c) char mi\_array[];
  - d) char [][] mi\_array;
13. Respecto a las estructuras iterativas o bucles en Java:
  - a) Existen 2 tipos, el while y el for
  - b) Existen 4 tipos, el while, do-while, repeat-until y el for
  - c) Existen 4 tipos, el while, do-while, switch y el for
  - d) Existen 3 tipos, el while, do-while, y el for
14. ¿Qué es un casting?
  - a) Una evaluación de Java para determinar el modelo de ejecución óptima
  - b) El proceso de selección de Java para inferir la clase Main
  - c) Es el proceso de conversión de un tipo de datos a otro equivalente
  - d) No existe ese concepto en Java

- 
15. El modificador static, será empleado cuando:
    - a) Queremos que una clase hija, herede de una clase padre
    - b) Queremos definir métodos o atributos a nivel de objeto
    - c) Queremos definir métodos o atributos a nivel de clase
    - d) Todas las anteriores son ciertas
  16. La palabra reservada super, hace referencia:
    - a) Al constructor de una clase
    - b) Al método main de la clase principal
    - c) Al método de la clase padre con el mismo nombre
    - d) Dentro del constructor de la clase hija, invoca al constructor del padre
  17. Los modificadores de visibilidad son:
    - a) 2 public y private
    - b) 3 public, protected y private
    - c) 4 public, protected private y package-protected (sin modificador)
    - d) 4 public, protected private y no-protected
  18. Respecto de las colecciones, seleccione las verdaderas:
    - a) Existen Mapas, Listas y conjuntos (representados por sendas interfaces)
    - b) ArrayList, LinkedList y Vector son tipos List
    - c) TreeMap hace inserción ordenada por la clave
    - d) TreeMap hace inserción ordenada por el valor
  19. Respecto de MVC:
    - a) Es un patrón arquitectónico para organizar un proyecto
    - b) La capa del Modelo, recibe las peticiones del cliente
    - c) En JEE, los Servlets hacen de controlador y los JSP de vistas
    - d) En Java, no podemos aplicar este patrón
  20. Respecto de Maven:
    - a) Propone una estructura estándar de un proyecto Java
    - b) Puedo ejecutarlo para obtener un jar
    - c) Es una tecnología poco utilizada
    - d) Obtiene copia local automática de las librerías almacenadas en un repositorio

## Soluciones

|          |            |
|----------|------------|
| <b>1</b> | a, c       |
| <b>2</b> | b          |
| <b>3</b> | a, b, c, d |
| <b>4</b> | e          |
| <b>5</b> | d          |

|           |      |
|-----------|------|
| <b>6</b>  | a    |
| <b>7</b>  | a, b |
| <b>8</b>  | a, c |
| <b>9</b>  | d    |
| <b>10</b> | c, d |

|           |      |
|-----------|------|
| <b>11</b> | b, c |
| <b>12</b> | a, c |
| <b>13</b> | d    |
| <b>14</b> | c    |
| <b>15</b> | c    |

|           |         |
|-----------|---------|
| <b>16</b> | c, d    |
| <b>17</b> | c       |
| <b>18</b> | a, b, c |
| <b>19</b> | a, c    |
| <b>20</b> | a, b, d |

## TEST PREFACIO

---

1. Respecto al nivel de partida del lector, marque las indicaciones correctas:
  - a) Saber Java sería el punto de partida ideal
  - b) Puedo aprender a programar en Android sin saber nada de programación
  - c) Para colaborar en un proyecto Android es imprescindible saber de programación
  - d) Saber HTML es el punto de partida ideal
2. ¿Cuántos lenguajes imperativos puedo emplear en una aplicación Android?
  - a) Uno
  - b) Dos
  - c) Tres
  - d) Cuatro
3. Los roles de un grupo de trabajo de una aplicación de Android grande son:
  - a) Programador
  - b) Programador y probador
  - c) Programador, probador y diseñador
  - d) Gestor, arquitecto, analista, diseñador, programador, maquetador y probador

- 
4. Android es:
    - a) Android es un sistema operativo
    - b) Android es un marco de trabajo
    - c) Android es un marco de trabajo y un sistema operativo
    - d) Android es una librería
  5. Marque la opción correcta:
    - a) Puedo encontrar la documentación oficial de Android en github.com
    - b) El portal stackoverflow es una referencia oficial
    - c) Puedo encontrar ejemplos de código tanto en stackoverflow como en github
    - d) Todas las anteriores son falsas
  6. Android, en una aplicación moderna:
    - a) Representa la vista, el cliente
    - b) Es sólo una parte más de algo más grande, por lo general
    - c) Su misión principal es la interfaz de usuario y recibir y enviar datos al servidor
    - d) Todas las anteriores son ciertas
  7. Respecto de las certificaciones en Android:
    - a) No existen certificaciones que acrediten conocimientos en Android
    - b) Existen 3 certificaciones distintas
    - c) Es básico adquirir al menos una para acreditar conocimientos
    - d) Hay 2: una de ATC y otra de Google
  8. El código en Kotlin se traduce a bytecodes de Java:
    - a) Verdadero
    - b) Falso

## SOLUCIONES

1a, 2c, 3d, 4c, 5c, 6d, 7d, 8a

# 1

---

## INTRODUCCIÓN

Ubicamos los elementos comunes de un proyecto Android, describiendo el marco de trabajo y sus utilidades más destacadas.

### 1.1 TAREAS PRÁCTICAS DEL TEMA 1

---

- ▶ Crear proyecto en Android
- ▶ Identificar carpetas clave
- ▶ Aprender a depurar y usar el log
- ▶ Personalizar atajos y vistas de Android Studio

### 1.2 SDK

---

El SDK de Android (*Software Development Kit*) es, de forma análoga al *JDK*, el conjunto básico de librerías que forman el núcleo del entorno que permite desarrollar aplicaciones Android (compilador, herramientas de depuración, librerías básicas, etc.).

Como un compendio de aplicaciones, el SDK es distribuido con un número de versión que lo identifica. Este número, también llamado API, es el que posteriormente va asociado a cada título o versión de un dispositivo concreto.

Aquí el listado de versiones del SDK hasta la fecha y su título asociado:

| Título             | Versión     | Fecha de lanzamiento     | SDK API |
|--------------------|-------------|--------------------------|---------|
| Apple Pie          | 1.0         | 23 de septiembre de 2008 | 1       |
| Banana Bread       | 1.1         | 9 de febrero de 2009     | 2       |
| Cupcake            | 1.5         | 25 de abril de 2009      | 3       |
| Donut              | 1.6         | 15 de septiembre de 2009 | 4       |
| Eclair             | 2.0 – 2.1   | 26 de octubre de 2009    | 5 – 7   |
| Froyo              | 2.2 – 2.2.3 | 20 de mayo de 2010       | 8       |
| Gingerbread        | 2.3 – 2.3.7 | 6 de diciembre de 2010   | 9 – 10  |
| Honeycomb          | 3.0 – 3.2.6 | 22 de febrero de 2011    | 11 – 13 |
| Ice Cream Sandwich | 4.0 – 4.0.5 | 18 de octubre de 2011    | 14 – 15 |
| Jelly Bean         | 4.1 – 4.3.1 | 9 de julio de 2012       | 16 – 18 |
| KitKat             | 4.4 – 4.4.4 | 31 de octubre de 2013    | 19 – 20 |
| Lollipop           | 5.0 – 5.1.1 | 12 de noviembre de 2014  | 21 – 22 |
| Marshmallow        | 6.0 – 6.0.1 | 5 de octubre de 2015     | 23      |
| Nougat             | 7.0 – 7.1.2 | 15 de junio de 2016      | 24 – 25 |
| Oreo               | 8.0 – 8.1   | 21 de agosto de 2017     | 26 – 27 |
| Pie                | 9.0         | 6 de agosto de 2018      | 28      |
| "Q"                | 10.0        | agosto del 2019          | 29      |

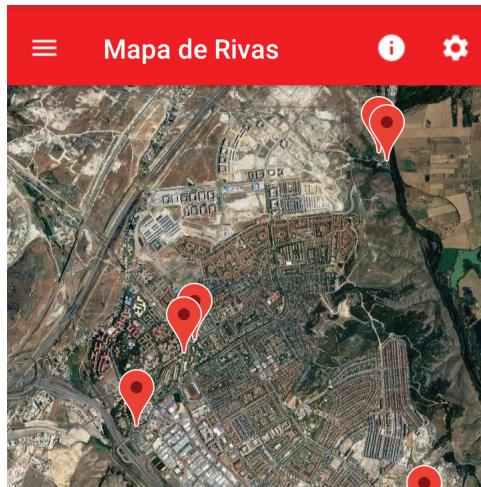
Si observas el título o nombre de cada versión, verás que va en orden alfabético ascendente, desde manzana hasta tarta (pie en inglés). Y además, siempre en relación a alimentos dulces. ¿Te atreves a apostar por el nombre definitivo de la versión Q?

### 1.3 CONCEPTO CLAVE DE API

Literalmente, es un acrónimo del inglés *Application Programmer Interface*. En términos llanos, el API es el conjunto de clases con sus respectivos atributos y métodos que quedan a mi disposición como programador y voy a poder invocar siempre que necesite. Por ejemplo, String y su método length (), son parte del API estándar de Java (JSE). Es decir, ya está programado y al ser parte del API, sólo debo usarlo a conveniencia.

Con el paso del tiempo, el SDK incluye nuevas clases y métodos y modifica otras, lo que termina constituyendo nuevas versiones, con novedosos recursos para el programador y finalmente, para el usuario. Eso da lugar a un nuevo API. Conocer el API de un lenguaje o tecnología es sinónimo de dominarla.

Un ejemplo claro que todo el mundo puede comprender es el ActionBar es, añadido en el API 11, que se corresponde con la barra horizontal en la parte alta de la pantalla. En la captura mostrada a continuación, el ActionBar es la porción roja que alberga el título de la pantalla <<Mapa de Rivas>>, y los tres iconos del menú. Los que tuvimos un móvil anterior, vimos cómo antes no teníamos ese objeto visual en nuestras pantallas y fue toda novedad en versiones posteriores.



Como programador, ese ActionBar es una clase nueva, que amplía el API; puesto que ahora puede crear y manipular ese nuevo elemento, haciendo uso de esa clase y sus métodos ya dados.

## 1.4 ENTORNO DE DESARROLLO

---

Para el desarrollo de aplicaciones Android necesitamos descargar Android Studio. Es el entorno integrado de desarrollo (IDE, en inglés), que usaremos para programar nuestras *apps*. Ir familiarizándonos con esta herramienta, ayuda mucho a aumentar nuestra productividad. Recomiendo encarecidamente definir y usar los atajos de teclado.

Antaño, era posible programar para Android desde Eclipse –entorno por excelencia para desarrolladores Java, que además permitía desplegar simultáneamente tu app y el servidor–, pero desde abril de 2016, se ha hecho obligado el empleo de esta herramienta.

Han sido años difíciles para lograr una versión madura del IDE, pero finalmente, desde la versión 3, parece lograda cierta estabilidad. Este IDE, trae consigo todo lo necesario para poder construir nuestras aplicaciones. Los elementos destacados son:

- ▶ **JDK** En su versión 8, contienen todo el API estándar de Java que necesitaríamos para hacer una app de Java (javac, javadoc, las clases como File, String, etc.) Por problemas de compatibilidad con el SDK, para usar características de la versión 8 de Java necesitaremos restringir su funcionamiento al API 24, 25 o incluso 26 del SDK. Aviso para amantes de ir a la última: no es una buena idea. Ni que decir que el uso de la última versión de Java estable, la 12, es inimaginable hoy por hoy.
- ▶ **SDK** Como hemos dicho antes, son todas las clases que nos dan ya en Android. Es decir, además de las clases de Java, comentadas en el punto anterior, quedarán a nuestra disposición clases como Button, Activity o ActionBar, que iremos empleando en la creación de nuestros ejemplos.
- ▶ **SDK Manager** Herramienta que permite descargarnos a nuestro equipo distintas versiones del SDK o complementos necesarios para el entorno, actualizaciones, etc. Será útil, si por ejemplo, trabajamos con el fuente de una aplicación hecha con la API 21 y en nuestro equipo sólo tenemos la API 28. No es preocupéis, que el entorno nos ayudará con esto llegado el caso.
- ▶ **Gradle** Herramienta que gestiona el ciclo de vida de mi aplicación. Desde la gestión de dependencias, compilación, testeo, empaquetado y despliegue. Análogo a Maven, para los que vienen de entornos Java modernos, esta fue desarrollada por la gente de Google. Evolución de herramientas tipo ant, o makefile. ¿Maven no les valía? No, ellos querían la suya, que empezase por G.

No será raro padecer severos quebraderos de cabeza para resolver ciertos fallos provocados por Gradle. Ríos de tinta en foros de Internet dan fe de ello. Esto, nos puede obligar a actualizarnos el IDE a otra versión más reciente o a redefinir nuestras aplicaciones, pues Gradle arrastra dependencias intrínsecas con el entorno que hace que proyectos desarrollados con una versión antigua de Gradle, no sean inteligibles por determinada versión de Android Studio. En fin, un follón. Gracias Google. Con G, sí.

Generalmente, y por suerte, para cubrir las necesidades normales de un programador, sólo tendremos que tocar algún parámetro de configuración en los ficheros con extensión *.gradle*

- ▶ **AVD** Android Virtual Device, es una extensión del IDE que nos permite definir dispositivos de manera virtual para volcar allí nuestra *app* y hacer pruebas en ellos. A pesar de las grandes mejoras logradas en este complemento, desde mi experiencia desaconsejo su uso. Prefiero siempre probar las aplicaciones en un dispositivo físico. Es mucho más rápido y realista.
- ▶ **Herramientas de la plataforma** En el subdirectorio *Android/Sdk/platform-tools* alberga una serie de herramientas útiles para la compilación y el despliegue de la app. Entre ellas, destaca *adb*, acrónimo de *android device bridge*, que se para comunicar nuestro ordenador y nuestro móvil, permitiendo la transferencia de ficheros y la ejecución de instrucciones en el dispositivo desde nuestro terminal del PC. No siendo imprescindible, sí recomiendo la experiencia de trastear con algún comando de los descritos en la documentación oficial.

## 1.5 UN PROYECTO ANDROID

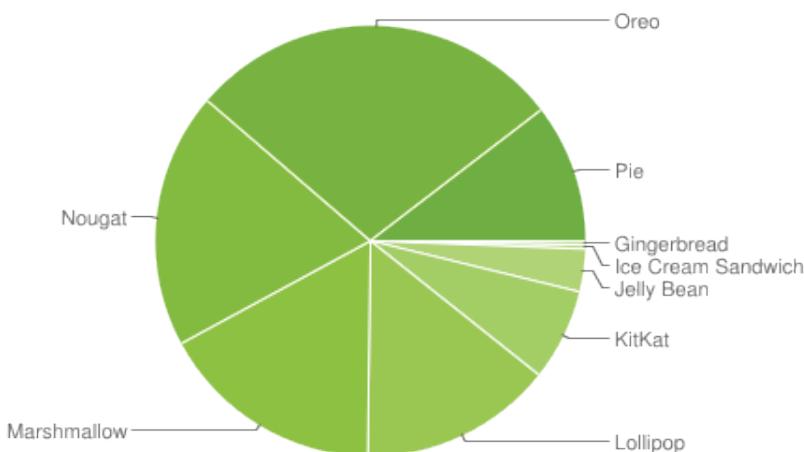
---

Cuando creamos una aplicación en Android, lo hacemos siguiendo el asistente del IDE, el cual nos solicita algunos atributos claves y genera un montón de ficheros y directorios que constituyen la base o el esqueleto de nuestra aplicación. Vamos a desmigar un poco los atributos de interés y la organización inherente a cualquier proyecto Android dado:

- ▶ **minSkdVersion:** Indica el API a partir del cual nuestro programa estará disponible. Es decir, si ponemos la 15, móviles con la versión 14 o inferiores no podrán instalar o siquiera ver en la tienda nuestra aplicación. Para decidir este atributo, es bueno basarse en la estadísticas que proporciona Android sobre el porcentaje de cada versión en uso. Adjunto la estadística en el momento de edición de esta sección:

| Versión       | Nombre clave       | SDK API | Porcentaje de uso |
|---------------|--------------------|---------|-------------------|
| 2.3.3 - 2.3.7 | Gingerbread        | 10      | 0.3%              |
| 4.0.3 - 4.0.4 | Ice Cream Sandwich | 15      | 0.3%              |
| 4.1.x         | Jelly Bean         | 16      | 1.2%              |
| 4.2.x         |                    | 17      | 1.5%              |
| 4.3           |                    | 18      | 0.5%              |
| 4.4           | KitKat             | 19      | 6.9%              |

| Versión | Nombre clave | SDK API | Porcentaje de uso |
|---------|--------------|---------|-------------------|
| 5.0     | Lollipop     | 21      | 3.0%              |
| 5.1     |              | 22      | 11.5%             |
| 6.0     | Marshmallow  | 23      | 16.9%             |
| 7.0     | Nougat       | 24      | 11.4%             |
| 7.1     |              | 25      | 7.8%              |
| 8.0     | Oreo         | 26      | 12.9%             |
| 8.1     |              | 27      | 15.4%             |
| 9       | Pie          | 28      | 10.4%             |



En teoría es deseable abarcar el máximo número de dispositivos posibles, con lo que está bien guiarse por estos datos para fijar el valor del `minSdkVersion`. A veces, cuando queramos emplear una característica moderna, sólo disponible en SDK tardíos, nos veremos obligados a modificar este atributo o a gestionar desde el código distintos caminos en función de la versión del dispositivo sobre la que corre nuestro aplicativo. Veremos cómo gestionar este escenario en el vídeo práctico que acompaña esta sección.

- ▶ **compileSdkVersion.** Versión del SDK con la que se está compilando nuestra aplicación. Normalmente, es la última disponible, ya que por el principio de compatibilidad hacia atrás, una aplicación construida con la versión más moderna del API garantiza en principio que pueda ejecutarse en los dispositivos más antiguos.

- ▶ **targetSdkVersion.** Versión del SDK para la que se asegura compatibilidad por parte del desarrollador, por haber sido empleada durante toda la fase de pruebas. En general, su valor es igual al anterior. El sistema operativo puede usar este atributo a modo de consulta, para gestionar algunas características modernas sólo aplicables a determinado API (por ejemplo temas o aspectos de apariencia). En caso de no definirse, se asume el valor del minSdkVersion.
  - ▶ **versionCode.** Número entero positivo que identifica la versión de nuestra app. Seguirá la secuencia 1, 2, 3, etc. y por cada nueva evolución que queramos publicar, deberemos incrementarla.
  - ▶ **versionName.** Empieza valiendo "1.0" por defecto y es el nombre de la versión de nuestra app. Al ser un literal, aquí podemos jugar un poco bautizar a nuestra obra con algún nombre algo más descriptivo, al estilo de los nombres dulces del SDK. Aunque para proyectos formales, se invita a seguir la nomenclatura Numero.Numero.Numero, por ejemplo 2.3.1, donde los números más a la izquierda indiquen cambios notables y los más a la derecha evoluciones más leves o correcciones insignificantes.
- Ambos atributos serán visibles en la descripción de las aplicaciones instaladas en el dispositivo, desde el menú Ajustes.
- ▶ **applicationId.** El identificador de nuestra aplicación en la tienda de Google. Debe ser por tanto, único y distinto para cada aplicación publicada. Se le llama también paquete, por estar conformado al estilo de la nomenclatura de los paquetes Java <carpeta>.<subcarpeta1>.<sub carpeta2>.\* y porque además, definirá cuál es el paquete raíz del código fuente en mi aplicación debajo de la carpeta src.

Como detalle sin aparente importancia, se suele dejar el que proporciona el asistente del tipo com.example.<nombre\_de\_mi\_app> lo cual resulta un revés a la hora de publicar nuestra aplicación en la tienda, ya que no se permiten identificadores del estilo com.example.\* Conviene por tanto, echarle una pensada al inicio o tenerlo presente. Algunos ejemplos con aplicaciones propias, por si sirven de inspiración:

- ▶ com.val.ebtm donde com, es abreviatura de comercial, val, derivado mi nombre, ebtm, acrónimo del nombre de mi aplicación "El boxeo tiene música".
- ▶ edu.val.idel.rivas donde edu, es el prefijo por ser un parte de proyecto educativo, val, de nuevo por mis iniciales, idel, por la compañía contratista, y rivas, del municipio madrileño sobre el que versa.

Pon el que quieras. Si tiene algo de sentido, mejor que mejor.

## 1.6 MANIFEST

El `AndroidManifest.xml` es fichero clave en la configuración del entorno Android. Su ubicación es invariante (`/app/src/main/`) y en él se declaran los aspectos principales de una aplicación. La extensión de este archivo, dan una idea de cómo de grande o compleja es una *app*.

Para su edición, resulta muy cómodo el asistente que incorpora Android Studio, pues como una especie de diccionario, nos predice las opciones a redactar según el contexto/posición del puntero de escritura. Algunos elementos de relevancia que deben declararse en el fichero de manifiesto:

- ▶ Permisos: algunas acciones llevadas a cabo por nuestra aplicación deben de ser consentidas y/o comunicadas al usuario. Por ejemplo, si nuestra aplicación necesita conectarse a Internet, leer la agenda de contactos o acceder a la galería, ello implicará que se describan esos permisos de forma explícita en la sección superior del *manifest*.
- ▶ Componentes: en Android, existe una categoría llamada componentes, que agrupa a tres tipos de clases fundamentales: las actividades, los servicios y los receptores. Las actividades, serán las distintas pantallas o vistas que compongan nuestra aplicación. Los servicios, serán clases que desarrollan algún proceso de interés de modo no visual. Y los receptores, serán una especie de sensores, que serán activados ante una alarma, el inicio del dispositivo móvil, la finalización de la descarga de un archivo y eventos del estilo.

Bien estos tres tipos de clases (*Activity*, *Service* y *Receiver*), que reciben el nombre genérico de componente, constituyen la parte esencial del núcleo de Android. Dominarlas, será dominar Android. Pues bien, dada su especialidad, estas deban ser declaradas de forma explícita en el fichero de manifiesto, entre otras cosas porque será Android el que lea este fichero e instancie las clases en él descritas. Si estás familiarizado con ello, esto no es ni más ni menos que la Inversión de Control o *IOC*.

Pero bien, no nos desviemos de objetivos sencillos. Otros aspectos que pueden definirse en este fichero, será la configuración vertical y horizontal de una pantalla, su estilo, el ícono de la aplicación, claves para el uso de APIs de Google y un sinfín de características relacionadas con la configuración. iremos desgranando su uso en ejemplos prácticos.

Ejemplo de un fichero de manifiesto:

```
.....  
<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
package="edu.val.idel.rivas.turismorivas">  
  
    <application  
        android:allowBackup="true"  
        android:icon="@mipmap/ic_launcher"  
        android:label="@string/app_name"  
        android:supportsRtl="true"  
        android:theme="@style/AppTheme">  
        <activity android:name=".actividades.SplashActivity"  
            android:configChanges="keyboardHidden|orientation|screenSize"  
            android:screenOrientation="portrait">  
            <intent-filter>  
                <action android:name="android.intent.action.MAIN" />  
                <category android:name="android.intent.category.LAUNCHER"/>  
            </intent-filter>  
        </activity>  
        <activity android:name=".actividades.CreditosActivity"  
            android:configChanges="keyboardHidden|orientation|screenSize"  
            android:screenOrientation="portrait"  
            android:label="Créditos"/>  
        <activity android:name=".actividades.InicioActivity"  
            android:configChanges="keyboardHidden|orientation|screenSize"  
            android:screenOrientation="portrait"  
            android:label="Inicio"/>  
        <activity android:name=".actividades.MapaActivity"  
            android:configChanges="keyboardHidden|orientation|screenSize"  
            android:screenOrientation="portrait"  
            android:label="Mapa de Rivas"/>  
        <activity android:name=".actividades.PuntoDeInteresActivity"  
            android:configChanges="keyboardHidden|orientation|screenSize"  
            android:screenOrientation="portrait"  
            android:label="Punto de interés"/>  
        <activity android:name=".actividades.AjustesActivity"  
            android:configChanges="keyboardHidden|orientation|screenSize"  
            android:screenOrientation="portrait"  
            android:label="Ajustes"></activity>  
    </application>  
  
</manifest>  
.....
```

## 1.7 EXTENSIONES

Por ser este un tema introductorio, vamos a hacer un repaso a las extensiones de archivos más comunes que podemos encontrarnos en el universo Android. Ello, nos dará la pista del contenido del fichero y nos ayudará a ubicarnos.

- ▶ **.java** La extensión por excelencia de todo archivo fuente Java. Un archivo con esta extensión contendrá la declaración de una clase con el mismo nombre que el archivo. Es una buena práctica que cada archivo contenga una única clase (evitando las clases anónimas). Ello, ayuda a la legibilidad y sostenibilidad de una aplicación.
- ▶ **.class** El archivo fuente, traducido a *bytecodes* de Java para su interpretación por la máquina virtual. Proceso cometido automáticamente por el IDE y archivos generalmente no visibles mientras trabajamos desde Android Studio.
- ▶ **.jar, .zip** Extensiones que contienen agrupaciones de .class; por lo general, bibliotecas/librerías de Java usadas en nuestro proyecto.
- ▶ **.apk** El fichero ejecutable Android, de inglés *Android Package*. Nuestra programa, quedará ensamblado y listo para su distribución y ejecución en un archivo con esta extensión. Será el compendio resultante de nuestra app que subiremos a la tienda y será instalado en nuestro dispositivo. Análogo a los .exe en entornos Windows.

Hay otras extensiones que aparecen en un proyecto como .xml, .png y otros ejecutables o de configuración. No son propiamente características de Java, por lo que dejamos su explicación para más adelante.

## 1.8 ESTRUCTURA DE UN PROYECTO

Una de las características comunes de los marcos de desarrollo es que definen una estructura de directorios común a todas las aplicaciones. Así se evita que cada programador de Android establezca de forma arbitraria la ubicación de su código fuente, los archivos de recursos, etc. Además que, cuando Android arranque, irá a buscar cada tipo de archivo a una ubicación concreta. Debemos por tanto, conocer la convención a este respecto.

A continuación, describimos las principales carpetas y ubicaciones de un proyecto Android:

- ▶ /app/build.gradle .- Fichero de configuración gradle de mi app (gestión de dependencias, versiones)
- ▶ /app/build/ .- Directorio para los .class
- ▶ /app/libs/ .- Directorio para importar librerías manualmente o privadas
- ▶ /app/src/androidTest/ .- Directorio para clases de test lanzadas desde el dispositivo o instrumentales
- ▶ /app/src/main/assets .- Para ficheros de propiedades o html
- ▶ /app/src/main/AndroidManifest.xml .- El fichero de manifiesto
- ▶ /app/src/main/java .- Directorio para el código fuente
- ▶ /app/src/main/test .- Directorio para clases de test lanzadas desde el PC o unitarios
- ▶ /app/src/main/res .- Recursos: imágenes, estilos, literales, etc.
  - /drawable/ .- Recursos gráficos genéricos
    - /drawable-ldpi (densidad baja)
    - /drawable-mdpi (densidad media)
    - /drawable-hdpi (densidad alta)
    - /drawable-xhdpi (densidad muy alta)
    - /drawable-xxhdpi (densidad muy muy alta )
  - /mimap/ .- Reservada para el icono de la aplicación
  - /layout/ .- Todos los xml que representan la interfaz de usuario
  - /anim/ .- Directorio para animaciones
  - /menu/ .- Directorio que contiene los menús de la aplicación
  - /values/colors.xml .- Archivo que define los colores empleados en la aplicación
  - /values/strings.xml .- Archivo con expresiones literales
  - /values/arrays.xml .- Archivos con array de valores estáticos o de carga
  - /values/dimens.xml .- Archivo para definir tamaños

Todos los valores que encontraremos en estos ficheros precedidos de una arroba @, serán recursos que podrán ser referidos desde la clase R.

## 1.9 LA CLASE R

Todos los archivos ubicados bajo la carpeta /res y los valores en ellos definidos, tienen su equivalencia automática en un atributo de la clase R. Esta clase se recrea automáticamente cada vez que incluyo o modiflico algún archivo de ese directorio.

La clase R es en realidad una facilidad que me propone el entorno, de modo que el uso y manejo de los recursos de la aplicación, sea más sencillo para el programador desde el código fuente. Los recursos engloban elementos tales como: ficheros, imágenes, sonidos, plantillas, botones, estilos, literales, etc.

Para que cada recurso se identifique por un número, Android crea un atributo de tipo entero en la clase R (con un valor único y arbitrario), pero con el nombre que yo he definido. El valor final del número, generalmente largo y extraño, no me importa, pues pese a que ese número será el id del recurso en cuestión en mi aplicación, precisamente gracias a este artificio podré acceder a él por su nombre.

A continuación, unos ejemplos de lo comentado:

fichero	tipo int
res/layout/fila_listado.xml	→ R.layout.fila_listado
res/raw/sonido_aviso.mp3	→ R.raw.sonido_aviso
res/drawable/icono_notificacion.jpeg	→ R.drawable.icono_notificacion

atributo/elemento	tipo int
<Button android:id="@+id/boton_aceptar"></Button>	→ R.id.boton_aceptar
<string name="saludo">¡BIENVENIDO!</string>	→ R.string.saludo
<color name="color_gris">#949494</color>	→ R.color.color_gris

### ① IMPORTANTE

Es de vital importancia respetar la nomenclatura, de modo que todos los archivos definidos bajo la carpeta /res no contengan mayúsculas ni espacios. Siempre deberán emplearse nombres en minúscula concatenados opcionalmente por guiones bajos. En caso contrario, la generación de la clase R resultará fallida.

Al ser esta clase generada automáticamente por el entorno, es relativamente frecuente que topemos con fallos causados por su cálculo, del tipo "no existe la clase R" o "no se pudo encontrar tal o cual identificador". Cientos de comentarios en foros de Internet denuncian el fallo y aportan soluciones, que van desde reiniciar el IDE, hasta forzar la recopilación del proyecto. No hay una fórmula mágica y deberemos probar con las alternativas encontradas navegando en búsqueda de ayuda.

## 1.10 DEPURACIÓN

---

Depurar un programa es ejecutarlo paso a paso, pudiendo observar los cambios en los estados de las variables y siguiendo el flujo de la ejecución al detalle. Por supuesto, el IDE de Android Studio, como entorno moderno, incorpora funcionalidades que nos ayudan a realizar la depuración de nuestro código. Y ello es, junto con el Log, las herramientas fundamentales que tenemos a la hora de localizar fallos y subsanarlos.

Por muy súper programador que sea uno, depurar el código será una tarea inevitable y consustancial al desarrollo. Dedicaremos a ello un vídeo demostrativo, por ser una habilidad eminentemente práctica.

## 1.11 EL REGISTRO O LOG

---

El registro de las acciones que se van acometiendo durante la ejecución de una aplicación es una necesidad inherente a cualquier sistema, por sencillo o sofisticado que sea. Es como la caja negra de un avión, o las migas en el cuento de Pulgarcito. Necesito un rastro, un registro, en inglés un *log*, que me ayude a visualizar qué acciones se han llevado a cabo, cuáles no y si se ha producido alguna incidencia o error durante la marcha.

Pues bien, para satisfacer esa funcionalidad, los creadores de Android nos ofrecieren en el SDK la clase Log del paquete android.util. Se recomienda encarecidamente su uso, especialmente en las partes funcionalmente importantes o susceptibles de error, pues aporta legibilidad y sostenibilidad. Aspectos claves en la calidad de una aplicación.

Sólo es necesario invocar a uno de los métodos estáticos definidos, previstos según la importancia o criticidad del mensaje a registrar. Su uso es sencillo y se ilustra a continuación, yendo de menos a más prioridad:

---

```
Log.v (String etiqueta, String mensaje) para mensajes con importancia VERBOSE  
Log.d (String etiqueta, String mensaje) para mensajes con importancia DEBUG  
Log.i (String etiqueta, String mensaje) para mensajes con importancia INFO  
Log.e (String etiqueta, String mensaje) para mensajes con importancia ERROR  
Log.e (String etiqueta, String mensaje, Throwable error)
```

---

El parámetro *etiqueta* es un texto que acompañará al mensaje que queremos registrar y que se recomienda declarar como constante para ser usada a lo largo del código. Nos servirá para identificar los mensajes propios de nuestra *app*, al margen de los cientos de mensajes de *log* vertidos por otras aplicaciones en el dispositivo.

```
public static final String MI_ETIQUETA_LOG = "APP_VAL";
```

El segundo parámetro, es el mensaje que queremos registrar. Un ejemplo sería: "Aplicación iniciada".

En el método e, el tercer parámetro viene a representar el fallo o error producidos en caso de excepción y describe el propio suceso anómalo.

### NOTA

Hay dispositivos en los que puede no funcionar un método. Recuerdo el caso de un alumno con un Sony, que se volvía loco porque no le iba el método para nivel DEBUG Log.d(). En su caso, la solución fue usar el nivel superior INFO Log.i() como alternativa.

## 1.12 TEST TEMA 1

1. El SDK de Android es:
  - a) Una clase Java
  - b) El IDE usado para programar en Android
  - c) El conjunto de herramientas y código fuente básico para el inicio de una app Android
  - d) Hay 5 versiones del SDK
2. Marque las afirmaciones correctas sobre una API:
  - a) Es el acrónimo de Application Production Interface
  - b) Como programador, son todas las clases y métodos que me son dadas
  - c) Por lo general, una API es sólo el título de una versión
  - d) Diferentes versiones del SDK proporcionan idénticas APIs

3. Marque la afirmación correcta sobre un IDE :
  - a) El IDE oficial para Android es ECLIPSE
  - b) Aprender a manejar el IDE me da productividad
  - c) Es el acrónimo de Investigation Development Envioroment
  - d) Android Studio ha alcanzado una buena estabilidad desde la versión 1
4. Marque las afirmaciones correctas respecto de Gradle:
  - a) Es el Maven de Android
  - b) Permite gestionar las dependencias de mi proyecto
  - c) Gestiona el ciclo de vida mi aplicación: compilación, pruebas, empaquetado, etc.
  - d) Todas las anteriores son ciertas
5. Hay varios atributos de interés en los ficheros de configuración. El applicationId, indica:
  - a) El id de la versión de compilación de mi proyecto
  - b) Es el id usado por Google en la tienda de aplicaciones
  - c) Da nombre a mi aplicación
  - d) No existe ese atributo en un proyecto Android
6. La extensión apk, hace referencia a:
  - a) Es la extensión de una clase de Java
  - b) Es la extensión del ejecutable Android
  - c) A la extensión de los ficheros de recursos
  - d) Todas las anteriores son ciertas
7. Respecto a la clase R, marque las afirmaciones correctas:
  - a) El marco de trabajo genera automáticamente esta clase
  - b) La clase se actualiza si incluyo o modiflico un archivo bajo la carpeta res
  - c) Es una facilidad del entorno para acceder a los ficheros de recursos
  - d) Todas las anteriores son correctas

- 
8. Sobre la depuración de aplicaciones Android:
    - a) Depurar el código es un paso previo para generar el ejecutable
    - b) No puedo depurar el código Android
    - c) Depurar el código me ayuda a identificar errores y entender su funcionalidad
    - d) Es totalmente prescindible aprender a depurar el código
  9. Sobre el registro de las actividades y procesos de mi aplicación:
    - a) Cuento con la clase Log de serie para registrar los eventos de mi aplicación
    - b) Para usar el Log, necesito definir una etiqueta, preferiblemente como una constante
    - c) Hay diversos niveles de Log, según la prioridad del mensaje que quiera registrar
    - d) Todas las anteriores son ciertas
  10. Respecto del fichero de Manifiesto, podemos afirmar:
    - a) Existe un fichero con extensión .class
    - b) Su ubicación viene predeterminada por Android
    - c) Es empleado para la traducción a distintos idiomas de mi aplicación
    - d) En él se declaran todos las constantes de mi aplicación

## SOLUCIONES

1c, 2b, 3b, 4d, 5b, 6b, 7d, 8c, 9d, 10b

# 2

---

## ACTIVIDADES I

Nos introducimos en las Actividades, elemento fundamental de Android. Entenderemos qué representan y cómo definirlas, así como sus principales atributos y características.

