

# DESARROLLO DE APLICACIONES PARA MÓVILES





# ACTIVIDADES

- Las actividades representan una pantalla simple con una interfaz de usuario.
- Una App para gestionar pedidos podría tener una actividad para mostrar el listado de clientes, otra actividad para mostrar un producto determinado y una tercera actividad para mostrar los datos de un pedido concreto.
- Además de ser programas independientes, cada actividad puede ser ejecutada por otra App (si nuestra App lo permite).
- Para poder crear una actividad debemos crear una subclase de la clase Activity, con su propio ciclo de vida.



# ARQUITECTURA DE ANDROID

- Todos los componentes de nuestras Apps se programan en Java con las siguientes reglas:
  - El SO Android es multiusuario y cada App es un usuario diferente.
  - Por defecto, el sistema asigna un identificador a cada usuario (User ID) que es solo conocido por la aplicación y el SO. El sistema establece permisos para todos los ficheros de la App y solo la App puede acceder a ellos.
  - Cada proceso tiene su propia VM, con lo que el código de cada App se ejecuta aislado del resto de Apps, evitando así problemas de seguridad.



# ARQUITECTURA DE ANDROID

- Pese a estas restricciones hay formas para intercambiar información entre las distintas Apps.
- Para que una App pueda acceder a los recursos del dispositivo (contactos, cámara, bluetooth...) se le deben dar permisos en el momento de la instalación.

# ARQUITECTURA DE ANDROID



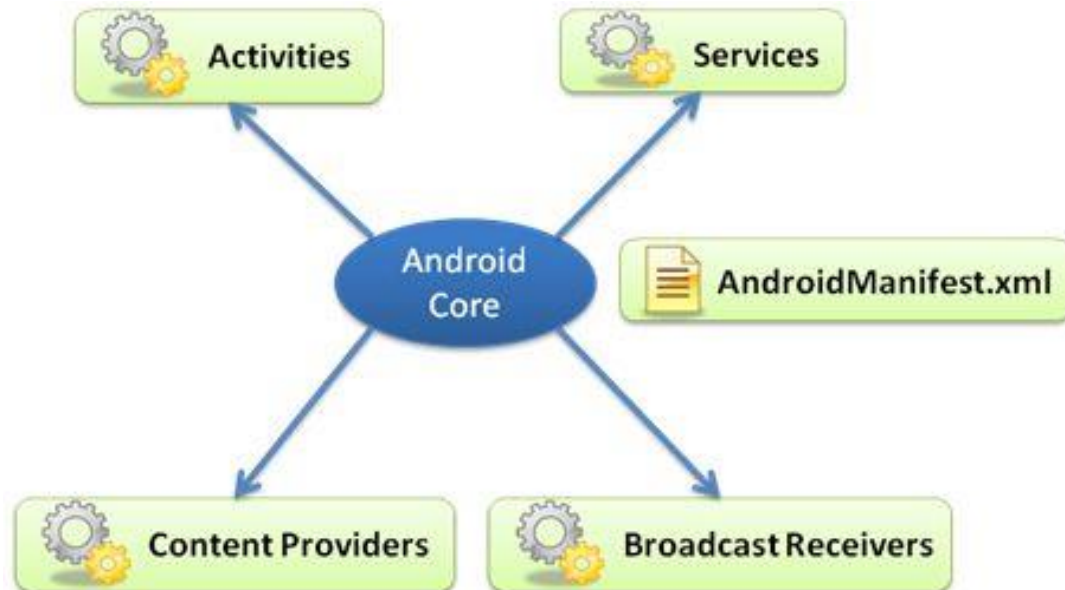


# COMPONENTES

- Cada componente de una aplicación es un punto de entrada a través del cual el sistema Android puede comunicarse con nuestras Apps.
- Cada tipo de componente tiene un propósito y un ciclo de vida propio que indica cuando un componente se crea y se destruye.

# COMPONENTES

- Hay cuatro tipos de componentes:
  - Actividades.
  - Servicios.
  - Proveedores de contenido.
  - Receptores de broadcast.





