

Módulo 8

Programación multimedia y dispositivos móviles

```
function updatePhotoDescription() {  
    if (descriptions.length > (page * 9) + (currentImageSubstring() - 1)) {  
        document.getElementById('bigImageDesc').innerHTML = descriptions[page * 9 + (currentImageSubstring() - 1)];  
    }  
}  
  
function updateAllImages() {  
    var i = 1;  
    while (i < 10) {  
        var elementId = 'foto' + i;  
        var elementIdBig = 'bigImage' + i;  
        if (page * 9 + i - 1 < photos.length) {  
            document.getElementById(elementId).src = 'images/min/' + photos[page * 9 + i - 1];  
            document.getElementById(elementIdBig).src = 'images/max/' + photos[page * 9 + i - 1];  
        } else {  
            document.getElementById(elementId).src = 'images/min/default.jpg';  
            document.getElementById(elementIdBig).src = 'images/max/default.jpg';  
        }  
        i++;  
    }  
}
```

UF1: DESARROLLO DE APLICACIONES PARA DISPOSITIVOS MÓVILES..... 4

1. Análisis de tecnologías para aplicaciones en dispositivos móviles.	4
1.1. Limitaciones que plantea la ejecución de aplicaciones en dispositivos móviles. ...	4
1.2. Entornos integrados de trabajo.	5
1.3. Módulos para el desarrollo de aplicaciones móviles.	7
1.4. Emuladores.	8
1.5. Configuraciones.	8
1.6. Perfiles.	9
1.7. Ciclo de vida de una aplicación.	10
1.8. Modificación de aplicaciones existentes.	14
1.9. Utilización de entornos de ejecución del administrador de aplicaciones.	15
2. Programación de dispositivos móviles.	15
2.1. Herramientas y fases de construcción.	15
2.2. Interfaces de usuario. Clases asociadas.	17
2.3. Contexto gráfico. Imágenes.	18
2.4. Eventos.	21
2.5. Técnicas de animación y de uso.	22
2.6. Descubrimiento de servicios.	23
2.7. Bases de datos y almacenamiento.	25
2.8. Persistencia.	27
2.9. Modelo de hilos.	32
2.10. Comunicaciones. Clases asociadas. Tipos de conexiones.	35
2.11. Gestión de la comunicación sin hilos.	37
2.12. Envío y recepción de mensajes de texto. Seguridad y permisos.	38
2.13. Envío y recepción de mensajería multimedia. Sincronización de contenidos. Seguridad y permisos.	40
2.14. Tratamiento de las conexiones HTTP y HTTPS.	40

UF2: PROGRAMACIÓN MULTIMEDIA 42

1. Uso de librerías multimedia integradas.	42
1.1. Conceptos sobre aplicaciones multimedia.	42
1.2. Arquitectura de la API utilizada.	42
1.3. Fuentes de datos multimedia. Clases.	43
1.4. Datos basados en el tiempo.	44
1.5. Procesamiento de objetos multimedia. Clases. Estados, métodos y eventos. ...	47
1.6. Reproducción de objetos multimedia. Clases. Estados, métodos y eventos.	48

UF3: DESARROLLO DE JUEGOS PARA DISPOSITIVOS MÓVILES..... 51

1. Análisis de motores de juegos.	51
1.1. Conceptos de animación.	51
1.2. Arquitectura del juego: componentes.	52
1.3. Motores de juegos: tipos y utilización.	53
1.4. Áreas de especialización, librerías utilizadas y lenguajes de programación.	55

1.5.	Componentes de un motor de juegos.	56
1.6.	Librerías que proporcionan las funciones básicas de un motor 2D/3D.	58
1.7.	API gráfico 3D.....	58
1.8.	Estudio de juegos existentes.....	59
1.9.	Aplicación de modificaciones sobre juegos existentes.	60
2.	Desarrollo de juegos 2D y 3D.....	60
2.1.	Entornos de desarrollo para juegos.	60
2.2.	Integración del motor de juegos en entornos de desarrollo.	62
2.3.	Conceptos avanzados de programación 3D.	63
2.4.	Fases de desarrollo.....	64
2.5.	Propiedades de los objetos: luz, texturas, reflexión y sombras.	66
2.6.	Aplicación de las funciones del motor gráfico. Renderización.	66
2.7.	Aplicación de las funciones del grafo de escena. Tipos de nodos y de su utilización.	67
2.8.	Análisis de ejecución. Optimización del código.	68
BIBLIOGRAFÍA		70

UF1: Desarrollo de aplicaciones para dispositivos móviles

1. Análisis de tecnologías para aplicaciones en dispositivos móviles.

1.1. Limitaciones que plantea la ejecución de aplicaciones en dispositivos móviles.

A la hora de desarrollar una aplicación para móviles existen algunos requisitos que son necesarios tener en cuenta. Cada plataforma plantea una serie de requerimientos que son necesarios a la hora de ejecutar una aplicación correctamente.

Estos son algunas de las consideraciones que hay que verificar antes:

- Los dispositivos móviles Android, generalmente, son móviles de tamaño pequeño o mediano y portables, por lo que el hardware tiene una limitación. Esto implica que a la hora de procesar la velocidad y capacidad de carga no sea tan alta como las aplicaciones para equipos de escritorio.
- La interfaz de usuario ha de ser ajustada a tamaños de pantalla pequeños, que obligan a organizar de una forma adecuada el contenido visible.
- La mayor parte de las aplicaciones requiere de una conexión de datos (4G, 3G, Internet) que está limitada por su ancho de banda. Pudiendo, en ocasiones, suceder que la señal de recepción de un dispositivo sea intermitente o nula.
- Un apartado importante, y que en muchas ocasiones no se tiene en cuenta, es la seguridad. El uso de estas aplicaciones que requieren de itinerancia de datos implica estar expuesto a un mayor número de problemas de seguridad, como, por ejemplo, virus. Ejecución de servicios por parte de terceros que están fuera del control del usuario. Aplicaciones que tienen permisos de acceso a datos personales almacenados en el teléfono, etc.

- Cada dispositivo tiene unas determinadas características de *hardware*. La memoria de los dispositivos puede variar dependiendo del fabricante. Esto puede dar como resultado una mala ejecución de una aplicación, incluso no permitiendo su instalación por no cumplir con los requerimientos mínimos.
- Otro de los mayores problemas que existen hoy en día es el consumo de batería por parte de algunas aplicaciones. Esta tarea es muy importante ya que es necesario optimizar el consumo de recursos de una aplicación.
- Almacenamiento: El almacenamiento interno, al igual que la memoria, son componentes *hardware* que dependen del fabricante de cada dispositivo móvil.

Por lo que una buena práctica es tratar de optimizar el código de la aplicación lo máximo posible, haciendo que el tamaño de esta no sea excesivamente pesado.

Además de estos, existen algunas limitaciones propias de Android. Cuando se va a desarrollar una aplicación que está pensada para ser comercializada, es importante establecer el rango de dispositivos para los que la versión de compilación de la aplicación será aceptada. Estableciendo cuál será la versión mínima y recomendada para su funcionamiento.

1.2. Entornos integrados de trabajo.

Existen diversas plataformas de desarrollo de aplicaciones Android hoy en día, que proporcionan a los desarrolladores todas las herramientas necesarias para poder crear aplicaciones para los dispositivos móviles. Estas plataformas reciben el nombre de **IDE** (*Integrated Development Environment*). Dos de los más destacados y usados con mayor frecuencia son **Eclipse** y **Android Studio**. La elección del IDE es una cuestión subjetiva, que en la mayoría de los casos viene determinada por el propio desarrollador de la aplicación.

A continuación, se mostrará una comparativa entre ambos IDE:

- **Eclipse:**
 - Es una plataforma muy potente con una gran cantidad de herramientas de depuración.

- Eclipse forma parte de una distribución de Java, cuyo lenguaje es la base de Android.
- Cuenta con una gran cantidad de plugins (ADT Plugin) con las herramientas de desarrollo de aplicaciones Android.
- Integración completa con el SDK manager desde el IDE con todo lo necesario para la instalación de todas las versiones de Android.



- **Android Studio:**

- Es la plataforma recomendada por la mayor parte de los desarrolladores.
- Es puramente Android, puesto que esta creado para desarrollar aplicaciones de este lenguaje.
- Permite una instalación de plugins e integración con mayor facilidad que eclipse.
- La compilación y exportación de los ".apk" es más fácil.
- Cuenta con la garantía de soporte de Google, ya que eclipse dejará de tener este soporte en el plugin de Android.
- Optimiza mejor los recursos destinados a los emuladores de Android.



Una vez valoradas las ventajas e inconvenientes de cada uno, el IDE utilizado para el desarrollo de aplicaciones será Android Studio. Este cuenta con mayores garantías de futuro dentro del desarrollo de aplicaciones móviles. Android es un lenguaje ya consolidado, que ha estado en continuo desarrollo durante los últimos años. Se ha expandido por todo el mundo siendo el sistema operativo más usado en dispositivos móviles. Ya antes de este crecimiento, durante el desarrollo de las primeras versiones, se empezó a documentar el lenguaje. Facilitando de esta forma la creación de aplicaciones para los desarrolladores.

1.3. Módulos para el desarrollo de aplicaciones móviles.

Antes de comenzar a desarrollar es necesario instalar el IDE correspondiente. En este caso Android Studio requiere de unas características previas antes de comenzar su instalación.

- **Sistema Operativo:** Windows 7/8/10 (32 o 64 bits).
- **Memoria RAM:** 2 GB mínimo (8 GB de RAM recomendado).
- **Disco Duro:** 2 GB de espacio libre mínimo (4 GB recomendado).
- **Resolución mínima:** 1.280 x 800.
- **Versión de Java:** Java 8.
- **Procesador:** 64 bits y procesador Intel (con tecnología de virtualización).

Android Studio realiza la compilación de las aplicaciones con el lenguaje Java. Por tanto, es necesario instalar el **JDK** (*Java Development Kit*) que provee las herramientas de desarrollo para aplicaciones Java.

Esta herramienta se puede encontrar en la web de Oracle, donde será necesario descargarla y posteriormente instalarla.

Una vez instalado esto, el siguiente paso es instalar Android Studio. Durante la instalación será necesario instalar un módulo fundamental para el desarrollo. Este es el SDK Manager para poder descargar todos los paquetes necesarios para la compilación de aplicaciones dentro del IDE.

Otro módulo que será necesario instalar es la máquina virtual de java (JVM). Esta será capaz de interpretar el lenguaje compilado java. Se encargará de ejecutar todas las instrucciones para ser emulado.

1.4. Emuladores.

Android proporciona una herramienta que permite comunicarse con un emulador o un dispositivo conectado. Esta se conoce **ADB** (*Android Debug Bridge*), permite realizar cualquier acción como si de un dispositivo físico se tratara, permitiendo depurar cualquier aplicación. El emulador propio de Android es **AVD** (*Android Virtual Device*), que permite emular todas las características de dispositivos Tablet, Android Wear, televisiones y móviles. Además del emulador propio de Android existen otros que realizan estas mismas acciones. Uno de los más utilizados y que mayor flexibilidad permite es Genymotion.

En este caso el emulador escogido será Genymotion. Se trata de un emulador muy eficaz, que además permite una integración completa con Android Studio a través de su plugin. Es posible elegir las características de un dispositivo concreto a emular muy rápidamente, incluso pudiendo crear varios dispositivos con diferentes versiones de Android.

1.5. Configuraciones.

-
- **Tipos y características:**

Genymotion permite emular incluso sensores y otras características propias de un dispositivo como son:

- **Batería:** al activar esta característica se permite ver el comportamiento del teléfono cuando la batería es baja, o cuando se muestran las ventanas de aviso.
- **GPS:** activar o desactivar el GPS haciendo uso de unas coordenadas específicas.
- **Cámara:** hacer uso de la cámara frontal o trasera del dispositivo.
- **Archivos del sistema:** acceder al sistema de ficheros del dispositivo.
- **Control Remoto:** manejar desde otro dispositivo físico el comportamiento del emulador.
- **Identificadores:** es posible ver y modificar el identificador de Android y del dispositivo.
- **Red:** controlar si se habilitan la red de datos del dispositivo o si permanece sin conexión a internet.
- **Llamadas:** emular el comportamiento del dispositivo al realizar una llamada telefónica o un mensaje de texto.

- **Dispositivos soportados.**

Genymotion permite crear emuladores de diferentes dispositivos, como son: Google 4, Google Nexus 5, Google Nexus 7, Samsung Galaxy S3, etc. Permite también poder emular un dispositivo totalmente personalizado, configurando el número de procesadores, tamaño de la memoria, resolución de pantalla, conexión a internet, interfaz de barra de navegación y versión de Android.

Su instalación y configuración es muy sencilla, basta con seguir los pasos que ofrece Genymotion en su documentación.

Su versatilidad permite que se puedan integrar con los principales IDE de desarrollo como son Android Studio y Eclipse a través del plugin que existe para cada uno de ellos.

1.6. Perfiles.

- **Características.**

Los emuladores ya cuentan con unos perfiles hardware ya definidos. De esta forma, si un dispositivo de los que ya están creados encaja con las características del dispositivo que se desea emular, no haría falta personalizar un nuevo perfil. Estos perfiles no se pueden modificar, pues estos ya están incluidos dentro del AVD.

La información que este tipo de perfiles muestra es la siguiente:

- **Nombre del dispositivo.**
- **Tipo de dispositivo:** si este dispositivo es Tablet, Android wear, móvil o Android tv.
- **Tamaño de la pantalla:** tamaño físico que tendría esta pantalla en pulgadas.
- **Resolución de pantalla:** resolución máxima que tiene el dispositivo. Esto se mide en número de píxeles.
- **Memoria RAM:** tamaño de la memoria RAM del dispositivo.
- **Entradas:** si el dispositivo cuenta con teclado hardware externo. Si esto se selecciona, el dispositivo no mostrará un teclado integrado, sino que recogerá los eventos de teclado del equipo.
- **Estilo de navegación:** modo en que se controlará el dispositivo.
- **Cámaras:** si tiene cámara frontal y posterior.
- **Sensores:** giroscopio, acelerómetro, GPS, etc.
- **Máscara:** control de la apariencia del dispositivo.

- **Arquitectura y requisitos.**

Para poder hacer uso de estos perfiles ya definidos es importante tener en cuenta algunos requisitos. Estos vienen ya definidos dentro de la documentación de Android. El requisito fundamental es que el equipo permita, por sus características de hardware, la emulación de dispositivos y la virtualización. Para un mayor detalle se tiene que consultar la documentación ofrecida por Android en su página oficial.

- **Dispositivos soportados.**

Todos los tipos de dispositivos son soportados y cuentan con un perfil dentro del AVD de Android. Es posible hacer uso como ya se ha comentado de Tablets, Android Wear, Móvil o Android TV.