### 2.1 TAREAS PRÁCTICAS DEL TEMA 2

---

- ▶ Definir nuevas actividades
- ▶ Modelar estéticamente diseños sencillos
- ▶ Jugar con los aspectos básicos de tamaño y posición de los controles

### 2.2 ACTIVIDADES

---

Una Actividad es una clase que representa una pantalla. Se llaman Actividades puesto que heredan de Activity, una clase dada en el SDK.

Cuando programemos nuestras actividades, tenemos que realizar dos fases de la forma más independientemente y secuencial posible: la fase estática primero y la fase dinámica después. El método expresado a continuación parece sencillo y hasta obvio, pero por mi experiencia, resulta clave en la consecución del éxito para programar nuestra actividad. Y el no seguirlo, deriva generalmente en fracaso.

Esta forma procedimental nos ayudará a definir de manera completa qué queremos, la funcionalidad esperada y la forma de representarlo.

1. La fase estática: Realizar un diseño visual lo más detallado posible de los elementos de la pantalla. Qué botones, qué texto, qué menú e imágenes tendrá nuestra actividad; las opciones y cuál será su distribución en la pantalla.

Todo ello, se traducirá en el producto final de esta fase que es archivo XML de layout. Pero es muy recomendable que tomes papel y lápiz y hagas un croquis lo más detallado posible antes de tocar el ordenador. O un dibujo con cualquier programa informático.

Una vez tengas claro cuál quieras que sea la apariencia, la pasaremos al XML que Android entiende. Interesante en esta fase aportar datos ficticios (aunque aún no se disponga de ellos), para ver cómo quedan dibujados en la pantalla.

2. La fase dinámica: Dar vida al dibujo anterior, programando su funcionalidad asociada. Ello implicará programar nuestros métodos e instrucciones en la(s) clase(s) Java. Es decir, todas las acciones que se pueden realizar desde el interfaz gráfico definido en la fase anterior, tendrá su traducción en el código que ejecute las acciones necesarias.

## 2.3 CONCEPTO CLAVE CLASE

Al usar Java, que es un lenguaje orientado a objetos, nuestro proyecto estará compuesto por decenas de clases. Y, ¿qué es una clase, además del átomo de cualquier aplicación Android? Bien, la clase sirve para abstraer y representar esa parte del mundo real sobre la que versa nuestra *app*. De modo que no te olvide nunca y tenlo siempre presente: una clase puede ser definida únicamente por dos motivos:

- ▶ Bien porque representa algo, por su naturaleza. Es decir, podemos por ejemplo tener una clase de un equipo de fútbol, la clasificación o un jugador, si hacemos un programa sobre La Liga o la clase Ingrediente o Receta si hacemos una aplicación sobre cocina.
- ▶ Bien por su funcionalidad, porque vale para algo. Es decir, una clase que sirve para llevar a cabo una acción: enviar datos por Internet, guardar datos en la memoria, grabar un mensaje de voz, etc.

Importantísimo será entender el significado de las clases que nos iremos encontrando en el SDK y poder usarlas así a conveniencia.

## 2.4 CREACIÓN DE UNA ACTIVIDAD

Después de este inciso, vayamos a la *Activity*, que ya hemos dicho será la clase que represente una pantalla en nuestra aplicación. Tantas pantallas tenga, tantas actividades tendré definidas.

En realidad, cuando definimos una actividad, no sólo tenemos que definir una clase. Por tratarse de una clase especial para Android, se hacen tres cosas:

1. Definir una clase que herede de *AppCompatActivity*
2. Definir un elemento XML en el *manifest*
3. Definir un archivo XML llamado *layout*

Ya que la definición de una nueva clase implica estas tres acciones, es imperativo realizar esta acción mediante el asistente de Android Studio. Se puede hacer manualmente, pero no compensa.

El archivo definido en punto 3, que llamamos de *layout*, no es ni más ni menos que un archivo XML que representa los elementos visuales y la disposición (de ahí el término inglés *layout*) que tendrán en el pantalla. Los que tienen alguna noción de programación web, es como si fuera el html que representa una página web. Posteriormente, Android interpretará este archivo e irá dibujando en la pantalla cada elemento correspondiente.

El elemento en el archivo de manifiesto no es ni más ni menos que una declaración para que el motor de Android sepa que tiene una pantalla y qué clase es la que tendrá que instanciar llegado el momento.

Y la clase del punto 1, vendrá a ser la parte dinámica de la pantalla. Digamos, donde está programada la funcionalidad o las acciones que pueden derivarse de los elementos visuales del fichero de *layout*. Para los que saben de web, si el fichero de disposición es el HTML, la parte estática; la clase sería el JavaScritp que lo acompaña y la dota de funcionalidad.

## 2.5 MÉTODOS IMPORTANTES EN UNA ACTIVIDAD

---

En una clase, podremos definir todos los métodos que consideremos necesario; pero al ser una clase que hereda de otra, ya tenemos métodos heredados que nos son dados. Entre ellos, destacan:

### 2.5.1 **onCreate (Bundle bundle)**

Para nosotros, de momento, será como nuestro método *main*, es decir, por donde comienza a ejecutarse nuestra aplicación. En realidad, ya veremos que es sólo una etapa más ciclo de vida de una Actividad. Pero podemos anticipar que en

este método, se prepara lo que más tarde será visible por el usuario. Dejamos el parámetro *bundle* para su estudio posterior.

### 2.5.2 `findViewById(int id)`

Método básico que nos permite obtener un objeto visual de la pantalla en la que estamos. Lo buscaremos pasando al método el id o identificador del objeto. Este id, es un nombre que le damos al elemento en el XML y que la clase R convierte en un número automáticamente. Por ejemplo, si quiero obtener el mensaje que introdujo un usuario supongamos, para enviarlo; lo primero será obtener la caja de texto que contiene el mensaje mediante este método. Esa caja de texto estará definida de antemano y tendrá su id, que habremos definido en el XML.

### 2.5.3 `setContentView(int id)`

Método que siempre será invocado en el interior de `onCreate()`. Su cometido es cargar el archivo de *layout* que aparecerá dibujado en nuestra pantalla o *Activity*. Ese archivo de *layout*, será identificado por el parámetro id, también generado por la clase R, a partir de su nombre.

### 2.5.4 `finish()`

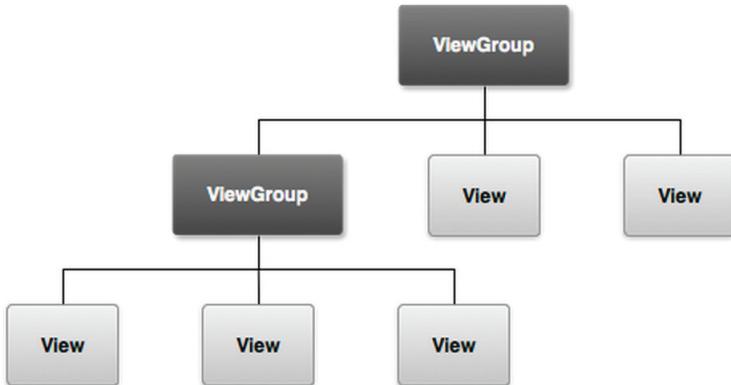
Este método finaliza la actividad, provocando que deje de verse la pantalla/actividad visible. Si es la última, se sale de la aplicación. Si no, tiene como un efecto de ir hacia atrás, pasando a ser visible la actividad anterior.

## 2.6 VIEWS Y VIEWGROUP

---

Imagina todos los objetos visuales que puedes encontrar en una pantalla: desde un botón, hasta una imagen, pasando por un menú. Pues bien, todos ellos, caen dentro de la categoría de vista o *View* en inglés. Y todos estos elementos visuales serán clases (*Button*, *ImageView*, *Menu*), que heredan de *View*.

Además, hay un tipo especial de vista, que son las que permiten agrupar o agregar dentro de ella a otras vistas. Imagina, por ejemplo, tu lista de contactos de WhatsApp. Cada ítem o elemento de la lista, representa un contacto, pero todos ellos, están englobados en un elemento de entidad superior que es la propia lista. Pues bien, esa lista es un *ViewGroup*, que no es sino un tipo particular de Vista.



Como ves, *Views* y *ViewGroups*, componen una estructura jerárquica donde puede entenderse que unos elementos <<hijos>> penden de otros <<padres>> que los agrupan y organizan. Esta relación quedará textualmente reflejada en un fichero XML de *layout* (que recuerda, será nuestra pantalla).

Aprenderemos a recorrer esta estructura de hijos, padres y hermanos en un video práctico.

## 2.7 ATRIBUTOS COMUNES DE UNA VISTA

---

Cada `ViewGroup` o `View`, tendrá sus propios campos exclusivos en función de su subtipo (botón, caja de texto, imagen, etc.) pero sí que como tipos concretos de Vista, comparten una serie de atributos comunes de especial relevancia que debemos conocer:

- ▀ **android:id** Identificador. Indispensable para poder referirse a la vista desde el código y distinguirla de las demás. El nombre que demos aquí tendrá su equivalente numérico en la clase R de modo automático.
- ▀ **android:layout\_width** Ancho. Cuál es la medida de ancho de un elemento.
- ▀ **android:layout\_height** Alto. Cuál es la medida de alto de un elemento.
- ▀ **android:gravity** Posición de un elemento (centrado, izquierda, derecha, etc.) respecto de sus márgenes.
- ▀ **android:layout\_gravity** Posición de un elemento (centrado, izquierda, derecha, etc.) respecto de su parente/elemento contenedor.
- ▀ **android:margin** Margen respecto de sus elementos circundantes (`marginBottom` –margen respecto del elemento inferior –, `marginTop` –

margen respecto del elemento superior –, marginLeft –margen respecto del elemento a la izquierda –, marginRight –margen respecto del elemento a la derecha –). Digamos este es el margen externo.

- ▶ **android:padding** Margen interno, respecto de sus límites. Igual que para el atributo anterior, hay opción de definir individualmente margen superior, inferior y laterales.

Al par de atributos **android:layout\_width** y **android:layout\_height** (ancho y alto respectivamente ) se les denomina en conjunto como **LayoutParams** y son obligatorios para toda vista

## 2.8 UNIDADES DE MEDIDA

A la hora de definir el tamaño de un elemento, su margen, ancho o alto debo usar la medida **dp**, que se escala automáticamente en función de la resolución del dispositivo. Como curiosidad dp proviene de pixels de densidad independiente.

Sólo para texto, debo emplear **sp**, que es como dp (adaptada a la resolución) pero además tiene en cuenta el tamaño de fuente elegido por el usuario en Ajustes (letra grande, muy grande, pequeña, etc.).

También puedo usar dos medidas relativas súper útiles para definir el tamaño de un control: **match\_parent** y **wrap\_content**. Si atribuyo la primera de ellas a un elemento, este ocupará todo el espacio disponible. Mientras que si uso la segunda, ocupara lo estrictamente necesario para representarse.

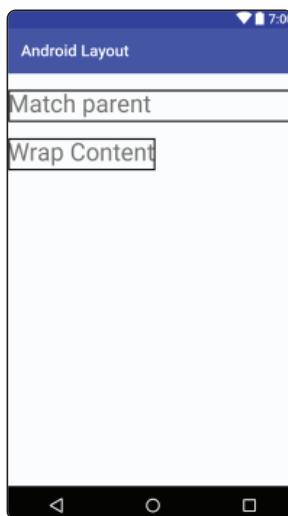


Figura 2.1. Ejemplo de un texto con valor de ancho match\_parent y wrap\_content.

## 2.9 TEST TEMA 2

---

1. Si mi aplicación tiene 4 pantallas distintas, tendré al menos:
  - a) 5 clases
  - b) 6 clases
  - c) 8 clases
  - d) 4 clases
2. Al definir una nueva Actividad:
  - a) Represento directamente los elementos XML
  - b) Pienso primero qué apariencia quiero darle, lejos del Android Studio
  - c) No es conveniente usar datos de prueba en la fase de diseño
  - d) Primero programo el comportamiento, luego la apariencia
3. Cuando creo una Actividad nueva en mi proyecto:
  - a) Creo un HTML que representa la pantalla
  - b) Tengo que añadirla al fichero Manifiesto, definir un clase Java y un XML asociado
  - c) No debo usar el asistente de Android Studio
  - d) Tan sólo dedo crear una clase de Java
4. Algunos métodos que aparecen predefinidos en mi Actividad:
  - a) Lo están por herencia y onCreate es uno de ellos
  - b) findViewById (string) es uno de ellos
  - c) Lo están por polimorfismo y onCreate es uno de ellos
  - d) Todas las anteriores son ciertas
5. Sobre la clase View, señale la correcta:
  - a) Una View es un caso especial de una ViewGroup
  - b) Una View es una ViewGroup porque puede albergar otras Views en su interior
  - c) Una ViewGroup es un caso especial de View
  - d) Un botón es un tipo particular de ViewGroup

- 
6. Los atributos indispensables al definir una vista son:
    - a) El alto y el ancho
    - b) El margen y el layout\_gravity
    - c) El alto, el ancho y el id
    - d) Ninguno es indispensable
  7. ¿Existe unidades de medida relativas en Android?
    - a) Sí, como fit\_parent y wrap\_content
    - b) No.
    - c) Sí, en pixels
    - d) Sí, como match\_parent y wrap\_content
  8. Se usa para definir el tamaño de la fuente:
    - a) Verdadero
    - b) Falso

## SOLUCIONES

1d, 2b, 3b, 4a, 5c, 6a, 7d, 8a

# 3

---

## VISTAS BÁSICAS

Toda Actividad va a llevar asociada un archivo de diseño o layout donde están declaradas las vistas que conforman el conjunto visual de la pantalla. Este archivo puede ser modificado e incluso creado sobre la marcha (como veremos en la sección de Inflar) aunque lo normal, es partir de una diseño definido en tiempo de compilación.

Veamos una primera tanda de los elementos visuales más comunes para poder ir construyendo nuestras primeras Actividades.

### 3.1 TAREAS PRÁCTICAS DEL TEMA 3

---

- ▶ Construir interfaces con controles básicos
- ▶ Combinar distintas distribuciones
- ▶ Incrustar una web en una actividad
- ▶ Gestionar la visibilidad de los controles y vistas
- ▶ Programar eventos

### 3.2 LAYOUTS BÁSICOS

---

Un *Layout* (plano o diseño, del inglés) es un elemento no visual dedicado a establecer la distribución y posición de los elementos que se definen en su interior. Si recuerdas la definición del tema anterior, efectivamente, un layout es un ViewGroup y por ende un caso particular de View, ya que agrupará un conjunto de controles declarados jerárquicamente como hijos suyos.

Pongamos un ejemplo de su utilidad. Supón un formulario, si yo quiero que aparezca una caja de texto y debajo un botón, usaré una distribución o layout vertical. Si quisiera que aparezca uno al lado del otro, englobaré a ambos elementos en un layout horizontal.

Demos un repaso a las distribuciones / layouts más relevantes:

### 3.2.1 LinearLayout

Los elementos en él definidos aparecen uno a continuación del otro. Bien en horizontal, bien en vertical, según el valor del atributo *orientation*. Es un clásico y combinándolo de forma anidada se puede hacer virguerías.

### 3.2.2 ScrollView

Usado para hacer que cuando el contenido de una actividad sobrepasa el tamaño de la pantalla, ésta deslice. Normalmente agrupa a otro layout y sólo puede tener un hijo directo como restricción. El desbordamiento puede ser horizontal o vertical.

### 3.2.3 FrameLayout

Sirve para superponer elementos. Los controles definidos en su interior aparecen dibujados uno sobre el otro, según el orden de aparición. Se puede usar para crear fondos, transparencias, etc.

### 3.2.4 RelativeLayout

Cuando sitúo un elemento, lo estaré haciendo respecto de su hermano o padre. La posición estará en función de otro elemento definido previamente (a su izquierda, a su derecha, a continuación). De ahí lo de *relative*.

### 3.2.5 ConstraintLayout

Parecido a la anterior, es la más moderna de las distribuciones. Está pensada para que pueda construirse la pantalla de modo gráfico, facilitando así el modelado a programadores noveles. Coloco los elementos arrastrándolos desde el menú. Es imprescindible definir al menos una restricción vertical y otra horizontal para cada elemento. Restricción toma aquí el sentido de distancia obligatoria a un borde de la pantalla u otro elemento.

Hay otras distribuciones que están demodé (TableLayout, GridLayout, ListView) al haber sido sustituidas por RecyclerView. Dicho layout, se emplea para mostrar colecciones de datos, como una lista de productos, contactos o cualquier otro tipo que se te ocurra. Dedicaremos a ella una sección en exclusiva más adelante.

Todas estas distribuciones o ViewGroups, albergarán en su interior controles o Views más sencillas. En las siguientes secciones veremos cómo definir algunas de las más comunes.

### 3.3 BOTONES

---

El elemento visual más icónico y empleado en la construcción de interfaces visuales, está representado por la clase Button. Esta clase, al igual que todos los controles básicos predefinidos, están agrupados en el paquete android.widget.

Hay varios tipos de botones, entre los que destacan:

- ▶ Button. El más sencillo, con texto en su interior
- ▶ ImageButton. En vez de texto, puedo definir una imagen en él
- ▶ Switch. Al estilo de un interruptor, apagado y encendido
- ▶ToggleButton. Alterna entre dos estados, que pueden ser textuales y gráficos

También tengo opción de crear mis propios botones a partir de personalizar cajas de texto e incluso, heredando de Button, definir botones redondeados o de otras formas geométricas.

Hay un caso especial de botón predefinido que queda al margen de esta sección de básicos y es botón flotante, *–Floating Action Button–* que veremos más adelante.

### 3.4 CAJAS DE TEXTO

---

Básicamente hay dos tipos de cajas de texto: las editables y las no editables. Están representadas respectivamente por:

- ▶ EditText que permiten que el usuario modifique el contenido textual de la caja
- ▶ TextView que muestran un texto fijo que no puede modificarse por el usuario

Con el atributo booleano *enabled* –habilitado– puedo jugar a habilitar o deshabilitar la edición de un *EditText*. Otro atributo interesante es *inputType*, que me permite preformatear la entrada en función del tipo de dato que desea albergar: fecha, número, teléfono, etc.

Jugaremos con estos atributos en algún formulario.

## 3.5 IMÁGENES

Para imágenes la clase prevista es *ImageView*. Los atributos *width* y *height* son el ancho y alto respectivamente. El *src* es el anagrama de *source* (fuente en inglés) y se refiere al nombre/ruta del fichero de imagen, albergado en la subcarpeta *drawable*. Al emplear este atributo, Android cargará automáticamente la imagen de la subcarpeta correspondiente, según la resolución del dispositivo (mdpi, ldpi, xhdpi, xxhdpi, etc.).

El uso de gráficos vectoriales mejora esta situación y permite que dado un único archivo de imagen, se escale sobre la marcha según la resolución, ahorrando así memoria. Sin embargo, el uso de formatos vectoriales está soportado sólo desde la versión 21. Para salvar esta circunstancia puedo usar el atributo *srcCompat*, que permitirá cargarlos en versiones anteriores, y además añadir el atributo *vectorDrawables.useSupportLibrary = true* en el fichero de gradle.

Android Studio incorpora dos herramientas para facilitarnos el trabajo con archivos de imagen: Vector Asset e Image Asset, que nos permiten importar gráficos vectoriales y no vectoriales y usar esas imágenes en nuestra app.

## 3.6 VÍDEO

Para insertar vídeo en una Actividad la clase prevista es *VideoView* y el archivo se tiene por costumbre de almacenar en la carpeta *res/raw*. Para definir un video que ocupe todo el espacio del padre, podría hacerlo así:

```
<VideoView  
    android:id="@+id/video_view"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" />
```

Para ponerlo en marcha, detenerlo y elegir la forma de reproducción, debemos jugar con las clases MediaController y MediaPlayer, que veremos en la sección práctica.

## 3.7 SELECTORES

---

Los controles previstos para permitir la selección del usuario son:

- ▶ CheckBox. La caja seleccionable
- ▶ RadioButton. El clásico botón redondeado que podemos marcar y desmarcar
- ▶ RadioGroup. Conjunto de RadioButtons
- ▶ Chips. Es lo último en diseño y son como las etiquetas temáticas de los blogs
- ▶ Spinner. Con él podemos hacer listas desplegables

Aunque todas estas clases son herederas de Button, las vemos aparte por ser más objetos que están pensados para darla la opción al usuario de elegir.

Terminaremos contando los trucos de cada uno en sus correspondientes videos.

## 3.8 PÁGINAS WEB

---

Android me permite visualizar una página web en una actividad, gracias a la vista WebView. El WebView va a ser como una ventana del navegador, pero sin la barra de navegación, ni el menú. En su interior, albergará una página web. Bien, almacenada localmente o bien remotamente (para este último caso, se deben tener permisos del uso de INTERNET en la aplicación).

El WebView se declara como cualquier vista en archivo de layout correspondiente:

---

```
<WebView  
    android:id="@+id/webView1"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content">
```

---

Y después se carga la página de manera dinámica en el código de la actividad, de este modo:

```
WebView wv = (WebView) findViewById(R.id.webView1);
wv.loadUrl("file:///android_asset/aviso_legal.html");
```

En este ejemplo, `aviso_legal.html` reside en el directorio `src/main/assets`, un sitio habitual para guardar archivos de configuración o diversos ejecutables. El término inglés `assets` significa activos, desde el punto de vista económico en su traducción al español. Entiéndase que es un directorio de archivos con diverso valor (aunque no del mismo tipo).

## 3.9 CALLBACK Y LISTENER

Hay dos conceptos tradicionales en la programación que toman vital relevancia en Android. Tal es así, que si los entendemos en profundidad, habremos ganado una batalla clave en la comprensión funcional del entorno.

La llamada por detrás o *callback* y el escuchador o *listener*, íntimamente relacionados y complementarios. Empecemos por el segundo.

Imagina que tocas un botón y se reproduce una canción, que tocas una flecha y se envía un mensaje, o que deslizas un elemento de una lista y desaparece. Seguro que son acciones con las que estás familiarizado como usuario.

Pues bien, por cada una de esas potenciales acciones, hay un proceso al acecho, que está esperando a que se produzcan para poder desencadenar las acciones asociadas. Ese proceso que espera, que está a la escucha de los gestos sobre un botón, imagen o cualquier otro elemento visual, es lo que entendemos como *listener*.

¿Y qué sucede cuándo se dispara el evento (tocaste el botón, deslizaste, le diste atrás, etc.)? Pues que se llevan a cabo las acciones previstas para él, ejecutándose la función o método que hayamos programado. Esa llamada a nuestro método es lo que llamamos *callback* o llamada por detrás. Y se llama así, ya que generalmente nuestra función será invocada desde Android.

Si nos elevamos un poco y generalizamos el funcionamiento de un dispositivo, podemos resumirlo como la secuencia reiterada de eventos y acciones asociadas (te hacen una llamada, suena el timbre, tocas el botón verde, contestas, tocas el botón rojo y cuelgas; recibes un mensaje, aparece un aviso, tocas un aviso y abres el mensaje, le das para atrás, y cierras etc.) Y toda esta cadena evento-acción, puede ser expresado en términos de *listener-callback*. Dos sustantivos que por sí solos, son casi una analogía del funcionamiento de cualquier aplicación y por tanto de Android.

En la sección práctica veremos las diversas formas de programar nuestros *listener* y nuestros *callback* sobre cualquier elemento y/o evento de nuestra app.

### 3.10 VISIBILIDAD: VISIBLE, INVISIBLE Y GONE

---

Un objeto visual en la pantalla puede transitar entre estos tres estados de visibilidad. Por defecto, todo elemento visual, al declararse en el fichero XML de *layout*, está en estado visible. Podemos hacer que desaparezca temporalmente (pasándolo a invisible) y podemos eliminarlo de la jerarquía, borrarlo del todo (pasándolo a gone).

Es una necesidad habitual en las aplicaciones, por lo que veremos el truco en la sección práctica.

### 3.11 TEST TEMA 3

---

1. Un Layout sirve para :
  - a) Definir el tema de una Actividad
  - b) Definir el color de fondo de una Actividad
  - c) Definir la distribución posicional de los controles que agrupa (horizontal, vertical, etc.)
  - d) Ninguna de las anteriores
2. Para conseguir controles superpuestos, debo emplear:
  - a) El LinearLayout
  - b) El FrameLayout
  - c) El ConstraintLayout
  - d) Ninguna de las anteriores
3. El último Layout que me permite construir la interfaz arrastrando con el ratón es:
  - a) El FrameLayout
  - b) El RelativeLayout
  - c) El ConstraintLayout
  - d) Ninguno de los anteriores

4. Para visualizar una web en Android:
  - a) Abrir el navegador
  - b) Usar un ScrollView
  - c) Definir una WebView
  - d) Ninguna de las anteriores
5. Puedo mezclar distintos tipos de selectores en una misma actividad:
  - a) Verdadero
  - b) Falso
6. Sobre Callbacks, marque la opción correcta:
  - a) Hay distintos tipos de Callback
  - b) Un Callback se da cuando desde mi código llamo al API de Android
  - c) Un Callback se da cuando una función programada por mí es invocada por un código que no he hecho yo
  - d) Ninguna de las anteriores
7. Sobre los listeners:
  - a) Hay distintos listener para distintos eventos
  - b) No puedo poner varios listeners al mismo objeto visual
  - c) Sólo puedo programar un listener declarativamente
  - d) Ninguna de las anteriores
8. Marque la opción verdadera:
  - a) Por defecto, la visibilidad de una Vista es invisible
  - b) Por defecto, la visibilidad de una Vista es gone
  - c) Si la vista tiene visibilidad gone, está en la estructura jerárquica, pero no se ve
  - d) Si la vista tiene visibilidad invisible, está en la estructura jerárquica, pero no se ve

## SOLUCIONES

1c, 2b, 3c, 4c, 5a, 6c, 7a, 8d

# 4

---

## ICONOS, ESTILOS Y TEMAS

Abordamos la creación de iconos para distintos uso de la aplicación: desde la creación del ícono principal, como los de menús y otros elementos. Se explora el uso de los gráficos vectoriales y su compatibilidad en distintas APIs. Comprenderemos la apariencia por defecto de nuestras aplicaciones y cómo podemos personalizarla por la extensión de temas y estilos dados.

### 4.1 TAREAS PRÁCTICAS DEL TEMA 4

---

- ▶ Crear el ícono de la aplicación
- ▶ Creación y uso de gráficos vectoriales
- ▶ Creación de íconos de menú y otros íconos
- ▶ Uso y personalización de temas
- ▶ Definición de estilos propios

### 4.2 ICONOS

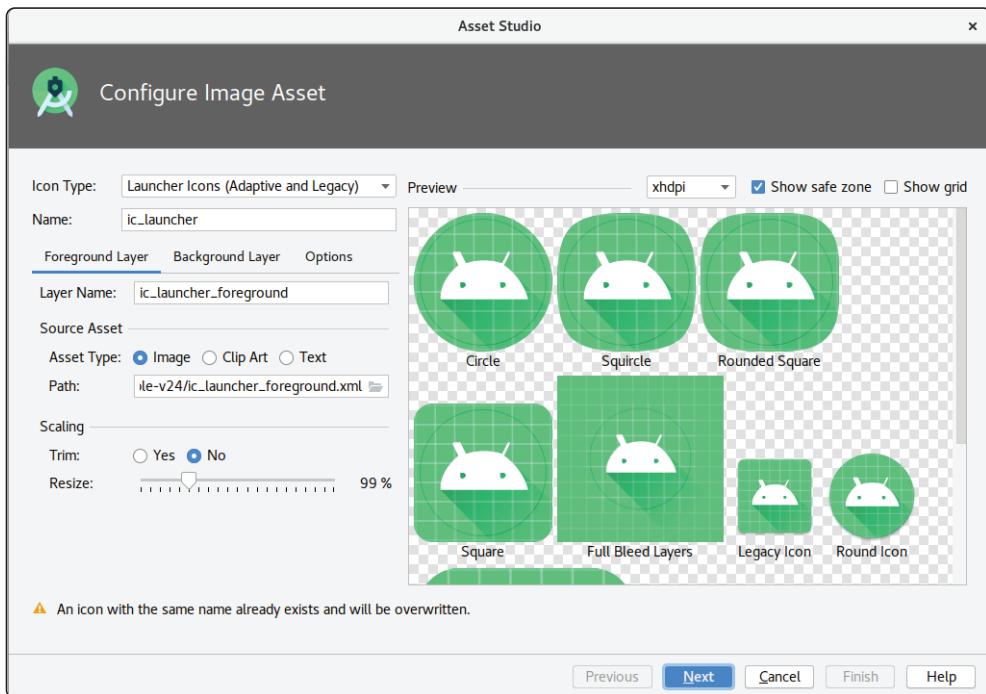
---

Gracias a las herramientas incorporadas en Android Studio es muy sencillo definir íconos para su uso en la aplicación, con un propósito diverso como:

- ▶ El ícono de la propia aplicación (*launcher icon* o de lanzamiento en inglés)
- ▶ Íconos para usarse en los menús
- ▶ Íconos para notificaciones

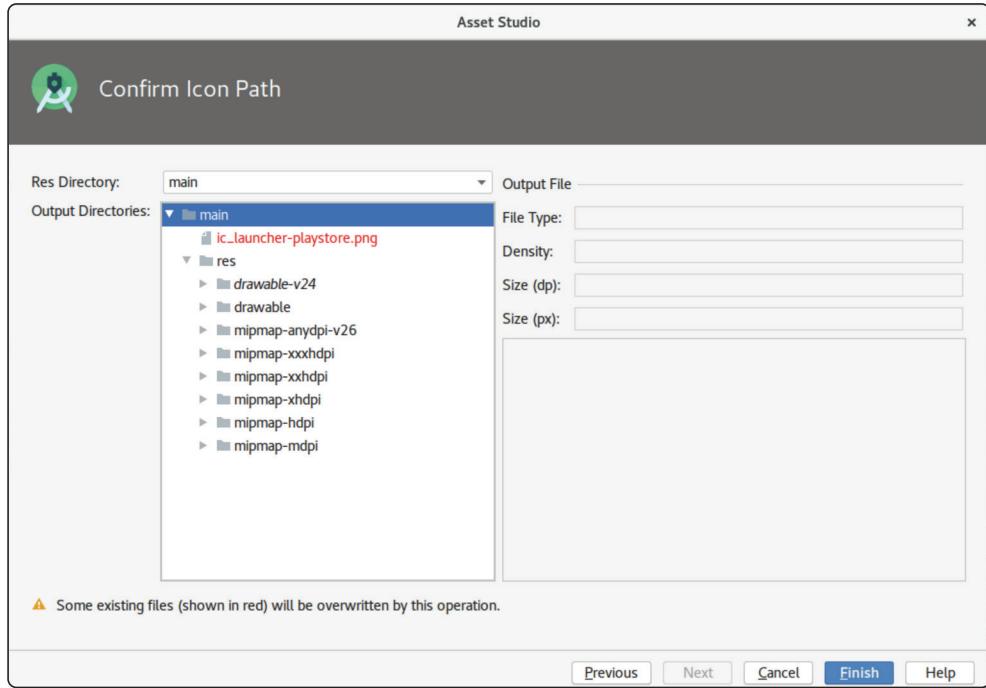
### 4.2.1 Iconos de la Aplicación y Menús

Las herramienta principal es Image Asset accesible desde File - New - Image Asset. Ahí podemos generar el ícono deseado partiendo de una imagen, una palabra o un ícono prediseñado llamado *Clip Art*.



El menú nos da la opción de generar el ícono para versiones antiguas o *legacy* y sólo modernas (a partir de la versión 8 Oreo) llamados iconos adaptativos. Estos últimos se definen con una capa superior (*foreground*) y otra de fondo (*background*) de modo que en distintos dispositivos, el mismo ícono puede verse de forma distinta y aparecer con brillos, recortado en rectángulo o con puntas redondeadas según el algoritmo particular de cada dispositivo.

Siguiendo el asistente podemos diseñar la imagen al gusto y la herramienta nos generará varias versiones en cada subcarpeta automáticamente:



La generada bajo main es la que nos exigen para cuando publicamos en la tienda, que tiene que ser un png de 512x512. La de los directorios mmap, se corresponden con los iconos de la propia app generados en todos las resoluciones y además la versión adaptativa (para versiones 8 o posteriores). Se genera en drawble la versión vectorial del ícono, por si se quiere usar en otra vista. Todo automáticamente, lo que ha simplificado la creación y el diseño de iconos respecto de las primeras versiones.

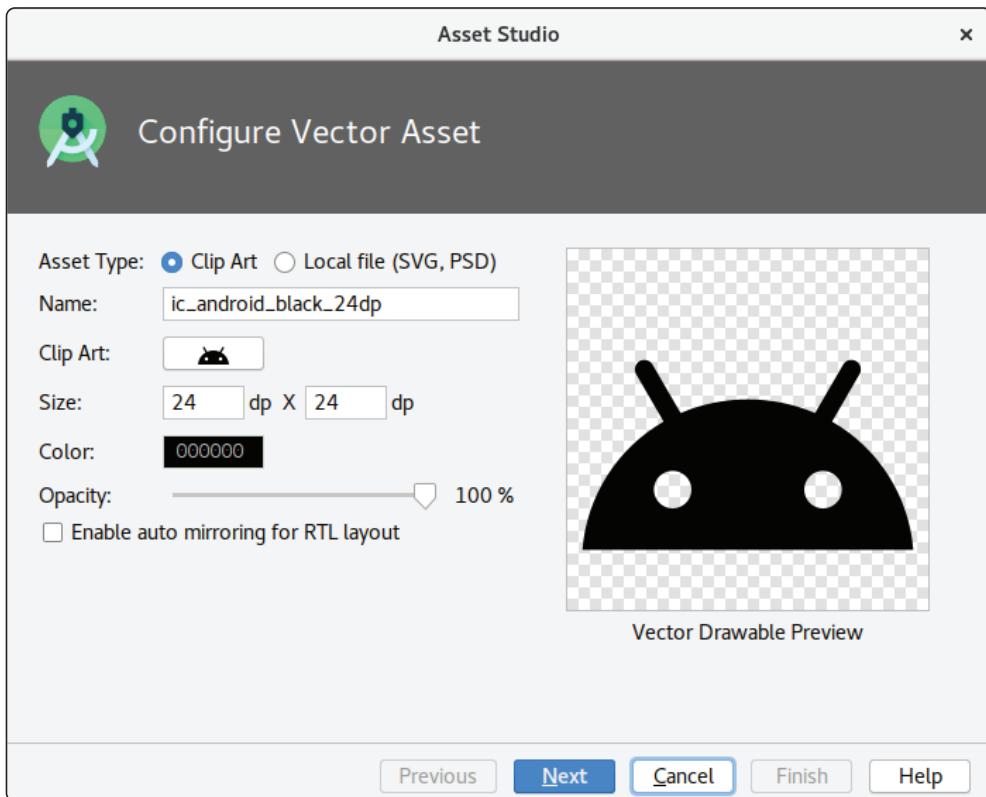
De manera idéntica puedes crear iconos para acompañar opciones del menú (seleccionando Action Bar and Tab Icons) y las notificaciones (Notification Icons) que generan respectivamente versiones en distintas resoluciones y gráfico vectorial.

## 4.2.2 Iconos Vectoriales

Hay otra herramienta llamada Vector Asset que permite importar gráficos vectoriales al proyecto.

Los gráficos vectoriales son archivos que a diferencia de otros formatos (png, jpg, bmp, etc.) se representan a escala automáticamente en el momento de dibujarse.

Se adaptan de forma dinámica a la resolución de la pantalla donde se muestran, por lo que no hay que preparar diferentes versiones más o menos densas de ellos.



Android recomienda el uso de vectoriales sencillos, de tamaño máximo 200 x 200 dp. Puesto que su representación en la pantalla requiere más tiempo de CPU que una imagen de trama o tradicional (jpeg).