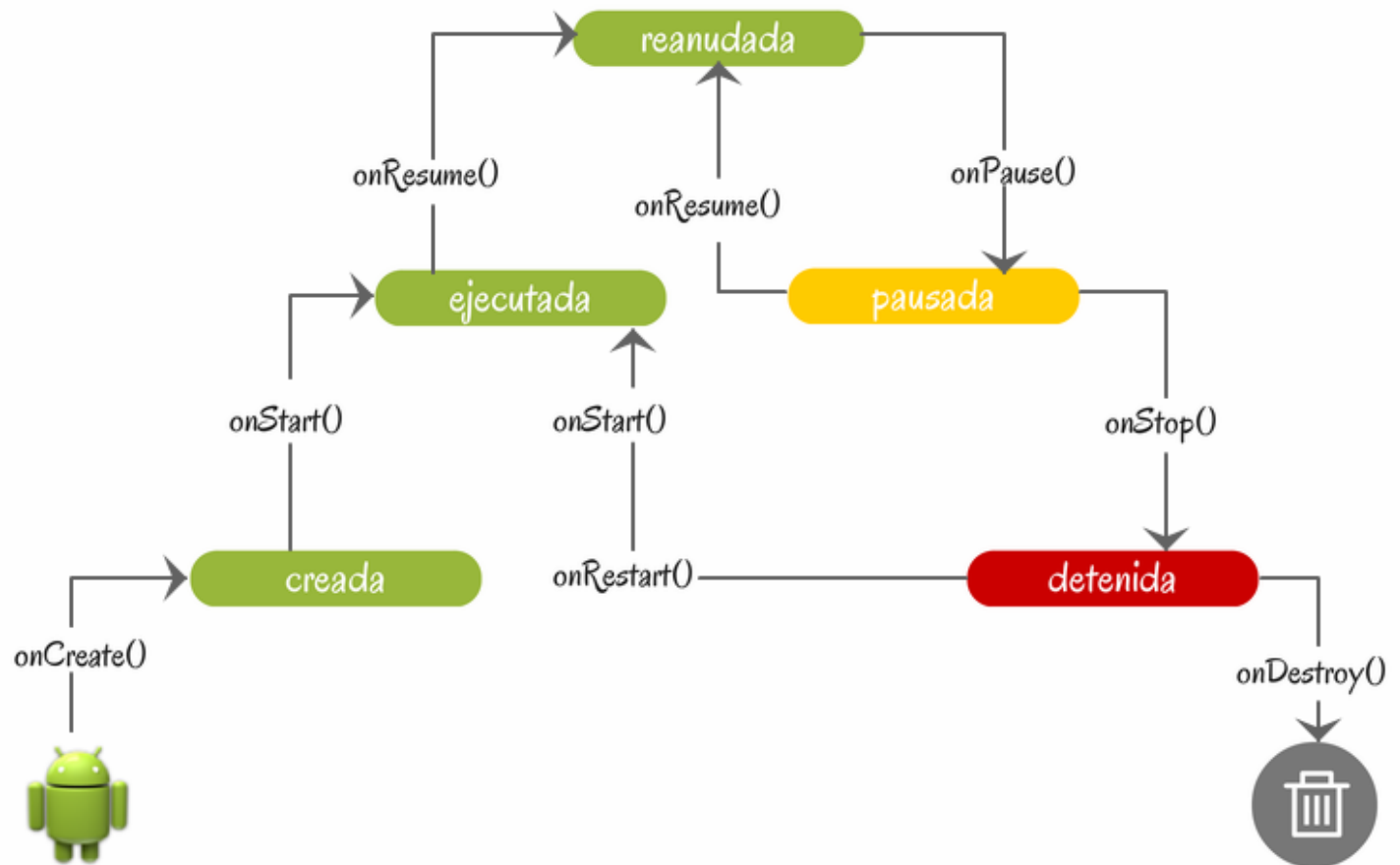
# ACTIVIDADES

- Las actividades representan una pantalla simple con una interfaz de usuario.
- Una App para gestionar pedidos podría tener una actividad para mostrar el listado de clientes, otra actividad para mostrar un producto determinado y una tercera actividad para mostrar los datos de un pedido concreto.
- Además de ser programas independientes, cada actividad puede ser ejecutada por otra App (si nuestra App lo permite).
- Para poder crear una actividad debemos crear una subclase de la clase **Activity**, con su propio ciclo de vida.