1.7. Ciclo de vida de una aplicación.

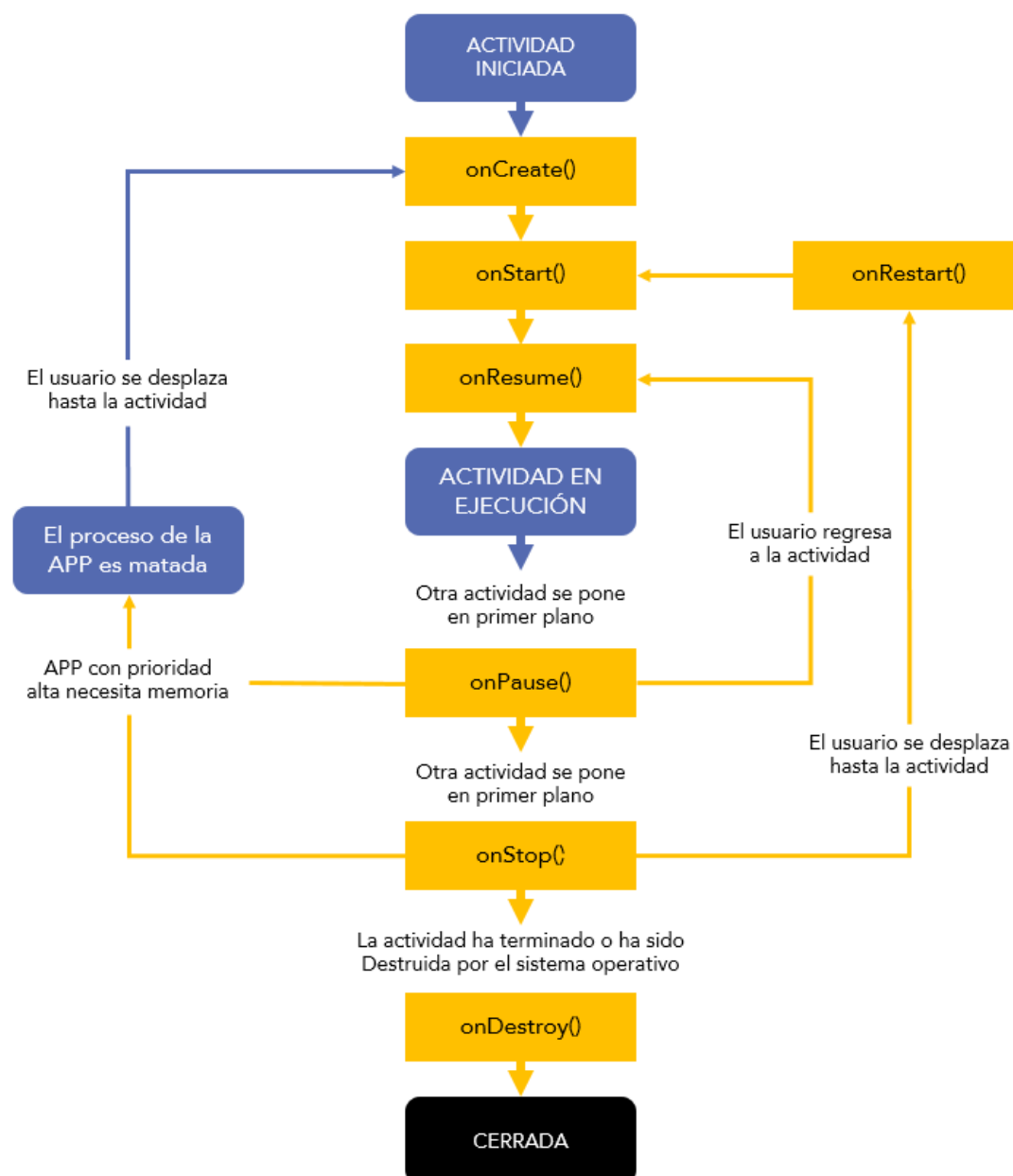
Una aplicación se compone de una o más actividades. **Una actividad es el componente de la aplicación que permite la interacción con el usuario.** Por lo que, una actividad es cada una de las pantallas que componen la aplicación.

Las actividades se dividen en dos partes, que son la capa lógica y la parte gráfica. La **capa lógica** es la encargada de establecer el funcionamiento de la aplicación, y se encuentra en los archivos .java del proyecto. La **capa gráfica** se compone de los archivos xml que forman los distintos layouts de la aplicación. Se encarga de especificar los elementos que forman las distintas actividades.

Por lo tanto, cada actividad de una aplicación necesita tener un archivo java y un archivo xml. El archivo de java será el encargado de llamar al archivo xml para cargar su contenido en la aplicación.

Las actividades tienen tres estados; **resumed**, **paused** y **stopped**, que irán cambiando según los eventos que se lancen.

- **Resumed:** la actividad se encuentra en ejecución.
- **Paused:** la actividad está parada, aunque es visible.
- **Stopped:** la actividad se encuentra parada, pero no es visible.



Como se observa en la imagen, existen una serie de eventos o sucesos que tienen lugar para realizar cambios de estado en la actividad.

- **onCreate:** es el evento producido al crearse la aplicación. Su función es establecer el layout correspondiente de la actividad y sus recursos importantes.
- **onRestart:** es el evento que salta cuando se para una actividad, antes de ser reiniciada.
- **onStart:** es el evento que se produce antes de mostrar la actividad.

- **onResume:** es el evento ejecutado antes de que el usuario interaccione con la actividad.
- **onPause:** es el evento producido cuando la actividad no es visible.
- **onDestroy:** es el evento que se produce al terminar la actividad, es decir, al llamar a la función `finish()`. También se produce automáticamente por el sistema operativo.

Todos estos eventos están relacionados, es decir, cada uno de estos eventos tiene su inverso.

- Con el evento `onCreate` se reservan los recursos, mientras que con el `onDestroy` se liberan.
- Con el evento `onStart` la actividad es visible, mientras que con el `onStop` pierde su visibilidad.
- Con el evento `onResume` la actividad pasa a tener el foco, mientras que con el `onPause` lo pierde.

Cuando se crea un nuevo proyecto, el IDE Android Studio ofrece la posibilidad de crear una actividad inicial para el proyecto. Pero, ¿cómo se crea una nueva actividad?

Paso 1.

Se crea una actividad dentro del proyecto, como se puede observar, heredará de la clase `Activity`, con su archivo java y su xml correspondiente. Si se crea como actividad, vendrá relacionado. Sino, debemos indicar en el evento `onCreate` el layout correspondiente.

CÓDIGO:

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
```

Paso 2.

Se declara la actividad en el `AndroidManifest.xml`.

CÓDIGO:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".Main2Activity"></activity>
</application>
```

Una vez que se ha creado la nueva actividad, se puede lanzar. Para ello se debe crear un objeto **Intent**.

Un intent es un elemento de comunicación entre los distintos componentes de una aplicación. Estos componentes pueden ser internos o externos, por ejemplo, es el encargado de lanzar una actividad o un servicio en nuestra aplicación, o de lanzar una página web.

Se debe crear un objeto de la clase *Intent*, indicando el contexto en el que nos encontramos, y la actividad que se quiere lanzar. Después, se iniciará la nueva actividad.

Existen dos métodos para lanzar la actividad, dependiendo si queremos recibir resultado o no de la nueva actividad:

- `startActivity(intent);`
- `startActivityForResult (intent, code);` En este método, se espera un resultado asociado al código establecido.

Además, es posible enviar parámetros a la nueva actividad, mediante el método `putExtras()`, y recibirlos en la nueva actividad mediante el método `getExtras()`.

CÓDIGO:

```
public void onClick(View view) {  
    Intent intent = new Intent (MainActivity.this, Main2Activity.class);  
    intent.putExtra("name", "Ilerna");  
    startActivity(intent);  
  
    Bundle extras = getIntent().getExtras();  
    String nombre = extras.getString("name");  
}
```

1.8. Modificación de aplicaciones existentes.

De la misma forma que se ha visto que es posible emular dispositivos ya definidos, este apartado es extendido a las aplicaciones que ya están desarrolladas. Android permite descargar y modificar desde su página oficial una gran cantidad de aplicaciones de ejemplo. Estas aplicaciones pueden servir de guía para el desarrollo de una nueva aplicación. Se pueden reutilizar funciones que ya funcionan correctamente en estos ejemplos, y añadirlas en el proyecto que se está desarrollando.

También es posible para desarrolladores con un nivel avanzado en Android añadir funcionalidades en aplicaciones que ya existen. Estas aplicaciones se pueden encontrar en un repositorio de acceso público, que mantiene un control de las versiones y cambios realizados por los distintos desarrolladores.

Otra posible situación que requiera de una modificación, son las labores de mantenimiento. Una vez se ha lanzado una aplicación, en muchas ocasiones los usuarios encuentran posibles fallos o *bugs* en la aplicación, y que posteriormente se arreglarán. Estas actualizaciones permiten modificar periódicamente todos los aspectos referentes a dicha aplicación.

Este tipo de aplicaciones se pueden encontrar en el apartado de muestras de la página oficial de desarrolladores de Android.

1.9. Utilización de entornos de ejecución del administrador de aplicaciones.

El entorno de ejecución de aplicaciones Android se conoce como **Android Runtime** (ART). Es el encargado de iniciar las aplicaciones de Android a partir de la versión 4.4 KitKat, como sustituta de Dalvik, y basado en la arquitectura just-in-time (JIT). La arquitectura usada por este entorno es Ahead-of-time (AOT), cuya funcionalidad es crear un archivo de compilación una vez se ha instalado la aplicación en el dispositivo. Este archivo creado es el que se usará una vez que se vuelva a iniciar esa aplicación. De esta forma no será necesario recompilar de nuevo la aplicación y liberar a su vez al procesador de carga. Este entorno ofrece una mejora en el rendimiento de los dispositivos: eliminando objetos que no se usan y permitiendo la depuración de aplicaciones, entre otras.

Estos entornos de ejecución se encargan de gestionar la ejecución de las aplicaciones, y son esenciales en el funcionamiento de las aplicaciones dentro de un dispositivo.

2. Programación de dispositivos móviles.

2.1. Herramientas y fases de construcción.

Toda aplicación o desarrollo en Android se realiza usando lo que se conoce como las herramientas del SDK de Android. Estas proveen a los desarrolladores de todo lo necesario para la compilación y ejecución de aplicaciones.

El proceso de construcción y desarrollo de una aplicación viene determinado por una serie de fases, que se pueden organizar de la siguiente manera:

Fase 1: Configuración.

En primer lugar, es necesario instalar todos elementos requeridos. Tanto los elementos de programación como los entornos de emulación (AVD) en el caso de Android.

Los elementos que se deben instalar son los siguientes: Android SDK, Android Development Tools y Android Platforms.

Una vez se han instalado estos, en último lugar se instalará el emulador de dispositivos ADV.

Fase 2: Desarrollo.

Aquí se desarrollará toda la parte de programación de la aplicación. Esto tiene que incluir el código fuente, todos los archivos o recursos utilizados, y el fichero Android *manifest* del proyecto.

Fase 3. Depuración y pruebas.

Toda aplicación requiere de un proceso que asegure que el resultado obtenido es el correcto y deseado. Esta fase separa dos procesos.

- Depuración: este proceso debe ser integrado junto con el desarrollo de la aplicación. De esta forma se detectarán y evitarán futuros errores, pudiendo observar el proceso de ejecución de la aplicación en tiempo real.
- Pruebas: una vez se ha finalizado el proceso anterior y en la etapa de desarrollo se obtuvo una versión final de la aplicación. Esta permitirá el uso de pruebas que ofrezcan resultados lo más fiables posibles. Este proceso también puede ayudar a detectar errores en la aplicación y poder corregirlos antes de su publicación.

Android ofrece herramientas de pruebas y depuración como Android Testing y documentos de log como los logging tools.

Fase 4: Publicación.

Es la última fase del proceso de construcción de una aplicación. En este momento se genera el archivo ejecutable de la aplicación.

Se tiene que definir la versión del SDK mínima y recomendada, los idiomas, definir las resoluciones gráficas, y los recursos requeridos para la correcta ejecución de la aplicación en todos los dispositivos. En caso de tener algún tipo de restricción se tiene que revisar el archivo `AndroidManifest.xml`.

Identificar el nombre del paquete también dentro del archivo `AndroidManifest.xml`.

Crear un certificado digital y firmar la aplicación. Esto permitirá la instalación de la aplicación en los dispositivos. Esta firma, además, permitirá identificar el autor y evitará que la aplicación pueda ser manipulada. Se generará una clave asociada a la aplicación y, por último, se obtendrá el ejecutable ya firmado.

Una vez se ha obtenido el ejecutable, este ya puede ser transferido a un dispositivo físico o publicarlo en Google Play. Para poder realizar una publicación en Google Play es necesario identificar esta aplicación y aceptar los términos que Google exige. Para un mayor detalle se puede consultar en su página oficial.

2.2. Interfaces de usuario. Clases asociadas.

Los “*layout*” son elementos no visibles cuya función es establecer la posición de los componentes gráficos (widgets) de la aplicación.

Existen distintos tipos:

- **FrameLayout:** dispone todos los elementos en la esquina superior derecha del mismo, es decir, los superpone. La utilidad de este *layout* es para hacer visible distintos elementos en una misma actividad en la misma posición.
- **LinearLayout:** dispone todos los elementos uno detrás de otro. Este layout tiene una propiedad llamada *Orientation*, que puede tener valor horizontal o vertical. Esto indica si los elementos estarán formando columnas, o formando filas respectivamente.
- **RelativeLayout:** dispone los elementos en cualquier punto del layout. Para ello, se suele colocar en función del componente padre o de otros ya colocados.

Puedes conocer las distintas propiedades de este tipo de *Layout* en el siguiente enlace:

<https://developer.android.com/reference/android/widget/RelativeLayout.LayoutParams.html>

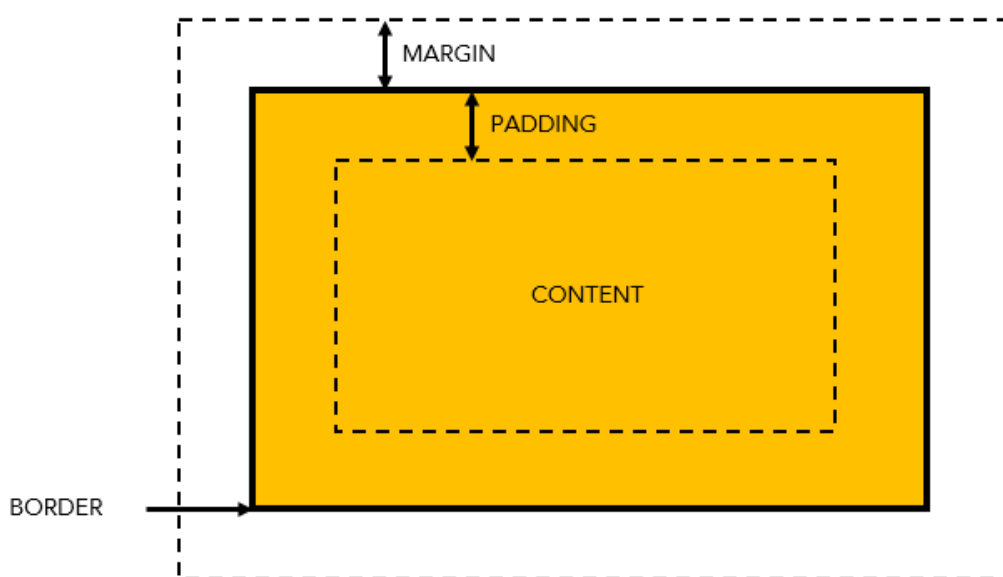
- **TableLayout:** dispone los componentes en forma de tabla. Se compone de etiquetas <TableRow>, que indican el número de filas que tendrán. El número de columnas dependerá del número de componentes que contenga la etiqueta <TableRow>.
- **GridLayout:** es otro tipo de tabla, con la diferencia que el propio layout tiene como propiedad el número de filas y columnas que contendrá. Después, se irán introduciendo componentes en el layout, y se distribuirán automáticamente.