El trabajo con vectoriales puede ser algo problemático, porque es una característica introducida en el API 21 –Lollipop 5– en la clase `VectorDrawable` (antes sólo existía el `BitmapDrawable`). Este asunto está relacionado con la sección `ImageView` y le será dedicado un video en la correspondiente sección práctica donde veremos cómo usar gráficos vectoriales en versiones anteriores.

En general es una buena regla usar vectoriales para menú y notificaciones (iconos pequeños) y usar imágenes de trama para el resto (ícono de la aplicación y demás imágenes).

La documentación de Android oficial propone usar una nomenclatura según el tipo de ícono que estemos creando, recogida en la siguiente tabla:

TIPO	PREFIJO
Icono	ic
Icono de la aplicación	ic_launcher
Icono de menú	ic_menu
Icono de notificaciones	ic_stat_notify
Icono de pestañas	ic_tab
Icono de diálogo	ic_dialog

Es sólo una recomendación, pero la idea es buena (aunque no sea obligatorio). Recuerda que lo que sí es obligatorio es que el nombre de todos los archivos de recursos sean en minúscula y con guiones bajos.

#### 4.2.3 Iconos Material

Además de estos recursos, existe también un catálogo en línea de iconos (mantenidos en parte por Android) que permiten su uso libre y descarga tanto en formato vectorial (svg) como formato de trama (png). Son llamados material icons y pueden accederse desde la página <https://material.io/resources/icons/>

#### 4.2.4 Fuentes Iconográficas

Las fuentes iconográficas o *icon fonts* en inglés son un tipo de letra que por cada carácter va a dibujar un ícono o símbolo. Al contrario de los tipos de letras tradicionales donde cada símbolo está representado explícitamente, en las fuentes iconográficas se usa un código y mediante un algoritmo, ese código se traducirá en el dibujo de un ícono gráfico. En realidad cada símbolo/letra es un gráfico vectorial que queda asociado a un código (generalmente hexadecimal). Así, podemos tener un montón de dibujos en todas las resoluciones con el uso de una sola fuente.

Hay miles de fuentes iconográficas descargables desde multitud de páginas web. Tan variadas como alfabetos de animales, frutas u objetos de oficina.

Para su uso, sigue estos pasos:

1. Elige tu fuente y descarga el archivo a la carpeta src/main/assets de tu proyecto.

- 
2. Cárgala mediante la clase Typeface, que nos devuelve un objeto del mismo tipo

```
.....  
otf = Typeface.createFromAsset(context.getAssets(), "tufuente.ttf");  
.....
```

3. Asigna esa fuente al objeto TextView donde quieras usarlo

```
.....  
textView.setTypeface(objtypeFace);  
.....
```

4. Usa los códigos hexadecimales para dibujar el icono deseado.

Veremos un ejemplo en la parte práctica asociada.

### 4.3 ESTILOS Y TEMAS

---

Los estilos se refieren a propiedades estéticas aplicadas a toda vista o control. Por ejemplo, se pueden describir propiedades como el color, el margen interno, externo, el tipo de letra y agruparlas bajo un único estilo, que se comparta a lo largo de diferentes TextView.

Los estilos se declaran con la etiqueta <style> en un archivo de recursos /res/values/styles.xml. Por ejemplo:

```
.....  
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
    <style name="CodeFont" parent="@android:style/TextAppearance.Medium">  
        <item name="android:layout_width">fill_parent</item>  
        <item name="android:layout_height">wrap_content</item>  
        <item name="android:textColor">#00FF00</item>  
        <item name="android:typeface">monospace</item>  
    </style>  
</resources>  
.....
```

Así definimos el estilo CodeFont y podemos usarlo después, referenciándolo así:

```
.....  
<TextView  
    style="@style/CodeFont"  
    android:text="@string/hello" />  
.....
```

El atributo *parent* usado en el elemento style es opcional. En caso de usarse, indica que nuestro estilo tomará todos los valores de ese estilo "padre", salvo que sean redefinidos en los elementos internos <item> propios.

Con el elemento <item> se declara el atributo y el valor que quiere aplicársele, que puede ser una medida, color o tipo de letra o cualquier atributo susceptible de ser parametrizado. Dependerá de qué objeto estemos tratando de estilizar (TextView, Butha, etc.) para saber qué atributos podemos personalizar. Y para saberlo, no nos queda más remedio que ir al API de Android y consultar los atributos XML que viene definidos en la sección *XML attributes*.

Como apunte práctico te diré que no emplees medidas fijas como pixels. Usa siempre relativas como dp o sp para objetos o fuentes respectivamente. Lo que en una pantalla puede quedar bien, en otro móvil puede ser de un tamaño inapropiado o hasta ilegible.

En lugar de asignar un valor explícito definido por mí, también puedo emplear un color o una medida predefinidas con el prefijo ? seguido del nombre del atributo ya definido en un tema concreto o por defecto. Por ejemplo, con:

- ▶ **?colorError** usamos el valor del atributo colorError definido en el tema en uso
- ▶ **?android:colorBackground** usaría el valor del atributo predefinido en Android

### 4.3.1 Temas

Los temas o *Theme* en inglés son una recopilación de estilos también, pero que aplican más que a objetos concretos, a la misma ventana o actividad en sí. Como por ejemplo, si existe o no la barra de menú, qué colores tiene esa barra, cuál es su ancho, si es visible la barra de estado (donde aparece la cobertura y la hora). Todo eso se define con un tema (normalmente ya predefinido) y nuestra app siempre funciona con un tema por defecto para todas las actividades, referido en el *manifest*.

---

```
<application
    ...
    android:theme="@style/AppTheme">
```

---

Un tema son muchos valores y estilos predefinidos, por lo que sí quiero definir uno propio, lo más cómodo es que "herede" de uno ya definido y los personalice a mi gusto, como por ejemplo:

```
<style name="NoActionBar" parent="AppTheme">
    <item name="windowActionBar">false</item>
    <item name="windowNoTitle">true</item>
</style>
```

Ahí defino mi tema NoActionBar, que hereda del AppTheme, y estaría haciendo que una actividad que use este tema no tenga barra de acción ni título. Normalmente, son todos los atributos que empiezan por window los aplicables a un tema.

Ahora para una Actividad en concreto quiero aplicar ese tema, puedo hacerlo con el mismo atributo theme a nivel de actividad:

```
<activity android:name=".actividades.main.MainActivity"
    ...
    android:theme="NoActionBar">
```

Los estilos y temas predefinidos, así como el diseño de los controles y elementos visuales están muy trabajados por el equipo de Android. Por tanto, salvo que quieras algo muy específico, no hay que tocar mucho los estilos por defecto para lograr una aplicación bien aparente.

#### 4.4 TEST TEMA 4

1. Android Studio incorpora una herramienta para la creación de iconos en la aplicación:
  - a) Verdadero
  - b) Falso
2. Respecto al ícono de la aplicación, marque la respuesta correcta:
  - a) Es un único archivo en una gran resolución
  - b) Hubo una redefinición de los iconos desde la versión 7
  - c) La carpeta destino del ícono de la aplicación es res/drawable
  - d) La carpeta destino del ícono de la aplicación es res/mimap

3. Respecto de los gráficos vectoriales, marque la respuesta incorrecta:
  - a) Los gráficos vectoriales son archivos que se escalan automáticamente al representarse
  - b) Están soportados desde la versión 21, Lollipop
  - c) Ahorran tamaño en la aplicación, pero se pierde tiempo de CPU para dibujarlos
  - d) Se recomienda usar siempre gráficos vectoriales
4. Marque la correcta, respecto de la nomenclatura:
  - a) Android obliga a usar un prefijo en función del tipo de ícono
  - b) Es obligatorio que el nombre de los iconos sea en minúscula y sin espacios
  - c) Debo guardar los iconos de notificaciones en carpetas distintas a los de menú
5. Android pone a disposición de los usuarios un conjunto de íconos de libre uso, dentro de su proyecto Material:
  - a) Falso, son de pago
  - b) Verdadero, pero hay que referenciar a Android en los créditos
  - c) Verdadero, simplemente pudo descargarlos desde internet en el formato vectorial o png
  - d) Verdadero, puedo descargarlos en formato jpg o png
6. ¿Puedo usar otras fuentes en Android cargando ficheros ttf mediante la clase TrueType?
  - a) Android no admite la carga de fuentes externas
  - b) Verdadero
  - c) Incorrecto, es mediante la clase Typeface
7. Las aplicaciones de Android usan un tema predefinido por defecto:
  - a) Verdadero
  - b) Falso

- 
8. ¿Puedo modificar el tema de una sola actividad?
    - a) Sí
    - b) No
  9. Al definir estilos propios, puedo partir de uno existente, heredando de él con el atributo parent:
    - a) Verdadero
    - b) Falso
  10. Marque la opción correcta sobre estilos y temas:
    - a) Es indispensable la creación de nuevos estilos en toda aplicación que se precie
    - b) Se recomienda usar medidas absolutas en la definición de estilos, como pixels
    - c) Salvo si deseas algo muy específico, no hay que tocar demasiado los estilos y temas por defecto

## SOLUCIONES

1a, 2d, 3d, 4b, 5c, 6c, 7a, 8a, 9a, 10c

# 5

---

## ACTIVIDADES II

Damos un paso importante en comprender el marco de trabajo Android y su dinámica provista en la ejecución de Actividades. Profundizamos en el comportamiento de este básico componente. Además vemos dos aspectos útiles: preparar nuestra aplicación para el funcionamiento en distintos idiomas y cómo asociar cualquier información a cualquier vista.

---

### 5.1 TAREAS PRÁCTICAS DEL TEMA 5

---

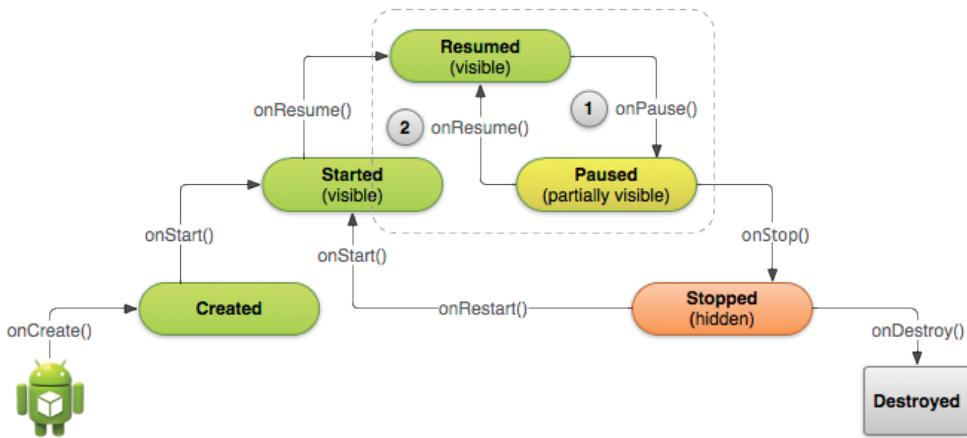
- ▶ Sobrescribir métodos que afectan al ciclo de vida de la actividad
- ▶ Gestionar cambios de orientación del dispositivo
- ▶ Aprender a programar el botón de hacia atrás
- ▶ Almacenar información en una vista
- ▶ Hacer nuestra aplicación disponible en otros idiomas

---

### 5.2 CICLO DE VIDA DE UNA ACTIVIDAD

---

Una actividad evoluciona desde que va a ser dibujada en la pantalla hasta que deja de ser visible. En ese tránsito, pasa por un conjunto de estados bien definidos, que se corresponden con la ejecución de una serie de métodos. Estos métodos son invocados de modo secuencial y automático por Android.



Cada método aporta una funcionalidad implementada ya en la clase padre. En caso de que necesitemos modificar o ampliar el comportamiento de estos métodos, podemos sobrescribirlos en nuestra actividad, añadiendo en ellos el código que queramos.

Supón por ejemplo un formulario. El usuario entra en nuestra actividad, rellena parcialmente el formulario y abandona. Por defecto, los datos introducidos se perderán, ya que al salir, el método que se ejecuta de modo automático es `onStop()` y este no preserva la información introducida. Seremos nosotros como programadores, los que sobrescribiendo el método `onStop()` en nuestra clase, podemos añadir las instrucciones con las almacenar los datos que deseamos preservar.

### 5.3 ESTADOS DE UNA ACTIVIDAD

<b>onCreate</b>	Inicializa. Prepara lo que se va a ver. A modo de constructor (sin serlo, ojo)
<b>onStart</b>	La actividad se hace visible
<b>onResume</b>	La actividad está en primer plano
<b>onPause</b>	La actividad está apilada / parcialmente visible
<b>onStop</b>	Actividad no visible
<b>onDestroy</b>	Actividad finalizada

Por necesidades de memoria el Sistema puede cerrar de forma inesperada mi aplicación. En tal caso, el único método que tenemos garantía se va a ejecutar es `onStop`. Deberemos aprovechar la ejecución de este método para guardar la información que deseamos conservar a toda costa.

## 5.4 GIRANDO EL DISPOSITIVO

Cuando giras un dispositivo, la actividad pasa a representarse de modo horizontal a vertical o viceversa. Esto provoca que la actividad se destruya y se vuelva a crear, lo que en ocasiones puede conllevar una pérdida de Información. Ante este hecho, podemos seguir dos estrategias como programadores:

1. Estrategia insensible: Podemos evitar el comportamiento por defecto y que cuando el dispositivo sea girado, la actividad permanezca inmutable. Esto hace que nuestra actividad sólo pueda mostrarse en una orientación y simplifica la gestión de la información que manejamos en ella.
2. Estrategia sensible: Igual que definimos un layout para la actividad en la carpeta res/layout, podemos definir otro archivo de layout con el mismo nombre bajo el directorio res/layout-land. Automáticamente, si volteamos el dispositivo de vertical a horizontal, Android va a dibujar en la pantalla el archivo de layout-land (*landscape* o apaisado del inglés). Al volver a situarlo verticalmente, Android volverá a representar el archivo de layout bajo res/layout.

Esta alternativa es más laboriosa, ya que debemos prever dos ficheros de layout para una misma actividad, pero puede resultar más atractiva para el usuario. Además ya que estamos permitiendo que la actividad se destruya y se recreé, debemos preservar la información como explicamos en el siguiente punto.

### NOTA

Igual que Android puede alternar el layout en ejecución según la orientación del dispositivo, también puede cargar un layout u otro en función de las características de la pantalla. Por ejemplo, si deseo un layout específico para tabletas de 7 pulgadas o superior, puedo hacer una carpeta llamada res/layout-sw600dp

## 5.5 SALVAR EL ESTADO DE UNA ACTIVIDAD

Como hemos comentado, al girar el móvil, la actividad puede recrearse y con ello perder datos. Pues bien, antes de morir, la actividad nos da la opción de guardar lo que queramos invocando al método *onSaveInstanceState* (*Bundle* estado). Es decir, se producirá un *callback* sobre este método. Nosotros podemos sobreescribirlo y guardar en él la información deseada. Pero, y ¿dónde la guardamos? En el objeto estado provisto por parámetro. Ese objeto, de tipo *Bundle*, es como una especie de bolso o saco, donde puedo almacenar objetos.

Si nos fijamos en la cabecera completa del método *onCreate (Bundle estado)*, vemos que recibimos un objeto del mismo tipo. Pues bien, resulta que es justamente el mismo saco o bolso que nos pasaron al ejecutar *onSaveInstanceState*. De modo que para distinguir si traímos algo guardado o no, bastará con preguntar por si vale null o no al principio de *onCreate*.

```
protected void onCreate (Bundle estado)
{
    if (estado!=null)
    {
        //la actividad se ha recreado y se ha guardado información
    } else
    {
        //la actividad se crea por primera vez o no se ha guardado información
    }
}
```

Veremos el ejemplo práctico para aclarar todo este punto.

## 5.6 BOTÓN DE IR HACIA ATRÁS

Todos los teléfonos incorporan el botón de ir hacia atrás, que permite navegar a la pantalla anterior o salir de la aplicación en caso de estar en la pantalla inicial. Cuando eso ocurre, se produce una llamada por detrás al método *onBackPressed()*, que es ejecutado de forma automática.

Por tanto, si deseamos mostrar algún menú o llevar a cabo una acción concreta cuando el usuario pulsa este botón, deberemos sobrescribir el citado método en nuestra actividad.

También podemos ir hacia atrás, incluyendo una flecha en la barra superior de la actividad. Algo que se ha convertido en un estándar del diseño y veremos en el capítulo dedicado a los menús.

## 5.7 EL MÉTODO SETTAG () DE LA CLASE VIEW

El método *setTag (Object o)* está definido en la clase View. Por ello, queda definido para todas las clases que heredan de ella: imágenes, botones, cajas de texto, layouts, etc.

Su utilidad es tan potente que nos puede ser útil en multitud de ocasiones. Y es que, gracias a este método, vamos a poder asociar cualquier tipo de información a una vista. Imagina por ejemplo, saber si el usuario tocó un elemento o no, guardando para ello un valor booleano asociado a él, o cuantas veces lo tocó, usando para ello un valor int anexo, o almacenamos información de valor a una foto. En fin, una infinidad de usos.

## 5.8 INTERNACIONALIZACIÓN O I18N

---

La internacionalización de aplicaciones o dicho de forma abreviada i18n (sí, hay 18 letras entre la primera i y la última n), versa sobre cómo hacer nuestras aplicaciones adaptadas a varios idiomas, de modo que nuestro programa pueda mostrarse en diferentes lenguas, según la preferencia del usuario. Esta preferencia o idioma elegido, se denomina localización o l11n y aunque viene predefinida en el dispositivo según su región de venta prevista, en la sección de ajustes de cualquier Android, podemos seleccionar el idioma en el que queremos funcione nuestro dispositivo y por ende sus aplicaciones.

La facilidad que proporciona el entorno para adaptar nuestra app a distintos idiomas consiste en definir todos los literales objetos de traducción en el fichero res/values/strings.xml identificados mediante una clave y asociados a un valor. Por ejemplo, definamos el literal ‘saludo’:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="saludo">Hola</string>
</resources>
```

---

Así, tenemos la clave saludo asociada al valor Hola. Ahora vamos a definir el saludo en inglés. Usando el asistente, diremos que saludo se corresponde con el valor Hello y Android Studio por si solo nos genera un fichero llamado res/values-en/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="saludo">Hello</string>
</resources>
```

---

Fíjate que se crea un nuevo fichero para el idioma inglés. En otra carpeta. Así será para cada idioma.

Por último hagamos ahora uso del valor definido. Para ello, dibujamos un botón que use el texto del saludo.

```
<Button android:text="@string/saludo"></Button>
```

Observa que en vez de usar el valor Hola o Hello, estamos usando la clave "saludo" como texto del botón. Esta configuración dada, implica el siguiente comportamiento: si el idioma del dispositivo es inglés, en la pantalla veremos el botón con Hello, y si el idioma es cualquier otro, veremos un botón con Hola. Esta decisión, la realiza Android automáticamente en tiempo de ejecución, obteniendo el valor del fichero correspondiente según las preferencias del idioma.

Como ves, gracias al marco de trabajo, tenemos una solución sencilla a una problemática general como la traducción de aplicaciones.

Veremos un ejemplo en la correspondiente sección práctica.

## 5.9 INFLAR

Inflar es un verbo clave en la jerga de Android. *To inflate*, del inglés, significa pasar de la definición textual de un objeto visual a su representación gráfica y como objeto. En realidad, es una tarea tan común, que ya veremos que Android tiene una clase dedicada a este menester.

declaración textual

```
<button android:text="ACEPTAR"/>
```



INFLADO

ACEPTAR

representación  
gráfica -objeto button-

Piensa que nuestras pantallas, quedarán en principio definidas en tiempo de compilación en un archivo XML y cuando nuestra actividad entre en ejecución, serán interpretadas para dibujarse. Pues bien, ese transformación es justamente inflar.

Además de definir la pantalla en tiempo de compilación, podemos inflar sobre la marcha nuevos objetos y así modificar la apariencia de nuestra pantalla en tiempo de ejecución, añadiendo nuevos elementos XML al fichero de *layout*. Esto de generar la pantalla desde código es un poco más avanzado, pero nos ayudará a entender la potencia de inflar.

LayoutInflater es la clase prevista en el SDK para inflar y pasar así de un elemento XML a su equivalente objeto Java. Y dentro de esta clase, tenemos el método inflate, que ofrece tres versiones\* y siempre nos devuelve una vista (View):

- a) inflate (ID)
- b) inflate (ID, ViewGroup)
- c) inflate (ID, ViewGroup, boolean)

En la primera versión, obtenemos un objeto de un elemento xml usando su id, sin más. Ese objeto NO es visible aún, puesto que para que lo sea, debe incluirse en la jerarquía del layout.

En la segunda versión, se da el id del elemento que queremos inflar y el padre donde se acoplará esta vista hija recién creada. Además, de este ViewGroup, la nueva vista inflada hereda sus atributos de anchura y altura (LayoutParams).

En la última versión, se añade un boolean como tercer parámetro. Su significado en caso de ser verdadero es que la nueva vista pasa a integrarse como nuevo hijo del ViewGroup. Si se omite (como en la segunda) se da por verdadero. Si es falso, el ViewGroup no es su padre, sólo habrá heredado su estilo (ancho y alto) y será necesario después llamar a addView de ViewGroup para hacer efectiva la representación.

Aquí el ejemplo práctico.

\*hay más versiones del método, pero estas recogen la esencia del proceso inflar

## 5.10 TEST TEMA 5

1. ¿Qué métodos se ejecutan en el ciclo de vida de una Actividad?
  - a) onCreate, onStart, onResume, onPause, onStop, onDestroy, onRecreate
  - b) onCreate, onStart, onResume, onPause, onDestroy
  - c) onCreate, onStart, onResume, onPause, onStop, onDestroy
  - d) onInit, onStart, onResume, onPause, onStop, onDestroy
2. ¿Puedo programar el dispositivo de modo que la actividad no obedezca giros de orientación?
  - a) No, necesito un permiso especial por parte del usuario
  - b) No, no puedo hacer nada al respecto
  - c) Sí, pero solo para una actividad de mi aplicación
  - d) Todas las anteriores son falsas
3. onDestroy siempre es invocado:
  - a) Verdadero
  - b) Falso
4. Al cambiar la orientación del dispositivo, cierta información puede perderse. ¿Qué puedo hacer al respecto?
  - a) Puedo guardar todo lo que me interese en otra Actividad temporalmente
  - b) No tengo opción de guardar nada
  - c) Puedo sobrescribir el método onLoadInstanceState para guardar en el objeto Bundle lo que desee
  - d) Puedo sobrescribir el método onSaveInstanceState para guardar en el objeto Bundle lo que desee
5. ¿Qué debo de hacer para capturar el evento de <<botón hacia atrás pulsado>>?
  - a) Programar el método onReturn
  - b) Sobreescribir el método onRestart
  - c) Sobreescribir el método onBackPressed
  - d) Ninguna de las anteriores

6. Marque la afirmación correcta respecto al método setTag:
  - a) Es un método sólo disponible para Button
  - b) Está definido para todas las clases que heredan de View
  - c) Su uso es limitado a muy pocas circunstancias
  - d) Sólo puedo usarlo para asociar números enteros a una Vista
7. Respecto de la internacionalización de aplicaciones en Android:
  - a) Necesito definir las traducciones en un solo archivo
  - b) El usuario sólo puede ver la aplicación en un máximo de dos idiomas distintos
  - c) Android selecciona el idioma de la aplicación en base a los Ajustes del dispositivo
  - d) Android pregunta al usuario en qué idioma prefiere mostrar la aplicación al abrirla
8. Inflar es sinónimo de:
  - a) Crear un XML a partir de un objeto Java
  - b) Comprimir el código Android
  - c) Transformar la declaración textual de un objeto a su representación visual y en memoria
  - d) Todas las anteriores son ciertas

## SOLUCIONES

1c, 2d, 3c, 4d, 5c, 6b, 7c, 8c



# 6

---

## INTENTS

Los Intent son una de clases principales del SDK de Android, pues son el precursor de toda Actividad.

Intent puede traducirse en español como intento; y el nombre no está mal traído, pues precisamente, cuando creemos un Intent, estaremos queriendo intentar algo. Es decir, llevar a cabo algún tipo de acción: saltar a otra pantalla, usar la aplicación de la cámara, leer la información de un contacto, seleccionar una foto de la galería, etc.

Básicamente el Intent es una estructura de datos que contiene una descripción abstracta de una acción a realizar. Normalmente, lo usaremos para lanzar actividades, aunque podemos usarlo para iniciar otros componentes que ya veremos (servicios, receptores, etc.). La Actividad lanzada puede ser de mi aplicación o de otra aplicación instalada en el dispositivo.

---

### 6.1 TAREAS PRÁCTICAS DEL TEMA 6

---

- ▶ Cambiar de pantallas intercambiando información
- ▶ Definir actividades que puedan ejecutarse desde otra aplicación
- ▶ Serializar objetos con parcelable
- ▶ Crear subactividades

## CONSTRUCTORES

El constructor está sobrecargado, ofreciendo varias versiones:

- a) Intent()
- b) Intent(Intent o)
- c) Intent(String action)
- d) Intent(String action, Uri uri)
- e) Intent(Context packageContext, Class cls)
- f) Intent(String action, Uri uri, Context packageContext, Class cls)

Los parámetros que de forma opcional pueden usarse, son:

- La acción a realizar (String)
- Los datos (URI)
- Los datos extras (Bundle)
- El Componente (Class –Actividad, Servicio, Receptor–)
- El Contexto (Context)

Salvo el Contexto, que es un concepto más especial al que dedicaremos un capítulo aparte, el resto de los parámetros y su significado son abarcados en las secciones posteriores.

## 6.2 INTENTOS EXPLÍCITOS

Los intents implícitos son los que al crearse, utilizan de forma explícita el nombre la actividad a la que quieren transitar o ejecutar. Típicamente son creados por el constructor e) de la lista anterior, donde Class, el segundo parámetro, representa el nombre de la Actividad destino.

```
Intent intent = new Intent(this, Main2Activity.class);
startActivity (intent);
```

En el intent estoy diciendo algo como "estoy aquí –this– y quiero ir allí" –Main2Activity.class–. Ya ahondaremos en el significado del primer parámetro, de tipo Context. La segunda instrucción, sirve para lanzar el intent una vez preparado y en este caso, como lo que quiero lanzar es una nueva actividad, uso el método startActivity.

Es el ejemplo más sencillo de Intent y habitualmente lo usamos para transitar de una pantalla a otra dentro de nuestra aplicación.

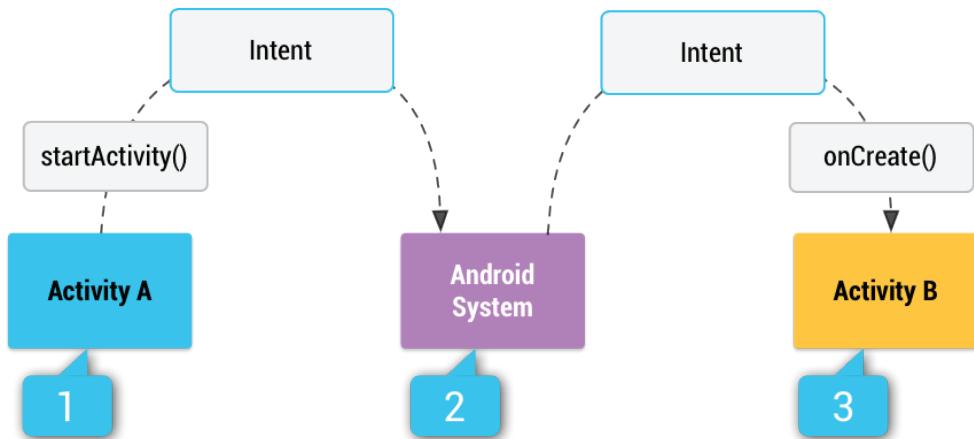
### 6.3 INTENTOS IMPLÍCITOS

Creamos un Intent implícito cuando expreso una acción del tipo "Quiero ver esta página" o "Quiero hacer una foto", pero no estoy indicando el nombre de la actividad concreta que quiero ejecutar (entre otras cosas porque no lo sé). Por ejemplo:

```
Intent i3 = new Intent(Intent.ACTION_VIEW, Uri.parse("http://google.com"));
startActivity (i3);
```

Mediante este constructor estoy diciendo quiero ver, una dirección web. Tras lanzarlo, Android verá qué aplicaciones de las instaladas en el dispositivo, pueden llevar a cabo esta acción; y si en su búsqueda da con varias candidatas, me saldrá un menú tipo de "elija la aplicación con que desea abrir este archivo".

El proceso de lanzamiento de un intent implícito queda recogido en el siguiente esquema:



Al no detallar el nombre de la actividad de forma explícita, Android va buscando las actividades candidatas a llevar a cabo mi petición entre las actividades de todas las aplicaciones instaladas en el dispositivo, filtrando las que concuerdan o casan con la descripción dada en el Intent.

Existe la posibilidad que en el paso 2 del diagrama anterior, Android no encuentre ninguna actividad que case con el intent implícito. En tal caso, Android acaba la búsqueda de vacío y en lanzamiento del intent provoca una excepción. Para evitar que esto pase, debemos siempre asegurarnos de que hay agua en la piscina, haciendo una sencilla comprobación:

```
.....  
    //SI HAY ALGUNA ACTIVIDAD QUE CASE CON EL INTENT  
    if (intent.resolveActivity(packageManager) != null)  
    {  
        startActivity(intent); //LANZO  
    } else {  
  
        //INFORMAMOS QUE NO HAY UNA APLICACIÓN EN EL DISPOSITIVO PARA LLEVAR  
        //A CABO LA ACCIÓN DESEADA  
    }  
.....
```

Hay miles de dispositivos Android y pudiera ser que no se encuentre una actividad candidata siempre. Por tanto, en pro de la robustez de nuestra aplicación, es siempre aconsejable hacer esta instrucción y evitar así que la aplicación se cierre de forma repentina.

En caso de haber varias candidatas, puedo personalizar el mensaje que invita a la selección al usuario usando el siguiente método estático.

```
.....  
Intent createChooser(Intent accion, String mensaje)  
.....
```

que nos devuelve un intent que ya podemos lanzar. Veremos en la sección práctica algún ejemplo.

## 6.4 INTENT FILTER

Cuando hablamos de filtrar en el punto anterior, estamos diciendo que Android selecciona una o varias candidatas. ¿Ahora, cómo lo hace? Pues atendiendo a una serie de atributos denominados Intent filter.

Los Intent filter, son elementos XML definidos en el interior de los componentes del Manifest y que de algún modo, describe las capacidades o potencialidades de la Actividad. Ejemplo:

```
.....  
    <activity android:name="ShareActivity">  
        <intent-filter>  
            <action android:name="android.intent.action.SEND"/>  
            <category android:name="android.intent.category.DEFAULT"/>  
            <data android:mimeType="text/plain"/>  
        </intent-filter>  
    </activity>  
.....
```

Si yo ahora monto un Intent donde digo que la Acción es un Enviar y el tipo texto plano, este Intent coincidirá con la Actividad anterior, de modo que se lanzaría o sería propuesta como candidata.

```
.....  
Intent intent = new Intent();  
intent.setAction (android.content.Intent.ACTION_SEND);  
intent.setType ("text/plain");  
startActivity (intent);  
.....
```

Este es un ejemplo sólo para que os hagáis la idea. Veremos un ejemplo completo en funcionamiento en la sección práctica.

Los Intent Filter pueden ser de tres subtipos: Acción, Datos y la Categoría. Acción y categoría son Stirings, mientras que los datos son expresados en forma de URI (Universal Resource Identifier) que ahora veremos.

- ▶ **ACCION.** Es un simple String, que normalmente adopta valores predefinidos como constantes en la clase Intent, como por ejemplo, ACTION\_VIEW, ACTION\_SEND, ACTION\_WEB\_SEARCH, ACTION\_VOICE\_COMMAND, etc. Hay una ingente cantidad de ellos. También puedo usar cualquier otro String que yo haya definido.
- ▶ **CATEGORÍA.** Es un atributo omitido la mayor parte de las veces pero digamos que es una descripción extra, que sirve para clasificar de manera más refinada la actividad. Por ejemplo, usariamos CATEGORY\_BROWSABLE si la actividad puede mostrar un vínculo o una imagen de Internet.
- ▶ **DATOS.** Los datos pueden expresar desde un tipo MIME en su versión más simple (para identificar con qué tipo de datos trabaja la actividad: texto plano, imagen, video, etc.) hasta una ruta concreta de un archivo o un recurso de Internet. Su forma completa definición es esta:

```
.....  
    <data android:scheme="string"  
        android:host="string"  
        android:port="string"  
        android:path="string"  
        android:pathPattern="string"  
        android:pathPrefix="string"  
        android:mimeType="string" />  
.....
```

Que puede quedar resumida en una URI por una cadena con este patrón:

```
.....  
cadena = <schema>://<host>:<port>[<path>|<pathPrefix>|<pathPattern>]  
.....
```

y después ser asignada al intent con el método setData (cadena).

Ejemplo de actividad e intent implícito que casarían:

```
.....  
<activity>  
    <intent-filter>  
        <action android:name="android.intent.action.VIEW"/>  
        <data android:scheme="http" />  
        <data android:scheme="https" />  
    </intent-filter>  
</activity>  
  
Intent intent = new Intent(Intent.ACTION_VIEW);  
intent.setData(Uri.parse("http://www.google.com"));  
activity.startActivity(intent);  
.....
```

## 6.5 EXPORTED

Una Actividad de mi aplicación ( o un servicio o un receptor, ya veremos esos componentes) podría en teoría, llegar a ser ejecutada por intento implícito lanzado desde otra aplicación si concuerda con él.

Pues bien, como dueño del componente, puedo hacer que esta actividad sea pública y por tanto candidata a ser ejecutada por un tercero, o bien puedo hacerla privada, de modo que su uso quede reservado en exclusiva a los límites de mi aplicación.

Para configurar este comportamiento, debo indicar de forma explícita el valor booleano del atributo exported de la Actividad en el fichero de Manifiesto. Si lo asigno a true, indico que mi actividad es pública y si le asigno false, indico que mi actividad es privada.

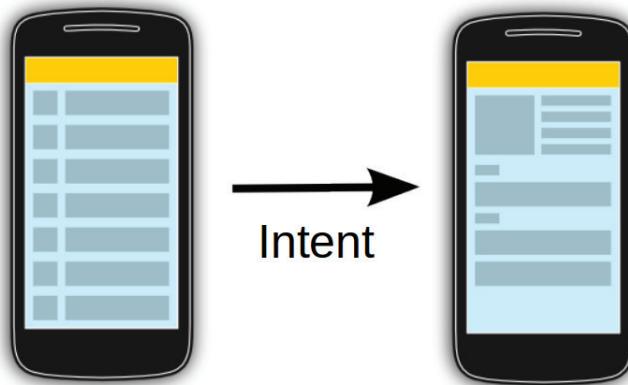
```
.....  
<activity  
    ...  
        android:exported="false"  
    </activity>  
.....
```

En el ejemplo previo, esa actividad no podría lanzarse nunca desde el exterior de mi aplicación.

## 6.6 BUNDLE EXTRAS

---

Cuando hablamos de lanzar un Intent, ello significa que se inicia otro proceso, otra actividad. Y esa Actividad, puede necesitar algunos datos o información de entrada. ¿Cómo transmitírselo?



En realidad es un escenario muy común. Piensa por ejemplo: estás visualizando la lista de contactos y seleccionas uno para editarlo. Pasas de una ventana (la lista) a otra (la edición de un contacto en concreto). De algún modo, la actividad de la lista de contactos debe informar a la de edición de cuál es el contacto seleccionado. Este escenario se repite en decenas de aplicaciones.