# ACTIVIDADES

## Ciclo de vida de una actividad

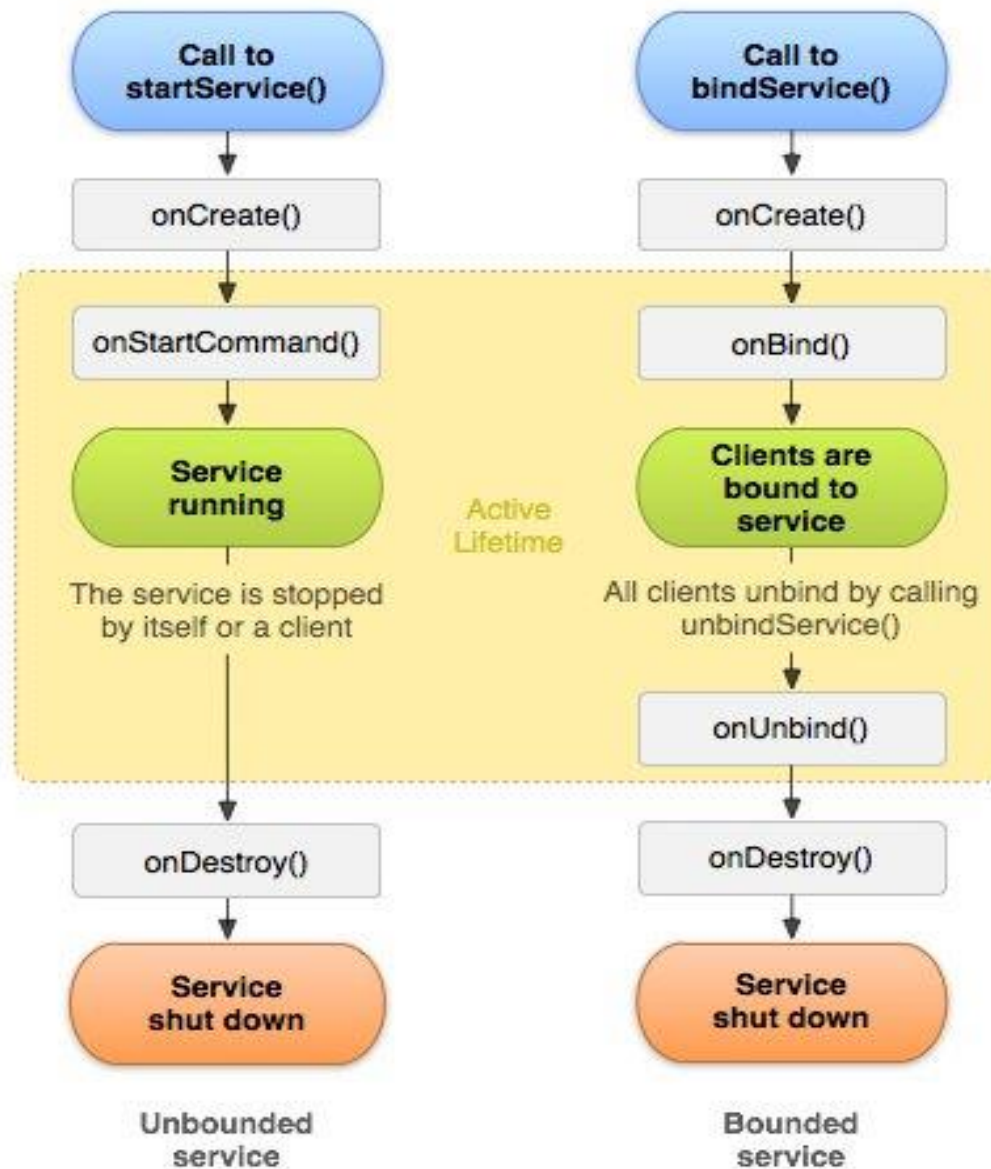




# SERVICIOS

- Un servicio es un programa que no tiene interfaz de usuario y que se ejecuta en *background*.
- Por ejemplo, un servicio podría estar conectado a la red para obtener los precios de los productos de nuestra App en tiempo real.
- Para poder crear un servicio debemos crear una subclase de la clase **Service**.