Dentro de un *layout* se puede encontrar otro *layout* o *widgets*, entre los que se engloban todos los elementos que heredan de la clase *Widget*.

Los más conocidos son: *Button*, *TextView*, *EditText*, *ListView*, *RadioButton*, *CheckBox* y *ToggleBar*, entre otros.

Todos estos elementos, tendrán una serie de propiedades. Las propiedades de tamaño, tanto para ancho como para alto, son obligatorias en todos los componentes de la actividad, es decir, tanto en los *layouts* como en los *widgets*. Además, podrán tener distintas propiedades para el margen, el *padding*, el texto, la letra, el fondo, entre otros. Otra de las propiedades más importantes es el identificador, para poder hacer referencia a este componente.

El **margen** es la distancia entre dos componentes, mientras que, el **padding** es el espacio entre el componente y su propio **contenido**.



Puedes conocer los distintos widgets en el siguiente enlace:
<https://developer.android.com/reference/android/widget/package-summary.html>

2.3. Contexto gráfico. Imágenes.

- Menús.

Un **menú** es un **componente** que contiene un conjunto de opciones para navegar por la aplicación de forma más rápida.

Tenemos diferentes formas de crear menús en una aplicación, tenemos los menús en la barra de tareas, los menús contextuales y los menús desplegables laterales.

Para crear un menú hay que crear un archivo xml en la carpeta de nuestro proyecto /res/menu. En este archivo indicaremos los distintos ítems que tendrá el menú y, después, será necesario cargar este menú en la actividad.

CÓDIGO:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/Configuracion"
        android:icon="@drawable/ic_conf"
        android:title="@string/conf">
    </item>
    <item android:id="@+id/Ayuda"
        android:icon="@drawable/ic_ayuda"
        android:title="@string/ayuda">
    </item>
</menu>

public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu1, menu);
    return true;
}
```

Para poder interactuar con este menú, se deben crear las *callback* que responden a estos eventos:

- **onOptionsItemSelected**. Se utiliza en los menús de la barra de tareas, y recogeremos el ítem seleccionado mediante el método `getItemId()`.
- **onContextItemSelected**. Se utiliza en los menús contextuales. Su uso es similar al anterior tipo.

Para los menús desplegables laterales existe la plantilla predefinida, en la cuál ya viene establecido el menú y su interacción con el usuario. Para ello, es necesario modificar el contenido del menú para personalizar tu aplicación, y establecer su funcionamiento según el elemento que se haya seleccionado.

- **Notificaciones.**

Distinguimos dos tipos de notificaciones:

Toast:

Una *toast* es el mecanismo por el cual se puede mostrar un mensaje en la actividad.

Este tipo de notificaciones son muy útiles para mostrar mensajes de poco interés al usuario, puesto que en ningún momento se puede tener la certeza de que el usuario lo ha visto. El programador sólo debe indicar el mensaje que quiere mostrar y su duración.

CÓDIGO:

```
public boolean onOptionsItemSelected(MenuItem item) {  
    //Posibilidades del menú  
    switch (item.getItemId()) {  
        case R.id.Configuracion:  
            Toast.makeText(getApplicationContext(), "Has pulsado CONFIGURACIÓN", Toast.LENGTH_SHORT).show();  
            return true;  
        case R.id.Ayuda:  
            Toast.makeText(getApplicationContext(), "Has pulsado AYUDA", Toast.LENGTH_LONG).show();  
            return true;  
        default:  
            return super.onOptionsItemSelected(item);  
    }  
}
```

Notificaciones en la barra de estado

Las notificaciones en la barra de estado constan de una interfaz y del elemento que se lanzará al ser pulsado, normalmente la actividad principal de la aplicación.

Estas notificaciones deben **tener obligatoriamente definidas en su interfaz un icono, un título y un mensaje**. Y se deberá especificar mediante código su eliminación al ser pulsadas, puesto que sino, quedarán fijas en la barra de estado.

En el siguiente código se puede observar como crear la notificación, al que hay que añadir el elemento que se lanzará al ser pulsado.

CÓDIGO:

```
NotificationCompat.Builder mBuilder = (NotificationCompat.Builder) new NotificationCompat.Builder(MainActivity.this)
    .setSmallIcon(android.R.drawable.stat_sys_warning)
    .setContentTitle("Notificación")
    .setContentText("Ilerna Online");

NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

mNotificationManager.notify(1, mBuilder.build());
```

2.4. Eventos.

Las aplicaciones Android están pensadas para dispositivos táctiles, por lo que muchos de sus eventos están relacionados con esta característica, es decir, son eventos de la clase **View**.

Los **listener** son las interfaces de la clase **View**, que se encargan de capturar los eventos, es decir, de detectar la interacción con el usuario y ejecutar las instrucciones correspondientes.

Para capturar un evento es necesario implementarlo mediante el método **setOnClickListener()** del objeto de la vista sobre el que queremos que se capture.

CÓDIGO:

```
Button cambiar = (Button) findViewById(R.id.cambiar);

cambiar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});
```

Los eventos más conocidos que se implementan son: **onClick**, **onLongClick**, **onFocusChange**, **onKey**, **onTouch** y **onCreateContextMenu**.

En el siguiente enlace puedes consultar más información sobre estos eventos:

<https://developer.android.com/reference/android/view/View.html>

2.5. Técnicas de animación y de uso.

Una animación es el cambio de alguna de las propiedades de un objeto que permiten ver a este a lo largo del tiempo con un aspecto diferente.

Android nos permite realizar tres tipos de animaciones:

- **Animación por fotogramas:** mediante la clase *AnimacionDrawable* es posible realizar una reproducción de diferentes imágenes.
 - **Animaciones de vistas o Tween:** permiten la modificación de la imagen mediante diferentes técnicas, como pueden ser: translación y rotación, tamaño o transparencia.
 - **Animaciones de propiedades:** modificaciones del objeto, no de la vista.
-
- **Animación por fotogramas.**

Es una de las posibilidades de la clase *Drawable*. Para crear una transición de imágenes es necesario crear un archivo xml en la carpeta *res/drawable*.

CÓDIGO:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/imagen1" android:duration="750"/>
  <item android:drawable="@drawable/imagen2" android:duration="750"/>
</animation-list>
```

Una vez que tenemos el archivo xml y las imágenes que se quieren mostrar en este directorio, es posible crear distintos ítems, uno para cada una de las imágenes. Estos ítems tendrán dos atributos:

- **android:drawable** -> la ruta del archivo, sin extensión.
- **android:duration** -> el tiempo durante el que se mostrará.

Después, en la actividad en la que se quiera mostrar, es necesario crear la animación, indicando el archivo xml, y comenzar la animación.

CÓDIGO:

```
miVista.setBackgroundResource(R.drawable.imagenes);
animation = (AnimationDrawable) miVista.getBackground();
```

Por último, se utiliza el método **start()** y **stop()** para realizar las operaciones de animación.

- Animación de vistas.

En el directorio res/anim hay que crear un documento xml, que contendrá las diferentes animaciones para el objeto en el orden en el que se quiere actuar. Las etiquetas que se utilizan para esto son: **<translade>**, **<rotate>**, **<scale>**, **<alpha>**. Además, es posible agrupar varias de estas etiquetas en grupos, para que se ejecuten simultáneamente, para ello se utiliza la etiqueta **<set>**.

Una vez que se tiene el xml es posible crear la animación, con la clase Animation y el xml creado.

New Resource File

File name: vista

Resource type: Animation

Root element: set

Source set: main

Directory name: anim

Available qualifiers:

Chosen qualifiers:

2.6. Descubrimiento de servicios.

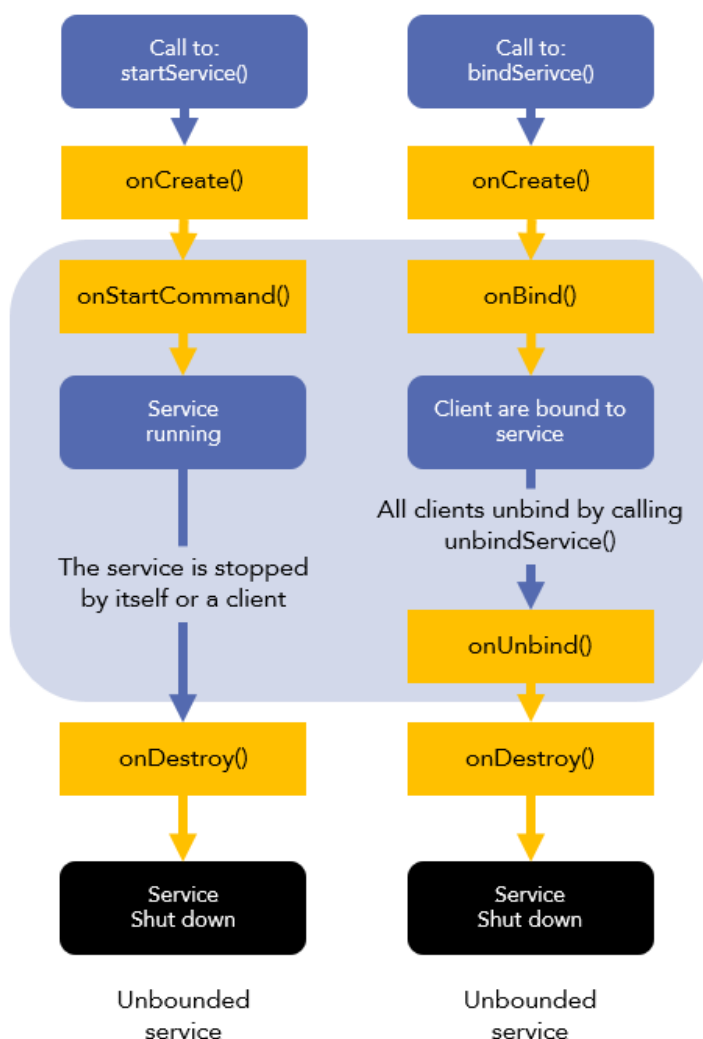
Un servicio es un proceso ejecutado de forma invisible para el usuario. Existen dos tipos de servicios: iniciados y de enlace.

- **Servicio iniciado.**

Un componente inicia un servicio mediante el método **startService()**. De esta forma, el servicio queda iniciado en segundo plano hasta que finalice su proceso. Aunque el componente que lo ha lanzado finalice, este servicio seguirá ejecutándose.

- **Servicio enlazado.**

Al contrario que en los servicios iniciados, los servicios enlazados son creados para vincular un componente con un servicio. Para ello, es necesario realizarlo mediante el método **bindService()**. Este tipo de servicios crean un interfaz cliente-servidor, que permite la comunicación entre los componentes y el servicio. Se pueden enlazar distintos componentes a un mismo enlace, pero cuando todos los componentes eliminan la comunicación con el servicio, el servicio finaliza.



Todos los servicios se deben declarar en el archivo `AndroidManifest.xml`, mediante la etiqueta `<Service>`.

Cualesquiera de estos tipos de servicios no pueden comunicarse directamente con el usuario, puesto que no tienen interfaz gráfica. Para ello, deben utilizar un mecanismo de comunicación, como puede ser las *Toast* o las *Notificaciones*.

En la documentación de Android puedes encontrar más información sobre la creación de un servicio:

<https://developer.android.com/guide/components/services.html?hl=es-419>

2.7. Bases de datos y almacenamiento.

Para el almacenamiento de información tenemos distintos mecanismos:

- **Bases de Datos internas:** mediante el API SQLite se pueden crear bases de datos en la aplicación.
- **Bases de Datos externas:** mediante servicios web se pueden crear conexiones a bases de datos en Internet.
- **Preferencias:** permite almacenar la configuración del usuario en la aplicación.
- **Proveedores de contenido:** son componentes que permiten la gestión de los datos con otras aplicaciones.
- **Ficheros:** permite crear ficheros, tanto en la memoria del dispositivo como en una tarjeta SD, y utilizarlos como recursos.
- **XML:** mediante diferentes librerías, como son SAX y DOM, permite manipular datos en un XML.

- **Bases de datos SQLite**

Las bases de datos SQLite se basan en el lenguaje SQL, es decir, se ejecutan en la aplicación Android sentencias SQL.

Para ello, se debe utilizar la clase `SQLiteOpenHelper`. Su uso, normalmente, es crear una clase que herede de esta e implemente sus dos métodos obligatorios: `onCreate()` y `onUpgrade()`. Esta clase tiene un constructor por defecto.

CÓDIGO:

```
public BaseDatos(Context context, String name, SQLiteDatabase.CursorFactory factory, int version) {
    super(context, name, factory, version);
}
```

Como se puede observar, se reciben cuatro parámetros:

- **Context context:** el contexto desde el que se utiliza la base de datos.
- **String name:** el nombre de la base de datos.
- **SQLiteDatabase.CursorFactory factory:** un objeto de tipo cursor, no es obligatorio.
- **int version:** la versión de la base de datos.

El método **onCreate()** es el encargado de crear la base de datos, por lo que, si ya existe dicha base de datos, sólo la abrirá. Mientras que el método **onUpgrade()** se encarga de actualizar la estructura de dicha base de datos. Es decir, si el número de versión es superior al que estaba establecido, se ejecutara este método.

CÓDIGO:

```
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase) {
    String consulta = "CREATE TABLE usuarios (_id INTEGER PRIMARY KEY AUTOINCREMENT, nombre TEXT, apellido TEXT, telefono INTEGER)";
    sqLiteDatabase.execSQL(consulta);
}

@Override
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i1) {
}
```

Por otra parte, el programador podrá implementar todos los métodos que necesite para la gestión de la base de datos.

Además, esta clase dispone de dos métodos que permiten abrir la conexión de la base de datos en forma de lectura o en forma de escritura, dependiendo del tipo de operación que necesitemos realizar. Los métodos, respectivamente, son **getReadableDatabase()** y **getWritableDatabase()**.

CÓDIGO:

```

    SQLiteDatabase bd = getWritableDatabase();

    bd.execSQL("INSERT INTO usuarios (nombre, apellido, telefono) VALUES (

    bd.close();

    SQLiteDatabase db = getReadableDatabase();

    Cursor cursor = db.rawQuery ("SELECT nombre, apellido, telefono FROM usuarios order BY nombre", null);

```

Entre los métodos que se utilizan para ejecutar sentencias SQL se encuentran:

- **execSQL**: cuando no hay valor de retorno.
- **query()** y **rawQuery()** para recuperar los datos de la base de datos. En el método *query()* se especifican los distintos parámetros para formar la consulta, mientras que en el método *rawQuery()* se envía la sentencia SQL en forma de String.

Para conocer todos los métodos, puedes mirar la documentación de la clase SQLiteDatabase:

<https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>

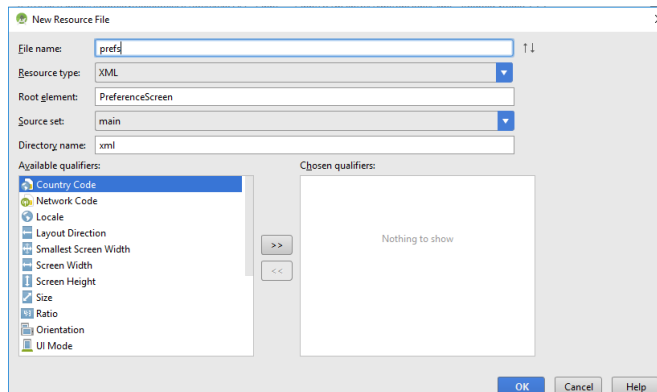
2.8. Persistencia.