Para transitar de una actividad a otra, se lanzará un intent. Pues bien, ese Intent, actúa como un almacén temporal, una especie de saco o bolso, en donde la actividad origen, introduce la información que quiere pasar a la actividad destino, y ésta, podrá a su vez leer o recuperar de ese saquito, la información introducida por la actividad precedente.

```
intent.putExtra(Intent.EXTRA_TEXT, textMessage); //lanzo en la Actividad Origen  
...  
Bundle bundle = getIntent().getExtras(); //recojo en la Actividad Destino
```

Veremos un ejemplo práctico en la sección correspondiente.

## 6.7 PARCELABLE Y SERIALIZABLE

¿Qué puedo escribir en ese Intent o saco de almacenamiento temporal y qué no? Por defecto, puedo guardar tipos primitivos como int, boolean u objetos String. Pero si quiero guardar por ejemplo, una instancia de una clase propia como pueda ser Persona, deberé asegurarme de que esta se convierta bien en un tipo Parcelable o Serializable.

Para los Javeros de pro, sólo informar que Parcelable, es una versión mejorada de Serializable, particularmente optimizada para Android.

Veremos un ejemplo práctico en detalle.

## 6.8 INTENTS COMUNES

Al hablar de Intents Comunes nos referimos a un tipo especial de intento implícito, que permite la ejecución de servicios y actividades de aplicaciones que vienen de serie en los dispositivos Android.

Por ejemplo, para poner una alarma, seleccionar un contacto, hacer una llamada, echar una foto, o enviar un SMS hay que preparar un intent tal y como nos indica la documentación y simplemente lanzarlo, para que se inicie la actividad deseada.

En la documentación podemos encontrar la lista actualizada de intentos comunes: <https://developer.android.com/guide/components/intents-common?hl=es-419>

En caso de la actividad lanzada con uno de estos intents comunes, recupere o devuelva alguna información o resultado útil (una foto seleccionada, la información de un contacto, etc.) debemos igualmente atender a las instrucciones de la documentación para obtenerlos.

## 6.9 SUBACTIVIDADES

---

Cuando hablamos de subactividades queremos referirnos a Actividades que ofrecen resultados de vuelta a las mismas desde donde son invocadas/lanzadas. En realidad, el apartado anterior de Intents Comunes, son mayormente un caso de subactividades. Lo que pasa es que aquí vamos a ver cómo se programa por dentro y tendremos opción de hacer nuestras propias subactividades.

El proceso es el siguiente:

1. Desde la Actividad A, se lanza la Actividad B, de la que esperamos un resultado de vuelta en A. El pseudocódigo sería así:

//Preparo un intent en la Actividad A

---

```
Intent i = new Intent(this, ActividadB.class); //podría ser implícito
int cod_peticion = 237; //en el párrafo siguiente explicamos este valor
startActivityForResult (intent, cod_peticion); //lo lanzo
```

---

Fíjate que lanzamos el intent con el método startActivityForResult (a diferencia del startActivity) y pasamos dos parámetros en vez de uno.

El primero es el intent, que como ya sabemos, identifica la acción a realizar. Y el otro, es un número entero. ¿Y para qué vale ese número?, te preguntarás. Pues bien, piensa que la a ActividadB la pueden invocar distintas actividades en distintos momentos Piensa además, que esa ActividadB, puede invocarse repetidas veces antes incluso de que haya acabado la primera petición que le llegó. Dado el caso, necesitamos de alguna forma "etiquetar" la conversación entre A y B, para poder identificar que es "nuestra petición", la que nosotros lanzamos desde A y no otra. Esa es la función del código numérico. En el apartado 3, a la vuelta, lo terminarás de entender.

- 
2. La Actividad B, realiza su función y guarda un resultado en otro Intent al salir.
- 

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    //la actividad realizar su cometido y devuelve un resultado, y  
    //supongamos devuelve un objeto que llamamos objetoResultado en  
    //un intent -debe ser Parecelable o Serializable, recuerda-  
  
    Intent i = new Intent ();  
    i.putExtra("RESULTADO", objetoResultado);  
  
    //ahora usamos el método setResult para almacenar el resultado  
    //y le damos el intent creado  
    //además hay un primer parámetro que es un entero usado a modo de  
    //indicador. Ejemplo con un 1, diremos que la actividad finalizó  
    //de forma existosa o con un 0 que la cosa fue mal  
    this.setResult(1, i);  
  
    //asignado el resultado, concluimos la actividad  
    this.finish();  
}
```

---

3. El flujo de ejecución vuelve a A, que puede acceder a los resultados obtenidos por B. Aquí fíjate que está preestablecido el callback al método onActivityResult, cuya cabecera es la siguiente:
- 

```
@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
    //TODO accedemos al resultado de la Actividad B  
}
```

---

El significado de los parámetros es:

- ▶ **requestCode:** ¿recuerdas que cuando lanzamos la petición con startActivityForResult dimos un entero? Pues aquí está de vuelta. Así sabemos que esta respuesta se corresponde exactamente con la petición que hicimos.

- ▶ **resultCode:** como podrás intuir, este número es el valor asignado en el primer parámetro de setResult en el apartado anterior. Con él podemos saber si la cosa fue bien, mal o regular. Hay algunas constantes predefinidas como RESULT\_OK o RESULT\_CANCELED que suelen usarse.
- ▶ **data:** el intent como tercer parámetro, es a su vez el intent usado como segundo parámetro de setResult, que alberga el objetoResultado devuelto por la ActividadB.

Veremos el ejemplo práctico correspondiente con detalle.

Programar subactividades no es lo más común del mundo, pero sí es conveniente cuando la actividad realiza un proceso útil para muchas otras actividades o aplicaciones. Piensa por ejemplo en seleccionar un contacto. ¿Cuántas aplicaciones necesitarán en un momento dado seleccionar un contacto de la agenda? Seguramente miles. Pues en vez de programarse mil veces, se reutiliza la selección de un contacto con este patrón.

## 6.10 TEST TEMA 6

---

1. ¿El constructor de Intent está sobrecargado?
  - a) No
  - b) No, el constructor es privado
  - c) Sí, hay dos versiones
  - d) Sí, hay hasta 6 versiones
2. Los parámetros del constructor de un Intent pueden ser:
  - a) El Contexto y la Clase/Actividad
  - b) La acción y la URI
  - c) Datos extras
  - d) La suma de las respuestas anteriores
3. Un Intent puede ser:
  - a) Implícito
  - b) Expreso
  - c) Implícito o Explícito
  - d) Declarativo o explícito

4. La diferencia entre un Intent explícito y otro implícito es:
  - a) Al contrario que el implícito describe sin ambigüedad la actividad que quiere lanzar
  - b) Al contrario que el implícito, el explícito describe ambiguamente la actividad que quiere lanzar
  - c) No hay diferencia
  - d) El implícito describe una acción general mientras que el explícito una acción concreta
5. Es posible que al lanzar un Intent implícito, no se ejecute ninguna actividad.
  - a) Verdadero
  - b) Falso
6. Es posible que al lanzar un Intent explícito, surjan varias aplicaciones candidatas a ejecutarse.
  - a) Verdadero
  - b) Falso
7. Los intent filter son:
  - a) Un conjunto de sentencias condicionales y repetitivas
  - b) Atributos de la clase View
  - c) Atributos de una actividad que describen su capacidad operativa
  - d) Elementos XML que describen la prioridad de ejecución de un componente
8. ¿Puedo usar un intent para intercambiar información entre actividades?
  - a) No es posible, ya que el Intent sólo sirve para cambiar de pantalla
  - b) Sí, ya que el intent puede ser usado también como un almacén temporal
  - c) Sólo es posible al lanzar un Intent implícito
  - d) Sólo es posible al lanzar un Intent explícito

9. Sobre Parcelable, marque la correcta.
  - a) Debo hacer que todas mis clases del modelo sean Parcelables
  - b) Android recomienda seguir usando Serializable antes que Parcelable
  - c) No puedo usar Serializable en Android
  - d) Todas las anteriores son falsas
10. Puedo hacer subactividades de modo que retornen un valor a la actividad desde que se lanzan.
  - a) Sí, es obligatorio desarrollar al menos dos subactividades en una aplicación Android
  - b) Sí, justo esa es la definición de Actividad subrogada
  - c) Sí, justo esa es la definición de Actividad dependiente
  - d) Todas las anteriores son falsas

## SOLUCIONES

1d, 2d, 3c, 4d, 5a, 6b, 7c, 8b, 9d, 10d



# 7

---

## MENÚS Y DIÁLOGOS

En este tema veremos dos elementos básicos para informar al usuario y guiarlo en su navegación por la aplicación: los menús y los cuadros de diálogo.

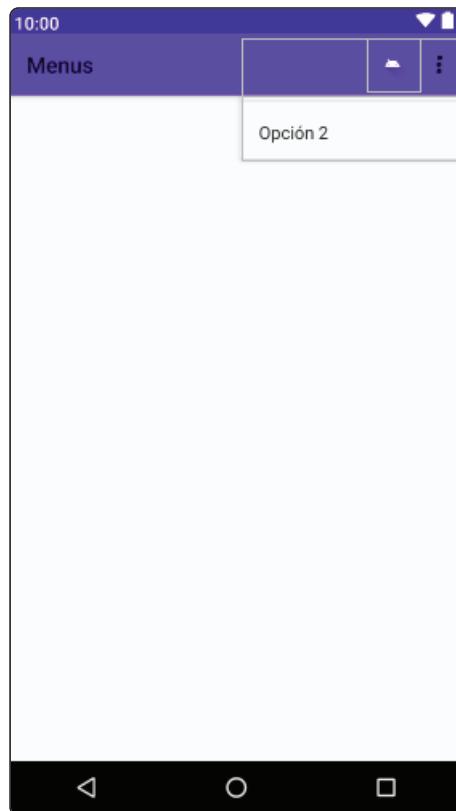
---

### 7.1 TAREAS PRÁCTICAS DEL TEMA 7

---

- ▶ Crear menús en la barra de una actividad
- ▶ Crear menús contextuales y emergentes
- ▶ Crear diálogos informativos y personalizarlos
- ▶ Creación de selectores de hora y fecha

## 7.2 MENÚS TEXTUALES



Nos referimos por menú textual al típico que nos sale en la barra superior de la pantalla o *AppBar*. Normalmente lo componen varias opciones que aparecen accesibles por un ícono (las más comunes) y otras que aparecen ocultas y se despliegan al tocar el botón punteado.

### 7.2.1 Definiendo el menú

Su definición es sencilla ya que realmente se hace tiempo de compilación y queda descrito en archivo de diseño xml, bajo la carpeta res/menu. Como ejemplo, el correspondiente con el dibujo anterior, sería:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"  
      xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
xmlns:tools="http://schemas.android.com/tools"
tools:context="com.example.myiconapp.MainActivity">
<item
    android:id="@+id/opcion1"
    android:orderInCategory="1"
    android:title="@string/opcion1"
    android:icon="@drawable/ic_launcher_foreground"
    app:showAsAction="always|withText"/>
<item
    android:id="@+id/opcion2"
    android:orderInCategory="2"
    android:title="@string/opcion2"
    app:showAsAction="never"/>
</menu>
```

---

Como se ve el elemento raíz es <menu> y cada opción es un <item>. Los atributos destacados de cada ítem son:

- ▶ **id:** el identificador de turno en todo objeto visual. Nos servirá a la poste para saber qué botón se ha tocado.
- ▶ **orderInCategory:** describe el orden de aparición en la lista desplegable. También puede servirnos para identificar la opción pulsada.
- ▶ **icon:** el ícono con que se muestra la opción. Relacionado con el último atributo.
- ▶ **title:** el texto que describe la opción. Siempre mejor usando la internacionalización. O sale el texto o sale el ícono. También relacionado con el último atributo.
- ▶ **showAsAction:** atributo que indica si la opción se muestra con el ícono o como texto. Asume este rango de valores, que pueden combinarse con la tubería (símbolo | ) que actúa como un OR lógico. Dependiendo del tamaño de resolución de cada dispositivo, puede verse alterado.
  - **always:** siempre se muestra en la barra.
  - **ifRoom :** si cabe, se muestra en la barra si no, en el desplegable.
  - **never:** nunca se muestra en la bar. Siempre en el desplegable.
  - **withText:** se muestra con imagen y texto en la barra.
  - **collapseActionView:** se muestra oculto. A efecto práctico igual que never.

Lo más conveniente porque asegura más compatibilidad con la mayoría de los dispositivos es dejar las dos primeras opciones en always y las demás en never. En algunas aplicaciones, a las opciones del menú desplegable les acompaña el ícono, pero no es un comportamiento estándar: está programado adhoc.

### 7.2.2 Dibujando las acciones del menú

El menú siempre irá asociado a una Actividad, por lo que para cargarlo, lo único necesario es sobrescribir el método que sigue en la misma actividad:

```
.....  
    @Override  
    public boolean onCreateOptionsMenu(Menu menu) {  
        MenuInflater inflater = getMenuInflater(); // objeto para inflar el menú  
        inflater.inflate(R.menu.menu, menu); // menu.xml se infla sobre menu (padre)  
        return true; // devolvemos verdadero para que se dibuje  
    }  
.....
```

Este método será invocado automáticamente por Android al crearse la Actividad y así se hará efectiva su representación.

El objeto menu que es pasado por parámetro representa el punto de anclaje en que se dibujan mis opciones. Sobre él, también puedo añadir de forma programática opciones –MenuItem– sobre la marcha con el método añadir (*add*):

```
.....  
    MenuItem add(int groupId, int itemId, int order, CharSequence title)
```

Por ejemplo:

```
.....  
    MenuItem mi = menu.add(Menu.NONE, 3, 3, "Opción 3");  
.....
```

### 7.2.3 Escuchando las acciones sobre el menú

Para estar atento a la opción seleccionada hay un método previsto a modo de escuchador, el `onOptionsItemSelected`, que debo sobrescribir también en la actividad. Este método recibirá una llamada por detrás cuando cualquier opción del menú sea tocada por el usuario.

Los parámetros que nos pasan son de tipo `MenuItem`, que representa la opción seleccionada. Ya que cada opción tiene un id que le asignamos al definirlo en

el xml, es habitual una estructura condicional sobre el id del elemento para discernir qué opción fue clicada y obrar en consecuencia.

```
.....  
    @Override  
    public boolean onOptionsItemSelected(MenuItem item) {  
        switch (item.getItemId()) {  
            case R.id.opcion1:  
                Log.d ("MIAPP", "Tocó la opción 1");  
            case R.id.opcion2:  
                Log.d ("MIAPP", "Tocó la opción 2");  
        }  
        return true; //indicamos que gestionamos la opción aquí  
    }  
.....
```

#### 7.2.4 Eliminando la barra del menú

A veces puede interesarnos eliminar la barra de menú que aparece por defecto en la parte superior de una actividad. Ya sea por cuestiones de diseño o funcionalidad, es posible eliminarla de dos maneras sencillas. Una programáticamente y otra declarativamente.

- De forma declarativa. La solución es ir al archivo de manifiesto y editar el estilo de la actividad, eligiendo uno del estilo pero que sea NoActionBar

```
.....  
    <activity android:name=".Activity"  
              android:label="@string/app_name"  
              android:theme="@style/Theme.AppCompat.Light.NoActionBar">  
    </activity>  
.....
```

- De forma programática, incluir este código donde se desee (normalmente en onCreate() )

```
.....  
    getSupportFragmentManager().hide();  
.....
```

y si deseas volverlo a mostrar:

```
.....  
    getSupportFragmentManager().show();  
.....
```

La forma b) es más versátil (puedo cambiar sobre la marcha) mientras que la a) es definitiva o estanca (la actividad irá sin barra siempre).

También se puede eliminar la barra de estado (la que tiene el reloj y la cobertura), como se indica en la documentación <https://developer.android.com/training/system-ui/status?hl=es#java>

### 7.2.5 Botón de ir hacia atrás

La navegación hacia atrás es casi un estándar en la usabilidad y navegación. Para dibujar y activar el botón hacen falta dos instrucciones, que hay que incluir dentro de onCreate ()

```
getSupportActionBar().setDisplayHomeAsUpEnabled(true);  
getSupportActionBar().setDisplayShowTitleEnabled(true);
```

Así quedaría la flechita de ir hacia atrás representada en la barra de menú.



Para detectar el evento, simplemente tengo que incluir una opción más dentro del método onOptionsItemSelected y preguntar por el id de la flecha, que por defecto es android.R.id.home.

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    if (item.getItemId() == android.R.id.home) {  
        Log.d("MIAPP", "Tocó la flecha de hacia atrás");  
        finish(); //normalmente, salimos de esta actividad  
    }  
    return true;  
}
```

## 7.3 MENÚS CONTEXTUALES

---

Los menús contextuales son el tipo de menú que se generan sobre la marcha entorno a un elemento gráfico cuando éste recibe un pulsado largo.

Para dibujar un menú contextual sobre un elemento visual, pongamos, `v`; debemos hacer la llamada al método `registerForContextMenu(v)` dentro del método `onCreate()`. Esto hará que el método `onCreateContextMenu()` sea invocado y en él, de forma programática definimos el menú a representar. Por ejemplo:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);
    menu.add(Menu.NONE, 8, 8, "Opcion 8");
    Log.d(getClass().getCanonicalName(), "Asociado");
}
```

---

## 7.4 MENÚS POP O EMERGENTES

---

Los menús pop son también generados sobre la marcha, pero el matiz que los diferencia de los contextuales es que las acciones de este menú no tiene efecto directo sobre el elemento donde aparecen. También se usan para añadir opciones extras no visibles por motivos de espacio.

Para definir un menú pop simplemente debo obtener el elemento y añadirle un *listener* donde se dibuja y representa el menú. De forma genérica, sería así:

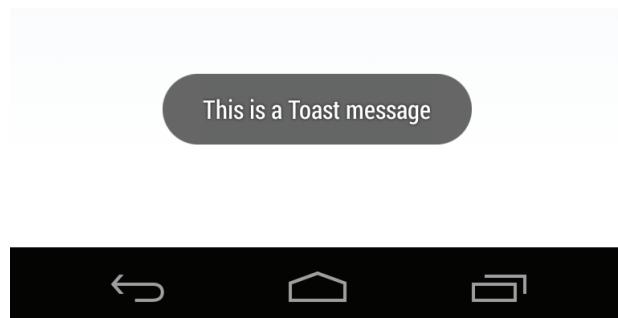
```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        PopupMenu p = new PopupMenu(MainActivity.this, v);
        p.getMenuInflater().inflate(R.menu.menu_layout, p.getMenu());
        p.show();
    }
});
```

---

Así, dibujamos el menú `menu_layout.xml` sobre la vista `v`.

## 7.5 TOAST

Los *Toast* son ese mensaje que aparece y desaparece automáticamente cuando por ejemplo, programamos una alarma.



Para crearlo hacen falta simplemente dos llamadas.

```
Toast toast = Toast.makeText(this, "Hola", Toast.LENGTH_LONG);
toast.show();
```

Con la primera construimos el aviso o toast. El método *makeText* tiene tres parámetros: el contexto, el mensaje textual a mostrar y una duración predefinida: larga como en el ejemplo o corta pasando *Toast.LENGTH\_SHORT*.

Con la segunda instrucción mostramos el mensaje. Tan importante como la primera y se suele olvidar. Con el método *setView* (*View*) de *Toast* puedo construir un aviso personalizado, como por ejemplo:

```
LinearLayout ll = new LinearLayout(this); //this es el contexto
ll.setBackgroundColor(getResources().getColor(R.color.colorAzul));
TextView tv = new TextView(this); //Creamos un texto
tv.setText("Gooool del Madrid");
ll.addView(tv); //Añadimos el hijo al padre
toast.setView(ll); //Personalizamos el aviso
toast.show(); //Lo mostramos
```

Con esto veríamos un aviso personalizado con el color y el texto definidos sobre la marcha.

## 7.6 ALERT DIALOG

---

El AlertDialog es el típico menú que nos encontramos al realizar una acción y nos pide confirmación del tipo: ¿Seguro que desea borrar? o ¿desea salir? y nos presenta al menos dos opciones: SÍ/NO ACEPTAR/CANCELAR.

A continuación un ejemplo de AlertDialog cuando el usuario presiona el botón físico de ir hacia atrás. Por su sencillez, exponemos directamente el código y lo comentamos a posteriori.

---

```
@Override
public void onBackPressed() {
    AlertDialog ad = new AlertDialog.Builder(this).create();
    //this es el contexto
    ad.setTitle("Salir");//definimos el titulo
    ad.setMessage("¿Desea salir?");//y el mensaje del aviso
    //y los botones con sus respectivas opciones ya programadas
    ad.setButton(AlertDialog.BUTTON_NEGATIVE, "NO",
        new DialogInterface.OnClickListener(){
            @Override
            public void onClick(DialogInterface dialog, int which) {
                dialog.cancel();
            }
        });
    ad.setButton(AlertDialog.BUTTON_POSITIVE, "SÍ",
        new DialogInterface.OnClickListener(){
            @Override
            public void onClick(DialogInterface dialog, int which) {
                MainActivity.this.finish();
            }
        });
    ad.show(); //por último, lo mostramos
}
```

---

Leyendo el código se deduce que hay dos opciones previstas en el diálogo: positiva y negativa, AlertDialog.BUTTON\_POSITIVE y AlertDialog.BUTTON\_NEGATIVE respectivamente. Hay posibilidad de añadir una tercera que es AlertDialog.BUTTON\_NEUTRAL.

En este ejemplo, si el usuario dice Sí, se sale de la aplicación. Ojo porque no podemos hacer this.finish ya que en esta sección del código this es el listener y no la actividad. Pero por suerte, en las clases anidadas, se reserva una referencia a la clase contenedora por medio del nombre de la clase seguido de punto this.

Si la respuesta es negativa, simplemente se cierra el propio diálogo con el método *cancel()*.

Para objetos predefinidos como estos Diálogos, –ya veremos también en Notificaciones–, existe un objeto *Builder* asociado que me permite construir y personalizar de manera sencilla el objeto visual a representar (el *Alert Dialog* en este caso).

## 7.7 DIALOG

Siendo los *AlertDialog* una vista habitual, puedo llegar al caso, tener la necesidad o gusto de fabricar mi propio diálogo con el contenido definido por mí mismo. Para ello, tengo la opción de clase *Dialog*.

Puedo definir el diseño o layout en un archivo aparte (como haría con una actividad) y después, emplearlo en el un diálogo personalizado como sería:

```
Dialog ad = new Dialog(MainActivity.this);
ad.setContentView(R.layout.dialogo_layout);
ad.show();
```

Además antes de mostrarse, podría personalizar o asignar valores a los elementos del layout.

## 7.8 SELECTOR DE HORA Y FECHA

Los selectores de fecha y hora son un caso especial de diálogo y pueden programarse de forma sencilla mediante las clases *DatePickerDialog* y *TimePickerDialog*.

La manera óptima para trabajar con ambas clases es crear una clase en nuestra aplicación que sea el Reloj o el Calendario y que hereden de *DialogFragment* (ya veremos los fragmentos, pero son como un trozo de *Activity*).

En esas clases, programaremos los eventos de escucha cuando la hora o la fecha sean seleccionadas y podremos almacenarlas.

Completaremos los ejemplos de selección de fecha y hora en la sección práctica correspondiente, que es más fácil de explicar y entender.

## 7.9 TEST TEMA 7

---

1. Al definir un menú textual es necesario definir:
  - a) Una nueva clase Java
  - b) Un nuevo XML
  - c) Un nuevo HTML
2. Para dibujar el menú, ¿basta con inflarlo?
  - a) Sí, con LayoutInflater
  - b) No, además hay que referir el XML en onCreate
  - c) No es necesario programar ninguna acción
  - d) Hay que programar un nuevo método y usar en él MenuInflater
3. ¿Debo programar un método nuevo para dibujar el menú textual?
  - a) Sí, onSelectedOptionsMenu
  - b) Sí, onCreateOptionsMenu, en una nueva clase
  - c) Sí, onCreateOptionsMenu, en la misma actividad
  - d) No hace falta programar un método nuevo
4. Para escuchar las opciones del menú textual, debo:
  - a) Programar una nueva clase Listener
  - b) Programar el método onClickMenu
  - c) Programar el método onClickMenuItem
  - d) Ninguna de las anteriores
5. ¿Qué opciones tengo para eliminar la barra de menú de la Actividad?
  - a) Cambiar el estilo de la Actividad
  - b) Ocultarlo programáticamente
  - c) Ambas opciones previas son correctas

6. Sobre dibujar la flecha de navegación hacia atrás:
  - a) Se hace necesario descargar el icono de Material apropiado
  - b) Debe declararse en el XML del menú
  - c) No es una práctica extendida en el mundo del diseño
  - d) Basta incluir dos instrucciones en el método onCreate para su representación
7. La diferencia entre el menú contextual y el pop es:
  - a) Ninguna
  - b) Que uno se declara programáticamente y otro declarativamente
  - c) Que el pop atiende a una opción del elemento seleccionado mientras que el contextual no
  - d) Que el contextual atiende a una opción del elemento seleccionado mientras que el pop no
8. Una notificación Toast se crea con un método:
  - a) Estático
  - b) Dinámico
9. Puedo crear un AlertDialog con una ventana personalizada:
  - a) Verdadero
  - b) Falso
10. ¿Qué clases del API de Android ayudan a construir un selector de Fecha y Hora?
  - a) El CalendarPickerDialog y el TimePickerDialog
  - b) El DatePickerDialog y el TimePickerDialog
  - c) El DatePickerDialog y el HourPickerDialog

## SOLUCIONES

1b, 2d, 3c, 4d, 5c, 6d, 7d, 8a, 9a, 10b

# 8

---

## PERSISTENCIA

Todos los botones, textos y menús existen mientras el programa está ejecutándose. Al verse la actividad, todos los objetos que representan la vista están instanciados y ocupan su espacio en la memoria principal o RAM. Pero cuando cierras la aplicación, toda esa información desaparece. Es volátil. A veces queremos guardar los datos de modo que cuando volvamos a abrir la aplicación, se hallen disponibles. Imagina por ejemplo, el nombre y la contraseña de un usuario, la puntuación de un juego, la conversación con un contacto, los números de la agenda. Todo eso nos interesa almacenarlo y poder recuperarlo aún después de apagar el dispositivo. Para ello, todos esos datos deben guardarse en archivos de la memoria secundaria o disco (que en los teléfonos suele llamarse memoria o almacenamiento interno).

A esa operación de almacenar los datos de modo permanente es lo que denominamos persistir y es precisamente lo que vamos a abordar en este tema: las distintas maneras de guardar información de forma persistente.

---

### 8.1 TAREAS PRÁCTICAS DEL TEMA 8

---

- ▶ Crear y usar vectores tipados como recursos persistentes
- ▶ Aprender a diferenciar las distintas ubicaciones de memoria
- ▶ Aprender a gestionar los archivos de preferencias
- ▶ Diseñar e implementar nuestras propias bases de datos relacionales

## 8.2 VECTORES TIPADOS

Cuando empecé a estudiar programación, una de las cosas que más loco me volvió fue la polisemia de los vectores. Vector, array, matriz, arreglo y hasta lista, resultaron ser sinónimos para mi desdicha iniciática. Pues bien, ese conjunto de datos relacionados (ya sea un listado de equipos de fútbol, de nombres de personas, o de provincias) los puedo tener redactados de antemano en un fichero XML y cargarlos después en memoria en lo que se llama un `TypedArray` o vector tipado en español.

Para ello, debo seguir el convenio establecido por Android que consiste en:

1. Definir un xml bajo la carpeta values  
`equipos.xml`
2. Escribir allí el array

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="equipos">
        <item>REAL MADRID</item>
        <item>BARCELONA</item>
        <item>SEVILLA</item>
        <item>ATLÉTICO</item>
        <item>VALENCIA</item>
    </string-array>
</resources>
```

3. Referirlo desde el código para su uso

```
TypedArray array_equipos = getResources().obtainTypedArray(R.array.equipos);
array_equipos.getString(0); //REAL MADRID
```

Lo malo de esta técnica es que sólo puedo guardar listas de números y cadenas (con la etiqueta `<integer-array>`). También se puede hacer arrays de arrays o vectores multidimensionales. Como por ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="equipos1">
        <item>REAL MADRID</item>
        <item>BARCELONA</item>
        <item>SEVILLA</item>
    </string-array>
```

```
<string-array name="equipos2">
    <item>RAYO VALLECANO</item>
    <item>XEREZ</item>
    <item>ALBACETE</item>
</string-array>
<array name="divisiones">
    <item>@array/equipos1</item>
    <item>@array/equipos2</item>
</array>
</resources>
```

---

Pero los datos finalmente almacenados serán siempre esos tipos: números o cadenas. No puedo hacer construir otros tipos propios más complejos. Es un mecanismo sencillo de persistencia, de modo que pueda cargar fácilmente algunos datos permanentes, en mi aplicación.

Un uso habitual se da en combinación con las listas desplegables (Spinner), donde puedo hacer que estas tomen los datos de vectores tipados (TypedArray).

### 8.3 MEMORIA INTERNA Y MEMORIA EXTERNA

---

Debido a la herencia del PC y a las pobres traducciones, viene siendo una confusión habitual la distinción entre memoria externa e interna cuando hablamos de ello en Android. Debe quedar claro que en ambos casos, estamos hablando de disco, de memoria secundaria o no volátil, que viene siendo una cantidad de memoria de 16, 32, 64 o hasta 128 GB en los teléfonos comerciales de hoy.

Esta memoria, se divide en tres espacios lógicos, desde la perspectiva de la aplicación.

1. Memoria interna –no confundir con RAM– (`data/data/mi.paquete.app/`)  
Sólo accesible desde el código de nuestra aplicación. En ella se almacenan preferencias, bases de datos locales (que veremos después) y archivos de caché además de otros archivos que yo puedo crear con el API File IO estándar de Java. Se borran al desinstalar la aplicación automáticamente y también el usuario puede forzar su borrado desde el menú Ajustes --> Borrar Datos.

Para acceder a esta dirección, lo haremos mediante:

---

```
context.getFilesDir().getPath()
```

---

2. Memoria externa pública o compartida. (sdcard//) Son las carpetas de CÁMARA, GALERÍA, DESCARGAS, CAPTURAS DE PANTALLA. Todos los usuarios y todas las aplicaciones pueden leer y escribir con el debido permiso. Si desinstalo la aplicación, los datos permanecen. Cuando hablemos de memoria externa, no estamos refiriéndonos a memorias SD extraíbles ni nada parecido (que han venido siendo un quebradero de cabeza y ya casi no se ven).

Para acceder a esta dirección, lo haremos mediante:

```
.....  
Environment.getExternalStorageDirectory().getPath()  
.....
```

3. Memoria externa privada (sdcard/Android/mi.paquete.app) . Estas carpetas son visibles y editables para todo el mundo. La diferencia con la anterior es que cuando desinstalo la aplicación, estas carpetas también desaparecerán.

Para acceder a esta dirección, lo haremos mediante:

```
.....  
context.getExternalFilesDir(null).getPath()  
.....
```

Qué archivos e información almacenar en cada una de las secciones, depende de la naturaleza de los datos y de las operaciones que quiero permitir al usuario. Piensa por ejemplo en la icónica aplicación Whatsapp. Los archivos de imagen que recibes y envías se guardan en una carpeta pública. Así, desde Galería, los puedes ver después. O eliminarlos. Sin embargo, las conversaciones y los contactos se almacenan en bases de datos de la memoria interna, para evitar que sean accesibles por otro programa.

## 8.4 ARCHIVOS DE PREFERENCIAS

Los archivos de preferencias también designados en inglés por las Preferences, son archivos en formato xml que cuentan con una API propia que permite, de manera muy sencilla, almacenar datos en el formato clave-valor. Un ejemplo:

```
.....  
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>  
  <map>  
    <string name="nombre">prueba</string>  
    <string name="email">modificado@email.com</string>  
  </map>  
.....
```

Tenemos dos claves, que son nombre y email asociadas a sendos valores: prueba y modificado@gmail.com. Este escenario recordará en algo a los ficheros de propiedades en Java a los más veteranos.

Bien, ese es el producto final, pero lo interesante y bueno de este mecanismo es que para construir, mantener y consultar los valores aquí almacenados me basta con unas pocas líneas de código.

Las operaciones con un archivo se pueden resumir en crearlo, editarlo y consultararlo. Y todo se hace a través de la clase SharedPreferences. Vamos viéndolas:

```
.....  
SharedPreferences prefs = getSharedPreferences("archivo", MODE_PRIVATE);  
.....
```

Con esta operación el objeto prefs representa al fichero con nombre "archivo". Hay una ubicación prevista automáticamente por Android por lo que no debo añadir o especificar ruta alguna. Si el "archivo" existe, se abre en modo lectura-escritura. Y si no, automáticamente es creado. Ahí está la magia simplificada de estas preferencias.

El segundo parámetro indica el modo. En un principio, los diseñadores del Android debieron pensar que este fichero podría ser compartido entre otras aplicaciones (de ahí lo de Shared - compartido) y daban opción de declararlo de lectura y escritura pública con los valores MODE\_WORLD\_READABLE, MODE\_WORLD\_WRITABLE. Pronto se arrepintieron y recomiendan siempre el modo privado.

Ahora, para leer y escribir un valor, haremos lo siguiente:

```
.....  
String correo = prefs.getString("email", por_defecto@email.com);  
SharedPreferences.Editor editor = prefs.edit();  
editor.putString("email", "modificado@email.com");  
.....
```

Con la primera instrucción leo el valor de la clave "email". Y qué pasa si no existe esa clave en el fichero. Pues ahí toma sentido el segundo parámetro por\_defecto@email.com, puesto que ése será el valor asumido por correo.

En la segunda y tercera líneas, edito el fichero, modificando o añadiendo nuevos valores.

```
.....  
editor.commit();  
.....
```

Importante esta instrucción, que debo ejecutar siempre para hacer los cambios efectivos. De otra manera no se guardarían bien.

Los valores que puedo almacenar en un fichero de preferencias son: booleanos, cadenas, y numéricos (enteros y reales).

Normalmente hay una clase en cada proyecto, dedicada a gestionar los archivos de preferencias. Veremos más utilidades prácticas en el correspondiente ejercicio.

## 8.5 API JAVA IO

El API de Java 7 de Entrada/Salida (Input/Output en inglés) tiene plena vigencia en el uso y manejo de ficheros. Gracias a él, podre:

- ▶ Crear y eliminar archivos y directorios
- ▶ Leer y escribir archivos de texto
- ▶ Leer y escribir archivos binarios (pdf, jpeg, mp3, etc.)
- ▶ Descargar y subir archivos de cualquier tipo por internet

Si es en la memoria interna, no requerimos de ningún permiso especial. Pero si queremos trabajar con archivos en la memoria externa, hay que obtener los permisos necesarios (ver apéndice sobre permisos). También si queremos enviar o recibir archivos por Internet, será necesario el permiso oportuno.

Para Android (igual que para Java), los archivos alojados en la memoria como los que son transferidos son igualmente tratados como *streams* (anglicismo que podemos traducir flujo o chorro de transmisión) por lo que esta API es aplicable para ambos.

En versiones posteriores de Java como la 8 o la 9, se introducen nuevas clases como Stream o expresiones lambdas, que permite trabajar con ficheros de modo más avanzado. El problema es que para usar características de Java 8, necesito la **minSdkVersion** con valor API 26 (8 - Oreo) lo que limita el uso de mi aplicación a móviles modernos o hacer unas modificaciones en la configuración con Gradle (no pongo la mano en el fuego con Gradle).

Vamos a repasar las principales clases y métodos que permiten estas operaciones.

- ▶ **File.** La clase File nos permite obtener información sobre los directorios y archivos de nuestro sistema de ficheros. Nos permitirá crear y eliminar

archivos dentro de nuestro sistema de ficheros. Pero no nos permitirá leer o escribir en ningún archivo o directorio.

Métodos destacados de la clase `java.io.File`:

```
.....  
public File (String ruta) //constructor  
boolean exists ()  
boolean isDirectory()  
boolean isFile()  
String[] list () //obtenemos el listado del directorio  
boolean createNewFile ()  
boolean delete()  
.....
```

- ▶ **FileReader.** Para leer ficheros de texto.