# SERVICIOS

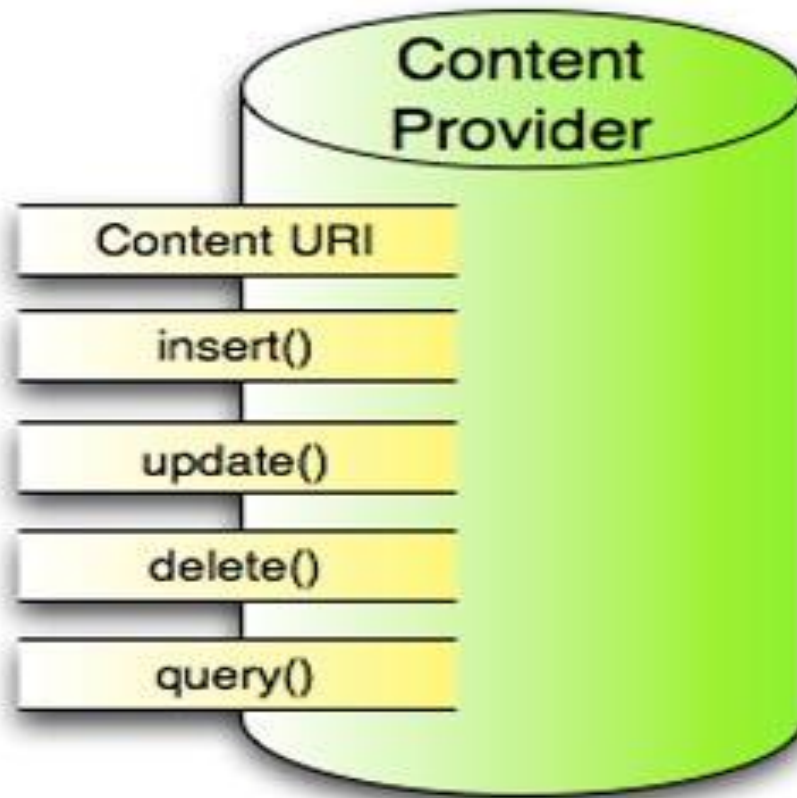




# PROVEEDORES DE CONTENIDOS

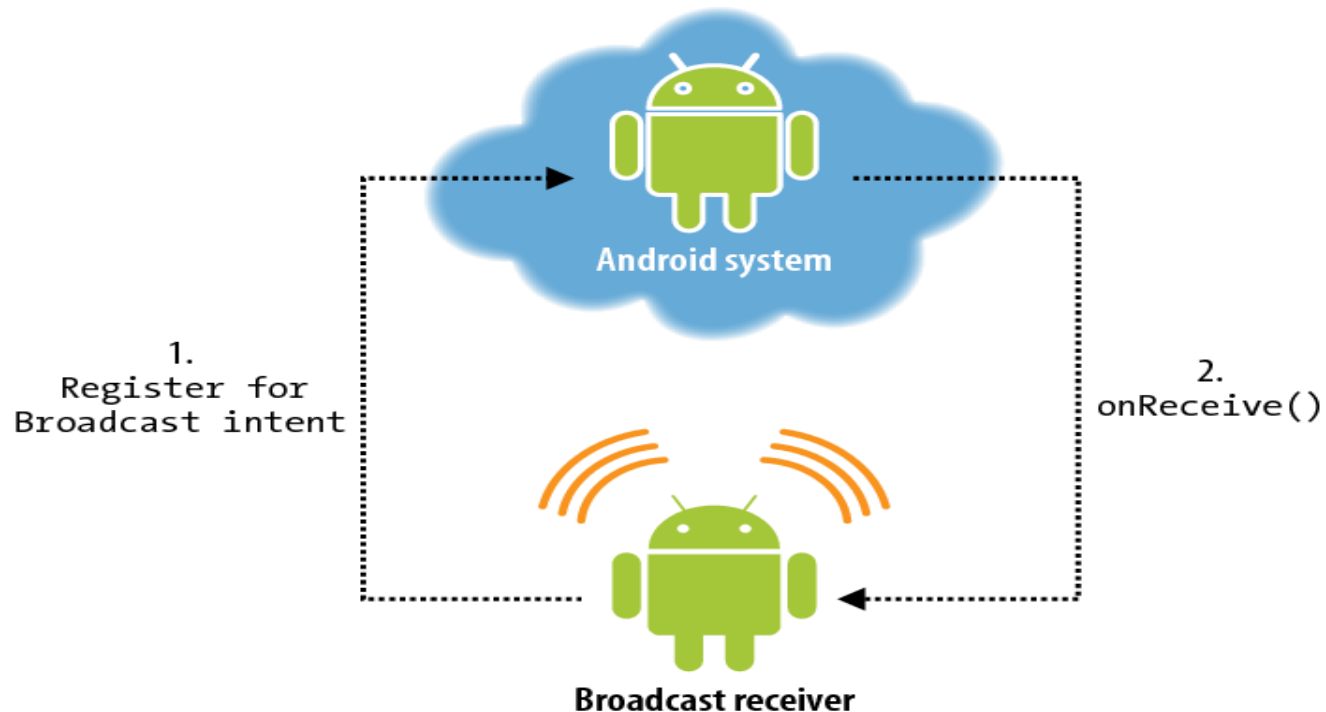
- Un proveedor de contenido manipula un conjunto compartido de datos de una aplicación.
- Permite a nuestra App publicar información para otras Apps, de tal forma que estas pueden acceder al contenido, y si tienen permisos incluso podrían hacer modificaciones.
- Por ejemplo, Android tiene un content provider para manejar los contactos del usuario. La App de gestión de pedidos, podría actualizar la información de los clientes o de los proveedores directamente en la agenda de contactos.
- Para poder crear un proveedor de contenido debemos crear una subclase de la clase **ContentProvider**.