- **Preferencias.**

Las preferencias se utilizan como mecanismo para almacenar datos de forma permanente, normalmente, la configuración de la aplicación. Para ello, se utiliza la clase **SharedPreferences**. El fichero creado para el almacenamiento de estos datos se encuentra en una carpeta llamada `shared_prefs` de la aplicación.

Para ello, lo primero es crear la parte gráfica. Es posible realizarlo de dos formas:

- Crear un archivo xml almacenado en la carpeta xml específico para trabajar con las Preferencias.



- Crear un archivo xml almacenado en la carpeta *layout*, que sea un *layout* como cualquier otro de las actividades.

Una vez que tenemos organizada la interfaz, hay que mostrarla.

- En el caso de utilizar el xml de PreferenceScreen:

La actividad heredará de `PreferenceActivity` en vez de `Activity` o `AppCompatActivity`.

CÓDIGO:

```
public class MainActivity extends PreferenceActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //Guardamos los cambios
        getSupportFragmentManager().beginTransaction().replace(android.R.id.content, new MyPreferenceFragment()).commit();
    }
}
```

Como se puede observar, es necesario trabajar con transacciones.

Una transacción es un conjunto de instrucciones que deben ejecutarse sin realizar cambios hasta que terminan todas. Es decir, si hay fallos en una instrucción, el resto de instrucciones no tendrán cambios.

En la transacción indicamos que debe llamar a una clase que hereda de `PreferenceFragment()` y cargarlo sobre la vista actual.

En la clase del fragmento, se indicará cual es el archivo xml que se debe mostrar.

CÓDIGO:

```
public static class MyPreferenceFragment extends PreferenceFragment
{
    @Override
    public void onCreate(final Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        //Cargamos el layout
        addPreferencesFromResource(R.xml.opciones);
    }
}
```

En el caso de utilizar un layout:

En el código Java de la actividad únicamente hay que cargar el layout correspondiente.

Para que los valores indicados en la aplicación se almacenen es necesario, como se ha indicado anteriormente, trabajar con la clase `SharedPreferences`.

Escribir datos en el fichero:

CÓDIGO:

```
SharedPreferences mypreferenceArchive = getSharedPreferences("fichero", 0);
SharedPreferences.Editor editor = mypreferenceArchive.edit();
```

Creamos un fichero, en este caso se llamará fichero, y el segundo parámetro se corresponde al modo de acceso.

Existe distintos modos de acceso:

- **MODE_PRIVATE:** sólo puede acceder esta aplicación al fichero. Su valor es 0.
- **MODE_WORLD_READABLE:** todas las aplicaciones del dispositivo pueden leer este fichero, pero sólo puede ser modificado por esta. Su valor es 1. No se recomienda su uso por fallos en la seguridad, y está obsoleto desde la versión 17.
- **MODE_WORLD_WRITEABLE:** todas las aplicaciones del dispositivo pueden leer o escribir en este fichero. Su valor es 2. No se recomienda su uso por fallos en la seguridad, y está obsoleto desde la versión 17.

- **MODE_MULTI_PROCESS**: se utiliza cuando más de un proceso de la aplicación necesita escribir en este fichero al mismo momento. Su valor es 4. No se recomienda su uso, y está obsoleto desde la versión 23.

Después se encontrarán los distintos métodos de la clase **Editor** para escribir en el fichero, como pueden ser **putString(clave, valor)**, **putBoolean(clave, valor)** o **putInt(clave, valor)** entre otros.

Puedes consultar todos los métodos de la clase en el siguiente enlace:
<https://developer.android.com/reference/android/content/SharedPreferences.Editor.html>

Por último, se debe cerrar el editor para que todos los cambios se hagan efectivos.

CÓDIGO:

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

Leer datos del fichero:

Se utilizan los métodos **get** para recoger los datos del fichero.

Puedes consultar los distintos métodos en el siguiente enlace:
<https://developer.android.com/reference/android/content/SharedPreferences.html>

Además del método **getSharedPreferences (String archivo, int modo)**, se puede utilizar la función **getPreferences (int modo)**, en el que el fichero es creado por defecto, y únicamente se podrá tener uno en la aplicación.

- **Ficheros**

Existen distintas formas de almacenamiento de los ficheros en un dispositivo Android. Podemos encontrar ficheros en la memoria interna del dispositivo, en la memoria externa o en los recursos. Los ficheros que se encuentran en

los recursos, únicamente son de lectura, por lo que no podremos almacenar información en ellos, sólo podemos realizar operaciones de lectura.

Almacenamiento interno:

Todas las aplicaciones contienen una carpeta para almacenar ficheros cuando se instalan en un dispositivo. Esta carpeta tiene la ruta `dada/data/nombrePaquete/files`. Esta carpeta también se desinstala automáticamente al desinstalar la aplicación.

El paquete que se utiliza para la lectura y escritura de ficheros es `java.io`, el cuál ya se ha estudiado en Programación.

Puedes conocer más información de este paquete en el siguiente enlace:

<https://developer.android.com/reference/java/io/package-summary.html>

Almacenamiento externo:

Como la memoria de los dispositivos es limitada, Android nos posibilita la opción de utilizar el almacenamiento externo, normalmente una tarjeta SD, para almacenar estos ficheros.

Como estos sistemas de almacenamiento no son fijos, antes de empezar a utilizarlo se debe comprobar que existe y que está preparada para su uso.

CÓDIGO:

```
if (Environment.getExternalStorageDirectory().equals(Environment.MEDIA_MOUNTED_READ_ONLY)) {  
    //Acceso sólo de lectura  
} else if (Environment.getExternalStorageDirectory().equals(Environment.MEDIA_MOUNTED)) {  
    //Acceso de lectura y escritura  
} else {  
    //Sin acceso  
}
```

Además, si la operación es de escritura, es necesario dar permisos a la aplicación para ello. Como se ha realizado anteriormente, hay que declarar el permiso `WRITE_EXTERNAL_STORAGE` en el archivo `AndroidManifest.xml`, con la etiqueta `<uses-permission>`.

Ficheros de recursos:

En la carpeta /res del proyecto Android se encuentran los recursos de la aplicación, que son sólo de lectura. Son aquellas carpetas que contienen las imágenes (res/drawable o res/assets) o la música (res/raw).

Desde el proyecto, podemos cargar cualquiera de estos recursos para hacer uso de lectura de ellos. Además, en la carpeta res/raw también es posible almacenar ficheros con extensión .txt.

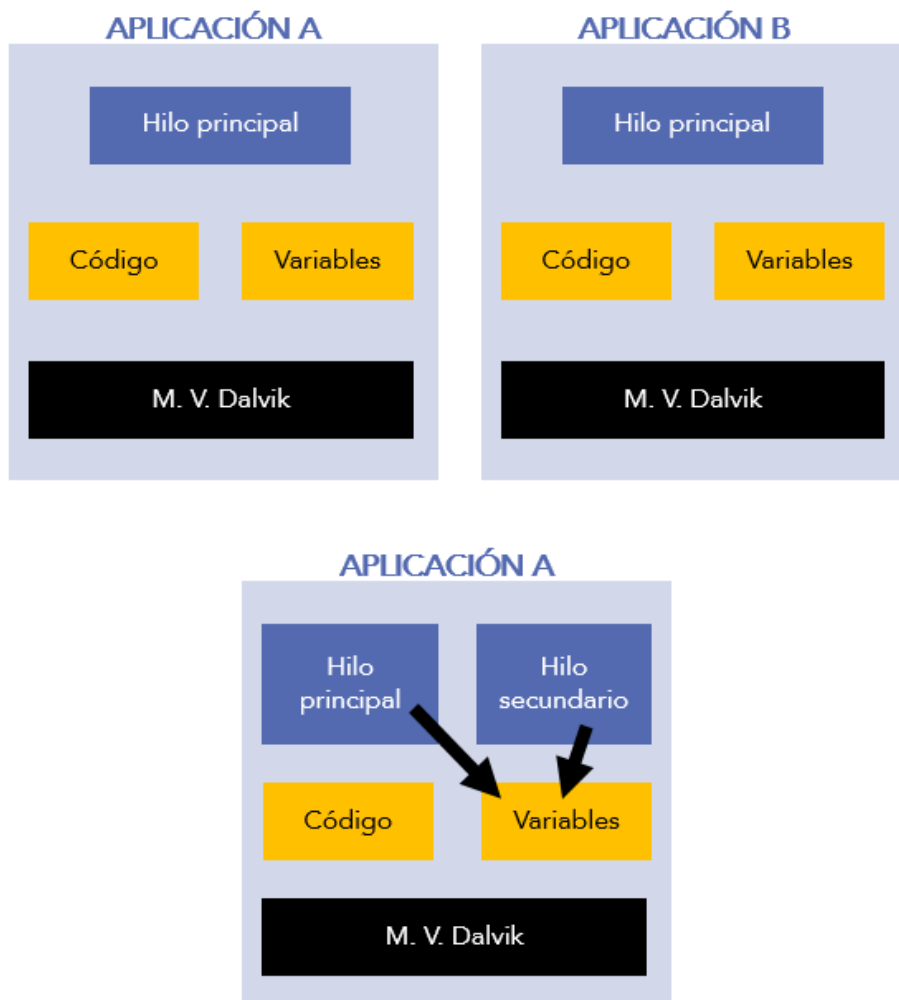
Normalmente, los ficheros de música e imágenes ocupan bastante espacio, y eso es una desventaja si forma parte de la aplicación, puesto que sería una aplicación de gran tamaño para descargar. Por ello, en las tarjetas SD existe la posibilidad de almacenar estos recursos que pertenecen a la aplicación. La ruta de estos ficheros es /Android/data/NombrePaquete/files.

CARPETA	CONSTANTE
Music	DIRECTORY_MUSIC
Download	DIRECTORY_DOWNLOADS
Pictures	DIRECTORY_PICTURES
DCIM	DIRECTORY_DCIM

Todas estas carpetas también pertenecen a la aplicación, por lo que serán eliminadas automáticamente al desinstalarla.

2.9. Modelo de hilos.

En cualquier sistema operativo es posible tener ejecutándose a la vez distintas aplicaciones, cada una en un hilo diferente. Pero también es posible tener varios hilos en una misma aplicación.



Todos los procesos de una aplicación se ejecutan dentro de la misma hebra, la elección de ejecución del proceso sigue un orden por prioridad. Este orden corresponde de menor a mayor con el siguiente:

- Procesos vacíos.
- Procesos en segundo plano.
- Servicios.
- Procesos visibles.
- Procesos en primer plano.

Al iniciar la aplicación, se creará una hebra, denominada principal. Es la hebra con la que se ha estado trabajando hasta ahora.

Una hebra es un subproceso o hilo de ejecución, es decir, es un conjunto de tareas que se ejecutan.

Las características del hilo principal son:

- Es la única capaz de interactuar con el usuario, se encarga de recoger los eventos.
- Es la única que puede modificar la interfaz gráfica, puesto que es la que puede acceder a los componentes de ella.

En una aplicación se tendrán tantas hebras principales como actividades tenga.

Si la hebra principal está ocupada con alguna operación, entonces no podrá recoger las interacciones con el usuario, y, por lo tanto, dará la apariencia de “aplicación bloqueada”, y si esto se alarga durante varios segundos, entonces el Sistema Operativo lanzará un mensaje de que la aplicación no responde.

La primera solución a este problema es la creación de nuevos hilos mediante la clase *Thread* de Java:

<https://developer.android.com/reference/java/lang/Thread.html>

Pero otra solución, y la que se suele llevar a cabo es crear tareas asíncronas.

Una tarea asíncrona es la ejecución de instrucciones en segundo plano. Estos objetos heredan de la clase *AsyncTask*.

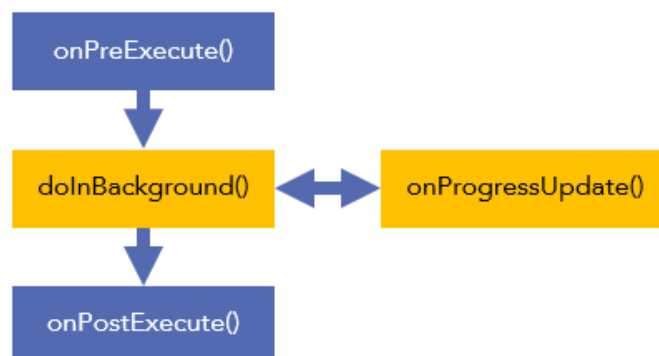
La clase *AsyncTask* es la encargada de ejecutar tareas en segundo plano, con la característica de permitir a estas tareas modificar la interfaz gráfica. Esto es posible por sus dos métodos obligatorios:

- **`doInBackground()`**

Es el encargado de ejecutar el código en la hebra secundaria.

- **`onPostExecute()`**

Es el encargado de ejecutar el código en la hebra principal.

Funcionamiento:

Cuando se crea una tarea `AsyncTask`, primero es necesario preparar los datos, aunque no obligatorio. Esto se realiza en la hebra principal, pero es necesario introducir el código dentro del método **`onPreExecute()`** de la tarea asíncrona. Por ejemplo, es común encontrar que en este método se indica el mecanismo por el cuál el usuario va a conocer el estado de la actividad, y se inicializa.

Después, en el método **`doInBackground()`** se ejecutan todas aquellas instrucciones que podrían detener la aplicación en caso de ejecutarse en primer plano, como por ejemplo, peticiones web.

Durante la ejecución de la tarea asíncrona, se puede invocar al método **`publishProgress()`**, que ejecutará en la hebra principal el código del método **`onProgressUpdate()`**. Por ejemplo, para actualizar el estado de la barra de espera.

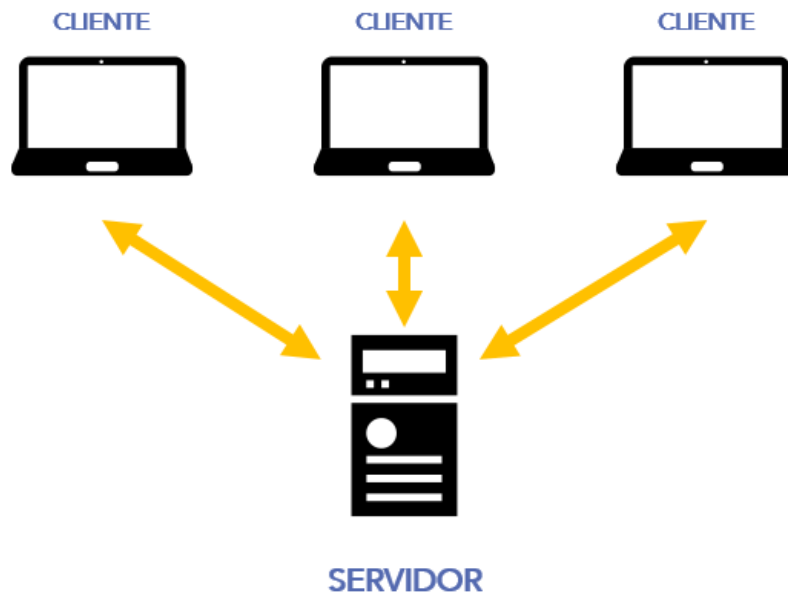
Por último, cuando termina el método **`doInBackground()`**, se ejecuta el código **`onPostExecute()`**. Es necesario saber que el valor o el objeto que devuelve *`doInBackground`* es el que recibe este último método como parámetro.