Métodos destacados:

```
.....  
public FileReader (String ruta)  
public FileReader (File file)  
int read ()  
.....
```

- ▶ **FileWriter.** Para escribir en ficheros de texto.

```
.....  
public FileWriter (String ruta, boolean añadir)  
public FileWriter (File file, boolean añadir)  
int write (String cadena int inicio int final)  
.....
```

Las dos clases anteriores se hacen poco eficaces si tengo que leer o escribir de carácter en carácter, por lo que hay otras clases previstas para leer y escribir por bloques. Esas son:

- ▶ **BufferedReader.** Permite leer por líneas un fichero.

```
.....  
public BufferedReader (FileReader)  
String readLine () //obtengo null si llegué al final  
.....
```

- ▶ **BufferedWriter.** Para escribir por bloques.

```
.....  
    public BufferedWriter(FileWriter)  
    void write(String cadena)  
    void newLine() //escribimos un salto de línea  
.....
```

A diferencia de los ficheros de texto, la cosa cambia cuando trabajamos con ficheros binarios (pdf, doc, jpg, mp3, etc.). Para ellos no usaré las clases FileInputStream y FileOutputStream.

- ▶ **FileInputStream.** Para leer ficheros binarios:

```
.....  
    FileInputStream (String ruta)  
    FileInputStream (File file)  
    int read (byte [] b) //lee b.length bytes  
    int read () //lee un byte  
.....
```

- ▶ **FileOutputStream.** Para escribir ficheros binarios:

```
.....  
    FileOutputStream (String ruta)  
    FileOutputStream (File file)  
    int wirte (byte [] b) //escribe b.length bytes  
    int write (int b) //escribe un byte  
.....
```

Para leer y escribir bytes de forma más eficiente, usaremos BufferedReader y BufferedWriter. No las incluimos porque son casi idénticas.

Lea o no por bloques, un fichero de texto o binario, lo que sí debo hacer siempre es cerrarlo. No hacerlo implica graves disfunciones. Lo más cómodo, para simplificar la gestión de excepciones en el tratamiento de ficheros, es usar el llamado try con recursos. Las instrucciones no quedan algo así:

```
.....  
try (BufferedReader br = new BufferedReader(new FileReader("fichero.txt")))  
{  
    return br.readLine();  
}  
.....
```

No estoy obligado a hacer br.close(), puesto que al declarar el objeto entre paréntesis del try, el objeto BufferedReader se cierra solo (en realidad implementa la interfaz *AutoClosable*).

El trabajo de ficheros implica la gestión de excepciones. Pero motivos didácticos, reservamos el trabajo de excepciones y ficheros en los ejemplos prácticos asociados a esta parte del temario.

## 8.6 BASES DE DATOS RELACIONALES CON SQLITE

---

El uso de una base de datos local puede ser oportuno cuando necesito guardar cierto volumen de información relacionada entre sí. Por ejemplo, para guardar contactos y sus respectivas direcciones de correo o por ejemplo, una colección de los sitios webs que voy leyendo y la fecha de visita.

En realidad, las bases de datos como tales se alojan en un servidor independiente y centralizan un gran volumen de información de todos los usuarios (véase el capítulo introductorio Android en las arquitecturas modernas), pero puede ser útil querer almacenar información local de sólo un usuario. Para este caso, tenemos la opción de Sqlite.

SQL es el lenguaje estándar de creación, mantenimiento y explotación de bases de datos relacionales. De él, hay muchas marcas famosas en el mercado como Oracle, MySQL, o SQL Server, pero para dispositivos Android, hay una versión reducida denominada SQLite. Este es un proyecto independiente, que tiene su propia página y documentación en <https://www.sqlite.org/>

Esa es la tecnología usada en Android en un API sencilla, que la describiremos en esta sección, compuesta básicamente por dos clases: SQLiteDatabase y SQLiteOpenHelper.

Las bases de datos son toda una materia de estudio aparte en mundo TIC, por lo que no pretendo en esta humilde sección abordar semejante contenido. Sí dar unas pinceladas que como siempre serán complementadas con la parte práctica.

En nuestro proyecto necesitamos una clase que herede de SQLiteOpenHelper. Ello nos obligará a sobrescribir dos métodos:

---

```
public void onCreate(SQLiteDatabase db)
public void onUpgrade(SQLiteDatabase oldVersion, int newVersion)
```

---

También es obligatorio crear un constructor que invoque al del padre.

---

```
public class MiBaseDatos extends SQLiteOpenHelper
    public public class MiBaseDatos extends SQLiteOpenHelper(Context contexto,
```

```
String nombre, SQLiteDatabase.CursorFactory factory, int version) {  
    super(contexto, nombre, factory, version);  
}
```

Cuando queramos usar la base de datos, llamaremos al constructor anterior y él, por dentro, invocará al método `onCreate` o `onUpgrade` según proceda de forma automática. Si Android detecta que base de datos con ese nombre y esa versión indicados en el constructor no está creada en el dispositivo, llamará a `onCreate`. En caso de que exista y se indique otro nuevo número de versión en la llamada, se llama a `onUpgrade` (algo raro este último caso).

La clase anterior, sirve para crear la base de datos. Pero para usarla, tenemos que usar la `SQLiteDatabase`. Obtendremos una instancia de ella por medio de la clase anterior, llamando a uno de estos métodos:

```
SQLiteDatabase getWritableDatabase()  
SQLiteDatabase getReadableDatabase()
```

Llamaré al primer método si quiero realizar alguna inserción o borrado en la base de datos y al segundo, si sólo quiero hacer una operación de consulta sobre ella.

De `SQLiteDatabase` me interesan los métodos:

```
void execSQL (String instrucion_sql)  
Cursor rawQuery (String instrucion_sql, String[] argumentos)  
void close ()
```

El primero para insertar, borrar y modificar registros.

El segundo método para ejecutar una consulta que nos permita extraer datos. El tipo devuelto `Cursor`, es una especie de lista recorrible que recoge los resultados obtenidos de por la consulta. Veremos su uso y algunos consejos prácticos en la sección correspondiente.

Toda la sintaxis para INSERTAR, ELIMINAR, MODIFICAR, CONSULTAR datos, así como para crear tablas y restricciones, quedan recogidas en la web del proyecto SQLite <https://www.sqlite.org/lang.html>, que aun siendo parecida al estándar, tiene sus particularidades.

La última API creada para el trabajo con Base de datos por Android es Room. Esta es sólo una mera capa sobre las clases que acabamos de estudiar. Por su inmadurez, escasa innovación y complejidad, nos hemos referido al concepto clásico, que además, se puede trasladar a las tecnologías de base de datos del servidor.

## 8.7 TEST TEMA 8

---

1. En un vector tipado puedo almacenar:
  - a) Sólo números enteros
  - b) Sólo cadenas
  - c) Cadenas, números y objetos complejos
  - d) Ninguna de las anteriores
2. La memoria interna y la memoria externa:
  - a) Son la RAM y el disco respectivamente
  - b) Son la RAM y la SDCard extraíble respectivamente
  - c) Son directorios distintos
  - d) Son la misma ruta en realidad
3. ¿Necesito permiso del usuario para escribir en la memoria interna?
  - a) Verdadero
  - b) Falso
4. Necesito permiso expreso del usuario para escribir en la memoria externa:
  - a) Verdadero
  - b) Falso
5. Los archivos de preferencias son de tipo:
  - a) .java
  - b) .xml
  - c) .doc
  - d) .jar

6. Es recomendable que un fichero de preferencias sea accesible por todas las aplicaciones:
  - a) Si, me interesa que tengan sobre él permisos de lectura y escritura públicos
  - b) Sí, si no, se ralentiza la ejecución de la aplicación
  - c) Mejor que cuente con permisos de lectura públicos y escritura privados
  - d) Mejor que cuente con permisos de lectura y escritura privados
7. Para operar con ficheros binarios debo usar la clase Java:
  - a) FileReader y FileWriter
  - b) FileInputStream y FileOutputStream
  - c) Ninguna de las anteriores
8. Respecto de las bases de datos en Android, marque la correcta:
  - a) Prácticamente todas las aplicaciones necesitan una base de datos SQLite
  - b) La base de datos soportada en Android es del tipo NOSQL
  - c) La base de datos soportada en Android es del tipo Relacional
  - d) Ninguna de las anteriores es cierta

## SOLUCIONES

1d, 2c, 3b, 4a, 5b, 6d, 7b, 8c

# 9

---

## VISTAS AVANZADAS

En este capítulo damos el salto de calidad definitivo a la interfaz de usuario, viendo cómo puedo mostrar listas de elementos, hasta ahora impensable.

Una actividad puede entenderse como una subdivisión de secciones reutilizables: he ahí donde nace el concepto de fragmento. Y también podemos transitar entre ellos sin salir de la misma Actividad.

Completamos el capítulo viendo todas las posibilidades que nos brinda la biblioteca Material Design, una librería desarrollada por el propio equipo Android que nos facilita modernos controles listos para usar como el menú desplegable, el botón flotante o las tarjetas entre otros.

Por último ilustramos el proceso de creación de vistas personalizadas, construyendo controles propios a partir de la clase View, por herencia.

---

### 9.1 TAREAS PRÁCTICAS DEL TEMA 9

- ▶ Mostrar colecciones de elementos / listas
- ▶ Crear fragmentos y transitar entre ellos
- ▶ Importar y usar la librería Material Design
- ▶ Crear vistas personalizadas

---

### 9.2 LISTAS DE ELEMENTOS

Al hablar de listas de elementos ya pasamos a representar colecciones de datos. Es decir, superamos el umbral de un botón, una opción de menú, una imagen, y empezamos a ver un listado de elementos homogéneos, como podrían ser la lista

de contactos de Whatsapp, el listado de correos electrónicos recibidos o una lista de sugerencias de Youtube.

A partir de la versión 5 Lollipop, se introdujeron las vistas recicladas o RecyclerView en el API de Android, que vinieron para mejorar a las *ListView*, verdaderamente demodés y de uso desaconsejado.

Con independencia del tipo de información que componga la lista, siempre se va a construir con la misma lógica. De modo que cuando vayamos a mostrar una colección en forma de tabla, vertical u horizontal, con datos recibidos desde Internet u obtenidos localmente, debemos abordar los mismos pasos y prestar atención a las clases siguiente, que vamos a ir detallando en esta sección y son: RecyclerView, Adapter, ViewHolder y LayoutManager.

### 9.2.1 RecyclerView

El RecyclerView es el elemento raíz o padre de una lista. Debajo suya, integrados como hijos, estará cada ítem o elemento que compone la lista. Su misión es como la de un punto de anclaje y una gestión transparente de los ítems que se van añadiendo a la estructura. Se declarará, generalmente en tiempo de compilación en el archivo de layout correspondiente de esta manera:

```
<android.support.v7.widget.RecyclerView  
    android:id="@+id/lista_rec"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" />
```

Antaño, con las estructuras precedentes al RecyclerView –ListView, GridView– por cada elemento que se dibujaba, se creaba un espacio en la memoria para él. De modo que si una lista tiene mil elementos, se representan en memoria los mil elementos aunque sólo sean visibles a la vez, pongamos, ocho. La novedad que aporta el Recycler es que él se encarga de mantener en memoria como ocho huecos y los rellena con la información visible en cada momento, según el usuario se desliza por la lista. De alguna manera, "se reciclan" los espacios, se reutilizan, lo que supone un ahorro muy considerable de memoria y recursos del dispositivo al tener ocho elementos en memoria al mismo tiempo como máximo y no mil.

El código obligatorio para funcionar con el Recycler es:

1. Obtener la referencia a él:

```
RecyclerView rv = findViewById(R.id.lista_rec);
```

## 2. Asignarle un Adaptador o Adapter:

```
.....  
rv.setAdapter (Adapter);  
.....
```

Veremos la importancia del adaptador o conector asociado al Recycler en el siguiente punto.

### 9.2.2 Adapter

El Adapter trabaja conjuntamente con un Recycler y es el proveedor de éste. Los datos que se agregarán al Recycler serán suministrados siempre por el Adapter. De algún modo, hay una conversación implícita entre ambos, de modo que cuándo toca representar la lista, el Recycler habla automáticamente con su conector y sin que nosotros mediemos, le dice a su adapter:

1. ¿Cuántos elementos tendrá la lista?
2. Y por cada uno de ellos, le dice: dame fila, dame fila, dame fila. etc.

Y es que, cuando creamos nuestra clase Adaptdor, lo hacemos heredando de la clase Adapter con una cabecera del estilo:

```
.....  
public class MiAdapter extends Adapter  
.....
```

y por ello nos vemos obligados a programar varios métodos abstractos, que son:

```
.....  
public abstract VH onCreateViewHolder(ViewGroup parent, int viewType)  
  
public abstract void onBindViewHolder(VH holder, int position);  
  
public abstract int getItemCount();  
.....
```

Empecemos explicando el último, el más obvio. Este método devuelve un número entero y es usado por el Recyler para saber cuántos elementos contiene la lista que se va a dibujar.

Los otros, actúan de forma complementaria. Si es la primera vez que se crea un elemento, es invocado el primero digamos para crear "la caja" que contendrá el

elemento. El segundo método, se invoca para llenar "esa caja" con la información de un ítem concreto.

En realidad, se aprecia la relación que hay entre estos métodos y el juego del reciclaje. Lo que se recicla o se reutiliza es la caja. Y ése es el VH acrónimo de ViewHolder, que detallaremos en el siguiente punto.

En el método *onCreateViewHolder*, inflaremos una "caja" o fila. Esta fila, deberá tener su propio archivo de diseño xml (*layout*) en el que esté claramente definido qué controles y vistas componen cada ítem. Este xml será inflado y posteriormente añadido al que será su padre, el Recycler.

Pero, si hemos dicho que el Adapter va a dar los datos de cada fila, ¿dónde están esos datos?. Pues bien, esos datos que posteriormente facilitara de uno en uno al Recycler, le son pasados al Adapter al invocar al constructor. Es decir, cuando creamos el conector, debemos pasarle los datos, normalmente un tipo List con la información neta de la lista en sí.

Por tanto, además de los métodos enumerados, es siempre habitual que exista un constructor con esta pinta:

```
public MiAdapter (List<> lista, Context contexto)
```

El primer parámetro es la lista con los datos (de lo que sea que fuere mi lista: de contactos, equipos de fútbol o productos de la compra) y el segundo es el famoso contexto, detallado en el Apéndice B y que sin ser obligatorio, nos será necesario.

Puede parecer un poco lioso, pero es importante seguir el procedimiento. Como dijo un exalumno mío, integrado plenamente de forma exitosa al mercado laboral, "hecho un Recycler, hechos todos".

### 9.2.3 ViewHolder

El ViewHolder tiene una traducción literal al español muy apropiada y es el Contenedor de la Vista. La vista, es aquí la fila o el ítem visual que representa para el usuario un elemento de la colección.

Por ejemplo, una conversación en la lista de chats WhatsApp o la cabecera de un correo en Gmail. Y el Holder es el contenedor de esta vista. Este ViewHolder es como digamos "la caja" que mencionábamos en la sección del Adapter. Es un "objeto padre" que alberga la información de fila concreta en un momento dado, pero que después, puede llenarse con otro contenido, recicrándose.

Nuestro Contenedor de Vistas será una clase que herede de ViewHolder.

```
public class MiHolder extends RecyclerView.ViewHolder
```

Esta clase tendrá las subvistas o hijas de la caja declaradas como atributos de la clase. Por ejemplo, en el caso de un chat de Whatsapp, se tendría en esta clase tres TextViews y un ImageView como atributos.

Es obligatorio declarar un constructor que invoque al padre:

```
public MiHolder (View itemView)
{
    super(itemView);
    ...
}
```

En ese constructor, debemos tomar referencia a los elementos hijos de la caja y guardarla en cada atributo de clase.

Y luego es aconsejable aquí también, hacer el relleno de esas subvistas con los datos oportunos en un método aparte. Siguiendo con el ejemplo de Whatsapp, tendríamos un objeto Conversación, con sus strings y el fichero de imagen, y se produciría la asignación de esos atributos a los de la vista (TextView e ImageView) en el interior de este método:

```
void cargarHolder (Conversacion conversacion)
```

No detallo más el código pero en el ejemplo ganaremos claridad.

## 9.2.4 LayoutManager

Una vez confeccionada la lista, podemos elegir mostrarla verticalmente, horizontalmente o en formato de tabla. Esa decisión queda delegada en esta clase, el LayoutManager o gestor de distribución.

Hay dos subtipos de LayoutManager. Uno para distribuciones lineales el LinearLayoutManager (vertical u horizontal) y otras en formato de filas y columnas el StaggeredGridLayoutManager.

Asignaré al Recycler mediante `setLayoutManager()` la distribución elegida.

### 9.2.5 Actualizando la colección

Con todos los elementos coordinados, seremos capaces de crear las filas, dotarlas del contenido y mostrar la vista. Pero qué pasa si necesitamos actualizar la lista (porque se ha eliminado/añadido o modificado un elemento). Tenemos dos opciones:

1. La opción más bruta, que es crear un nuevo Adapter con los datos actuales y volver a asignarle al Recycler el nuevo Adapater con `setAdapter()` los nuevos datos. Eso fuerza su representación nuevamente.
2. Usar los métodos de Adapter más específicos, según: se necesite:
  - a) Alterar la posición de un elemento:

```
.....  
    notifyItemMoved(posicion_vieja, posicion_nueva);  
.....
```

- b) Actualizar un elemento:

```
.....  
    notifyItemChanged(posicion_actualizada);  
.....
```

- c) Eliminar un elemento:

```
.....  
    notifyItemRemoved(posicion_elemento_eliminado);  
.....
```

- d) Insertar un elemento en una posición:

```
.....  
    notifyItemInserted(posicion_elemento_insertado);  
.....
```

## 9.3 FRAGMENTOS

Al decir fragmento el propio nombre nos indica que nos referimos al trozo, o parte de algo. Y es que un Fragment, es la clase que va a representar parte de una pantalla. De alguna manera, un Fragmento formará siempre parte de una Actividad, estará integrada en ella, pero a la vez será independiente: tendrá su propio ciclo de

vida y gestionará sus propios eventos. Porque, un Fragmento, tendrá una clase Java asociada y su propio diseño/fichero xml, de manera que puede reutilizarse en muchas otras actividades.

Cuando se pensó en los Fragmentos, se aspiró a construir componentes reutilizables en distintas Actividades, como módulos independientes o piezas de una pantalla. De hecho, hay famosos fragmentos diseñados por la propia gente de Android como un mapa de Google, integrados en multitud de aplicaciones, gracias a que son porciones reutilizables.

Los métodos importantes de un Fragmento, son :

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

Para inicializar/recrear controles. Es invocado al inicio de representarse el fragmento por primera vez.

```
public View onCreateView (LayoutInflater, ViewGroup, Bundle )
```

Este método es realmente el obligatorio, pues devuelve una vista que es la propia que representa al fragmento. Aquí se infla y se devuelve la View raíz.

```
public onPause()
```

Puede ser útil para guardar el estado del Fragment, invocado al dejar de verse.

Otro detalle importante es que debo escribir el constructor por defecto y en él, invocar al padre con *super()*.

Puedo declarar el Fragment explícitamente en el archivo de diseño de la actividad padre, como por ejemplo:

```
<fragment  
    android:name="com.example.ArticleListFragment"  
    android:id="@+id/list"  
    android:layout_weight="1"  
    android:layout_width="0dp"  
    android:layout_height="match_parent" />
```

O también puedo instanciar el *Fragment* desde código, en la Actividad padre, para lo cual deberé usar la clase *FragmentManager*.

Poco más de lo esencial en relativo a Fragmentos. Veremos como siempre un detallado ejemplo en la sección práctica.

## 9.4 VISTAS DESLIZANTES

Cuando estamos en una pantalla podemos conseguir un efecto deslizante, que nos permite transitar a otra pantalla, pero sin salir de la misma Actividad. Ello se consigue gracias a la clase *ViewPager*, que en realidad nos está permitiendo saltar entre Fragmentos.

### 9.4.1 ViewPager y PagerAdapter

El *ViewPager* se declarará en el XML de la actividad donde queramos y puede ser a pantalla completa u ocupar la sección o proporción deseada.

```
<androidx.viewpager.widget.ViewPager  
    android:id="@+id/pager"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">
```

Este será como el marco o punto de anclaje en el que se irán insertando los sucesivos Fragmentos. Lo que necesitaré hacer desde el código, es obtener una referencia a este elemento y asignarle un objeto de tipo *PagerAdapter*.

Existe una gran analogía entre el *RecyclerView* y el *Adapter*, pues ya dijimos que el primero le iba demandando al segundo, <<dame fila, dame fila, dame fila>>. Pues aquí, el que hace de pidón es el *ViewPager* y el que le va suministrando ítems es el *PagerAdapter*. De modo, que cada vez que deslicemos, el *ViewPager* estará por debajo pidiendo su *Adapter* "dame fragment, dame fragment".

Un código parecido a éste deberemos emplear en la Actividad donde deseemos fragmentos deslizantes.

```
//obtengo la referencia al ViewPager declarado en el archivo de diseño  
viewPager = (ViewPager) findViewById(R.id.pager);  
//Y le asigno su adapter  
pagerAdapter = new MiPagerAdapter(getSupportFragmentManager());  
viewPager.setAdapter(pagerAdapter);
```

Aquí la clase nueva que aparece es MiPagerAdapter, que será una clase independiente que herede de PagerAdapter. Hay varios subtipos, pero vamos a dejar de introducir nombres y quedémonos con la idea. El PagerAdapter tendrá dos métodos obligatorios:

El constructor, donde debemos llamar al padre pasándole el FragmentManager (clase que se encarga de crear y destruir los Fragment y se obtiene del contexto).

```
public PageAdapterPropio(FragmentManager fm) {  
    super(fm);  
}
```

Y el método *getItem (int)*, que recibe un número, que representa el número de fragmento deseado, y devuelve un objeto que es el propio fragmento inflado, que tiene su clase correspondiente.

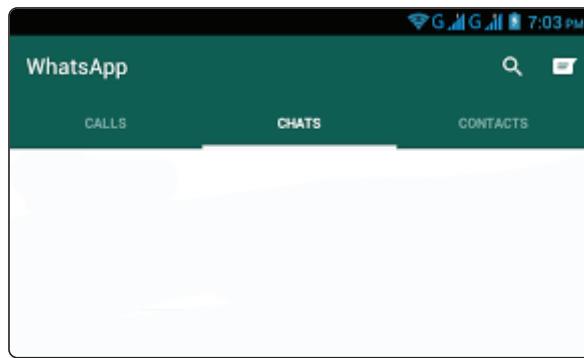
```
@Override  
public Fragment getItem(int position) {  
    Fragment fragment = null;  
    switch (position)  
    {  
        case 0: fragment = new ScreenFragment1();  
        break;  
        default: fragment = new ScreenFragment2();  
    }  
    return fragment;  
}
```

Hay algún método más, pero esto es lo básico. Hasta aquí la sección teórica sobre vistas deslizantes. Un ejemplo será detallado en la sección práctica correspondiente.

## 9.5 PESTAÑAS

---

Las pestañas son un elemento de diseño que permiten crear varias pantallas en una. Su nombre en inglés es TabLayout y también forma parte de la biblioteca Material Design. Un ejemplo clásico sería las usadas por Whatsapp, que divide en tres secciones la pantalla: chats, estados y llamadas.



Si pensamos como un usuario, al pulsar las distintas pestañas, estamos viendo distintas pantallas, pero si lo apreciamos como programador no se está produciendo un salto de Actividad. Ello es debido a que en realidad, lo que estamos consiguiendo es hacer transitar entre distintos fragmentos.

Las pestañas no son más que un intermediario entre el usuario y los fragmentos. De hecho, si deslizamos lateralmente, también conseguimos cambiar de pestaña, por lo que tenemos el escenario anterior: un ViewPager que nos permite transitar entre distintos fragmentos, acompañado esta vez por un TabLayout (o pestañas).

La declaración en el archivo de diseño sería la siguiente:

```
<android.support.design.widget.TabLayout
    android:id="@+id/tablay"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    style="@style/NavigationTab"
    android:layout_gravity="top" />

<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</android.support.v4.view.ViewPager>
```

La configuración será esencialmente la misma. El ViewPager, tendrá su PagerAdapter, que le dará los fragmentos. Lo que sí debemos hacer extra desde el código es conectar el TabLayout con el ViewPager.

```
.....  
//obtengo referencia al tablayout  
TabLayout tabLayout = (TabLayout) findViewById(R.id.tablay);  
tabLayout.setupWithViewPager(viewPager); //lo asocio al viewpager  
.....
```

De este modo, cuando seleccione un tab o pestaña, será como pedírselo al ViewPager.

Las pestañas propiamente dichas, son llamadas TabItem. Y se pueden declarar en el archivo de diseño como hijos del TabLayout, por ejemplo:

```
.....  
<com.google.android.material.tabs.TabItem  
    android:text="Tab 1"/>  
.....
```

O también pueden añadirse de forma dinámica en el código mediante el método:

```
.....  
tabLayout.addTab(tabLayout.newTab().setText("Tab 1"));  
.....
```

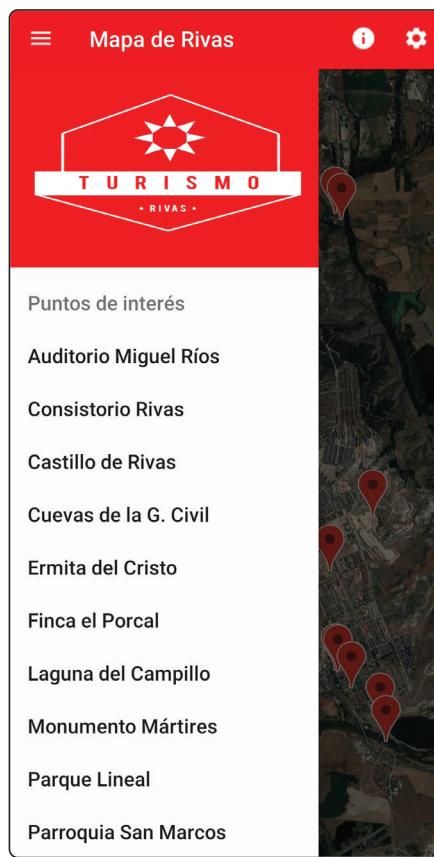
Lo detallaremos en la parte práctica, pero hay una particularidad destacable y no obvia de las pestañas sobre la que me gustaría incidir:

Las primeras dos pestañas (en el ejemplo de WhatsApp chats y estados) se cargan e instancian las dos al inicio, aunque sólo se vea una (los chats). Por eso, es importantísimo que no exista una dependencia funcional entre ambas. Pues si la segunda, depende de algún dato obtenido de la primera, no podrá cargarse correctamente.

## 9.6 MENU LATERAL DESPLEGABLE

---

El menú lateral desplegable se ha convertido en un elemento habitual de muchas aplicaciones que ocultan un menú de tamaño grande y que ofrecen muchas opciones.



Aunque parezca muy sofisticado, su uso es realmente sencillo, pues es uno de los elementos incluidos en Material Design, que no son más que un conjunto de controles en una librería aparte (antes Support Design Library, hoy agrupadas bajo otro nombre) que pretenden simplificar, homogeneizar y decorar las aplicaciones Android.

Lo primero que debemos hacer es importar la librería –Apéndice A–. Despues declarar en el archivo de diseño un elemento DrawerLayout como elemento raíz y bajo él, el diseño de la actividad seguido por último del elemento NavigationView. Por ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:id="@+id/drawer_layout"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    tools:context=".actividades.MapaActivity">

    <include layout="@layout/activity_mapa" />

    <android.support.design.widget.NavigationView
        android:id="@+id/navview"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/cabecera_menu"
        app:menu="@menu/menu_navegacion"/>

</android.support.v4.widget.DrawerLayout>
```

---

Para simplificar el diseño, hemos extraído el propio de la pantalla a otro xml llamado activity\_mapa.xml. Y el propio menú, representado por el NavigationView, se divide en dos apartados: cabecera –headerLayout y cuerpo menu–, que son las opciones propias del menú. Un ejemplo del primero es éste:

```
<?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="wrap_content"
    android:orientation="vertical">

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="180dp"
        android:scaleType="fitXY"
        android:src="@drawable/menu_fondonav"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/aviso"
        android:textColor="#000"
        android:layout_gravity="center_vertical"
        android:layout_marginBottom="10dp"
        android:layout_marginLeft="10dp" />

</LinearLayout>
```

---

Y las opciones del menú, en res/menu:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:title="@string/pdi">
        <menu>
            <item
                android:orderInCategory="0"
                android:title="@string/auditorio" />
            <item
                android:orderInCategory="1"
                android:title="@string/consistorio" />
            <item
                android:orderInCategory="2"
                android:title="@string/castillo" />
            <item
                android:orderInCategory="3"
                android:title="@string/cuevas" />
            <item
                android:orderInCategory="4"
                android:title="@string/ermita" />
            <item
                android:orderInCategory="5"
                android:title="@string/finca" />
            <item
                android:orderInCategory="6"
                android:title="@string/laguna" />
            <item
                android:orderInCategory="7"
                android:title="@string/martires" />
        </menu>
    </item>
    <item android:title="@string/tipodemapa">
        <menu>
            <item
                android:orderInCategory="16"
                android:title="@string/hibrido"
                android:icon="@drawable/ico_tmapa"/>
            <item
                android:orderInCategory="17"
                android:title="@string/normal"
                android:icon="@drawable/ico_tmapa"/>
            <item
                android:orderInCategory="18"
```

```
        android:title="@string/satelite"
        android:icon="@drawable/ico_tmapa"/>
    </menu>
</item>
</menu>
```

---

Unido a esto, hay que declarar un listener del menú, que puede estar en la misma actividad o en una clase aparte, y asignárselo al NavigationView. En este caso, programamos una clase aparte.

---

```
//código en la actividad
NavigationView navigationView = (NavigationView) findViewById(R.id.navview);
navigationView.setNavigationItemSelectedListener(new MenuNavListener(this));
```

---

La clase debe implementar la interfaz `OnNavigationItemSelectedListener`

---

```
public class MenuNavListener implements NavigationView.OnNavigationItemSelected-
Listener {

    //referencia a la actividad que contiene el mapa y el menú
    private Context context;

    //guardamos el contexto, porque sabemos que nos vendrá bien después
    public MenuNavListener (Context context)
    {
        this.context = context;
    }

    /**
     * Método que captura las selecciones del menú lateral (Navigator), y
     * gestiona la petición del usuario.

     * @param item el menú seleccionado
     * @return el valor devuelto por el padre*/
}

@Override
public boolean onNavigationItemSelected(@NonNull MenuItem item) { ... }
```

---

No tiene más. Sólo algún truco o detalle más que por conveniencia didáctica, lo veremos en el ejemplo práctico.

## 9.7 FORMULARIOS ANIMADOS

También formando parte de la librería de diseño de Material, se definió una versión animada de las cajas de texto, para hacer más atractivos y dinámicos los formularios.



Ellos son los TextInputLayout. Un ejemplo sería este:

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/tilcajamail"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/cajamail"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Email"
        android:inputType="textEmailAddress"/>
</com.google.android.material.textfield.TextInputLayout>
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/tilcajatelf"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@+id/tilcajamail">
    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/cajatelf"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
    android:hint="Teléfono"
    android:inputType="phone"/>
</com.google.android.material.textfield.TextInputLayout>
```

---

Hasta la fecha, Android Studio no lo incluye como elemento gráfico arrastrable desde la paleta de edición visual, por lo que hay que copiar el texto de alguna parte o usar el asistente una vez importada la librería.

La parte dinámica es prácticamente igual que siempre, con la salvedad que pueden asociarse clases listener fuera y jugar con estados de validación o error. Por ejemplo, si implementamos la interfaz View.OnFocusChangeListener, y le asociamos a la caja de texto ese escuchador, podemos definir un método, que sea invocado cada vez que el usuario abandona el campo del formulario ( pierde el foco ) o accede a él (lo gana).

---

```
@Override
public void onFocusChange(View v, boolean hasFocus) {
    if (!hasFocus) {
        EditText cajatextomail = (EditText)v;
        String mailintroducido = cajatextomail.getText().toString();
        TextInputLayout tmail = null;
        tmail= (TextInputLayout) actividad.findViewById(R.id.tilcajamail);
        if (!emailValido (mailintroducido))
        {
            tmail.setError("Mail Incorrecto");
        } else
            { //si el mail está bien
                tmail.setErrorEnabled(false);
            }
    }
}
```

---

La clase TextInputLayout es como un envoltorio (*wrapper* en inglés) que permite animar el hijo o campo textual.

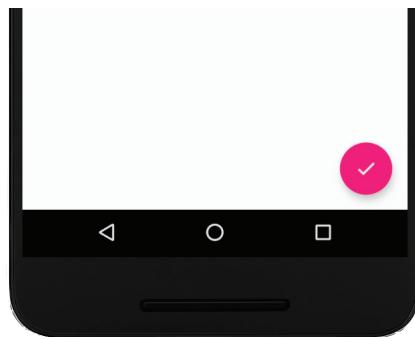
Veremos un ejemplo en marcha en la práctica asociada.

## 9.8 BOTÓN FLOTANTE

---

El botón flotante o botón de acción flotante (*Floating Action Button* en inglés o FAB) es otros de los símbolos introducidos por la iniciativa de Material Design.

Querían con este botón, tener un acceso directo para el usuario a la que el diseñador considere la acción más habitual en la Actividad en que se agregue.



Para su uso hay que definirlo en archivo de diseño y asignarle algunos atributos básicos como el tamaño, la posición o el ícono que muestra en su interior. Hay un atributo novedoso, *elevation*, que indica la altura en el eje imaginario zeta.

```
<android.support.design.widget.FloatingActionButton  
    android:id="@+id/fab"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:src="@drawable/ic_info_outline_white_24dp"  
    android:elevation="6dp"  
    android:layout_gravity="end|bottom"  
    android:layout_marginBottom="40dp"/>
```

Como siempre, debemos incluir un código asociado que gestione los eventos: el escuchador sobre el botón. Este puede ser declarado programáticamente y hacer que el evento se escuche en la misma actividad que contiene el botón o en una clase aparte. Optaremos por la segunda opción en este ejemplo.