Además, es posible finalizar esta hebra, para ello, habrá que implementar el evento **`onCancelled()`** mediante el método **`cancel()`**.

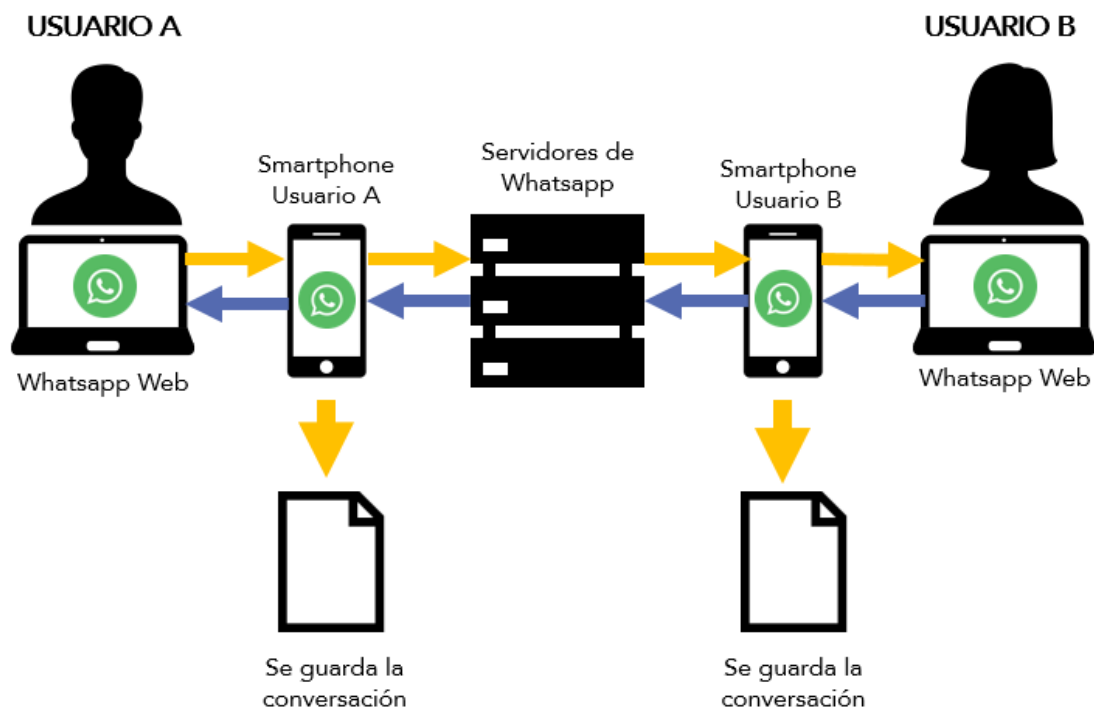
2.10. Comunicaciones. Clases asociadas. Tipos de conexiones.

El modelo cliente-servidor se basa en una arquitectura en la que existen distintos recursos, a los que denominamos servidores, y un determinado número de clientes, es decir, de sistemas que requieren de esos recursos. Los servidores, además de proveer de recursos a los clientes, también ofrecen una serie de servicios que se estudiarán en el siguiente capítulo.

En este modelo, los clientes son los encargados de realizar peticiones a los servidores, y estos responden con la información necesaria.



En la actualidad existen distintas aplicaciones que utilizan este modelo, como pueden ser: el correo electrónico, mensajería instantánea y el servicio de internet. Es decir, que se utiliza siempre que accedemos a una página web.



Se debe tener en cuenta que, en este modelo, no se intercambian los roles entre clientes y servidores.

Los sockets también son un ejemplo del modelo cliente-servidor.

Un **socket** es un mecanismo que permite la comunicación entre aplicaciones a través de la red, es decir, abstrae al usuario del paso de la información entre las distintas capas. Su función principal es crear un canal de comunicación entre las aplicaciones y simplificar el intercambio de mensajes.

Existen dos tipos de sockets, los orientados a conexión y los no orientados a conexión.

En el módulo 9, Programación de servicios y procesos, se conocerán las diferencias entre los tipos de sockets, y las clases Java utilizadas dependiendo del tipo de socket.

Al ser un servicio que necesita conexión a Internet para la comunicación, es necesario que tenga definido el permiso de internet.

CÓDIGO:

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

2.11. Gestión de la comunicación sin hilos.

Un **broadcast receiver** es un componente de la aplicación que se encarga de recibir los mensajes enviados por el Sistema Operativo y por otras aplicaciones.

Para declarar un broadcast receiver en la aplicación, es necesario indicarlo en el archivo AndroidManifest.xml, mediante la etiqueta **<receiver>** y **<intent-filter>** para indicar la acción a la que responden. De esta forma, estará

disponible siempre, es decir, desde que se instala la aplicación hasta que se desinstala.

Para evitar esto, es posible declarar el broadcast receiver para momentos puntuales, mediante `Context.registerReceiver()` y `Context.unregisterReceiver()`, que suelen encontrarse en los métodos `onResume()` y `onPause()`, respectivamente.

Los mensajes de *broadcast* pueden ser enviados de dos formas:

- Se envían mediante el método `Context.sendBroadcast`. Si se utiliza este método, los mensajes son asíncronos, y pueden llegar desordenados.
- Se envían mediante el método `Context.sendOrderedBroadcast`. Se envían de uno en uno, garantizando su orden, de forma asíncrona.

En la recepción de mensajes se verá una de las utilidades de este concepto.

2.12. Envío y recepción de mensajes de texto. Seguridad y permisos.

Para poder enviar mensajes de textos desde la aplicación Android, es necesario dar permisos para ello en el `AndroidManifest.xml` como ya hemos realizado anteriormente. El permiso que se ha de indicar para esto es `SEND_SMS`, mediante la etiqueta `<uses-permission>`.

El envío de mensajes de texto se puede realizar de dos maneras, puede ser desde otra actividad o directamente.

Para enviar un mensaje de texto desde otra actividad hay que crear un elemento de comunicación en el que indiquemos el número de teléfono al que se va a enviar este mensaje. Este *intent* tendrá también el contenido del mensaje y el tipo `"vnd.android-dir/mms-sms"`, y después enviar el mensaje como si fuera una nueva actividad:

CÓDIGO:

```
Intent enviar = new Intent(Intent.ACTION_VIEW, Uri.fromParts("sms", numero, null));
enviar.putExtra("sms_body", "Contenido del mensaje");
enviar.setType("vnd.android-dir/mms-sms");
startActivity(enviar);
```

Pero también es posible enviar los mensajes directamente, para ello se necesita la clase **SMSManager**. Esta clase tiene el método **sendTextMessage** que es el que enviará el mensaje de texto.

La sintaxis de este método es la siguiente: **void sendTextMessage** (*String destinationAddress, String scAddress, String text, PendingIntent sendIntent, PendingIntent deliveryIntent*)

En este método los campos necesarios para rellenar son *destinationAddress* con el número de teléfono al que se quiere enviar el mensaje, el *Texto* que se quiere enviar y el elemento de envío, que contiene la información relevante del mensaje.

Un PendingIntent es una comunicación con el Sistema Operativo que no se sabe cuando se llevará a cabo.

CÓDIGO:

```
PendingIntent enviar = PendingIntent.getActivity(MainActivity.this, 0, new Intent(MainActivity.this, Main2Activity.class), 0);
SmsManager manager = SmsManager.getDefault();
manager.sendTextMessage(numero, null, "mensaje", enviar, null);
```

Además, es interesante que después de enviar el mensaje, confirme su entrega. Para ello, la clase **SMSManager** nos permite conocer el estado del mensaje, mediante el campo **deliveryIntent**.

Para la recepción de un mensaje es necesario dar un nuevo permiso **RECEIVE_SMS**. Pero, además, en el *AndroidManifest* también hay que registrar el **receiver** que se encargará de la recepción de los mensajes de texto.

CÓDIGO:

```
<receiver android:name=".ReceiverSMS">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>

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

2.13. Envío y recepción de mensajería multimedia. Sincronización de contenidos. Seguridad y permisos.

Para el envío de mensajes multimedia únicamente es necesario cambiar el contenido y el tipo de mensaje.

CÓDIGO:

```
Intent enviar = new Intent(Intent.ACTION_SEND);
enviar.putExtra("sms_body", "Contenido del mensaje");
enviar.setType("image/png");
```

Para recibir un mensaje multimedia, el permiso que hay que dar a la aplicación es **RECEIVE_MMS**.

2.14. Tratamiento de las conexiones HTTP y HTTPS.

El protocolo HTTP es un protocolo cliente-servidor que se encarga de intercambiar información entre los navegadores web y sus servidores.

El protocolo HTTPS es un protocolo de la capa de aplicación, basado en el protocolo HTTP. Por así decirlo, añade seguridad a dicho protocolo. Utiliza un cifrado basado en SSL/TLS. Mientras que el protocolo HTTP utiliza el puerto 80, HTTPS utiliza el 443

Si se quiere realizar una aplicación con acceso a internet hay que tener presente el permiso necesario.

CÓDIGO:

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

Android tiene dos paquetes que permiten el desarrollo de este tipo de aplicaciones, `java.net` y `Android.net`.

Al igual que en una aplicación Java, para la conexión a una página web, es necesario realizar una conexión HTTP, para ello se debe crear el objeto URL, con la dirección web, y después realizar la petición con la clase HttpURLConnection.

CÓDIGO:

```
URL url = new URL ("http://google.com");  
connection = (HttpURLConnection) url.openConnection();
```

Estas conexiones es posible implementarlas como peticiones a un servidor en el que esté almacenada una base de datos, de esta forma, es posible acceder a datos externos a la aplicación.

Para ello, se recomienda comenzar con un servidor local, como puede ser WAMP, que nos ofrece una conexión a un servidor APACHE, con PHP y MySQL, y después, implementarlo en el servidor web elegido.

PHP es un lenguaje de programación que se ejecuta en el servidor, y se utiliza para el desarrollo web de contenido dinámico.

El funcionamiento de esto se resume a:

- Realizar los distintos archivos con extensión **".php"** que son los encargados de la comunicación entre la aplicación y la base de datos. En estos ficheros se encuentran las distintas consultas a la base de datos.
- Realizar una petición web al servidor: <http://localhost> o <http://127.0.0.1>, indicando la ruta de los ficheros. <http://localhost/archivosPHP/mostrarUsuarios.php>

UF2: Programación multimedia

1. Uso de librerías multimedias integradas.

1.1. Conceptos sobre aplicaciones multimedia.

Una **aplicación multimedia** es aquella que contiene diferentes tipos de información integrados de forma coherente. Estos tipos de información pueden ser: texto, audio, imágenes, vídeos, animaciones o interacciones.

Casi todas las aplicaciones que existen actualmente en el mercado de Android se consideran aplicaciones multimedia, puesto que todas contienen algunos de estos elementos.

Todas estas aplicaciones deben gestionar toda la información. No es lo mismo el trato de un audio que de una imagen, cada uno de estos tipos de información tienen unas clases y métodos que son las que permiten su funcionalidad.

En este capítulo vamos a estudiar estas clases, y sus métodos más conocidos.

1.2. Arquitectura de la API utilizada.

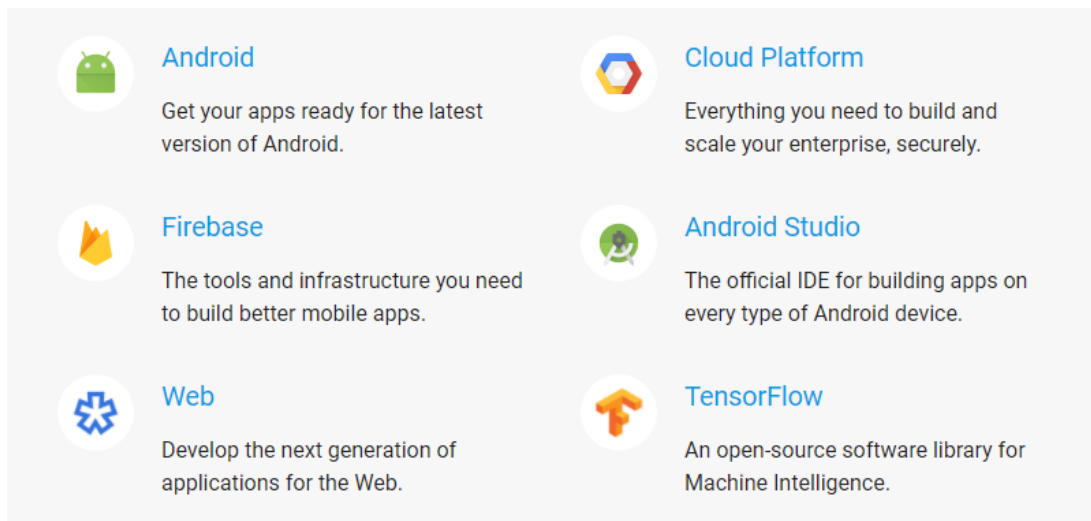
La arquitectura de Android se divide en distintas capas, teniendo siempre en cuenta, que el núcleo es Linux:

https://developer.android.com/guide/platform/images/android-stack_2x.png?hl=es-419

Google ha realizado un API de desarrollo para las aplicaciones Android, en el que algunos de los productos como son **Maps**, **Firebase** o **Notificaciones PUSH** han triunfado en el último tiempo.

En la siguiente página, el desarrollador tiene todas las herramientas que ofrece Google que puede utilizar para el desarrollo de su aplicación.

<https://developers.google.com/?hl=es-419>



1.3. Fuentes de datos multimedia. Clases.

Un proveedor de contenido es el mecanismo que permite compartir los datos con otras aplicaciones, y también, obtener datos de aplicaciones externas.

Las clases utilizadas para obtener información son:

- **Browser:** para obtener información sobre el historial de navegación o el historial de búsquedas.
- **Calendar:** para obtener la información de los eventos del calendario.
- **CallLog:** para obtener un registro de las últimas llamadas, tanto entrantes o perdidas, como salientes.
- **Contacts:** para obtener la lista de contactos del dispositivo.
- **Document:** para obtener los distintos ficheros de texto.
- **MediaStore:** para obtener los distintos ficheros de audio, imágenes, vídeos, etc. Que se encuentran en el dispositivo, tanto en memoria interna como externa.
- **Setting:** para obtener las preferencias del sistema.

- **Telephony:** para obtener los distintos mensajes tanto de texto como multimedia, recibidos o enviados.
- **UserDictionary:** para obtener las palabras definidas por el usuario y las más utilizadas.

Es importante tener en cuenta que, para hacer uso de esta información, es necesario que el usuario acepte el acceso a ello. Por eso, en la aplicación es necesario especificar el permiso de aquella información con la que se va a trabajar, en el `AndroidManifest.xml`, como ya se ha comentado anteriormente.

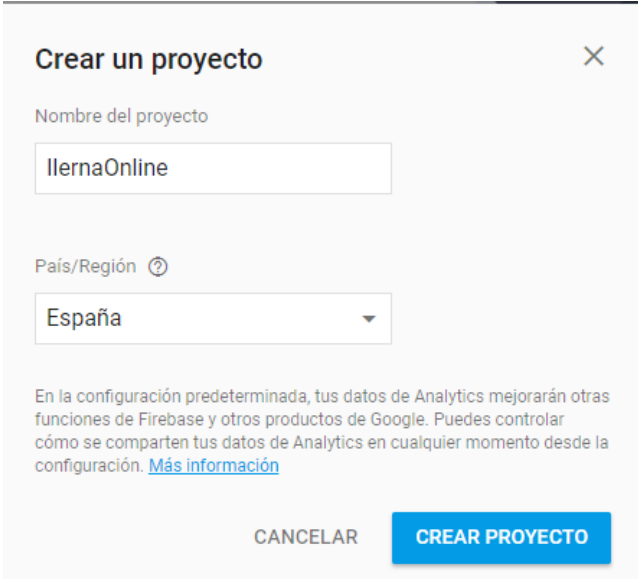
Por ejemplo, `READ_CALENDAR`, `READ_CALL_LOG`, `READ_CONTACTS` o `READ_SMS`.

En este enlace se pueden consultar todos los permisos que se pueden establecer en una aplicación:

<https://developer.android.com/reference/android/Manifest.permission.html>

1.4. Datos basados en el tiempo.

Firestore es una base de datos en tiempo real que se encuentra en la "nube". Los datos se almacenan por nodos, no como registros de SQL. Tendremos un nodo con diferente información y después otro nodo.



Crear un proyecto ✕

Nombre del proyecto

IlernaOnline

País/Región ⓘ

España ▼

En la configuración predeterminada, tus datos de Analytics mejorarán otras funciones de Firebase y otros productos de Google. Puedes controlar cómo se comparten tus datos de Analytics en cualquier momento desde la configuración. [Más información](#)

CANCELAR CREAR PROYECTO



Agrega Firebase a tu app para Android

Nombre de paquete de Android ?

Sobrenombre de la app (opcional) ?

Certificado de firma SHA-1 de depuración (opcional) ?

Obligatoria para Dynamic Links, Invites y la asistencia con un número telefónico o el Acceso con Google en Auth. Puedes editar la clave SHA-1s en Configuración.

CANCELAR REGISTRAR APP

El asistente irá mostrando los cambios que se tienen que realizar en la aplicación. Después de realizarlos, ya existente la comunicación entre el proyecto y la base de datos.

Es posible crear nodos en la base de datos, en la siguiente imagen se muestra la creación del primer nodo.

ilernaonline-281c6: null ✕

Nombre Almacen1 + ✕

Nombre cantidad Valor 445 ✕

CANCELAR AGREGAR

CÓDIGO:

```
DatabaseReference db = FirebaseDatabase.getInstance().getReference().child("almacen");
```

Para recoger un dato de la base de datos es necesario realizarlo con un **Listener**, puesto que es una base de datos en tiempo real, y nos mostrará los datos modificados. Los datos recogidos los parsea a un objeto, si se recoge más de uno.

CÓDIGO:

```
ValueEventListener evento = new ValueEventListener() {  
    @Override  
    public void onDataChange(DataSnapshot dataSnapshot) {  
        Clase c = dataSnapshot.getValue(Clase.class);  
    }  
  
    @Override  
    public void onCancelled(DatabaseError databaseError) {  
    }  
};  
  
db.addValueEventListener(evento);
```

Para enviar un nuevo dato al servidor de *Firebase*, es necesario también enviarlo parseado, es decir, el objeto completo.

CÓDIGO:

```
db.push().setValue(p);
```

1.5. Procesamiento de objetos multimedia. Clases. Estados, métodos y eventos.

- Mapas

Como se ha comentado anteriormente, en la consola de Google nos permite crear un proyecto de Google API Maps. Una vez creado, en vez de ser asignado a un proyecto Android como en Firebase, nos ofrece una clave que se podrá pegar en todos los proyectos que utilicen el mapa.

Android Studio nos ofrece la posibilidad de crear una actividad con la plantilla de un mapa, por lo que nos facilitará crear el layout desde el principio. Una vez que tenemos nuestro mapa creado, podemos indicar la clave obtenida en el Google API Maps en el archivo `google_maps_api.xml` que se encuentra en la carpeta `res/values`.

CÓDIGO:

```
<string name="google_maps_key" templateMergeStrategy="preserve" translatable="false">YOUR_KEY_HERE</string>
```

Clases para el uso del mapa:

- CameraUpdate.
- GoogleMap.
- LocationManager.

- Sensores

Los sensores son dispositivos que recogen información del medio exterior. Todos los dispositivos Android contienen una serie de sensores que permiten un mejor uso del mismo.

Las clases utilizadas para el uso de sensores se encuentran en el paquete `android.hardware` y son: `Sensor`, `SensorEvent`, `SensorManager` y `SensorEventListener`.

Los sensores en los dispositivos Android no siempre están disponibles, por ello, siempre es necesario comprobarlo antes de utilizarlos.

Algunos de los sensores más importantes de los dispositivos son:

- Acelerómetro.
- Gravedad.
- Giroscopio.
- Acelerador lineal.
- Rotación.
- Sensor de proximidad.
- Luminosidad.
- Presión.
- Temperatura.
- Humedad.

1.6. Reproducción de objetos multimedia. Clases. Estados, métodos y eventos.

Existe una gran cantidad de clases de Android que permiten utilizar distintos recursos multimedia. Estas son algunas de ellas.

Para audio:

- **AudioManager**: gestiona distintas propiedades del audio.
- **AudioTrack**: reproduce audio.
- **AsyncPlayer**: reproduce audio en una hebra secundaria.
- **SoundPool**: reproduce audio.

Para audio y vídeo:

- **JetPlayer**: reproduce audio y vídeo.
- **MediaController**: permite visualizar los controles de MediaPlayer.
- **MediaPlayer**: reproduce audio y vídeo.
- **MediaRecorder**: graba audio y vídeo.

Para vídeo:

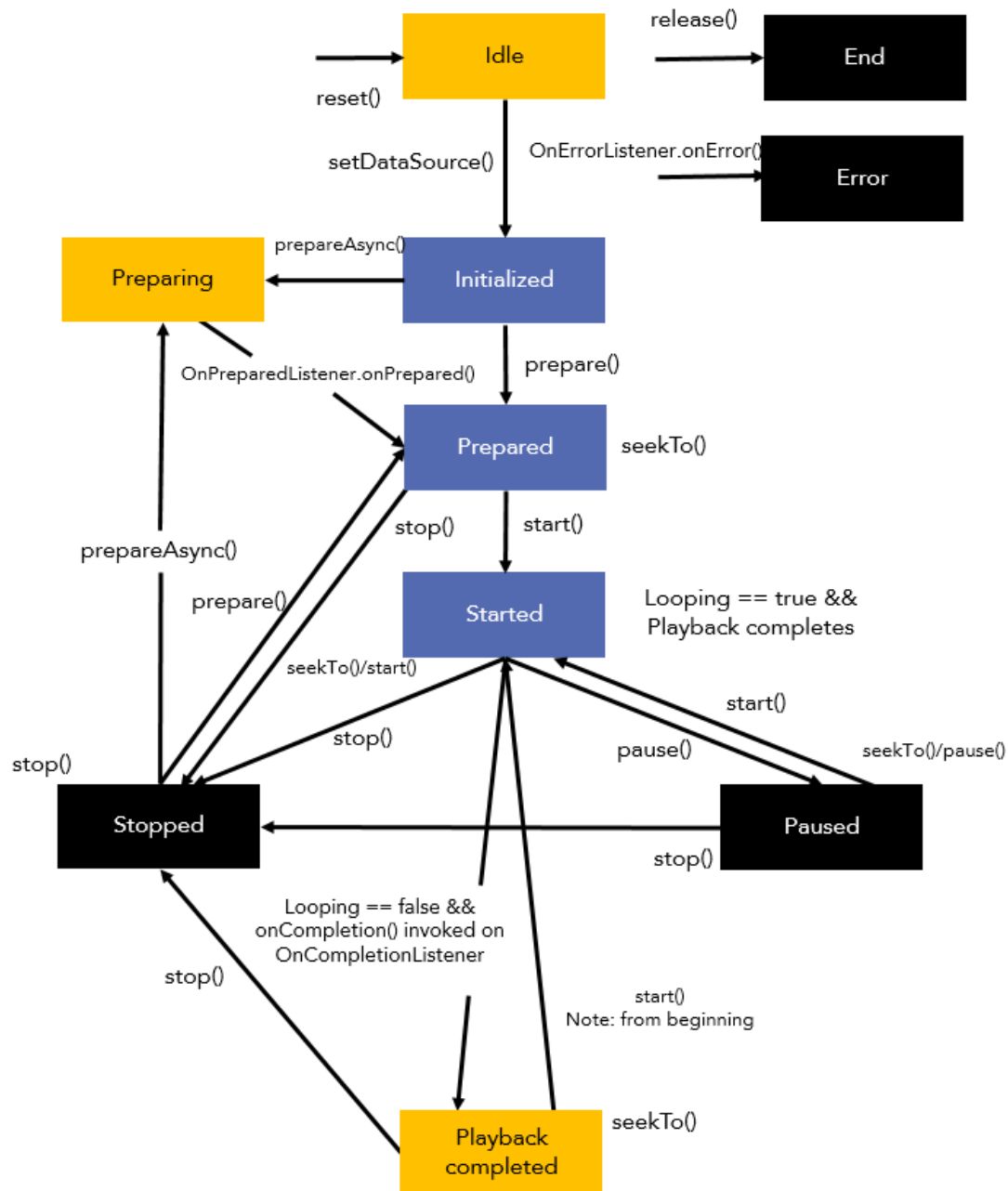
- **VídeoView**: reproduce vídeo.

Para la cámara:

- **Camera**: permite utilizar la cámara para fotos y vídeos.
- **FaceDetector**: detecta caras de la cámara.

Una vez que se han conocido las clases que permiten trabajar con objetos multimedia, vamos a centrarnos en MediaPlayer y Camera.

- MediaPlayer



Como se puede observar en la imagen, un audio pasa por muchos estados, desde que se llama al recurso hasta que finaliza su reproducción.

Siempre que se quiera reproducir un archivo de música, primero hay que indicar la ruta del archivo. Después es necesario preparar el audio para su posterior ejecución.

Hay que tener en cuenta que la preparación del audio puede llevar un tiempo, por lo que es necesario realizarlo en segundo plano.

- **Camera**

Para poder utilizar la cámara de fotos en una aplicación, se puede realizar de dos formas: utilizando la cámara directamente, o utilizando la aplicación cámara.

Si realizamos la primera de las opciones, es necesario dar permiso CAMERA a la aplicación para poder hacer uso de ella. Si utilizamos la segunda opción, no es obligatorio tener acceso a la cámara.

Una vez que se ha abierto la cámara en la propia aplicación, es necesario detectar la cámara, y si es así, acceder a ella mediante el método **Camera.open()**. Una vez que se ha terminado de utilizar la cámara, es necesario liberar los recursos con el método **release()**.

UF3: Desarrollo de juegos para dispositivos móviles

1. Análisis de motores de juegos.

1.1. Conceptos de animación.

En la actualidad, los dispositivos móviles forman parte del día a día de las personas, agrupando dentro de un mismo dispositivo todas las necesidades de los usuarios. Este hecho hace que los dispositivos móviles sean usados tanto para motivos laborales o de comunicación, como también para el ocio. Aquí es donde aparece entonces el desarrollo de aplicaciones animadas para los dispositivos.

Una animación es el cambio de alguna de las propiedades de un objeto que permiten ver a este a lo largo del tiempo con un aspecto diferente.

La base de creación de estas animaciones reside en la programación. Android ofrece una serie de mecanismos dedicados a la creación de animaciones para objetos tanto 2D como 3D.

- **Canvas:** la plantilla o lienzo que permite definir en las aplicaciones un control a nivel de interfaz de usuario. Este canvas puede suponer la representación de cualquier objeto como óvalos, líneas, rectángulos, triángulos, etc.
- **Animadores (Animators):** es la propiedad que permite añadir a cualquier objeto una determinada animación mediante el uso de propiedades o estilos de programación.
- **Dibujables Animados (Drawable Animation):** permite cargar una serie de recursos *Drawable* para crear una animación. Utiliza animaciones tradicionales como, por ejemplo, poner una imagen detrás de otra en orden como si fuera una película.
- **OpenGL:** es una de las librerías para gráficos de alto rendimiento 2D y 3D más importantes. Android incluye soporte para su utilización.

1.2. Arquitectura del juego: componentes.

Antes de comenzar con el desarrollo de un juego para plataformas móviles, es necesario previamente definir cuál va a ser su arquitectura.

Esta arquitectura permitirá detallar cómo será la estructura de desarrollo de la aplicación.

Dicha estructura estará formada por una serie de bloques con una función específica dentro del juego.

- **Interfaz de usuario:** se encargará de recoger todos los eventos que el usuario ha creado.
- **Lógica del juego:** es la parte central del videojuego. Se encargará de procesar todos los eventos del usuario y dibujar continuamente la escena del juego. Esto es lo que se conoce como **Game loop**. Se comprueba continuamente el estado del juego y se dibuja para cada estado una nueva interfaz, repitiendo este proceso de forma casi infinita.

En este bloque se controlarán también las colisiones y los *sprites*. Mediante el uso de librerías como OpenGL se modelan y diseñan los diferentes personajes dentro del juego. A través de esta, ella es posible haciendo uso de *sprites* (imágenes con transparencia) generar animaciones en personajes. Esto se conoce como renderizado de canvas.

Cada renderizado quedará encapsulado en un frame de animación.

- **Recursos utilizados:** dentro de la lógica del juego también es necesario controlar todos los efectos de sonido e imágenes.

Este bloque es la parte fundamental para el desarrollador. Todas las funciones tienen que ser programadas y controladas a través de código.

Los juegos a diferencia de las aplicaciones necesitan de un mayor consumo de los recursos del dispositivo, pudiendo, en muchos casos, utilizar casi la totalidad de los mismos. Es, por tanto, tarea del programador optimizar el uso de estos recursos.

- **Framework Android:** Android proporciona un framework potente que permite animar una gran cantidad de objetos, y representar dichos objetos de una gran multitud de formas. Para ello cuenta con diferentes mecanismos que ofrecen al desarrollador estas funcionalidades:

- Animación de propiedades (Property Animations): permite definir algunas propiedades de un objeto para ser animado, definiendo opciones como son: duración de una animación, tiempo de interpolación, repetición de comportamientos o animaciones, agrupar series de animaciones de forma secuencial, frames de actualización de una animación.
- Animación de vistas (View Animations): permite hacer uso de los diferentes mecanismos de animación de vistas, como son: translaciones, rotaciones, escalados, etc. Dando la sensación de transformación de una imagen en otra en un determinado momento.
- Animación de Dibujables (Drawable Animations): haciendo uso de los recursos *Drawable* permite crear una serie de animaciones como si de una película se tratara.
- **Salida**: Serán la sucesión de escenas que se van actualizando en la interfaz de usuario.

Estos frames son procesados uno detrás de otro ofreciendo al usuario una sensación de continuo movimiento y animación. Todos estos bloques de frames dan como resultado la escena del juego en los diferentes momentos del tiempo.

1.3. Motores de juegos: tipos y utilización.

Un motor gráfico o motor de videojuego es la representación gráfica de una serie de rutinas de programación, que ofrecen al usuario un diseño en un escenario gráfico 2D o 3D.

La principal tarea de un motor es proveer al juego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, scripts de programación, animaciones, inteligencia artificial, gestión de memoria y un escenario gráfico.

En la actualidad existen una gran variedad de motores gráficos, como: Ogre (que es open source), Doom Engine, Quake Engine, Unity, cryengine, source engine, Unreal engine, Game Maker, etc.

Estos motores suelen proporcionar:

- API y SDK para el desarrollo.
- Algunos motores permitir crear juegos sin necesidad de escribir código. Tan sólo es necesario hacer uso de los diferentes mecanismos implementados y documentados en su API. Algunos requieren del uso de su propio lenguaje de programación.
- Conjunto de herramientas de edición.
- A pesar de no proporcionar conjuntos ya creados de elementos visuales, permiten su creación a través de las herramientas de edición del software proporcionado.

Es posible clasificar los motores por tipos en función de:

- **Según las facilidades ofrecidas:**
 - o Librerías gráficas: según sus facilidades para el desarrollo y uso. SDL, XNA, DirectX, OpenGL.
 - o Motores: si el motor ya tiene un desarrollo visual completo o requieren scripts de programación para la utilización de elementos visuales, por ejemplo: OGRE, Unreal o id Tech, son algunos de los que requieren del uso de scripts de apoyo para la funcionalidad.
 - o Herramientas de creación especializadas: algunos de los motores se han desarrollado con un carácter exclusivo, orientadas en su finalidad como pueden ser videojuegos u otros tipos, por ejemplo: GameMaker o ShiVa, son para el desarrollo exclusivo de aplicaciones de juegos. Unity, por ejemplo, es posible utilizarlo para diferentes géneros.
- **Según la licencia:**
 - o Motores privados (algunos con licencia gratuita).
 - o Motores OpenSource.

La elección de un determinado motor gráfico es muy subjetiva. En la mayor parte de las ocasiones esta elección estará condicionada por el tipo de aplicación del juego a desarrollar, y los recursos disponibles de los que se dispone para su desarrollo.

1.4. Áreas de especialización, librerías utilizadas y lenguajes de programación.

La elección y utilización de librerías gráficas durante el desarrollo de un juego siempre es necesaria. Esto nos permite realizar determinadas animaciones sobre objetos de una manera más sencilla. Estas librerías se pueden clasificar por áreas de especialización en función de su funcionalidad:

Renderizados y efectos:

– OpenGL, Direct3D, GKS, PHIGS, PEX, GKS, etc...

Basados en gráficos de escena:

– OpenGL Performer, Open Inventor, OpenGL Optimizer, PHIGS+, etc...

Librerías de herramientas gráficas:

– World Toolkit, AVANGO, Game Engines, etc...

A la hora de programar un videojuego, se puede hacer en una multitud de lenguajes. Los lenguajes más utilizados en el desarrollo de videojuegos son **C, C++, C# y java**.

El uso de un lenguaje u otro viene definido por el tipo de juego que se quiere desarrollar. Los juegos 2D o plataformas, que trabajan con sprites sencillos, normalmente están basados en el lenguaje C. En el caso de juegos cuya complejidad gráfica es mayor, sobretodo cuando se va a trabajar con objetos tridimensionales y sus propiedades, la mayor parte de programadores usan C++. C# y Java, al requerir del uso de una máquina virtual, en ocasiones son menos elegidos, aunque su potencia en desarrollo de juegos también es alta.

Existe una gran cantidad de librerías que se pueden utilizar durante el desarrollo de un juego. Algunas de las librerías son las siguientes:

- **Allegro**: librería libre y de código abierto basada en lenguaje C. Permite el uso de elementos gráficos, sonidos, dispositivos como teclado y ratón, imágenes, etc.
- **Gosu**: librería que permite el desarrollo de juegos en 2D y basada en lenguaje C++ y Ruby. Es de software libre bajo licencia del MIT. Provee de una ventana de juego, que permite del uso de teclado, ratón y otros dispositivos. Se caracteriza por el uso para sonidos y música dentro de un juego.
- **SDL**: conjunto de librerías para el diseño de elementos 2D. También de software libre. Permite la gestión de recursos multimedia como

sonidos y música, así como el tratamiento de imágenes. Esta basada en lenguaje C, aunque permite el uso de otros lenguajes, como: C++, C#, Basic, Ada, Java, etc.

- **libGDX**: basada en Java. Librería orientada a su uso en aplicaciones multiplataforma, permite escribir el código en un solo lenguaje y posteriormente exportarlo a otros. Permite una integración fácil con otras herramientas Java.
- **LWJGL**: librería destinada al desarrollo de juegos en lenguaje Java. Proporcionada a su vez acceso al uso de librerías como OpenGL.
- **OpenGL**: librería de gráficos para el desarrollo de juegos 2D y 3D. Es una de las librerías más utilizadas hoy en día. Es de software libre y código abierto. Permite el uso de elementos básicos como las líneas, puntos, polígonos, etc. Así como otros elementos de mayor complejidad, como son: texturas, transformaciones, iluminación, etc.
- **Direct3D**: conjunto de librerías multimedia. Es el gran competidor de OpenGL en el mundo de los juegos. Permite el uso de elementos como líneas, puntos o polígonos. Así como gestión de texturas o transformaciones geométricas. Propiedad de Microsoft.

1.5. Componentes de un motor de juegos.

Un motor de juegos es una parte fundamental del código de programación de un juego. Este motor gráfico se encarga de la mayor parte de los aspectos gráficos de un juego.

Una de sus tareas es establecer la comunicación y aprovechar todos los recursos que una tarjeta gráfica ofrece.

Los componentes principales de un motor de juegos son:

- **Librerías**: todas aquellas librerías de las que se hace uso para el desarrollo de figuras, polígonos, luces y sombras, etc.
- **Motor físico**: es el encargado de gestionar las colisiones, animaciones, scripts de programación, sonidos, físicas, etc.
- **Motor de renderizado**: es el encargado de renderizar todas las texturas de un mapa, todos los relieves, suavizado de objetos, gravedad, trazado de rayos, etc.

Estos componentes recogen de manera global todos los elementos que aparecen dentro de un juego. Cada uno de estos elementos forma parte de un conjunto de recursos que en todo motor gráfico se pueden encontrar:

- **Assets:** todos los modelos 3D, texturas, materiales, animaciones, sonidos, etc. Este grupo representa todos los elementos que formarán parte del juego.
- **Renderizado:** todas las texturas y materiales en esta parte hacen uso de los recursos diseñados para el motor gráfico. Esto mostrará el aspecto visual y potencial de un motor gráfico.
- **Sonidos:** es necesario configurar dentro del motor como serán las pistas de audio. El sonido del videojuego dependerá de la capacidad de procesamiento de estos sonidos. Algunas de las opciones configurables son: modificación del tono, repetición en bucle de sonidos ("*looping*"), etc.
- **Inteligencia Artificial:** es una de las características más importantes que puede desarrollar un motor gráfico. Esto añade estímulos al juego, permitiendo que el desarrollo del mismo suceda en función de una toma de decisiones definidas por una serie de reglas. Define el comportamiento de todos los elementos que no son controlados por el jugador usuario, sino que forman parte de los elementos del juego.
- **Scripts visuales:** no sólo es posible ejecutar porciones de código definidas en el juego, sino que además se pueden ejecutar en tiempo real dentro del aspecto gráfico del juego.
- **Sombreados y luces:** el motor gráfico dota de los colores y sombras a cada uno de los vértices que forman parte de la escena.

Como se ha podido comprobar las tareas que componen un motor gráfico exigen la utilización de un gran número de recursos dentro del equipo. De ahí que, cuanto mayor sea la capacidad de procesado y velocidad de una tarjeta gráfica, mejor será el resultado de la escena de un juego. Para reducir el coste de esto, algunos motores emplean una serie de técnicas que permiten renderizar los terrenos o los materiales y que no consuman recursos, sino que aparecen dentro del espacio visual, esto se conoce como *culling*.

Los motores gráficos son un aspecto clave dentro de un juego. Han sido creados exclusivamente para el desarrollo de los mismos, y hoy en día son la herramienta fundamental de creación de videojuegos. La evolución de los

juegos y el entretenimiento está ligada a la evolución de los motores de juegos.

1.6. Librerías que proporcionan las funciones básicas de un motor 2D/3D.

Como se ha comentado ya en algunos de los apartados anteriores, las librerías son un apartado clave en el proceso de desarrollo de un juego. Para dotar a un objeto o elemento de un aspecto más realista es necesario que los motores gráficos procesen una serie de funciones, que dibujan en 2D o 3D dichos objetos. Ya que el diseño de un objeto requiere de una gran labor de programación, y este será representado en muchas ocasiones en un juego, la creación de estos objetos queda recogida en una serie de funciones que son proporcionadas por las librerías.

Estas librerías permiten abstraer al programador de los aspectos más complejos de representación de elementos visuales, tan sólo es necesario llamar a la función de la librería encargada de esto y recoger el objeto devuelto para su representación en la escena.

Las funciones básicas utilizadas por los motores gráficos son aquellas que permiten trabajar con elementos visuales, como son puntos, rectas, planos o polígonos. Y proveen de los recursos fundamentales en un juego como sonidos y música. El apartado de modelado de personajes deberá recoger el uso de sprites para 2D, y el uso de modelos (assets) para el desarrollo de plataformas de juego 3D.

1.7. API gráfico 3D.

OpenGL es una API de dibujo 3D, y permite realizar aplicaciones que producen gráficos. Esta API se compone de un gran número de funciones para la creación de elementos y objetos tridimensionales.

El objetivo de estas API es proveer al desarrollador de un documento donde poder encontrar todos los recursos, y, de esta forma, disminuir la complejidad en la comunicación con las tarjetas gráficas.

El funcionamiento de este tipo de librerías consiste en tratar de aceptar como entrada una serie de primitivas, como: líneas, puntos y polígonos, y convertirlos en píxeles.

OpenGL actualmente tiene la versión 4. Cada una de estas versiones ha ido desarrollando una evolución en cuanto a texturas, formas y transformaciones de los objetos. Es posible hacer uso de cualquiera de ellas, empleando la documentación proporcionada en su página oficial. Esta documentación ofrece numerosos códigos de ejemplo, libros o videotutoriales para hacer uso de la librería deseada.

A partir de la versión 3, OpenGL desarrolló su propio lenguaje de renderizado llamado GLSL. Este permite llevar a cabo mediante programación el desarrollo de una escena.

Otra API que nos ofrece este mismo tipo de gráficos 3D es Direct 3D. Ofrece una API 3D de bajo nivel, en ella se pueden encontrar elementos básicos, como: sistemas de coordenadas, transformaciones, polígonos, puntos y rectas.

Es una librería con recursos gráficos, que exigen de un nivel de programación experimentado en este tipo de recursos. Uno de los puntos fuertes de este API es que es independiente del tipo de dispositivo a utilizar, lo que permite un desarrollo más versátil.

1.8. Estudio de juegos existentes.

En la actualidad, el mercado de juegos para dispositivos es muy grande. Ya existen una gran cantidad de juegos para todos los tipos de géneros posibles. Esto, en muchas ocasiones, dificulta el éxito de algunos juegos. Por ello, es recomendable antes de su desarrollo hacer un estudio de mercado, centrando la atención en aquellos juegos de carácter similar al juego que se va a crear.

Si el juego va a ser publicado en Internet, es importante conocer a qué tipo de público está destinado. De igual forma es bueno conocer cuáles son las limitaciones de desarrollo, así como medir la cantidad de recursos necesarios para su creación, desarrollo y publicación.

En muchas ocasiones el desarrollo de un tipo de género de videojuego está relacionado con el éxito de alguno de ellos. Cuando las descargas de un juego determinado aumentan considerablemente, significa que ese tipo de juego es un buen reclamo para los usuarios. Este hecho puede ser aprovechado por desarrolladores de juegos con menos recursos para introducirse dentro del mercado.