```
//en el interior del método onCreate()  
FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);  
fab.setOnClickListener(new ListenerFAB());
```

Donde ListenerFAB es la clase:

```
public class ListenerFAB implements View.OnClickListener {
```

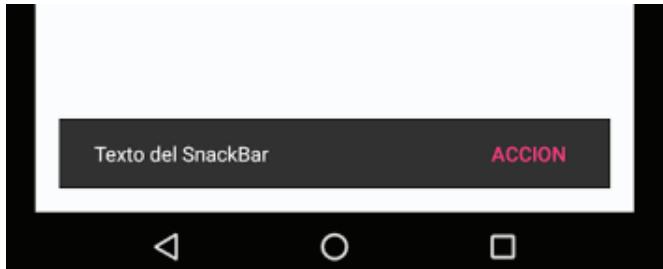
```
    @Override  
    public void onClick(View v) {  
        Log.d ("APP", "Se ha pulsado el botón flotante");  
    }  
}
```

---

## 9.9 BARRA EMERGENTE

---

La barra emergente, en inglés bautizada como *SnackBar*, es una notificación pensada para mostrar al usuario durante un pequeño lapso de tiempo, tras realizarse una acción.



Está también incluida en Material Design.

Para su uso, basta invocar al método estático `make()`, similar a lo que hacíamos con `Toast`. Si por ejemplo modificamos el ejemplo anterior del botón flotante y hacemos que cuando se pulse este, aparece la barra emergente, sería así:

---

```
public class ListenerFAB implements View.OnClickListener {  
    @Override  
    public void onClick(View v) {  
        Snackbar.make(v, "Se presionó el FAB", Snackbar.LENGTH_LONG).show();  
    }  
}
```

---

Usamos `show` para mostrar la notificación una vez creada.

El parámetro `v`, es simplemente la vista de referencia donde a los pies se dibujará la barra emergente. Digamos, el contexto.

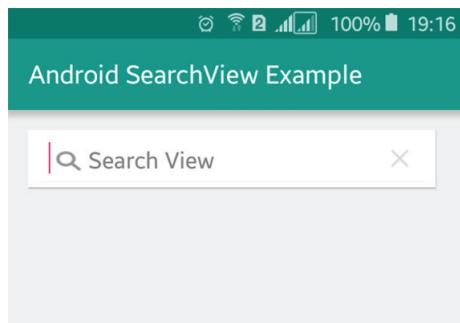
Se puede definir una Acción, caracterizada por un String, que aparezca en un color determinado al margen derecho. Otros métodos de Snackbar destacados serían:

```
setAction (String, Listener) // para llamar a una acción cuando se toque  
setActionTextColor (int) // para modificar el color del String anterior  
setCallback (Snackbar.Callback) //para invocarse al cambiar su visibilidad
```

Veremos un ejemplo en marcha con los dos elementos anteriores en la correspondiente sección práctica.

## 9.10 CAJA DE BÚSQUEDA

La caja de búsqueda es un elemento habitual en muchas aplicaciones y suele aparecer en la parte superior de un listado. La clase que la representa se denomina SearchView.



Para definir el elemento, basta con incluirlo en el fichero de diseño:

```
<SearchView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/cajabusqueda">  
</SearchView>
```

Después podemos referirlo desde el código y asociarle varios escuchadores como por ejemplo el OnQueryTextListener que nos permiten seguir eventos como:

- ▶ Si el texto ha cambiado
  - ▶ La lupa se ha pulsado
- 

```
SearchView searchView = (SearchView) findViewById(R.id.cajabusqueda);
searchView.setOnQueryTextListener(new SearchView.OnQueryTextListener() {
    @Override
    public boolean onQueryTextSubmit(String s) {
        Log.d("Texto buscado ", s);
        return false;
    }
    @Override
    public boolean onQueryTextChange(String texto_introducido) {
        Log.d("Texto introducido", texto_introducido);
    }
})
```

---

Hay una guía bastante extensa en la documentación oficial que cubre aspectos avanzados sobre el uso de cajas de búsqueda <https://developer.android.com/guide/topics/search>. En la sección práctica haremos algún ejemplo más sencillo.

## 9.11 TARJETAS

---

Las tarjetas fueron popularizadas por algunas aplicaciones de noticias de Google en sus inicios. La idea es combinar imagen y texto, agrupados bajo una forma rectangular. Su clase es la CardView. También hay que importarlo desde la librería de diseño Material.

En realidad las tarjetas son hijas del FrameLayout. Es decir, una GroupView cuyos hijos se muestran apilados, con la peculiaridad que tienen las esquinas redondeadas. Un ejemplo de declaración en el archivo de diseño sería:

```
<android.support.v7.widget.CardView
    android:id="@+id/card1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    card_view:cardCornerRadius="6dp"
    card_view:cardElevation="10dp"
    card_view:cardUseCompatPadding="true"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto">
    <ImageView
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        android:src="@drawable/ic_madrid"
        android:scaleType="fitCenter"
        android:layout_marginBottom="15dp"/>
    <TextView
        android:id="@+id/txt2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:text="GRAN VÍA DE MADRID"
        android:layout_gravity="bottom|center"
        android:background="#8c000000"
        android:textColor="#ffe3e3e3"
        android:textSize="18sp"
        android:textStyle="bold"/>
</android.support.v7.widget.CardView>
```

Los atributos destacados de esta vista son:

- ▶ cardElevation: Permite definir una sombra cuanto mayor sea el valor
- ▶ cardBackgroundColor: Para el color de fondo de la tarjeta
- ▶ cardCornerRadius y el método *setRadius()*; para jugar con la apariencia de las esquinas

## 9.12 VISTAS PERSONALIZADAS

A estas alturas hemos visto un gran número de vistas, controles y distribuciones que permiten configurar cualquier pantalla. Pero en ocasiones, las clases aportadas por el SDK y las librerías de diseño extras como Material, no ofrecen todo.

Por ejemplo, hace años que surgió la tendencia de una imagen redondeada. La propia imagen de perfil de Whatsapp o la usada también en los perfiles de la popular aplicación de Instagram. Si ya tenemos la ImageView, no hay nada así como ImageViewRedonda o similar.

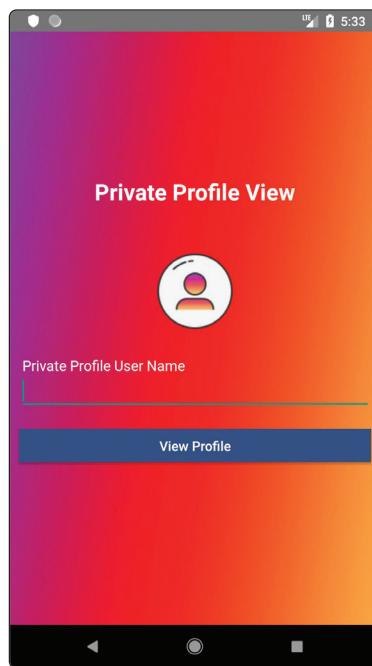
Las opciones son múltiples para conseguir efecto, pero una muy potente y curiosa, gracias a Java ser orientado a objetos, es crear una clase propia que herede de ImageView y sobrescribir el método *onDraw()*, que es el que realmente dibuja sobre el plano la vista visible.

```
.....  
public class RoundedImageView extends AppCompatImageView  
{  
    public RoundedImageView(Context context) {  
        super(context);  
    }  
  
    public RoundedImageView(Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
  
    public RoundedImageView(Context context, AttributeSet attrs, int defStyle{  
        super(context, attrs, defStyle);  
    }  
  
    @Override  
    protected void onDraw(Canvas canvas) {  
        Drawable drawable = getDrawable();  
        if (drawable == null) {  
            return;  
        }  
        if (getWidth() == 0 || getHeight() == 0) {  
            return;  
        }  
        Bitmap b = ((BitmapDrawable) drawable).getBitmap();  
        Bitmap bitmap = b.copy(Bitmap.Config.ARGB_8888, true);  
        int w = getWidth();  
        @SuppressWarnings("unused")  
        int h = getHeight();  
        Bitmap roundBitmap = getCroppedBitmap(bitmap, w);  
        canvas.drawBitmap(roundBitmap, 0, 0, null);  
    }  
    public static Bitmap getCroppedBitmap(Bitmap bmp, int radius) {  
        Bitmap sbmp;  
        if (bmp.getWidth() != radius || bmp.getHeight() != radius) {  
            float smallest = Math.min(bmp.getWidth(), bmp.getHeight());  
            float factor = smallest / radius;  
            sbmp = Bitmap.createScaledBitmap(bmp, (int) (bmp.getWidth() / factor),  
                (int) (bmp.getHeight() / factor), false);  
        } else {  
            sbmp = bmp;  
        }  
        Bitmap o = null;  
        o = Bitmap.createBitmap(radius, radius, Bitmap.Config.ARGB_8888);  
        Canvas canvas = new Canvas(o);  
        final String color = "#BAB399";  
        final Paint paint = new Paint();  
        final Rect rect = new Rect(0, 0, radius, radius);  
        paint.setAntiAlias(true);  
        paint.setFilterBitmap(true);  
        paint.setDither(true);  
.....
```

```
        canvas.drawARGB(0, 0, 0, 0);
        paint.setColor(Color.parseColor(color));
        canvas.drawCircle(radius / 2 + 0.7f, radius / 2 + 0.7f,
        radius / 2 + 0.1f, paint);
        paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.SRC_IN));
        canvas.drawBitmap(sbmp, rect, rect, paint);
    return o;
}
}
```

Una vez definida la clase, puede declararse en archivo de diseño y será representada como otra vista más.

```
<com.example.imagenredondeadaapp.extendedesviews.RoundedImageView
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:src="@drawable/fcb"
    android:scaleType="fitXY"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```



La clase, el elemento en archivo de diseño y la imagen, son sólo ilustraciones a modo de muestra. Veremos un ejemplo concreto de la creación de una Vista propia en la sección correspondiente. También hay multitud de librerías de terceros con Vistas que podemos reutilizar importándolas a nuestros proyectos.

### 9.13 TEST TEMA 9

---

1. Para mostrar colecciones de elementos, debo usar un RecyclerView:
  - a) Verdadero
  - b) Falso
2. ¿Qué función tiene el Adapter respecto del Recycler?
  - a) Actúa como proveedor de estilo, maquetando su apariencia
  - b) Actúa como proveedor de datos, proporcionando los ítems que componen la lista
  - c) Le indica cuántos elementos debe representar
  - d) Las dos anteriores son correctas
3. Los fragmentos son como una sección de una pantalla, pensada para poder reutilizarse dentro de distintas actividades:
  - a) Verdadero
  - b) Falso
4. Marque la afirmación correcta:
  - a) ViewPager es como el ViewHolder
  - b) Debo definir en un xml aparte cuál es el formato de fila o ítem, que después se inflará
  - c) RecyclerView es análogo a un PagerAdapter
  - d) Todas las anteriores son falsas
5. La librería de Material Design forma parte del SDK:
  - a) Verdadero
  - b) Verdadero, pero sólo a partir del API 25
  - c) Falso, debo importarla como dependencia mediante Gradle

- 
6. Un TabLayout sirve para:
    - a) Dibujar un menú desplegable
    - b) Renderizar una lista
    - c) Definir un botón flotante
    - d) Ninguna de las anteriores
  7. El TextInputLayout:
    - a) Me permite usar el formato svg
    - b) Permite usar botones animados
    - c) Se emplea para crear Formularios interactivos
    - d) Todas las anteriores son ciertas
  8. Sobre Floating Action Button, marque las correctas:
    - a) Su uso debe reservarse para ejecutar la tarea más común para el usuario en esa actividad
    - b) Debe emplear un color que destaque
    - c) El icono del botón no es personalizable
    - d) Todas las anteriores son ciertas
  9. La SearchView me permite realizar búsquedas a través de Google:
    - a) Verdadero
    - b) Falso
  10. Puedo crear elementos visuales personalizados mediante clases que hereden de View:
    - a) Verdadero
    - b) Falso

## SOLUCIONES

1a, 2d, 3a, 4b, 5c, 6d, 7c, 8a, 9b, 10a

# 10

---

## HTTP DESDE ANDROID

Constantemente las aplicaciones envían y reciben datos de otros dispositivos. Fotos, texto, videos, que son transmitidos y en muchos casos almacenados en el terminal Android. Detrás de todo ello, el protocolo HTTP. Veremos su uso y manejo desde Android, lo que nos permitirá transferir y recibir información que viaja por Internet.

Comprenderemos también la esencia de estructurar la información para hacerla transmisible y legible en los extremos de la comunicación, lo que viene a denominarse serializar.

### 10.1 TAREAS PRÁCTICAS DEL TEMA 10

---

- ▶ Aprender las clases implicadas en la comunicación y su uso
- ▶ Enviar y recibir información mediante http

### 10.2 HTTP

---

Prácticamente toda la información que nuestro dispositivo recibe o envía desde Internet lo hace por el protocolo HTTP –protocolo de transferencia de hipertexto–. De modo que lo que enviamos o recibimos, es siempre básicamente lo mismo. Un mensaje HTTP de ida que es la petición y otro de vuelta que representa la respuesta.



El cliente, aquí, es nuestro dispositivo. Y el servidor, es un ordenador, generalmente identificado y accesible por una dirección web o URL (<http://nombresservidor/nombresservicio>).

Los videos, mensajes de chat, las imágenes, incluso las notas de voz viajan por este canal. Puedes tener asociado HTTP al mundo web. Y es verdad, se emplean también en la transmisión de páginas HTML, pero en realidad, por su madurez y su extensión, se usa prácticamente para toda comunicación.

Para manejar el protocolo desde Android, están prevista las siguientes clases: `AsyncTask`, `HttpURLConnection`, `InputStream`, `OutputStream`, `URL` y el API del JAVA IO, visto en el capítulo 8 de persistencia.

### 10.3 ASYNCTASK

Es la clase principal, donde realmente se produce la comunicación entre nuestro dispositivo y el servidor de Internet de turno. Su nombre viene de tarea asíncrona y es que Android obliga a que la comunicación se produzca en un proceso independiente.

Normalmente, nuestra aplicación tiene un proceso o hilo, que es el principal, que muestra la Actividad y responde a la interacción con el usuario. Para que la experiencia del usuario no se vea interrumpida por los retardos de la comunicación vía internet, Android obliga a realizarla fuera, en otro proceso aparte. Y para simplificar la gestión de hilos, crearon la clase `Aysnctask`, que crea un nuevo proceso por nosotros.

Al heredar de `AsyncTask`, nuestra clase se verá obligada a implementar los siguientes métodos:

```
protected abstract Result doInBackground (Params... params)
protected void onPostExecute (Result result)
```

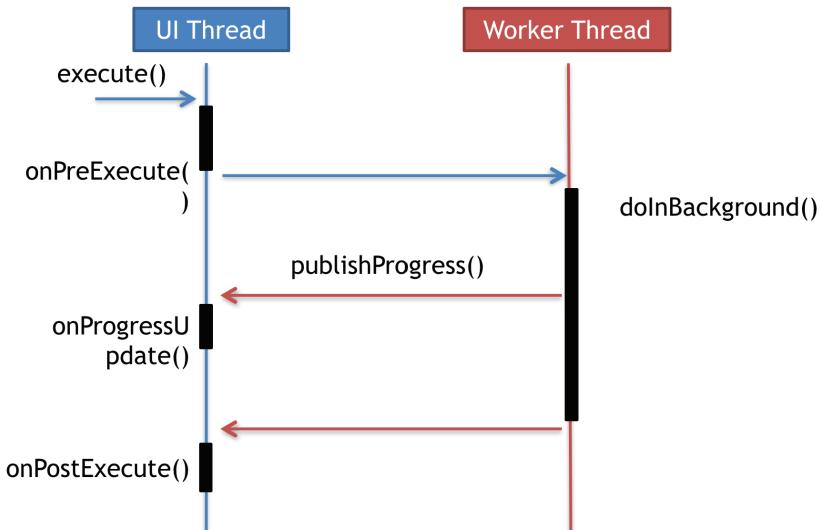
De forma opcional podrá sobrescribir:

```
protected void onProgressUpdate (Progress... values)
protected void onPreExecute ()
```

Describiremos después los parámetros de estos métodos, pero demos una visión de la secuencia de ejecución antes:

Para lanzar la comunicación deberemos instanciar la clase `AsyncTask` y después invocar al método `execute()`. Después, el método `doInBackground()` será ejecutado y es realmente en su interior donde se programa la conversación remota. Cuando esta finalice, se producirá un *callback* en `onPostExecute()`. Ahí está la asincronía. No sabemos cuánto va a durar la interacción con el servidor. Sólo sabemos que cuando acabe, será invocado `onPostExecute()`.

El siguiente esquema dibuja el flujo que se da entre el hilo principal (azul) y la clase asíncrona (roja):



Al definir el `AsyncTask<Params,Progress,Result>`, se declaran parámetros en su cabecera. Estos tipos genéricos, representan:

- ▶ **Params:** Es el tipo de datos que toma la entrada el proceso de envío. Si por ejemplo, estás enviando una lista de Personas, el tipo Persona podría ser el referido. En el método `doInBackground()` se recibe un array de tipo Params, es decir, de personas en este caso, usando para ello la anotación varargs `doInBackground(Personas ... lista_personas)`.
- ▶ **Progress:** O es Void o un tipo numérico, que es el valor que va a indicar el porcentaje de progreso de la tarea en segundo plano. Opcional. Será el tipo que se pase a `onProgressUpdate (Progress... values)`.
- ▶ **Result:** Es el resultado devuelto u obtenido por la comunicación. Normalmente será un código de control, un tipo entero o un booleano, para saber si la cosa ha acabado bien o mal, o también puede ser cualquier tipo de dato obtenido como respuesta del servidor: un String o un objeto propio.

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {  
    ....//cuerpo de la clase  
}
```

## 10.4 ATRIBUTOS IMPORTANTES DE HTTP

Si bien no es necesario conocer los detalles del protocolo al completo para trabajar con él, sí resulta conveniente conocer algunos.

Un mensaje HTTP (con independencia de la versión del protocolo) se divide en cabecera (header) y cuerpo (body). En el cuerpo, viaja el mensaje que queremos transmitir como tal o el que recibimos del servidor. Puede ir vacío si desde el cliente, mandamos una petición en la que no adjuntamos información. Por ejemplo, si pedimos una página web la dirección de la página es un atributo de la cabecera pero en el cuerpo no estamos enviando nada.

### 10.4.1 URL

La dirección del servidor con el que me comunico. Está formada a su vez por IP, el puerto y el nombre del servicio.

### 10.4.2 Contenido

En el cuerpo, viaja cualquier tipo de información: puede ser una imagen, un pdf, un texto, una página web, etc. Pues para identificar el tipo de contenido del cuerpo (ya que en el fondo, es un chorro de letras), hay un atributo de la cabecera que es el *Content-Type* o tipo MIME. Digamos que es como la extensión del archivo. Hay un listado oficial de tipos mime en [https://developer.mozilla.org/es/docs/Web/HTTP/Basics\\_of\\_HTTP/MIME\\_types/Common\\_types](https://developer.mozilla.org/es/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types).

### 10.4.3 Status

La comunicación ha podido ser un éxito o tener algún problema en el lado del servidor, o verse interrumpida por cualquier otra circunstancia. Para poder saber cómo han ido las cosas desde el cliente, una vez recibido el mensaje de respuesta, hay un código numérico que nos ayuda a saberlo que es el *Status Code*.

Las hay de 5 tipos, según empieza por números del uno al cinco. Los más habituales son los que empiezan por 2 (que nos dicen que la cosa ha ido bien) y los que empiezan por cinco (que nos informan de un fallo en el servidor). El listado completo puede hallarse aquí <https://developer.mozilla.org/es/docs/Web/HTTP/Status>.

### 10.4.4 Método

El método es un atributo que le da una semántica al paquete que enviamos desde nuestro dispositivo. Básicamente hay dos métodos: *GET* y *POST*. El primero es cuando nos comunicamos con el servidor y queremos obtener una información de vuelta. Algo así como una consulta. Por contra, empleamos el segundo cuando queremos adjuntar información o datos desde el cliente hacia el servidor, cuando queremos subir algo.

## 10.5 JSON

---

Muchas veces, lo que enviamos y recibimos al servidor es información textual, pero con una estructura. El formato JSON (JavaScritp Object Notation) se ha convertido en un estándar en la representación de información para su envío, adjuntándose en el cuerpo y siendo su tipo mime reservado el application/json.

Será habitual que mientras trabajamos en Java, tengamos por ejemplo la clase persona representada con estos atributos:

```
public class Persona {  
    private String nombre;  
    private String apellido;  
    private Date fechaNacimiento;  
    private Boolean soltero;  
    private Integer cantHijos;  
    // ... Los setters y getters para cada atributo ...  
}
```

Y que pasemos de un objeto persona a su representación textual en formato JSON para poder enviarlo, a algo así:

```
{  
    "nombre" : "Adrian",  
    "apellido" : "Paredes",  
    "cantHijos" : 0,  
    "soltero" : false,  
    "fechaNacimiento" : {  
        "year" : 82,  
        "month" : 5,  
        "day" : 4,  
        "date" : 3,  
        "hours" : 0,  
        "minutes" : 0,  
        "seconds" : 0,  
        "time" : 391921200000,  
        "timezoneOffset" : 180,  
    }  
}
```

Estas dos acciones, pasar de objeto de Java a texto en formato JSON y viceversa, son complementarias y se denominan serializar y deserializar respectivamente.

Mucha es la teoría que hemos visto en este capítulo. Pero es lo esencial para manejarnos y enviar y recibir todo lo que necesitemos por Internet. En la práctica asociada, haremos un detalle de la comunicación exhaustivo.

## 10.6 TEST TEMA 10

1. El acrónimo de HTTP es:
  - a) Html Type Transfer Protocol
  - b) Protocolo de Transferencia de Hipertexto
  - c) Protocolo de Transferencia de Html
  - d) Hipertext Transfer Timeline Portocol

2. En HTTP ¿cuántos agentes tengo?
  - a) 4 la presentación, la base de datos, el servidor y el controlador
  - b) 3 el cliente, el servidor y la base de datos
  - c) 2 el cliente y el servidor
  - d) Ninguna de las anteriores
3. HTTP se ha convertido en un estándar para el intercambio de información entre cliente y servidor:
  - a) Verdadero
  - b) Falso
4. Respeto de AsyncTask :
  - a) Debo implementar esta interfaz para comunicarme por HTTP
  - b) Es obligatorio que sobreesciba dos métodos al crear mi AsyncTask
  - c) Necesito pedir permiso al usuario en ejecución para llamar a AsyncTask
  - d) Ninguna de las anteriores
5. La ejecución de un AsyncTask:
  - a) Se hace en el mismo hilo que en el de la interfaz de usuario
  - b) Se produce en un hilo separado
  - c) Se hace en primer plano
  - d) Ninguna de las anteriores
6. El atributo Status de HTTP nos informa sobre:
  - a) El tipo de información intercambiada
  - b) El resultado semántico de la comunicación (positivo, negativo u otro)
  - c) El tiempo empleado en la comunicación
  - d) Ninguna de las anteriores
7. La URL empleada en un AsyncTask nos habla de:
  - a) Una página web
  - b) Un archivo JSON
  - c) Un servicio web
  - d) Todas las anteriores son ciertas

8. JSON es un formato para representar información:
  - a) Sí, de modo textual, y así poder intercambiarla
  - b) Sí, equivalente a un formato ejecutable
  - c) Falso, es la notación estándar de objetos Java
  - d) Ninguna de las anteriores son ciertas

## SOLUCIONES

1b, 2c, 3a, 4b, 5b, 6b, 7c, 8a

# 11

---

## CLASES PRINCIPALES

Describir el API íntegro de Android, sería una enciclopedia inabordable. Pero sí quiero intentar en este capítulo, sintetizar las clases más comúnmente utilizadas y daros a conocer su uso.

### 11.1 TAREAS PRÁCTICAS DEL TEMA 11

---

- ▶ Comprobar la conectividad a internet
- ▶ Reproducir videos y sonidos
- ▶ Descargar archivos mediante el servicio de descargas
- ▶ Programar y gestionar alarmas internas
- ▶ Consultar proveedores de contenido

### 11.2 CONNECTIVITYMANAGER

---

Con esta clase, puedo comprobar si el dispositivo tiene conexión a Internet y de qué tipo se trata (WIFI, MOBILE, ETHERNET, etc.).

Lo primero, necesito obtener una referencia al objeto , y lo hago a través del Contexto, de esta manera:

---

```
ConnectivityManager cm = null; //declaro e inicializo  
cm = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
```

---

En realidad, esta clase es un Servicio provisto por Android y lo identificamos por su nombre en forma de constante Context.CONNECTIVITY\_SERVICE.

Ya veremos más adelante los Servicios, de hecho ya hemos usado alguno, pero básicamente son clases que desempeñan una tarea no visual.

Una vez tengo el objeto, puedo obtener información de la red a través del método:

```
NetworkInfo ni = cm.getActiveNetworkInfo();
```

Y a su vez, ver mediante el objeto NetworkInfo comprobar si hay red ni.isConnected() y de qué tipo es ni.getType().

Para que esto funcione, debemos además declarar el permiso ACCESS\_NETWORK\_STATE en el fichero de Manifiesto, puesto que Android así lo considera.

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

Veremos un ejemplo en la sección práctica correspondiente.

### 11.3 MEDIAPLAYER

Gracias a esta clase, puedo reproducir con facilidad vídeo y audio.

Para instanciarlo, invoco al método estático *create* y le paso el contexto y el archivo que quiero reproducir. Que bien puede ser un archivo local o remoto. Tras ello, puedo iniciar su reproducción con *start*.

```
MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sound_file_1);
mediaPlayer.start(); // reproducción
```

Hay opciones de pararlo, reproducir en bucle, etc.

Es importante, según nos advierte la documentación oficial, de liberar el objeto mediaPlayer tras su uso, pues puede consumir una cantidad ingente de recursos. Para ello basta con ejecutar estas instrucciones:

```
mediaPlayer.release();
mediaPlayer = null;
```

También puede ser habitual que necesite llevar a cabo alguna acción, tras finalizar la reproducción del archivo en curso. Para ellos, la clase donde defina el MediaPlayer, puede implementar la interfaz MediaPlayer.OnCompletionListener. Así, puedo realizar esta asignación:

```
.....  
    mediaPlayer.setOnCompletionListener(this);  
.....
```

Y cuando el audio o video termine, el método siguiente será invocado de forma automática:

```
.....  
    @Override  
    public void onCompletion(MediaPlayer mediaPlayer) {  
        Log.d("MIAPP", "La canción ha terminado de sonar");  
    }  
.....
```

## 11.4 DOWNLOADMANAGER

La descarga de archivos de Internet a nuestro equipo es una tarea común de muchas aplicaciones. Por ello, para facilitar la vida al programador, el API de Android nos lo da hecho a través de esta clase.



Lo primero es que declaremos el permiso de acceso a Internet en el Manifest.xml

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

Si queremos guardar lo que descargamos en una carpeta pública (de la memoria externa a nuestro programa), debemos declarar también este permiso:

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

Y solicitar permiso al usuario en tiempo de ejecución, por considerarse que es un permiso peligroso (Ver Apéndice C GESTIÓN DE PERMISOS).

Hay una clase clave, `DownloadManager.Request`, que alberga los atributos básicos de la descarga y que debemos definir antes de iniciarla. Estos son:

- ▶ La dirección del recurso remoto que quiero descargar (String / URL)
- ▶ La ruta local o destino donde quiero almacenar el recurso descargado
- ▶ El tipo de contenido a descargar (pdf, mp3, jpg) o también denominado Tipo MIME
- ▶ Una descripción textual del contenido, que acompaña la ventana de descarga

A continuación, un ejemplo de cómo asignar estos valores en un objeto `Request`:

```
DownloadManager.Request request = null;
Uri uri = Uri.parse("www.descargas.es/archivo.mp3");
request=newDownloadManager.Request (uri);
request.setDescription("Musica ");
request.setTitle("Canción muestra mp3 ");
request.setMimeType("audio/mp3");

//conformamos la ruta de destino
String ruta_descarga = null;
ruta_descarga = getExternalFilesDir(null).getPath() + "/cancion1.mp3";
f_destino = new File(ruta_descarga);
Uri uri_descarga = Uri.fromFile(f_destino);
request.setDestinationUri(uri_descarga);
```

De modo opcional, también puedo indicar la visibilidad de la notificación que sale en la zona alta de la pantalla, mientras se produce la descarga, mediante el método `setNotificationVisibility()`. Esta puede ser oculta, visible y desaparecer al completarse, o visible hasta que el usuario desee. Respectivamente, estas son las constantes que representan las opciones:

- ▶ DownloadManager.Request.VISIBILITY\_HIDDEN
- ▶ DownloadManager.Request.VISIBILITY\_VISIBLE
- ▶ DownloadManager.Request.VISIBILITY\_VISIBLE\_NOTIFY\_COMPLETED

Acto seguido, debemos realizar esa petición al servicio de Descarga, poniendo a la cola nuestro encargo, ya que el DownloadManager también puede ser demandado por otras aplicaciones. Para ello, obtenemos la instancia al servicio y le encomendamos la misión de descargar nuestra petición.

---

```
DownloadManager dm = null;
dm = (DownloadManager) this.getSystemService(Context.DOWNLOAD_SERVICE);
int id_descarga = manager.enqueue(request);
```

---

En la última línea, al "encolar" nuestra petición, nos da un identificador, algo así como un número de ticket con nuestro pedido. Este es un detalle importante, que debemos almacenar en una variable temporal, para saber cuándo ha terminado nuestra descarga.

Nos queda ver de qué modo sabemos que el proceso de descarga ha finalizado. Para ello, en el siguiente apartado, os presentaré a los Receptores –*Receivers*–, tercer Componente básico de Android junto con las Actividades y Servicios.

## 11.5 BROADCASTRECEIVER

---

Los Receptores o *Receivers* están representados por esta clase. Gracias a ella, voy a poder recibir un aviso por parte del Sistema, cuando la descarga haya finalizado.

En realidad, los BroadcastReceiver son una especie de *listeners* o clases escucha-eventos, que yo puedo utilizar para recibir avisos en ellas ante diversos eventos como: cuándo se enciende el terminal o cuando un servicio ha finalizado.

Para poder usarla, lo primero que tengo que hacer es definir la clase y es mejor usar el asistente, pues no basta con crear el archivo java, sino que hay que declararla en el Manifest.xml, por ser un Componente.

---

```
<receiver
    android:name=".DescargaCompletaReceiver"
    android:enabled="true"
    android:exported="true">
```

```
</receiver>
public class DescargaCompletaReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    ...
}
```

El método *onReceive* será el que reciba la llamada por detrás cuando la descarga finalice. Pero, ¿cómo ligamos esta clase, con el servicio de descargas?: vamos a verlo.

Digamos que cuando el DownloadManager acaba, emite una señal. Esa señal, de "ya he terminado", está representada por un IntentFilter. Entonces, para que nuestro receptor "esté pendiente de esa señal", necesito asociarle ese IntentFilter.

```
IntentFilter filter = null; //declaro el IntentFilter "la señal"
DescargaCompletaReceiver receiver = null; //declaro el receptor
//instancio la señal y el receptor
filter = new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE);
receiver = new DescargaCompletaReceiver(this);
//y los asocio así, donde this es el contexto
this.registerReceiver(receiver, filter);
```

Tras finalizar la descarga, el método *onReceive* será invocado, y allí puedo averiguar cómo ha finalizado la descarga –si ha fallado o ha ido bien– y gestionar cada caso, informando al usuario o actualizando la pantalla, como veremos en la sección práctica.

Hay un último detalle fundamental para garantizar el correcto funcionamiento de la descarga, y es que *onReceive*, debo realizar la operación inversa a *registerReceiver* invocando al método.

```
context.unregisterReceiver(this);
```

Así, este objeto receptor que ha tramitado la señal de descarga completada deja de estar escuchando. Si no lo hiciera, la siguiente vez que finaliza una descarga, se invocaría dos veces al método *onReceive*, lo que podría llevar a inconsistencias.

Veremos con detalle los intríngulis del proceso de descarga en el ejemplo práctico correspondiente.

## 11.6 ALARMMANAGER

---

Nuestro dispositivo realiza tareas de manera rutinaria y cíclica, sin que seamos conscientes de ello. Por ejemplo, para comprobar si tenemos nuevos mensajes de Whatsapp o un correo electrónico nuevo, las aplicaciones –WhatsApp y Gmail– lanzan de forma periódica una tarea, que realiza esa comprobación.

Aquí es donde nos ayuda la clase AlarmManager. Con este servicio de Android, podemos programar un reloj, de modo que salte una alarma periódicamente, y se inicie la ejecución de determinada acción. Podemos entender a AlarmManager como un programador de tareas.

Para su uso, primero obtengo el servicio AlarmManager a través del contexto.

---

```
AlarmManager am = context.getSystemService(Context.ALARM_SERVICE);
```

---

Y después uso el método *set* para programar la acción y el momento, cuya cabecera pasamos a analizar:

---

```
public void set (int tipo, long milisegundos, PendingIntent acción)
```

---

**long milisegundos.** Expresa una cantidad de tiempo en milisegundos, que es el tiempo que falta para que salte la alarma.

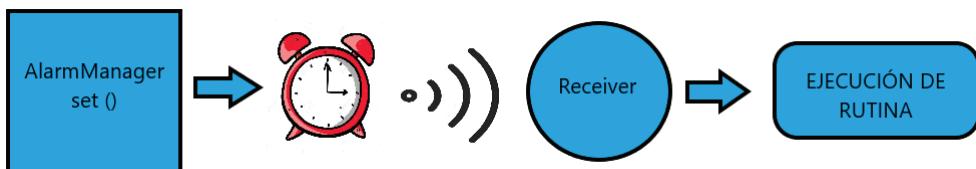
**int tipo.** Este parámetro asume una de las siguientes constantes:

- ▶ **ELAPSED\_REALTIME** o **ELAPSED\_REALTIME\_WAKEUP**. Los milisegundos expresados, son el tiempo transcurrido desde que el dispositivo se ha encendido–`SystemClock.elapsedRealtime()`–.
- ▶ **RTC** o **RTC\_WAKEUP**. Los milisegundos expresados, son interpretados como la fecha actual del Sistema –`System.currentTimeMillis()`–.

Las variantes con el sufijo WAKEUP, hacen que la alarma salte aun cuando el dispositivo está bloqueado.

- ▶ **PendingIntent acción.**– Este parámetro encapsula el receptor de la alarma, que es un BroadCastReceiver, declarado previamente en nuestra aplicación.

Por su importancia y uso en distintos escenarios, pasamos a detallar esta clase en la siguiente sección.



## 11.7 PENDINGINTENT

Pues bien, los PendingIntent son como un Intent, pero con una variante, y es que van a permitirnos lanzar un componente de nuestra aplicación (una Actividad, Servicio o Receptor), desde fuera de los límites de ésta.

En el caso anterior, cuando salte la alarma, la aplicación desde la que la programé no estará activa ni visible. Digamos que la alarma, "suena fuera de mi app", pero sin embargo, quiero que sea un Receiver mi app, el que reaccione. Pues eso, lo consigo mediante un PendingIntent.

Primero debo crear un Intent explícito, que alberga el BroadCastReceiver, receptor de la alarma.

```
Intent intentAlarm = new Intent(this.context, AlarmaReceiver.class);
```

Y después obtengo una instancia de PendingIntent a través del método estático `getBroadcast`, que tiene la siguiente cabecera:

```
public static PendingIntent getBroadcast (Context context, int requestCode, Intent intent, int flags)
```

El primero es el contexto, que puede ser `this` si estoy en una actividad.

El tercero es el Intent creado en el paso anterior.

El segundo y el cuarto son:

- ▶ requestCode.- Número arbitrario que identifica la petición que lanza el propio PendingIntent.
- ▶ flags.- Este parámetro responde a la pregunta, ¿si se ha lanzado ya un PendingIntent como éste, debe volver a crearse otro nuevo, se reutiliza el mismo –con la misma información en su interior–, se cancela el anterior y se crea otro, etc.? Los valores más habituales son alguna de entre estas constantes:
  - FLAG\_NO\_CREATE.- Si ya existe uno igual, no lo crea, sino que devuelve *null*
  - FLAG\_ONE\_SHOT.- Si ya se lanzó uno igual, no vuelve a lanzarse
  - FLAG\_UPDATE\_CURRENT.- Si ya existe, se lanza uno que actualiza el estado del Intent (la información en su interior, estado o "saquito")

Tras componer el PendingIntent, ya tengo el código completo para programar la alarma y que sea recibida en la clase AlarmaReceiver.

---

```
AlarmManager am = null;
PendingIntent pi = null;
Intent ia = new Intent(this.context, AlarmaReceiver.class);
am = (AlarmManager) this.context.getSystemService(Context.ALARM_SERVICE);
pi = PendingIntent.getBroadcast(this.context, 101, ia, FLAG_ONE_SHOT);
//en 5 segundos saltará la alarma
long tiempo = System.currentTimeMillis() + 5000;
am.set(AlarmManager.RTC_WAKEUP, tiempo, pi); //también puedo usar setExact
```

---

También se pueden emplear los PendingIntent al crear Notificaciones (son avisos externos que conducen a mi app) y para instar la ejecución de servicios. Lo veremos en el siguiente capítulo.

## 11.8 CONTENTPROVIDER

---

El ContentProvider o proveedor de contenidos, como su nombre indica en español, es un mecanismo que nos permite compartir información entre aplicaciones. Provee contenidos de una *app*, a otra.

Piensa por ejemplo en la aplicación de Contactos. ¿Cómo accedemos a la información de un contacto en nuestra agenda, desde otra aplicación? La aplicación de contactos consta de una base de datos interna, donde se almacena de modo relacionado, la información de nuestra agenda. Nosotros, desde fuera no podemos acceder directamente a esa base de datos (pues podríamos operar indebidamente), pero sí que podemos usar el ContentProvider correspondiente provisto.

Hay diversas aplicaciones que usan ContentProvider para exponer su datos a terceros. Entre ellas, destacan: Calendario, Navegador (Favoritos, Historial), Historial de Llamadas (CallLog), MediaStore (Multimedia fotos, videos), Contactos, Ajustes , Diccionario de usuario (Palabras propias).

Con un ContentProvider puedo:

1. Consultarlo
2. Insertar, modificar, borrar registros en él
3. Crear mi propio proveedor

Siendo la primera opción la más habitual.

Si quisiera declarar un ContentProvider en mi aplicación para proporcionar acceso a mis datos, debería declararlo en el Manifest. Aunque no vayamos a estudiarlo en principio, conviene que sepas que esta clase constituye el cuarto tipo de Componente en Android junto a Actividades, Servicios y Receptores y por tanto se debe registrar en el Manifesto en el caso 3.

Para poder usar el ContentProvider como fuente de datos en mi aplicación, debo declarar el permiso correspondiente en el archivo de Manifiesto. Por ejemplo, si deseo usar el Diccionario, debo añadir:

```
<uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

Cada proveedor indica el suyo en la documentación.