En la industria de los juegos para dispositivos móviles tienen aceptación tanto los juegos 2D como los 3D, por lo que el abanico de posibilidades es ilimitado.

1.9. Aplicación de modificaciones sobre juegos existentes.

Los juegos Android ocupan prácticamente el mayor porcentaje de descarga de aplicaciones relacionadas con el ocio. Por lo que es posible encontrar ejemplos de juegos ya probados y cuyo éxito ha sido medido. El proceso de creación de un nuevo juego no es sencillo, y muchas veces esto obliga a que este desarrollo sea llevado a cabo por un gran número de personas de especialidades diferentes, que realizan trabajo dentro del proyecto de forma conjunta. Sin embargo, el ritmo de vida en la sociedad obliga continuamente a estos equipos a renovarse, por lo que han de hacer que ese juego se adapte a los nuevos tiempos, añadiendo nuevas funcionalidades y optimizando su rendimiento en los dispositivos.

A diferencia de las aplicaciones, los juegos por lo general no suelen ser de código abierto, por lo que no es posible añadir de forma legal nuevas modificaciones al juego si no formas parte del equipo de desarrollo o eres el autor de uno de ellos.

Cuando se publica un juego en Internet en plataformas como Google Play, se adquiere un compromiso de mantenimiento de esa aplicación. En la que los desarrolladores han de corregir en la medida de lo posible todos aquellos errores que se detecten, tanto por parte de los usuarios como de los propios desarrolladores. Este control de modificaciones es muy importante para el posible éxito de un juego.

2. Desarrollo de juegos 2D y 3D.

2.1. Entornos de desarrollo para juegos.

Hacer un juego no es una tarea sencilla, puesto que requiere conocimientos de diferentes especialidades, como son: programación, diseño y animación. Para hacer un poco más fácil esta tarea están los entornos de desarrollo. Estos son unas plataformas software que ofrecen una interfaz gráfica para la creación de juegos mediante el uso de una serie de herramientas.

Existen diferentes tipos de entornos que están orientados a un tipo de juegos en concreto bien sea para 2D o para 3D. También es posible encontrar diferentes entornos dependiendo de la complejidad del juego a realizar.

Si el objetivo es realizar juegos sencillos cuya interfaz no sea muy exigente para plataformas en 2D, es posible hacer uso de entornos como:

- **Stencyl:** es una plataforma que permite creación de juegos en 2D mediante el uso de bloques de código, que ayudan comprender las estructuras básicas de programación, por lo que no es necesario desarrollar líneas de código. Permite añadir imágenes para los personajes, que se añaden a una escena simplemente arrastrándolos. Se trata de una plataforma sencilla y fácil de utilizar.
- **Pygame:** entorno de desarrollo de juegos que utilizan el lenguaje Python. Permite la creación de juegos en 2D. Se basa en el uso de *sprites* para los personajes y bibliotecas de recursos de sonido y multimedia. La programación es algo más compleja, ya que es necesaria realizar las estructuras de control y las variables a través de código.

Cuando el juego a realizar requiere de una potencia gráfica mayor, como es el caso del 3D, es necesario que los entornos de desarrollo a su vez sean más completos.

Algunos de los más importantes son:

- **Unity 3D:** a día de hoy Unity es una de las herramientas más utilizadas en el mundo de los juegos, así como una de las mejor valoradas. Unity permite exportar un juego creado en cualquiera de los distintos tipos de dispositivos. Unity está basado en el lenguaje C#. Unity tiene un motor propio para el desarrollo de la parte gráfica lo que permite llevar a cabo un desarrollo muy completo de todas las escenas de un juego. Es posible configurar todos los elementos necesarios, como pueden ser: la iluminación, las texturas, los materiales, los personajes, los terrenos, los sonidos, las físicas, etc.
- **Unreal Engine:** junto con Unity 3D, es uno de los entornos más conocidos y valorados dentro del mundo del desarrollo de juegos. Permite la configuración y diseño de recursos gráficos avanzados de la misma forma que Unity.

Ambos entornos requieren de un nivel significativo de programación.

2.2. Integración del motor de juegos en entornos de desarrollo.

Una vez se ha configurado Android en el equipo, es necesario de igual forma configurar la integración de Android dentro del entorno de desarrollo. En este caso el entorno escogido es Unity. Este debe conocer dónde está el SDK para poder compilar y, posteriormente, enviar la aplicación al dispositivo.

Para ello, el primer paso es, una vez seleccionado el proyecto, ir al menú de edición y seleccionar preferencias. Esto mostrará la ventana de preferencias de Unity, dentro de ella, en el apartado de herramientas externas, se podrán visualizar los distintos parámetros de configuración del compilador. Es posible elegir el editor para la programación de las líneas de código, así como la ruta donde se encuentra en el equipo el SDK de Android instalado.

Una vez se ha seleccionado esta ruta, Unity será capaz de compilar un proyecto para Android. Para realizar la compilación hay que seleccionar el menú configuración de la compilación. En esta ventana se podrá elegir cuál será la plataforma de compilación, en este caso Android. Esto realizará todo el proceso de renderizado de gráficos, así como de programación para dicha plataforma. En este punto se añadirán las escenas deseadas para compilar. Y una vez escogida la plataforma, se tiene que seleccionar la opción de compilar.

Android no permite compilar sin un identificador de paquete. Por lo que será necesario definir dicho identificador que, posteriormente, será usado por Google Play para su publicación. Dentro del apartado de ajustes del proyecto se especificarán todos los apartados del paquete.

Estos apartados son los siguientes:

- Resolución y presentación de la aplicación.
- Icono de la aplicación.
- **Splash image:** será la imagen previa al comienzo del juego una vez se inicia la aplicación.
- **Renderizado:** ajuste de parámetros de renderizado para Android.
- **Identificación:** en este apartado se especificará el identificador del paquete, que, por lo general, suelen ser el nombre de la estructura del proyecto.
- Versión del código.

- Nivel mínimo del API de Android.
- **Versión de gráficos utilizados.** Lugar de instalación de la aplicación por defecto en el dispositivo.

Por último, una vez completados todos estos apartados, se generará en el equipo un **.apk** (extensión de las aplicaciones en Android), que será el archivo ejecutable que instalaremos en el dispositivo deseado.

2.3. Conceptos avanzados de programación 3D.

El desarrollo y programación de un juego tridimensional conlleva aplicar algunos conceptos que son de carácter avanzado, como son: los movimientos, las físicas, y las colisiones. Estos harán que el juego sea lo más próximo a una escena real. Unity ofrece una serie de clases que permiten definir y configurar estas propiedades sobre modelos de personajes y objetos.

Esta clase se denomina controlador de personajes (*character controller*), esto permite aplicar físicas y colisiones en forma de capsula a los personajes. Proporciona un simple colisionador (*simple collider*). Esto hará que el personaje camine por el suelo y no suba por las paredes.

Los tipos de colisionador que existen son:

- **Box collider:** se trata de una colisión en forma de cubo. Estos generalmente son empleados en objetos de forma cúbica, como, por ejemplo, una caja o un cofre.
- **Capsule collider:** se trata de una cápsula de forma ovalada formada por dos semi-esferas.
- **Mesh collider:** son colisionadores más precisos, que van asociados a objetos 3D ya diseñados. Esto permite crear un colisionador ajustado a la forma del objeto completamente.
- **Sphere collider:** es un colisionador básico de forma esférica. Suele ser aplicado en objetos esféricos, como: pelotas, piedras, etc. Este efecto tiene un gran impacto en objetos, que aparecen rodando en la escena o que se están cayendo, por ejemplo.

Otro de los conceptos que son frecuentemente usados en el desarrollo de juegos, es el sistema de partículas de Unity. No siempre los objetos que se van a representar en una escena son sólidos o son elementos con formas bien definidas. Por ello cuando se requiere realizar la representación de fluidos,

por ejemplo, o líquidos que están en movimiento, humo, nubes, llamas, etc. Es necesario hacer uso de los efectos que proporciona el sistema de partículas. Este sistema de partículas son imágenes simples y generalmente pequeñas, que aparecen en la escena repitiéndose continuamente en una dirección. Esto hace que todas ellas representen, en conjunto, un elemento único para el usuario. Es necesario definir cuál será la forma de estas pequeñas imágenes, durante cuánto tiempo se mostrarán estas imágenes, y con qué frecuencia y cantidad aparecerán en la escena.

Por último, otro de los conceptos y, probablemente el más complejo, es la **Inteligencia Artificial (IA)**. Esta permite en Unity crear personajes que sean capaces de interactuar en la escena incluso evitando colisiones entre los elementos de la misma. Esta herramienta en Unity recibe el nombre de **NavMesh**.

Para ellos a través del inspector de creación de un agente se definen las propiedades que van a caracterizar al mismo, como son:

- **Radio:** radio que el personaje tendrá a la hora de moverse para evitar colisiones.
- **Altura:** define la altura máxima de los obstáculos por los que el personaje podrá acceder pasando por debajo de ellos.
- **Velocidad:** velocidad máxima en unidades/segundo que tendrá el personaje.
- **Aceleración:** aceleración del movimiento y acciones del personaje.
- **Área:** definirá el camino que tomará el personaje y cuáles no podrá escoger.

2.4. Fases de desarrollo.

La creación de un nuevo juego exige de una gran cantidad de tareas que se pueden agrupar en distintas fases de la siguiente forma:

1º) Fase de Diseño: es el paso previo a la programación. En esta fase es necesario determinar cuáles serán todos los aspectos relevantes del juego. Escoger la temática y el desarrollo de la historia. También es importante establecer cuáles serán las reglas del juego.

Una vez documentada la historia, es necesario separar el juego en partes. Cada una de estas partes conformarán las pantallas del juego. Se tienen que

definir también cuál será el aspecto del menú dentro de las pantallas, así como la colocación de los objetos dentro de la misma.

2º) Diseño del código: se especificarán en esta fase todas las capas de las que se compone el juego. Se trata de separar todos los aspectos básicos del juego de la funcionalidad del mismo. Esto es lo que se conoce como *Framework*.

El *Framework* definirá cómo será el manejo de las ventanas del juego. Asegurar que los objetos ocupan el espacio correcto dentro de la ventana que corresponde. También se encargará del manejo de eventos de entrada de usuario. Estos en la mayoría de los casos serán recogidos del teclado o ratón del equipo.

Otra de las tareas del *Framework* es el manejo de ficheros, donde se llevarán a cabo las tareas de lectura y escritura. Como, por ejemplo, guardar las preferencias y puntuaciones del juego.

Manejo de gráficos, que determinan los píxeles mapeados en las diferentes pantallas. Es necesario determinar la posición a través de coordenadas de cada uno de los píxeles, así como del color.

Manejo de audio, para poder reproducir, por ejemplo, música de fondo en el juego.

3º) Diseño de los Assets: es una de las fases más complicadas y que mayor repercusión tiene en un juego. Son las creaciones de los diferentes elementos o modelos que se pueden utilizar dentro del juego, como pueden ser: los personajes, los logos, los sonidos, los botones, las fuentes, etc.

4º) Diseño de la lógica del juego: en esta fase es donde se define cómo se comportará el juego. Se aplicarán las reglas ya diseñadas, así como la programación del comportamiento de cada uno de los eventos del juego.

5º) Pruebas: una de las fases más importantes. Pues en este momento es cuando se realiza una comprobación de toda la aplicación, con el fin de comprobar el comportamiento del juego y la aplicación correcta del resto de fases.

6º) Distribución del juego: una vez se han finalizado todas las fases de desarrollo, el objetivo es que el producto sea distribuido. Para ello, es necesario exportar este juego de la misma forma que una aplicación.

2.5. Propiedades de los objetos: luz, texturas, reflexión y sombras.

Cada uno de los objetos representados dentro de una escena tiene unas determinadas propiedades, como son: luz, texturas, reflejos y sombras.

- **Luz:** es la que permite observar puntos de iluminación dentro de la escena. Esto dotará de vida al juego. La luz o un punto de luz sobre una parte en particular de una escena conllevan a centrar su atención en ella. Indicará la proyección de la cámara dentro del juego. Es posible añadir diferentes puntos de luz en una escena, así como configurar el color de una luz.
- **Texturas:** reflejan la calidad con la que pueden apreciar todos los objetos que aparecen dentro del entorno del juego. Forman parte de las texturas, por ejemplo, los materiales. Dentro de los materiales se pueden ver algunos como son: el agua, el metal, la madera, los tejidos, etc. Para conseguir un efecto real en una textura, a estos materiales se le aplican unos algoritmos matemáticos denominados *shaders*, estos permiten definir cuál será el color de cada uno de los píxeles que componen un objeto.
- **Reflejos y sombras:** estos añaden a los objetos una representación más realista. Para conseguir este efecto lo que se hace es añadir una especie de contorno a los componentes gráficos. Se define el color de esta sombra y cuál será la distancia aplicada en cada objeto. Las sombras suelen ir acorde a la proyección de la luz, de esta forma la sombra aparece como un efecto de dicha iluminación. Con los reflejos y sombras es posible establecer la posición que ocupa un objeto dentro de la escena.

Estas son algunas de las propiedades básicas de los objetos. Estas serán configuradas de forma particular para cada objeto en función de la posible interacción dentro de una escena.

2.6. Aplicación de las funciones del motor gráfico. Renderización.

Una de las labores de mayor complejidad que tiene un motor gráfico es el renderizado de los objetos que componen una determinada escena. El procesamiento de cada uno de ellos requiere de cierta cantidad de recursos

gráficos que, en la mayor parte de los casos, son ofrecidos por las tarjetas gráficas.

Se puede definir la renderización como el proceso de creación de una imagen 2D o 3D real dentro de una escena, aplicando una serie de filtros a partir de un modelo diseñado.

Algunas de las propiedades que definen el proceso de renderizado son:

- **Tamaño:** define el tamaño de la renderización en píxeles. En la mayor parte de los casos esto se llevarán a cabo sobre texturas.
- **Anti-Aliasing:** se utiliza para aplicar un filtro de suavizado sobre los objetos que, al ser renderizados, aparecen con formas escalonadas.
- **Depth buffer:** se encarga de definir la profundidad de los objetos 3D en una escena. Esto tiene un gran impacto en la calidad de la escena producida.
- **Wrap mode:** utilizado para definir el comportamiento de las texturas. Por ejemplo, en un terreno se define la repetición de una textura en concreto que será aplicada en toda la escena.

En entornos como Unity, es posible configurar estas propiedades de renderizado a través de la función *Render Texture*.

2.7. Aplicación de las funciones del grafo de escena. Tipos de nodos y de su utilización.

Unity ofrece una herramienta sencilla para la organización y gestión de animaciones llamada **Animator Controller**. Este permite crear dentro de Unity un grafo de acciones, que permiten controlar todas las animaciones de un personaje u objeto. Es posible establecer un orden de ejecución de las mismas en función de algunas de las reglas o condiciones del juego.

Esto, por ejemplo, permite definir el comportamiento de un personaje que de forma normal anda en una dirección y que, al pulsar la barra espaciadora, realizará un salto.

Cada uno de estos nodos será la representación de una acción del personaje. Estos reflejarán las transiciones entre los estados más básicos.

Su utilización suele ser durante el empleo de movimientos direccionales del personaje, que se repiten de forma periódica hasta el suceso de otro de los eventos. Estos movimientos suelen ser caminar hacia delante, hacia atrás o en diagonal. Otros de los nodos definirán estados como serán la muerte del personaje, caídas o colisiones con otros objetos de la escena.

2.8. Análisis de ejecución. Optimización del código.

Durante el desarrollo de un juego será necesario compilar y depurar el código muchas veces. Unity ofrece un IDE integrado (*Mono Develop*). Este será el editor predefinido para la programación del código del juego.

Cuando se está editando un archivo dentro del proyecto este aparecerá como una pestaña. El editor de texto permite añadir *breakpoints* en los márgenes al lado de cada una de las líneas de código que se desea. Una vez seleccionados estos puntos de parada se comienza la depuración del código a través del botón *debug*. Esto ejecutará el código, quedando parado en el primer punto de parada encontrado en el código. Esto permite ver los valores que han tomado todas las variables hasta ese momento.

Es posible además navegar entre los distintos puntos de parada para comprobar el correcto comportamiento de la aplicación.

En caso de producirse errores en la compilación, Unity contiene un archivo de *logs* denominado *Debug.log*, donde se almacenarán todos los mensajes mostrados en la consola. Lo más común es que en caso de existir errores en el código, el propio Unity, al compilar, no permitirá la ejecución del juego, y mostrará en la parte inferior un mensaje que referencia el error o los errores encontrados.

Otra herramienta que es útil dentro del IDE de Unity es el **Unity Test Runner**. Esta herramienta lo que hace es comprobar el código de programación en busca de errores antes de realizar una compilación. Lo que puede ser útil de cara a corregir errores de sintaxis, por ejemplo.

A parte de tener todas estas herramientas de depuración, es conveniente que el desarrollador tenga adquiridas una serie de buenas prácticas de programación y estructuración de código.

El código tiene que estar lo más limpio posible, esto ayudará posteriormente a la corrección y mejora de algunas funciones. En proyectos con un desarrollo

de código muy extenso esto puede sino suponer un problema de optimización muy grande.

Las funciones declaradas deben estar bien definidas, no deberían existir varias funciones cuyo comportamiento es el mismo.

Bibliografía

El gran libro de Android. Jesús Tomas. Marcombo.

Programación multimedia y dispositivos móviles. César San Juan Pastor. Garceta.

Android. Programación multimedia y dispositivos móviles. Iván López Montabán, Manuel Martínez Carbonell, Juan Carlos Manrique Hernández. Garceta.

Webgrafía

<https://developer.android.com/index.html>

<http://www.sgoliver.net/blog/curso-de-programacion-android/indice-de-contenidos/>

ILERNA

Online

```
function updatePhotoDescription() {  
    if (descriptions.length > (page * 9) + (currentImageSubsting() - 1) / 1) {  
        document.getElementById('bigImageDesc').innerHTML = descriptions[page * 9 + (currentImageSubsting() - 1) / 1];  
    }  
}  
  
function updateAllImages() {  
    var i = 1;  
    while (i < 10) {  
        var elementId = 'foto' + i;  
        var elementIdBig = 'bigImage' + i;  
        if (page * 9 + i - 1 < photos.length) {  
            document.getElementById(elementId).src = 'images/min/' + photos[page * 9 + i - 1];  
            document.getElementById(elementIdBig).src = 'images/max/' + photos[page * 9 + i - 1];  
        } else {  
            document.getElementById(elementId).src = 'images/min/default.jpg';  
            document.getElementById(elementIdBig).src = 'images/max/default.jpg';  
        }  
        i++;  
    }  
}
```