Cada ContentProvider es referenciado e identificado por una URI, una ruta de acceso que sigue el siguiente esquema:

```
<prefijo>://<autoridad>/<datos>/<id>  
content://app-package/tabla/id
```

Por ejemplo, el de Contactos es:

content://com.android.contacts/contacts

En realidad, esta URI es como la ruta a una base datos.

Para acceder al ContentProvider, lo hago a través de una clase llamada ContentResolver, que obtengo del contexto con esta simple llamada:

```
ContentResolver contentResolver = getContentResolver();
```

y el método que uso para consultar el ContentProvider es `contentResolver.query`, cuya sintaxis es esta:

```
Cursor query (Uri uri,  
              String[] columnas,  
              String condicion,  
              String[] argumentos,  
              String orden)
```

Donde los parámetros son :

- ▶ Uri uri: la ruta o Id del Content
  - ▶ String[] columnas: qué columnas quiero (null si quiero todas)
  - ▶ String condicion: WHERE filtrar filas
  - ▶ String[] argumentos: ? parámetros
  - ▶ String orden: ASC, DESC

Realmente, estamos operando con una especie de base de datos, pues los parámetros, así como el tipo devuelto –Cursor–, nos evocan claramente a una base de datos relacional (SQL). Ver el apartado de BASES DE DATOS RELACIONALES CON SQLITE en el capítulo de Persistencia.

Veamos un pequeño ejemplo de código:

```
contentResolver.query(uri_contactos,
    null,
    ContactsContract.Contacts.DISPLAY_NAME + " LIKE ?",
    {"%"},  
    null);
```

Mediante esta sentencia, estaría seleccionando todos los atributos -columnas- de todos los contactos que empiezan por M, sin tener en cuenta el orden.

Posteriormente, debo recorrer el cursor que me devuelve esta operación *query*, y así acceder a los resultados. Veremos un ejemplo completo en la correspondiente sección práctica.

## 11.9 FILEPROVIDER

Esta clase es un caso especial de ContentProvider, ya que hereda de ella. Igual que su padre, su misión es dar una protección a los datos propios de una aplicación, al mismo tiempo que los hace accesibles a otras.

Al final, los ficheros de una aplicación se hallan en un directorio de su propiedad. Pero si quiero que otra aplicación acceda, se deben cambiar los permisos de manera permanente o hacer y deshacer cambios en cada uso: el FileProvider es una respuesta a esta situación.

FileProvider me va a permitir declarar unas direcciones de forma privada, donde se van a almacenar los ficheros de mi aplicación, y hacer de puente para cuando un tercero vaya a necesitarlas, no les da esas rutas originales, sino facilitar una ruta lógica equivalente. En este sentido, FileProvider es una especie de traductor entre rutas físicas privadas y rutas lógicas públicas.

Para declarar el FileProvider no hace falta crear una clase. Basta declararlo en el Manifest.xml

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="com.example.miapp.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/ruta_provider" />
</provider>
```

Con este elemento, le doy un nombre lógico a mi FileProvider "mi.paquete.app.fileprovider" y referencia un archivo que debo crear en res/xml/ruta\_provider.xml y es como sigue:

```
.....  
<paths xmlns:android="http://schemas.android.com/apk/res/android">  
    <files-path name="mis_canciones" path="songs"/>  
</paths>  
.....
```

Es en este archivo asocio lo que será mi carpeta interna songs, con el nombre externo o lógico mis\_canciones.

De este modo, el FileProvider traduciría una ruta interna como esta:

```
.....  
file:///data/user/0/com.example.miapp/files/songs/musica.mp3  
.....
```

En otra como esta, al invocar al método *getUriForFile*

```
.....  
content://com.example.miapp.fileprovider/mis_canciones/musica.mp3  
.....
```

Supón que quiero que mi canción la reproduzca otra app. Me creo un intent y sobre esta ruta, concedo un permiso temporal de acceso del siguiente modo:

```
.....  
//fcancion es un objeto File que alberga la ruta original  
Intent intent = new Intent();  
intent.setAction(Intent.ACTION_VIEW);  
intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);  
Uri uri_c = FileProvider.getUriForFile(this,  
"com.example.miapp.fileprovider", fcancion);  
intent.setDataAndType(uri_c, "audio/mp3");  
//la actividad destino podrá así acceder a mi archivo mp3  
startActivity(intent);  
.....
```

Otro ejemplo clásico es el de la cámara de fotos: mi aplicación, le pide a la cámara que tire una foto. La cámara, que un proceso externo, dispara la foto, pero esa foto voy a querer almacenarla en una carpeta de mi aplicación. Ahí cobra sentido el uso del FileProvider. También si quiero que el DownloadManager descargue los archivos en carpetas privadas de mi aplicación, deberé usar un FileProvider.

Veremos un ejemplo combinado del uso del FileProvider y la cámara en el tema 13.

## 11.10 TEST TEMA 11

1. Para comprobar si un dispositivo tiene conectividad a Internet debo usar:
  - a) Un Actividad
  - b) Un Receptor
  - c) Un servicio del Sistema
  - d) Ninguna de las anteriores
2. Necesito declarar el permiso en el Manifest:
  - a) ACCESS\_NETWORK\_STATE
  - b) ACCESS\_NETWORK\_STATE
  - c) ACCESS\_NETWORK\_STATE
  - d) No necesito declarar ningún permiso
3. Con MediaPlayer puedo reproducir:
  - a) Sólo Archivos locales
  - b) Sólo remotos
  - c) Archivos remotos y locales
  - d) Archivos remotos
4. Respeto de DownloadManager:
  - a) Es un servicio del sistema
  - b) Puedo hacer algo equivalente programando un AsyncTask
  - c) Permite descargar archivos de cualquier tipo
  - d) Todas las anteriores son ciertas
5. ¿Necesito un Receptor vinculado al DownloadManager?
  - a) Sí, para saber cuándo empieza la descarga
  - b) No, es meramente opcional
  - c) Sí, para saber el progreso de la descarga
  - d) Sí, para saber cuándo finaliza la descarga

6. Con DownloadManager puedo descargar archivos de cualquier tipo:
  - a) No, sólo pdf o documentos
  - b) No, sólo archivos de imagen y video
  - c) Ciento
  - d) Ninguna de las anteriores
7. Al programar un nuevo BroadcastReceiver, debo declararlo en el Manifest:
  - a) De modo opcional
  - b) No es necesario
  - c) Es obligatorio
8. AlarmManager sirve para:
  - a) Realizar tareas al inicio del dispositivo
  - b) Realizar tareas repetitivas
  - c) Recibir avisos al finalizar una descarga
  - d) Ninguna de las anteriores son ciertas
9. Un PendingIntent es:
  - a) Un Actividad externa a mi aplicación
  - b) Lo mismo que un Intent explícito
  - c) Como un intent, pero permite lanzar un componente desde fuera de la aplicación
  - d) Ninguna de las anteriores
10. ContentProvider es un caso especial de FileProvider:
  - a) Verdadero
  - b) Falso

## SOLUCIONES

1c, 2d, 3c, 4d, 5d, 6c, 7c, 8b, 9c, 10b



# 12

---

## NOTIFICACIONES Y SERVICIOS

Las notificaciones sean seguramente el aspecto icónico y funcional más característico de los dispositivos móviles. En este capítulo veremos cómo crearlas.

Los servicios son procesos en segundo plano, que desarrollan una actividad a espaldas del usuario. Veremos qué tipos de servicios hay y cómo trabajar con ellos.

Programar servicios sea seguramente la frontera siguiente entre el programador Android junior y el más experto.

### 12.1 TAREAS PRÁCTICAS DEL TEMA 12

---

- ▶ Crear y personalizar notificaciones
- ▶ Programar algún servicio

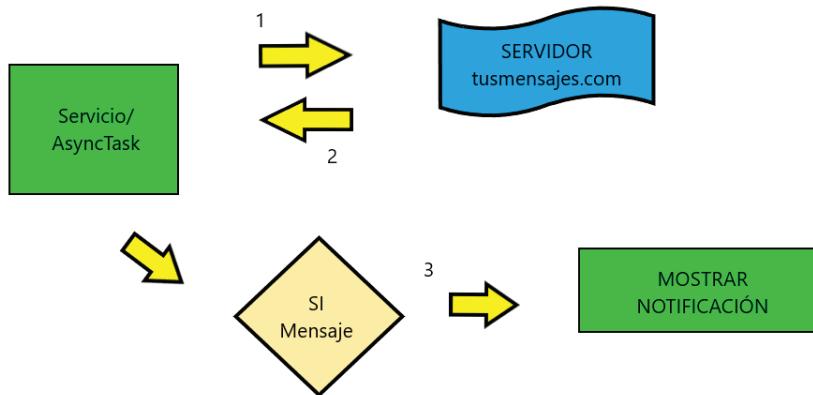
### 12.2 NOTIFICACIONES

---

Seguramente sea uno de los objetos visuales más característicos y reconocibles por los usuarios. Y es que la aparición de este elemento ha modificado un tanto los hábitos en que recibimos las noticias. Notificación, es sinónimo generalmente, de noticia o aviso más o menos relevante.

Esa funcionalidad de aviso que conlleva la mayoría de las notificaciones, hacen intuir al incipiente programador que la notificación es algo que nos llega vía Internet. Y en cierto modo, es así. Pero conviene aclarar que la notificación como tal, es generada de manera local, endógena al dispositivo y se construye de manera sencilla gracias al API provisto.

Por tanto, el ciclo completo de una notificación de Whatsapp, por ejemplo, sería el siguiente:



Un Servicio opera de manera transparente al usuario (en segundo plano) y él mismo o a través de un AsyncTask, lanza una interacción remota con un servidor al que consulta: (1)<<oye, soy Fulanito, ¿tengo algún mensaje nuevo?>>. Si el servidor tiene pendiente de entrega algún mensaje para él, se lo devuelve en la misma respuesta y en caso contrario, le dice, (2)<<no tengo nada nuevo>>. Generalmente, el servicio es lanzado periódicamente por un AlarmManager.

Al recibir un mensaje con contenido, se lanzaría una notificación con un código parecido al siguiente:

Lo primero es contar con el objeto NotificationCompat.Builder, que nos ayuda a crear y personalizar nuestra propia notificación de manera sencilla.

```

//esta clase nos ayuda a construir nuestra notificación
NotificationCompat.Builder nb = null;
nb = new NotificationCompat.Builder(context); //nos pide el contexto
nb.setSmallIcon(R.mipmap.ic_launcher); //le decimos qué ícono queremos
nb.setContentTitle("INFORME ENVIADO"); //el mensaje que mostrará
//con esta opción la notificación desparece al ser tocada
nb.setAutoCancel(true);

```

¿Y a donde vamos cuando la notificación es pulsada? A una Actividad que es lanzada por su intent correspondiente. Pero al ser la notificación un mensaje que se produce desde el exterior de nuestra aplicación y nos lleva adentro, usamos un PendingIntent que lo envuelva.

```

Intent resultIntent = new Intent(context, MainActivity.class);
resultIntent.putExtra("MENSAJE", "VENGO DE LA ALARMA");

```

Puesto que en este caso el PendingIntent envuelve a una Actividad y no a un Receptor, usamos el método estático getActivity, que recibe el contexto, un código de petición arbitrario, el intent y las opciones de creación de dicho intento o flags.

```
PendingIntent resultPendingIntent = PendingIntent.getActivity(context, 755,  
resultIntent, PendingIntent.FLAG_ONE_SHOT);
```

Ya tengo todo casi listo. Asocio el PendingIntent anterior a la notificación.

```
nb.setContentIntent(resultPendingIntent);
```

Y uso el servicio de notificación NotificationManager que es el que tiene la misión de dibujar la notificación que he ido creado mediante el método *notify*.

```
NotificationManager nm = null;  
nm =(NotificationManager)context.getSystemService(NOTIFICATION_SERVICE);  
//537 es un nº arbitrario como id de la petición de notificación  
nm.notify(537, nb.build());
```

Hay alguna característica novedosa como los NotificationChannel, desde el API 8 Oreo, que permite agrupar a las notificaciones bajo un canal, de manera que el usuario pueda decir, pues quiero recibir sólo avisos del canal fichajes, o noticias, o goles, pero su uso es algo residual y lo detallaremos en la sección práctica correspondiente.

## 12.3 SERVICIOS

---

Hay varios tipos de Servicios, según su funcionamiento y particularidades, pero hay una característica común y es que el servicio desarrolla una tarea que no es visual. Empecemos haciendo una clara clasificación entre Servicios del Sistema y servicios que podemos programar nosotros mismos.

## 12.4 SERVICIOS DEL SISTEMA

---

Los Servicios del Sistema son clases que podemos usar desde cualquier aplicación y nos ayudan a programar tareas muy diversas. Cada servicio se identifica por un nombre, y es accesible través del contexto con la llamada.

```
context.getSystemService (String nombre_servicio)
```

Donde el parámetro, es una constante que identifica al servicio solicitado y el objeto devuelto representa una instancia del mismo. A continuación, un listado de los más frecuentes y que hemos ido utilizado en capítulos anteriores:

NOMBRE (Constante Clase Context)	SERVICIO	USO
LAYOUT_INFLATER_SERVICE	LayoutInflater	Inflado de vistas
ALARM_SERVICE	AlarmManager	Programar alarmas
DOWNLOAD_SERVICE	DownloadManager	Descargas ficheros
NOTIFICATION_SERVICE	NotificationManager	Lanzar notificaciones
CONNECTIVITY_SERVICE	ConnectivityManager	Comprobar estado de la red

Debemos consultar el API específica de cada cual para poder usarlo.

Puede encontrar la lista completa de Servicios del Sistema en el API de Context.

## 12.5 SERVICIOS PROPIOS

Dentro de esta clasificación, tenemos las clases que nosotros mismos definiremos dentro del código. Éstas además, deberán declararse en el Manifest, puesto que los Servicios, son al igual que las Actividades y Receivers un tipo de Componente, –clases de un significado especial en para Android–.

Su uso es algo avanzado y menos frecuente, puesto que no todas las aplicaciones necesitan de Servicios Propios, si bien se adaptan a la necesidad de hacer una tarea repetitiva y periódica, sin necesidad de interfaz gráfica.

Ejemplos paradigmáticos de Servicios son un servicio que comprueba si tenemos un mensaje nuevo en un servidor, u otro que se dedica a ordenar una colección de datos cualquiera o transmitir estadísticas por Internet.

Como usuario, puedo acceder al Menú Ajustes y comprobar qué servicios están en ejecución en incluso forzar su detención. Veremos que aunque una aplicación no esté abierta, sí es habitual que esté ejecutándose algún servicio de ella en segundo plano o *background*.



El hecho que estas tareas sean opacas al usuario, unidas a que pueden ser repetitivas y complejas, han puesto a los servicios en el ojo del huracán de los creadores de Android, imponiendo restricciones a su uso y desarrollo desde el API 8 Oreo; ya que pueden mermar el rendimiento de los dispositivos de forma notable y silenciosa.

Básicamente hay cuatro tipos de servicios; a saber:

- ▶ Los Servicios Iniciados o *Started services*
- ▶ Los Servicios Enlazados o *Binded services*
- ▶ Los *Foregorund Services* o servicios de primer plano
- ▶ Y Los IntentService

### 12.5.1 Servicios Iniciados

Para usarlos declarar una clase nueva que herede de Service y programar ciertos métodos en el ella. Para declararlos usamos el asistente New-->Service-->Service; cosa que nos facilita la tarea y nos crea además la declaración en el Manifest del nuevo servicio, obligatorio por su condición de Componente.

El elemento `xml` es tal que así, al que puedo añadir un atributo `description`, donde detallo textualmente la función del mismo, que puede ayudar al usuario a entender su alcance:

```
.....  
    <service  
        android:name=".MyService"  
        android:enabled="true"  
        android:exported="true"  
        android:description="Comprueba el estado de la red">  
    </service>  
.....
```

Y la clase de Java asociada que hereda de `Service` queda así:

```
.....  
    public class MyService extends Service {  
        public MyService() {}  
    }  
.....
```

Para lanzar el servicio debo, desde una *Activity* o un *Receiver* hacer lo siguiente:

```
.....  
    Intent intent_serv = new Intent(context, MyService.class);  
    context.startService(intent_serv);  
.....
```

Esto deriva en la ejecución del método de `MyService` `onStartCommand` que debo programar.

```
.....  
    @Override  
    public int onStartCommand(Intent intent, int flags, int startId)  
.....
```

Donde los dos primeros parámetros referencian al Intent que lanza el servicio y el tercero, es el id o el número que ha dado Android a este servicio y que lo identifica.

Dentro de este método, ejecutamos las tareas que deseamos (no será visibles para el usuario). Conviene destacar que desde una clase que hereda de `Service` no puedo realizar conexión a Internet, pues se ejecuta en el hilo principal.

Cuando acabe, necesito llamar a `stopSelf` para detenerlo, pasándole ese id que recibí.

```
.....  
stopSelf(startId);  
.....
```

Eso a su vez provoca un *callback* al método `onDestroy` de la misma clase, que debo sobrescribir si quiero hacer algo al finalizar el servicios.

```
.....  
@Override  
public void onDestroy()  
.....
```

Un servicio debe ser detenido por el mismo u otro Componente mediante `stopService`, pero debe finalizarse explícitamente porque si no, se queda gastando recursos absurdamente.

También el propio Sistema puede decidir eliminar un servicio arbitrariamente, ante una necesidad de memoria, teniendo estos más posibilidades de ser eliminados que una Actividad, al desempeñar una actividad no visible para el usuario. Para esta eventualidad, está previsto el valor entero que el servicio devuelve, que debe ser una de las siguientes constantes:

START\_NOT\_STICKY //No se reinicia  
START\_STICKY //Se reinicia sin el Intent que lo lanzó  
START\_REDELIVER\_INTENT //Se reinicia con el intent que lo lanzó

### 12.5.2 Servicios en Primer Plano

Los *foregorund* services son servicios que se programan igual que los iniciados, pero cuya actividad va a tener visibilidad o reflejo en la pantalla. Por ello, el Sistema le dará una prioridad mayor a la hora de preservarlos, respecto de otros. El clásico ejemplo de servicio en primer plano es el reproductor musical, que al reproducir la música, va asociados a una notificación.

Para lanzar un foregorund service debo invocar dentro de `onStartCommand` al método:

```
.....  
startForeground (int id, Notification notification)  
.....
```

Con un id arbitrario (no puede ser cero) y una notificación creada a tal efecto. Veremos un ejemplo de cómo crear un reproductor musical, ejecutado por un servicio en la sección práctica.

### 12.5.3 Servicios Enlazados

Extraigo de la documentación oficial la siguiente cita:

"Crea un servicio de enlace cuando deseas interactuar con el servicio desde las actividades y otros componentes de tu aplicación o cuando quieras exponer algunas de las funciones de tu aplicación a otras aplicaciones a través de la comunicación entre procesos (IPC)."

En la práctica, la complejidad y alcance de estos, hace que sea de un interés residual para la mayoría de los programadores y sea más una utilidad para los desarrolladores del entorno Android como tal. Se lanzan desde el método *Context.bindService()*, pueden ser utilizados por varios Componentes y mueren al dejar de ser usados por todos ellos.

Este sería el método que deberíamos programar:

```
.....  
    @Override  
    public IBinder onBind(Intent intent) {}  
.....
```

No merece más comentario. En mis años como programador, no he tenido la necesidad de emplearlos.

### 12.5.4 IntentService

Los IntentService se desarrollan en un hilo independiente (como las AsyncTask y a diferencia de los Service), por lo que se prestan a tareas que requieran colectividad con Internet. Puedo lanzar desde ellos conexiones remotas.

Para crear un IntentService debo usar el asistente con New-->Service-->IntentService , lo que me crea el registro en el Manifest y una clase que hereda de IntentService.

```
.....  
    public class MyIntentService extends IntentService {  
.....
```

Debo crear un constructor, que es obligatorio que llame al constructor padre con un String. Normalmente se emplea el nombre de la clase.

```
.....  
public MyIntentService() {  
    super("MyIntentService");  
}  
.....
```

El asistente me crea bastante código basura. Pero realmente el único método que tengo que programar además del constructor es *onHandleIntent*. Donde desarrollo la funcionalidad íntegra que deseo acometer.

Estamos trabajando en otro hilo, por lo que puedo jugar con conexiones a Internet y otras tareas concurrentes.

```
.....  
@Override  
protected void onHandleIntent(Intent intent) {  
    try {  
        Thread.sleep(5000); //detengo 5 segundos este hilo  
    } catch (InterruptedException e) {  
        Thread.currentThread().interrupt();  
    }  
}  
.....
```

Como vemos, el método devuelve void. Al contrario que los otros servicios, no tengo un tipo especial que prever en caso de que se interrumpa. El Sistema finaliza el servicio al terminar la ejecución del método.

Para lanzarlos, igual que los servicios iniciados, basta con crear un Intent explícito y llamar con *startService*.

```
.....  
Intent intent = new Intent(this, MyIntentService.class);  
context.startService(intent);  
.....
```

Y hasta aquí la teoría de servicios. Como siempre, veremos en la sección práctica asociada la ampliación y profundización del capítulo.

## 12.6 TEST TEMA 12

1. Las notificaciones son mensajes que recibimos por Internet:
  - a) Verdadero
  - b) Falso
2. Para construir una notificación existe una clase prevista que es:
  - a) NotificationService
  - b) NotificationManager
  - c) NotificationCompatBuilder
3. Es necesario un PendingIntent para crear una notificación:
  - a) Verdadero
  - b) Falso
4. Marque la respuesta que más se aproxima a la definición de Servicio:
  - a) Un servicio es un tipo especial de Actividad
  - b) Un servicio hereda de un Receptor
  - c) Un servicio realiza una tarea no visual
  - d) Ninguna de las anteriores
5. Para acceder a los Servicios del sistema lo hago:
  - a) A través de una Actividad
  - b) A través del Contexto
  - c) Ambas son correctas
6. ¿Cuántos tipos de servicios propios hay?
  - a) 5
  - b) 4
  - c) 6
  - d) 3
7. Puedo ver los servicios en ejecución desde los Ajustes del dispositivo:
  - a) Verdadero
  - b) Falso

8. Los Servicios propios, al igual que los del Sistema, deben ser declarados en el Manifest:
  - a) Verdadero
  - b) Falso
9. Respeto de las prioridades entre componentes para Android:
  - a) Los servicios tienen mayor prioridad que una Actividad
  - b) Los servicios tienen la misma prioridad que una Actividad
  - c) Los servicios en primer plano tienen mayor prioridad que otros servicios
10. Sobre concurrencia, marque la correcta:
  - a) Todo servicio se ejecuta en un hilo independiente del de la interfaz de usuario
  - b) Un servicio iniciado se ejecuta en un hilo distinto
  - c) En un Intent Service puedo establecer una conexión a Internet, sin necesidad de llamar a una AsyncTask
  - d) Los servicios y las Actividades se ejecutan siempre en hilos distintos

## SOLUCIONES

1b, 2c, 3a, 4c, 5c, 6b, 7a, 8a, 9c, 10c



# 13

---

## PERIFÉRICOS Y APIS DE GOOGLE

La cámara de fotos y el GPS seguramente son dos de los elementos hardware más característicos de Android. Veremos cómo usar estos periféricos desde nuestras aplicaciones.

Por otra parte, Google ofrece un montón de servicios web a través de sus famosas APIs. Usaremos el API de mapas de Google para obtener un mapa y poder dibujar en él nuestra posición.

---

### 13.1 TAREAS PRÁCTICAS DEL TEMA 13

- ▶ Cómo tirar fotos, mostrarlas y almacenarlas
- ▶ Usar el GPS para obtener la ubicación del dispositivo
- ▶ Aprender a obtener un clave del Api de Google
- ▶ Usar el Api de mapas desde la aplicación
- ▶ Dibujar mapas, ubicaciones y obtener direcciones

---

### 13.2 CÁMARA

Prácticamente la totalidad de dispositivos Android lleva una cámara integrada. Para su uso, viene preinstalada una aplicación por el fabricante. Eso no impide que nosotros, como programadores, podamos usarla desde el código.

Hay dos alternativas:

1. Uso directo vía las clases del paquete API android.hardware.camera2.
2. Uso indirecto vía Intent.

Lo más usual es el modo indirecto. Sólo sería apropiado hacerlo por uso directo si la cámara es algo fundamental o quiero hacer una app de captura de vídeo.

Para la opción 2 debo lanzar un Intent implícito con la acción ACTION\_IMAGE\_CAPTURE si quiero tirar una foto o la acción ACTION\_VIDEO\_CAPTURE si quisiera hacer un vídeo. A ese Intent, debo añadir la ruta donde quedará almacenada la foto, mediante el método *putExtra* (MediaStore.EXTRA\_OUTPUT, URI);

Para generar la URI debo:

1. Generar un nombre de fichero nuevo (normalmente se usan atributos de la fecha y hora actuales para componer un nombre único).
2. Elegir un directorio donde irá ese fichero y crearlo si no existe. Lo más cómodo es usar Environment.getExternalStoragePublicDirectory (Environment.DIRECTORY\_DCIM), que nos da el directorio público donde la Cámara almacena las fotos.
3. Traducir esa ruta a URI mediante el método estático *Uri.fromFile(file)*.

Con la URI obtenida, puedo invocar la cámara mediante estas instrucciones:

```
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
Uri photoURI = obtenerURIFoto();
intent.putExtra(MediaStore.EXTRA_OUTPUT, photoURI);
//CÓDIGO_ACTIVIDAD es un código de petición arbitrario
startActivityForResult(intent, CODIGO_ACTIVIDAD);
```

Tras la ejecución de la captura se retorna la ejecución de la actividad en el método *onActivityResult*, recordad las subactividades vistas en el tema 6, cuya cabecera es:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
```

En este método puedo comprobar gracias al segundo parámetro si la captura fue bien y usar la URI que envíe, que ya contendrá la foto, para asignársela a un ImageView por ejemplo, mediante *setImageURI*.

Debo además declarar los permisos en el Manifest y pedirlos en ejecución, pues son de los llamados peligrosos.

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

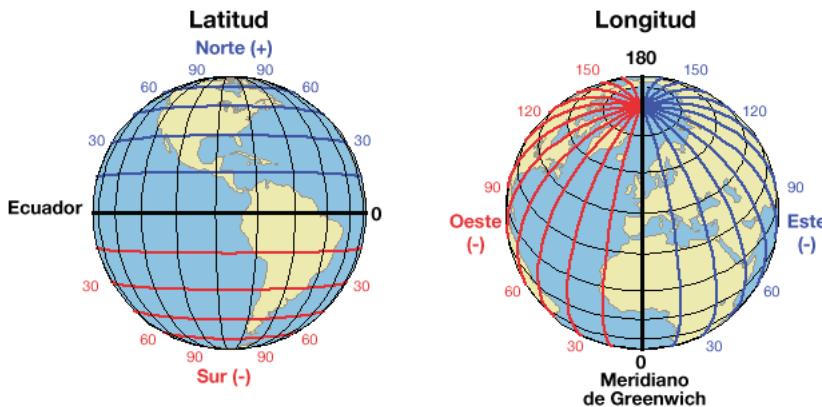
---

Si deseo guardar las fotos en una carpeta propia de la aplicación (ya sea la memoria interna o externa), debo además emplear un FileProvider. Veremos un ejemplo completo en la correspondiente sección práctica.

### 13.3 UBICACIÓN

---

La ubicación terrestre de un objeto viene dada por la latitud (la posición vertical respecto del Ecuador) y la longitud (la posición horizontal respecto del Meridiano cero), siendo la intersección de ambos ejes la posición relativa (0,0).



Así por ejemplo, Madrid se ubica en punto geográfico aproximado de  $40^{\circ}$   $-3^{\circ}$ .

La ubicación vendrá representada en Android por un objeto de la clase *android.location.Location*, cuyos atributos longitud y latitud, pueden ser accedidos por los respectivos métodos *getLongitud* y *getLatitude*. Pero ¿cómo llegamos a obtener un objeto Location con la información de la ubicación actual? Veamos.

Antes de nada, el conocer la ubicación de dispositivo requiere la aceptación expresa del usuario, por lo que hay que declarar los permisos en el Manifest, además de preguntar por ellos en ejecución (ver el apéndice de gestión de permisos):

```
.....  
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />  
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />  
.....
```

El primero habla de conocer la posición aproximada (*COARSE* del inglés, grueso), basándose en la ubicación de antenas de telefonía próximas, mientras que el segundo nos permite acceder a la posición exacta o fina, mediante GPS.

Para dispositivos con API mayor o igual a 10, necesito además el permiso *ACCESS\_BACKGROUND\_LOCATION* en caso de requerir acceso en segundo plano.

Las coordenadas pueden ser obtenidas desde un dispositivo Android mediante:

- ▶ Antenas de telecomunicaciones próximas –redes móviles–
- ▶ Puntos de acceso a internet inalámbricos –WiFi–
- ▶ El GPS (comunica con satélites en el espacio para determinar tu posición exacta)

Estas diferentes fuentes de ubicación están representadas como constantes en la clase *LocationManager*. (*NETWORK\_PROVIDER*, *GPS\_PROVIDER*) y junto con el servicio del sistema del mismo nombre, *LocationManager*, puedo comprobar si está o no disponible. Por ejemplo:

```
.....  
LocationManager lm = getSystemService(Context.LOCATION_SERVICE);  
if lm.isProviderEnabled(LocationManager.GPS_PROVIDER)  
{  
    //el usuario tiene la ubicación gps activada  
    obtenerUbicacion();  
} else {  
    //el usuario NO tiene la ubicación gps activada  
    pedirActivacionGPS();  
}  
.....
```

En la función *pedirActivacionGPS*, debo tirar un Intent que lleve al usuario a ajustes y active manualmente el acceso a la ubicación y recoger el resultado a la vuelta:

```
Intent i = null;
i = new Intent(android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS);
//500 código arbitrario de petición
startActivityForResult(viewIntent, 500);
```

---

A la vuelta, (recuerde las subactividades), puedo comprobar si quedó activo el acceso o no y obrar en consecuencia (desde salir, o repetir la petición, informarle o pasar a obtener su ubicación).

En este momento podemos tener la autorización de usuario y el GPS activado, pero aún no hemos obtenido la ubicación: vamos a ver cómo sería el método *obtenerUbicacion*.

Para ello necesito tres objetos más:

1. Una petición que defina los detalles de precisión y periodicidad
2. El objeto que resuelva dicha petición
3. El objeto invocado cuando la petición quede resuelta

Estas tres clases son respectivamente LocationRequest, FusedLocationProviderClient y LocationCallback.

Estas clases vienen fuera del SDK, y son provistas por Google en la librería: com.google.android.gms:play-services-location: 17.0.0 que debo importar en el fichero build.gradle (vea el apéndice de cómo importar librerías).

Un esbozo de *obtenerUbicacion* con estas clases sería el siguiente:

1. Preparo la petición: el objeto LocationRequest.

```
LocationRequest lr = LocationRequest.create();
lr.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
//precisión alta
lr.setInterval(5000); //quiero informarme cada 5 segundos
```

---

2. Instancio el objeto que atenderá esa petición: el FusedLocationProvider Client. Es la última clase diseñada por Google para obtener ubicaciones. Su nombre, del inglés fusionado o unido, es un objeto que nos va a procurar la ubicación con independencia de la fuente o proveedor (GPS, red móvil, Wi-Fi, histórico, etc.).

```
.....  
FusedLocationProviderClient flpc = null;  
flpc = LocationServices.getFusedLocationProviderClient(context);  
.....
```

3. Defino el objeto de la clase LocationCallback, que será quien reciba el aviso con la ubicación obtenida por la clase anterior.

```
.....  
LocationCallback locationCallback = new LocationCallback() {  
    @Override  
    public void onLocationResult(LocationResult locationResult)  
    {  
        Location location = locationResult.getLastLocation();  
        if (null != location) {  
            Log.d("MIAPP", "LATITUD = " + location.getLatitude());  
            Log.d("MIAPP", "LONGITUD = " + location.getLongitude());  
        }  
    };  
.....
```

4. Preparados la petición y el punto de vuelta, solicito la ubicación.

```
.....  
flpc.requestLocationUpdates(locationRequest, locationCallback, null);  
.....
```

El último parámetro, lo dejamos a null, pues carece de importancia.

Tras resolver la ubicación, la función *onLocationResult* de LocationCallback sería invocada, recibiendo la posición obtenida.

Recapitulando, hemos tenido que obtener los permisos en primer lugar, procurar la activación del GPS y finalmente obtener la ubicación. Son necesarios muchos pequeños pasos, por lo que un ejemplo práctico que ilustre esta sección se hace más que necesario.

Una vez obtenida la ubicación, tenemos las coordenadas, pero lo deseable es poder dibujarla en un mapa, e incluso saber qué dirección postal corresponde a

mi ubicación. Y no, no es magia. Lo podemos hacer fácilmente gracias a librerías de Google ya existentes, como veremos en el siguiente y último apartado del libro.

## 13.4 MAPAS DE GOOGLE

---

Para trabajar con mapas de Google, debemos importar la librería externa mediante gradle com.google.android.gms:play-services-maps:17.0.0. Ésta nos permitirá dibujar un mapa, poner un determinado nivel de zoom, añadir marcadores, etc.

Pero antes de que obtener el mapa, necesitamos comunicarnos con Google de un modo un tanto especial: a través de su API de Maps.

## 13.5 APIS DE GOOGLE

---

Las API de Google son un mecanismo provisto por Google de modo que expone sus servicios "en la nube", a los que tendremos acceso siempre que hayamos obtenido antes un código que nos dará el propio Google tras el registro.

Las API's de Google son cada vez más usadas por miles de programas, con una cantidad de propósitos enorme. Algunos ejemplos son:

- ▶ API Places: para conocer las valoraciones y comentarios de un local o servicio
- ▶ API Maps: en diferentes versiones para Android, IOS o JavaScript (web)
- ▶ API Gmail: para usar el correo
- ▶ API de YouTube: para acceder a datos de YouTube
- ▶ API de Traducción: para traducir idiomas en tu aplicación

Y hay muchas más. La idea siempre será la misma. Yo le pediré algo al API de Google, y él me lo devolverá. En nuestro caso le diremos: dame un mapa, y el API MAPS de Android me lo proveerá.

Lo primero para usar cualquier de estas APIS, es:

1. Crearse una cuenta en la Google Cloud Platform (con un correo de Gmail)
2. Después, creamos un proyecto dentro de la plataforma

- 
3. Y dentro de ese proyecto, habilitamos un API, proceso por el cual, obtenemos un código único, que debemos integrar en nuestra aplicación y se adjuntará en las conversaciones entre nuestro dispositivo y Google.

Ese código se denomina clave o en inglés *key* y en nuestro caso, (hay que seguir las cambiantes instrucciones de cada API en la página de Google) hay que declararlo en un fichero nuevo en /res/google\_maps\_api.xml con el siguiente contenido:

```
.....  
<resources>  
    <string name="google_maps_key" templateMergeStrategy="preserve"  
        translatable="false">AIzaSyAp9e0yV4na102J4gGAr38mB-8zasaPL-EU</string>  
</resources>  
.....
```

Siendo la clave ese chorro de bytes que empieza por "AIza..." –Aviso al lector que la expuesta es sólo un ejemplo y no es una clave válida–.

Una vez importada la librería y configurada la clave, se procede a definir el elemento visual que será el mapa, en el fichero de layout.xml. En este caso es un fragmento cuya clase es una de las importadas en la librería, llamada SupportMapFragment. Por ejemplo:

```
.....  
<?xml version="1.0" encoding="utf-8"?>  
<fragment xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:map="http://schemas.android.com/apk/res-auto"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:id="@+id/map"  
    android:name="com.google.android.gms.maps.SupportMapFragment"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    tools:context=".MapsActivity" />  
.....
```

Esto sería un mapa a pantalla completa. Y para dibujarlo, hay que programar dos cosas más en la actividad:

1. Hacer que nuestra clase implemente OnMapReadyCallback: así sobreescrivimos el método onReadyMap
2. Programar la llamada para obtener el mapa en el método onCreate()

Quedando así:

```
.....  
public class MapsActivity extends AppCompatActivity implements OnMapReadyCall-  
back {  
  
    private GoogleMap mMap;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_maps);  
        //solicitamos el mapa  
        SupportMapFragment mapFragment = (SupportMapFragment)  
            getSupportFragmentManager().findFragmentById(R.id.map);  
        mapFragment.getMapAsync(this); //llamamos al API  
        ...  
    }  
  
    //de vuelta de comunicarse con Google, este método será invocado  
    @Override  
    public void onMapReady(GoogleMap googleMap) {  
        //guardo una referencia para tener acceso al mapa  
        this.mMap = googleMap;  
        Log.d("MIAPP", "El mapa de google ha sido cargado con éxito");  
        ...  
    }  
.....
```

El mapa, ya habrá quedado representado. Ahora si quiero, puedo añadir un marcador en determinada posición y centrar el mapa con un nivel de zoom determinado. Ubiquemos por ejemplo el estadio del Real Madrid:

```
.....  
LatLang posicion_nueva = new LatLang(40.4530387d, -3.688333715d);  
mMap.moveCamera(CameraUpdateFactory.newLatLangZoom(posicion_nueva, 13));  
mMap.addMarker(new MarkerOptions().position(posicion_nueva)  
.title("Real Madrid"));  
.....
```

El nivel de zoom va entre 2 y 21, siendo 13 un rango intermedio aceptable.

Y el marcador icónico aparecerá en esa posición:



Por último, vamos a ver cómo dada una localización, podemos obtener a qué dirección postal corresponde gracias a la clase Geocoder del paquete android.location.

```
.....  
//this es el contexto y el segundo parámetro indica el lenguaje español  
Geocoder geocoder = new Geocoder(this, new Locale("es"));  
List<Address> dirs= geocoder.getFromLocation(40.453038d,-3.68833371d, 1);  
if (dirs!=null && dirs.size()>0)  
{  
    Address direccion = dirs.get(0);  
    Log.d("MIAPP", "Dirección = " + direccion.getAddressLine(0) + " CP "  
        +direccion.getPostalCode() + " Localidad "  
        +direccion.getLocality());  
}  
.....
```

Que mostraría la dirección del estado del Real Madrid:

Dirección = Av. de Concha Espina, 1, 28036 Madrid, España CP 28036  
Localidad Madrid.

De este sencillo modo, puedo obtener la dirección dada unas coordenadas.

Podríamos ver alguna cosa más avanzada, sobre cómo reacciona nuestra aplicación cuando pasa de primer a segundo plano, pero tenemos material suficiente para elaborar un ejemplo completo que aúne las dos últimas secciones: Obtener la ubicación y dibujarla en un mapa. Como siempre, acudir a la sección práctica para visualizar el ejemplo correspondiente.

## 13.6 TEST TEMA 13

---

1. Puedo obtener una foto lanzando un Intent explícito:
  - a) Verdadero
  - b) Falso
2. Para obtener las ubicación GPS de un dispositivo necesito habilitar el API Key de Google :
  - a) Verdadero
  - b) Falso
3. Es necesario usar un FileProvider si deseo almacenar las fotos capturadas en carpetas propias de la aplicación:
  - a) Verdadero
  - b) Falso
4. La obtención de permisos para usar la cámara consiste en:
  - a) Declarar en el Manifest los permisos relativos a la cámara
  - b) Pedir los permisos en tiempo de ejecución
  - c) Las dos respuestas anteriores son correctas
  - d) Ninguna de las anteriores

5. Respecto a las coordenadas de un dispositivo:
  - a) La latitud es relativa al Meridiano 0
  - b) La longitud es relativa al ecuador
  - c) Las dos respuestas anteriores son correctas
  - d) Ninguna de las anteriores
6. Respecto a los permisos de ubicación y la precisión de los mismos:
  - a) La aprobación del permiso ACCESS\_COARSE\_LOCATION permite una localización por GPS
  - b) La aprobación del permiso ACCESS\_COARSE\_LOCATION permite una localización por Redes de Telecomunicaciones o WIFI
  - c) La aprobación del permiso ACCESS\_FINE\_LOCATION permite una localización por Redes de Telecomunicaciones o WIFI
  - d) Ninguna de las anteriores
7. Para usar un mapa de Google en mi aplicación:
  - a) Debo descargar una librería adicional de google maps
  - b) Debo obtener una clave de acceso al API de Mapas de Google
  - c) Las dos anteriores son correctas
  - d) Ninguna de las anteriores
8. ¿Cómo se llama la clase que me permite traducir unas coordenadas a una dirección postal?
  - a) No hay una clase que realice esa traducción
  - b) GeoTranslate
  - c) GeoCoder
  - d) GeoStreet

## SOLUCIONES

1a, 2b, 3a, 4c, 5d, 6b, 7c, 8c

# Apéndice A

## IMPORTAR LIBRERÍAS Y PROYECTOS

### A.1 IMPORTACIÓN DE LIBRERÍAS DE TERCEROS

Cuando estamos programando, contamos con el API de Java y el SDK de Android, al que añadimos nuestras clases y confeccionamos así nuestra aplicación. Pero además, puedo reutilizar librerías de terceros, que hayan quedado dispuestas en otros repositorios.

Al inicio, los proyectos de Android tiraban de Maven Central. Posteriormente decidieron utilizar JCenter como repositorio de cabecera (el cual usa Maven por detrás) y más recientemente han definido uno propio de Google como prioritario. Esto aparece configurado en el fichero build.gradle del proyecto:

```
repositories {  
    google() //estos son los servidores donde descargamos dependencias  
    jcenter() // o librerías de terceros  
}
```

En el otro fichero build.gradle (dentro de la carpeta app) podemos encontrar la sección de dependencias, donde viene detallado los nombres de las librerías que usa nuestro proyecto (descargadas de los repositorios comentados).

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation 'androidx.appcompat:appcompat:1.1.0'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'com.google.code.gson:gson:2.8.6'
```

```
implementation 'com.google.android.material:material:1.1.0'
testImplementation 'junit:junit:4.13'
androidTestImplementation 'androidx.test.ext:junit:1.1.1'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
implementation 'com.android.volley:volley:1.1.1'
implementation ('com.google.android.gms:play-services-auth:18.0.0')
// {      exclude group: 'androidx'    }*/  
}
```

La primera línea configura el directorio libs como lugar donde depositar librerías de java manualmente (a la antigua) y así contar con ella en nuestro proyecto.

Para localizar librerías de moda, como por ejemplo, Picasso, y descargarlas de los repositorios configurados, simplemente debo entrar en el menú Build-->Edit libraries and dependencies, buscar por el nombre en el cuadro de diálogo y aceptar.

Al hacer esto, la línea se añadirá a la sección de dependencias y se descargará automáticamente una copia de la biblioteca descargada, pudiendo usar entonces las nuevas clases y métodos en mi proyecto.

```
implementation 'com.squareup.picasso:picasso:2.71828'
```

Si bien el uso de librerías de terceros puede ayudarnos en la rapidez del desarrollo, es mejor aprender primero bien el API estándar y usar con recelo y precaución el código de otros. Piensa que la propiedad intelectual del código es de los dueños de la biblioteca, lo cual puede generar desde una drástica evolución a conflictos de compatibilidades entre versiones o incluso a dejar de estar disponible.

Veremos en la sección práctica el manejo de esta potente opción.

## A.2 IMPORTACIÓN DE PROYECTOS DESDE GIT

Otra gran forma de aprender, es hacerlo a través de ejemplos. Hay multitud de proyectos y aplicaciones con su código fuente, que son publicados en repositorio públicos de Git como Github y que podemos usar libremente para propósitos académicos. Así puedo ejecutar, analizar y observar aplicaciones ya hechas o pequeñas demostraciones. El repositorio oficial de Android –<https://github.com/android>– nos puede servir para descargar aplicaciones dadas.

Para obtener una copia desde un repositorio público e importarla a mi equipo, simplemente debo ir al menú VCS-->Get from version control e introducir

la URI del proyecto que deseo copiar y seguir las indicaciones del asistente. VCS es el acrónimo de Sistema de Control de Versiones, del cual Git es un ejemplo.

Es muy posible que el proyecto que descargo arrastre dependencias de otro SDK, de otras versiones de Gradle e incluso del propio Android Studio, lo que puede ocasionarme no pocos quebraderos de cabeza antes de poder lanzar el ejemplo y trastear con él. Os recomiendo siempre tener actualizado vuestro entorno para evitar estos más que probables conflictos de configuración. Aunque por mi experiencia, ir a la última tampoco te garantiza el éxito.

Veremos en la sección práctica la importación de proyectos desde Git.



# Apéndice B

## EL CONTEXTO

### B.1 LA CLASE CONTEXT

La clase Context seguramente sea el concepto más utilizado y ambiguo en el API de Android. El Contexto es solicitado como parámetro en multitud de llamadas y me es necesario para cosas tan dispares como:

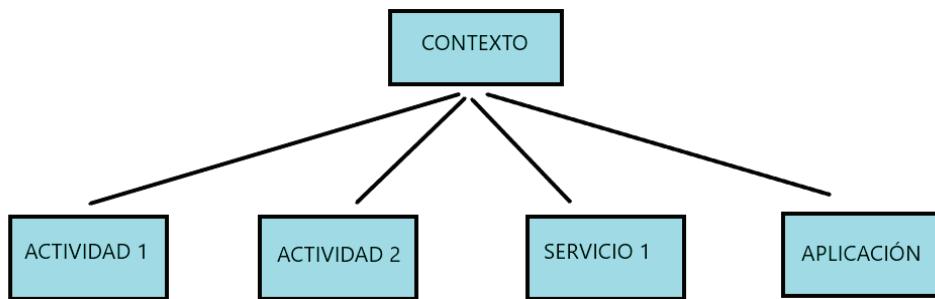
- ▶ Acceder a las vistas de una Actividad
- ▶ Acceder a un recurso (imagen, archivo, literal, etc.)
- ▶ Acceder a un Servicio del Sistema
- ▶ Acceder a ficheros de Preferencias y Bases de datos de la aplicación
- ▶ A la hora de lanzar una Actividad o Servicio
- ▶ A la hora de lanzar una notificación

Para Android puede entenderse que "contexto" equivale a "aplicación". Es decir, el motor de Android usa el contexto para distinguir a unas aplicaciones de otras ya que cada una tiene su propio espacio de memoria, sus archivos y sus recursos, de manera exclusiva. En este sentido, se produce una analogía con los contenedores de aplicaciones JEE (o servidores de aplicaciones web Java), donde se crea un contexto para cada aplicación en él levantada y así manejar de forma independiente el proceso que representa cada programa en ejecución.

Pero también el API de Android nos hace ver que el contexto, en ocasiones, equivale a una simple Actividad o Servicio. Entonces, ¿cuántos contextos hay?, ¿cuál debo usar?, ¿cómo accedemos a él? Respondamos a cada una de estas preguntas.

## B.2 NUMERO DE CONTEXTOS

Observemos el siguiente diagrama, donde tengo una aplicación que hay dos actividades y un servicio:



Cada Actividad y cada servicio son un contexto, puesto que son clases que heredan de Context. A su vez, y de forma automática, por cada aplicación se instancia una clase Application que hereda también de Context. Por tanto, en este ejemplo tengo cuatro contextos. Uno es de la aplicación y luego otro por cada componente.

De modo general, este escenario se repite siempre. Por lo que voy a tener siempre un contexto por la aplicación y otro por cada componente instanciado (Activity, Service o Receiver) en ella.

La pregunta siguiente es cómo acceder a los distintos contextos y cuál usar.

## B.3 ACCESO AL CONTEXTO

Dentro de una Actividad o Servicio, lo más habitual es usar la palabra reservada this. Ya que this es el propio objeto (es decir la actividad o el servicio) y este hereda de Context, tenemos acceso a él de modo sencillo y directo.

También desde un Componente puedo usar los métodos `getApplicationContext()` y `getSystemService()` con lo que estaría accediendo al Contexto de la aplicación.

Si necesito acceder a las vistas, o tema de una actividad, usará el del Componente. Por lo general, el de la Aplicación es apropiado. En cualquier caso puedo probar y usar uno u otro casi indistintamente.

Pero y qué pasa si estoy en una clase que no hereda de Context pero necesito acceder a él. Ahí tenemos un truco que vamos a explicar:

Supón que creas una clase nueva, sin heredar de Context o Activity. Y no tienes acceso al contexto, pero lo necesitas. Por ejemplo, para acceder a un literal. ¿Cómo haces? Pues bien, la técnica es sencilla:

1. Declaro una propiedad Context en la clase
2. Defino un constructor que acepte un parámetro de tipo Context
3. En el interior de ese constructor, asigno el parámetro del Contexto a la propiedad de la clase
4. Cuando necesite en la clase nueva el contexto, ya lo tengo "guardado" en el atributo de la propia clase

Ejemplo: MiController

```
public class MiController {  
  
    private Context contexto; //paso 1  
  
    public MiController (Context ctx) //paso 2  
    { this.contexto = ctx; } //paso 3  
  
    ...  
}
```



# Apéndice C

## GESTIÓN DE PERMISOS

Hay acciones de una aplicación que pueden invadir la privacidad o intimidad del usuario, de modo que antes de llevarse a cabo, necesitan una aprobación expresa por parte del mismo. Supón por ejemplo que una aplicación quiere acceder a la memoria externa. No sólo podría guardar y leer allí sus archivos, sino que podría leer todos los ficheros que residen en la memoria pública, incluyendo tus fotos. Querer saber la posición geográfica del dispositivo por su GPS, también puede ser considerado como potencialmente peligroso o invasivo.

Hasta el API 23, bastaba con declarar los permisos que la aplicación necesitaba en el fichero de manifiesto y al instalarse, Android preguntaba si aceptabas o concedías los permisos que la aplicación solicitaba. Si los concedías, la aplicación se instalaba. Si no, no.

Ejemplo de enumeración de permisos en el archivo de manifiesto, siempre con la etiqueta *uses-permission*.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="com.android.alarm.permission.SET_ALARM"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

Desde entonces, cambió la cosa y ahora los permisos se dividieron en dos: peligrosos y no peligrosos. Con estos últimos (como por ejemplo, usar Internet) no hace falta hacer nada especial: basta con declararlos en el Manifest. Pero los catalogados como peligrosos, deben además, solicitarse expresamente en tiempo de ejecución. Y ojo porque no basta con hacerlo una vez. Hay que controlar que el permiso esté vigente cada vez que vaya a necesitarse, porque el usuario puede en cualquier momento activar y desactivar individualmente los permisos de cada aplicación desde el menú Ajustes.

El listado de permisos peligrosos, agrupados según su tipo:

GRUPO DE PERMISOS	PERMISOS
CALENDAR	Read_Calendar Write_Calendar
CAMERA	Camera
CONTACTS	Read_Contacts Write_Contacts Get_Accounts
LOCATION	Access_Fine_Location Access_Coarse_Location
MICROPHONE	Record_audio
PHONE	Read_Phone_State Call_Phone Read_Call_Log Write_Call_Log Add_Voice_Mail Use_Sip Process_Outgoing_Calls
SENSORS	Body_Sensors
SMS	Send_Sms Receive_Sms Read_Sms Receive_Wap_Push Receive_Mms
STORAGE	Read_External_Storage Write_External_Storage

Si solicitamos un permiso y el usuario lo concede, habrá concedido automáticamente el permiso a todos los del grupo al que pertenece.

Por mi experiencia, seguir al pie de la letra la guía oficial es algo tedioso. Y hay un truco práctico, que consiste en pedirlo siempre. Si ya está concedido, Android no lo vuelve a pedir y así simplificamos el código.

Debemos realizar una llamada a este método:

```
.....  
void requestPermissions(context, String[] permisos, int cod_peticion)  
.....
```

Y esperar la respuesta con un callback sobre este método:

```
.....  
void onRequestPermissionsResult  
(int cod_peticion,  
String[] permisos,  
int[] resultados)  
.....
```

En el que recuperaremos el flujo de ejecución, con la respuesta del usuario obtenida. Lo veremos el ejemplo práctico asociado.



# Apéndice D

---

## UML

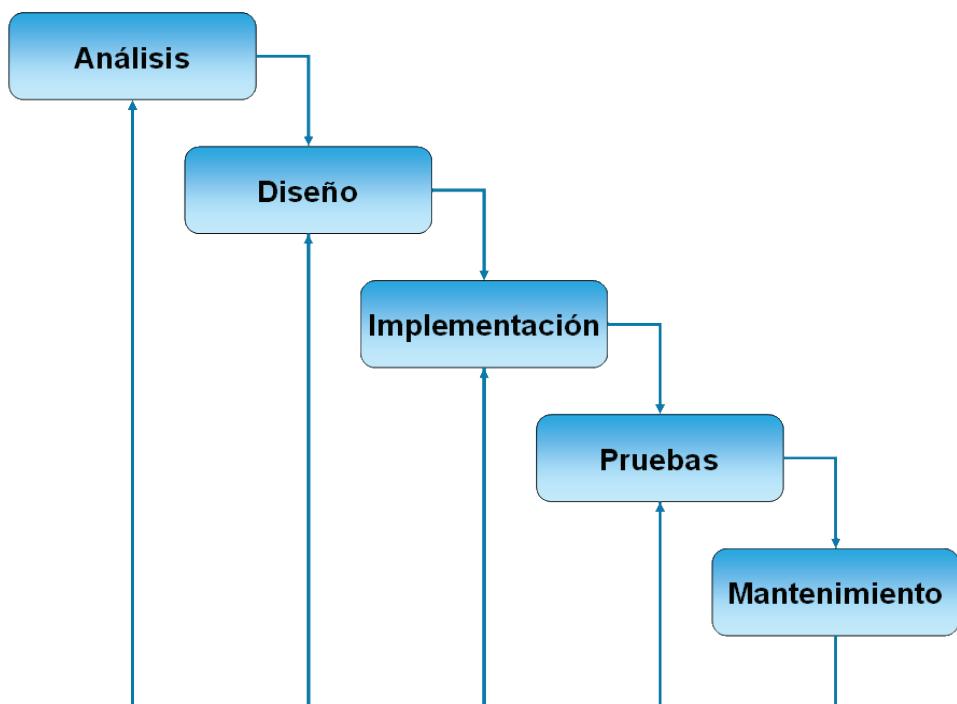
---

### D.1 LA NECESIDAD DE MODELAR

---

Piensa que hacer un programa es como un edificio. Si bien en la construcción hay un arquitecto, unos planos, un presupuesto y un proyecto de la obra que requiere de un seguimiento, cuando hablamos de hacer un software, nos enfrentamos a fases análogas. Debemos definir qué se quiere hacer, cómo, con qué recursos, plantear la solución, y llevarla a cabo. Por supuesto que no es lo mismo querer levantar un complejo residencial de mil viviendas, que hacer una caseta para el perro en el jardín. Es obvio que en función de la complejidad del producto, se debe optar por seguir más o menos estrictamente un protocolo de gestión y documentación. Pero resulta decisivo que para asegurar una calidad del proyecto, se lleve un cierto estudio previo y complementario al código en sí.

El Lenguaje Unificado de Modelado, UML de sus siglas en inglés, es una herramienta concebida como un estándar para cubrir toda la documentación a lo largo de un proyecto: desde su concepción hasta su materialización. A continuación el dibujo del ciclo de vida en cascada de un proyecto software:



La documentación UML de un proyecto, cuenta con las siguientes ventajas:

- ▀ Nos ayuda a definir y visualizar cada una de estas fases.
- ▀ Es una excelente herramienta de comunicación entre miembros de un equipo.
- ▀ Ayuda a plantear la solución.
- ▀ Obliga a pensar antes de hacer y por tanto a evitar y propagar errores.
- ▀ Una vez completada, representa una documentación inteligible por cualquiera que necesite evolucionar o corregir el software en el futuro.

¿Te imaginas que en un edificio se pica una tubería y ante la ausencia de planos, el fontanero se ve obligado a ir levantando el suelo a ciegas hasta que dé con la avería? Algo parecido le pasaría a un programador que necesite mantener un software sin la existencia de documentación: le tocaría empezar a leer y probar todo el código fuente.

No voy a desarrollar en esta humilde sección, toda la teoría que concierne al diseño y modelado de software con UML, pero sí capacitar a estudiantes que persiguen una rápida productividad, dándoles unas nociones de diagramas que considero básicos e imprescindibles antes de ponerse a codificar líneas como un loco.

El diseño es una rama transversal en el mundo de la programación y por tanto, tangencial a todo proyecto IT. Al que le interese profundizar, puede encontrar más información sobre UML en <https://sparxsystems.com/resources/tutorials/uml/dynamic-model.html>

## D.2 DIAGRAMAS UML

---

El lenguaje UML es un lenguaje formal. Es decir, se compone de una serie de símbolos y elementos gráficos con los que se construyen los diagramas que permiten describir y documentar el sistema. En UML hay dos grandes grupos de diagramas:

- ▶ Los ESTRUCTURALES: que permiten recoger la estructura, organización y componentes de una solución software. Es decir qué partes forman el proyecto y como se relacionan y despliegan.
- ▶ Los FUNCIONALES: que permite expresar cuál es el comportamiento funcional de la aplicación, detallando la operativa de los procesos y procedimientos que la componen, haciendo hincapié en las tareas más complejas.

Por lo dicho me parece sustancial abordar tres diagramas:

- ▶ El diagrama de casos de uso
- ▶ El diagrama de actividad
- ▶ El diagrama de navegación de pantallas

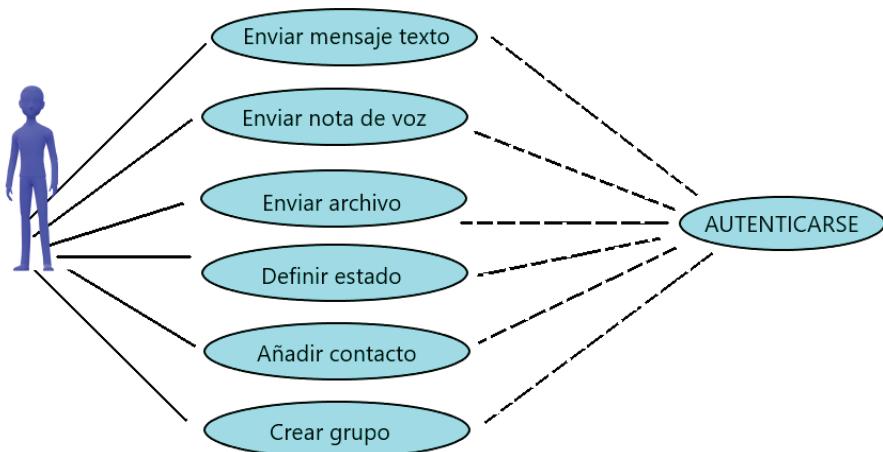
## D.3 DIAGRAMA DE CASOS DE USO

---

El diagrama de casos de uso describe qué hace el sistema desde el punto de vista del usuario. Gracias a él, podemos coleccionar a vista de pájaro, un dibujo que describe a grandes rasgos qué hace nuestra aplicación y qué espera el usuario de ella.

Una vez conseguido, pasaremos a descomponer y a analizar cada uno de estos casos de uso, pero de entrada, conseguimos un puzzle sencillo y completo que nos sirve de guía y resumen sobre las funcionalidades que debe contener nuestra aplicación.

Para ilustrar el diagrama de casos de uso, hagámoslo a una aplicación conocida por todos: el WhatsApp.



Analicemos el significado del diagrama:

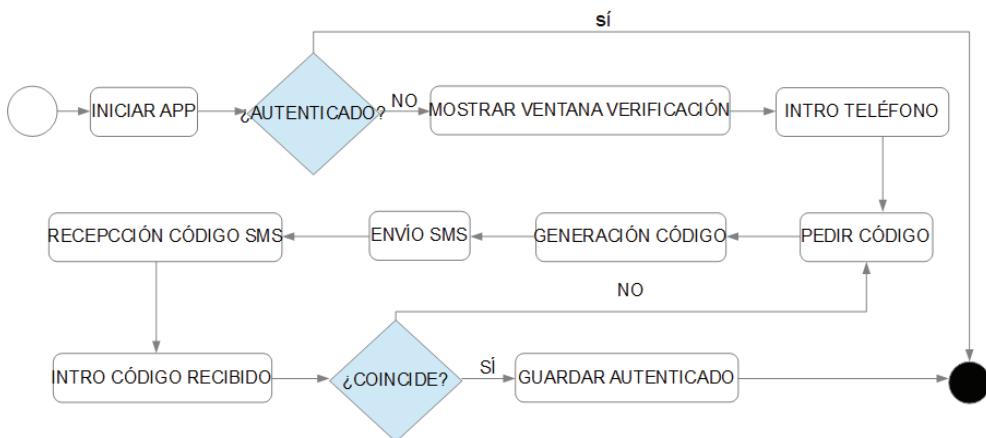
- ▶ El muñeco representa a un usuario. En este caso, sólo tenemos a un tipo de usuario, ya que no hay otro perfil tipo administrador, supervisor o comercial que pueda desempeñar otras funciones. En caso de darse, habría que reflejarlo con tantos muñecos como roles o tipos de usuario haya.
- ▶ Las viñetas ovaladas contienen una descripción de la acción que puede realizar el usuario. Como se observa, son casi las opciones de menú que más tarde podrá encontrar en la interfaz de nuestra aplicación, pero son sobre todo las operaciones que cobran sentido a un alto nivel desde el punto de vista del usuario final. Es decir, se omiten ordenaciones, grabado, transmisión de paquetes HTTP y todo detalle que vendrá estudiado después; pues la clave que contempla este diagrama es enumerar los usos de la aplicación desde el punto de vista del usuario.
- ▶ Las flechas unen las acciones con el usuario que puede realizarlas. Y hay un caso especial de flecha punteada, que se traduce por <<implica>>, como queriendo decir que la realización de esa acción (de donde parte la flecha punteada) es precedida implícitamente por la actividad donde va a parar la flecha. Quiere decirse, para poder enviar un mensaje, necesariamente has debido identificarte antes.

En el caso de WhatsApp, habría un diagrama más grande y con más casos de uso. Pero para hacernos la idea, nos basta esta versión reducida.

## D.4 DIAGRAMA DE ACTIVIDAD

Si ampliamos el foco en uno de estos casos de uso y nos centramos en él, podemos bajar el nivel de detalle y definir en qué consiste su realización de modo más concreto. Justo eso, es la tarea que llevamos a cabo con el diagrama de actividad.

El diagrama de actividad guarda un gran parecido con el diagrama de procesos de negocio con siglas BPD en inglés y nos sirve para detallar las secuencias de pasos en que se descompone una actividad o proceso de la aplicación. Pongamos un ejemplo para que se entienda mejor. Tomemos el proceso de Autenticación, definido como Caso de Uso en el diagrama anterior. Su diagrama de actividad sería algo así:



Pasemos a analizar brevemente el dibujo anterior:

- ▀ La actividad tiene un inicio y un final (punto blanco y relleno respectivamente).
- ▀ Las flechas sirven para indicar el flujo de ejecución de las acciones.
- ▀ Las acciones vienen descritas en un rectángulo de forma somera.
- ▀ Usamos un rombo si hay opción de bifurcar, indicando los casos.

Hay algunos elementos más, pero estos son los habituales.

Como se puede comprobar, el diagrama describe los pasos a seguir para solucionar el Caso de Uso y lo hace entendible a cualquiera. Así, para los programadores, es un punto de partida fantástico para continuar desgranando la solución de cada acción y acercarse al código con un sentido más fiel a lo que deben desarrollar y su alcance. Y para los ajenos a la programación, es un documento que les ayuda a entender, visionar e incluso supervisar la lógica y el desarrollo del proceso.

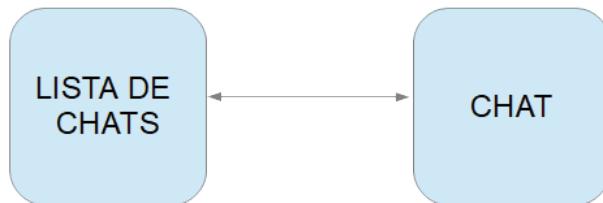
## D.5 DIAGRAMA DE NAVEGACIÓN DE PANTALLAS

Decidir qué pantallas forman tu aplicación y un boceto de su apariencia es algo esencial a la hora de dar forma al programa. Además, expresar cuales están conectadas y a qué pantalla puedo llegar desde otras, nos permite visualizar la navegabilidad de nuestra aplicación.

No es necesario detallar a este nivel los estilos o colores que quieren usarse, pero sí decidir qué elementos gráficos o controles habrá y su disposición en la pantalla. Tampoco es conveniente pensar y bocetar todas las pantallas desde un principio. Pero sí al menos las que están relacionadas y abarcan una funcionalidad común.

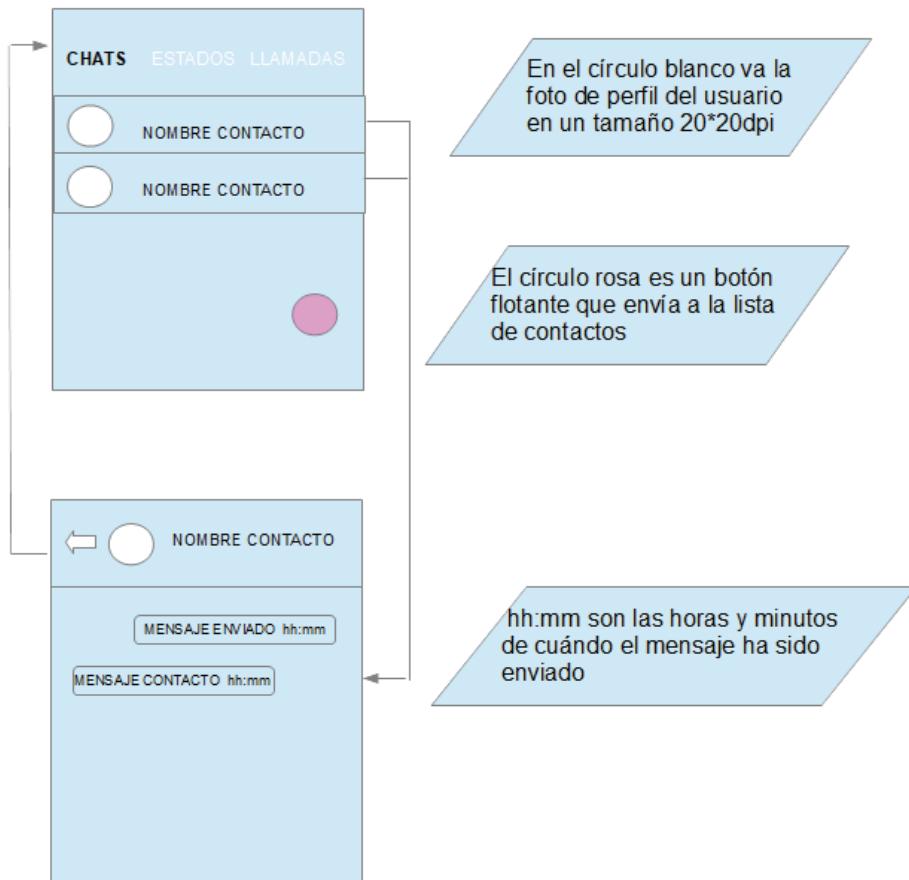
Con esto estamos reflejando cómo quedará expresada la funcionalidad de la aplicación a ojos del usuario final, lo que nos acerca a materializar el programa con mayores garantías de éxito.

Sirva de ejemplo las pantallas de conversación de WhatsApp.



Los rectángulos representan cada una de las actividades/pantallas.

Las flechas que conectan a las pantallas desde que puedo transitar y el sentido.



Los rectángulos se emplean para añadir comentarios a las pantallas y son válidos en cualquier diagrama del lenguaje UML para aclarar, matizar o ampliar el significado de un caso especial o símbolo.



# Apéndice E

## PUBLICACIÓN DE UNA APP

Para la publicación de una aplicación en la tienda oficial de Android –Google Play Store– necesitamos dar lo siguientes pasos:

- ▶ Obtener una cuenta de desarrollador de Google, para lo cual deberás confirmar un único pago de unos 25 \$ USD. Esto nos da derecho a publicar en la tienda por tiempo indefinido y sin límite de aplicaciones.
- ▶ Obtener una versión firmada del ejecutable APK. Para lo cual, crearemos un fichero (llamado almacén de claves) que contendrá la clave (algo así como nuestra firma). Tanto este fichero como la clave que contiene están protegidos por sendas contraseñas que debemos definir. Será imprescindible su uso para actualizar nuestra aplicación. Mucho ojo con no perderlo.

No entraremos en mayor detalle, puesto que nos basta el conocimiento funcional. Pero sí destacar, que el proceso de firma digital permite garantizar tanto la integridad como la autenticidad y la autoría de un archivo que ha sido firmado.

- ▶ Acceder a la Consola de Desarrollador de Google y llenar un formulario con los datos que nos piden, donde subiremos el archivo apk firmado en el punto anterior. Tras un periodo de revisión, será publicada.

El procedimiento ha ido variando ligeramente y han surgido alternativas al modo de firmar las aplicaciones y al contenido que subimos a la tienda. Pero básicamente se puede resumir en los puntos expuestos. Veremos un ejemplo de subida a la tienda en la sección práctica correspondiente.

Por último, destacar que hay otras páginas donde podemos publicar nuestra aplicación, al margen la de la oficial de Google y que pueden presentar menos restricciones o exigencias, tales como F-Droid o APKPure entre otras.

# Apéndice F

## LÍNEAS FUTURAS

El contenido de esta obra capacita plenamente a todo programador a construir y a colaborar en cualquier tipo de aplicación Android, así como supone una base idónea para afrontar las certificaciones de desarrollador Android existentes con garantías.

Pese a ello, cualquier programador, como profesional de una materia en constante evolución, debe actualizarse y aspirar a mejorar y completar sus conocimientos de modo constante. Es por ello por lo que dejamos aquí una evolución lógica en lo que sería ampliar las capacidades sobre Android una vez completada esta obra:

- ▶ Testar aplicaciones Android: lo que incluye una amalgama amplia de tecnologías, algunas de ellas aún no demasiado maduras, como: Junit, Thruth, Espresso y el API androidx.test
- ▶ Aprendizaje de librerías no estándares pero ampliamente usadas como: Picasso, Volley, Glide, Retrofit (usadas en HTTP y tratamiento de imágenes principalmente)
- ▶ Aprender Kotlin, que ha sido introducido como lenguaje de nueva generación en competencia de Java y que aporta más características de la programación funcional, lo que permite principalmente, economizar el código –a costa muchas veces de hacerlo menos legible–. Ojo con las modas. Kotlin no es ninguna panacea. Y un programador Java puede perfectamente continuar desarrollando en Android bajo Java.



---

## MATERIAL ADICIONAL

El material adicional de este libro puede descargarlo en nuestro portal web:  
<http://www.ra-ma.es>.

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

### INFORMACIÓN ADICIONAL Y GARANTÍA

- ▀ RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- ▀ Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- ▀ RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- ▀ Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