# PROVEEDORES DE CONTENIDOS



# RECEPTORES DE BROADCAST

- Un *Broadcast Receiver* es un programa que recibe un mensaje de multidifusión de alguna App.
- Por ejemplo, una cámara que acaba de capturar una foto, una App que publica que ha cambiado el precio de algún producto...

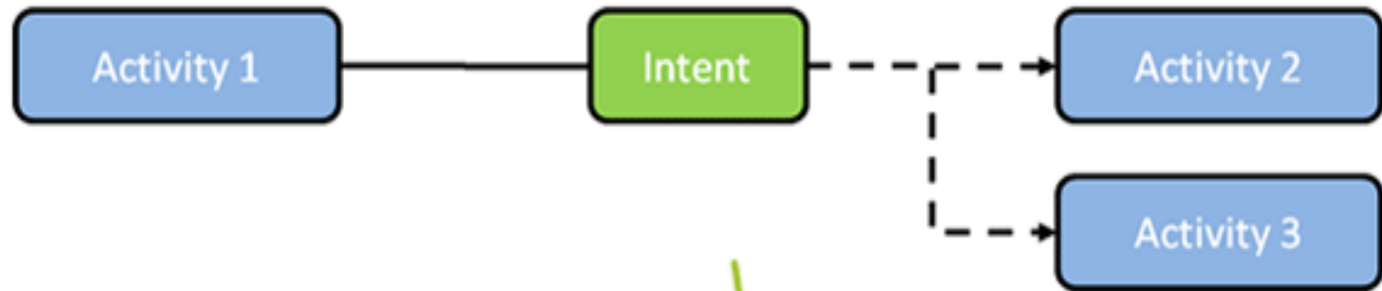




# COMUNICACIÓN ENTRE COMPONENTES

- Cuando un componente intenta hacer una llamada a otro (una actividad invoca a otra actividad), se crea un mensaje del componente origen al componente destino, declarando la intención de solicitar un servicio.
- Esta intención (**intent**), es la forma asíncrona que tienen los componentes de comunicarse en tiempo de ejecución.
- El intent activa el componente invocado, enlazando los componentes:
  - Para actividades y servicios el intent define la acción a realizar.
  - Para receptores de broadcast el intent define el anuncio a realizar.
  - El componente content provider no es activado por intents.

# COMUNICACIÓN ENTRE COMPONENTES



Intents







# EL FICHERO DE MANIFIESTO

- Antes de ejecutar una App el SO lee el fichero `AndroidManifest.xml` que se encuentra en el directorio raíz del proyecto.
- En este fichero se declaran todos los componentes que forman parte de nuestra App:
  - Los permisos que requiere la App.
  - Mínimo nivel de API de Android requerido por la App.
  - Características hardware y software que necesita la App (bluetooth, cámara...).
  - Librerías que necesitan ser enlazadas a la App.
  - Otras cosas...



# EL FICHERO DE MANIFIESTO

- El atributo `android:name` establece el nombre de la actividad y el atributo `android:label` la etiqueta que se mostrará.
- Hay que declarar de la misma forma todos los componentes que vaya a usar la App:
  - `<activity>` → Componentes de tipo actividad.
  - `<service>` → Componentes de tipo servicio.
  - `<receiver>` → Componentes de tipo receptores de broadcast.
  - `<provider>` → Componentes de tipo proveedor.



# EL FICHERO DE MANIFIESTO

- Aunque es posible iniciar un componente simplemente escribiendo su nombre y utilizando la clase Intent, si lo declaramos en el archivo de manifiesto, el sistema podría seleccionar nuestra aplicación como una posible App que fuera capaz de responder a ese Intent.

# EL FICHERO DE MANIFIESTO



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.o7planning.explicitintentexample" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportRtl="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=".GreetingActivity" >
        </activity>

    </application>

</manifest>
```



# LA PILA DE SOFTWARE DE ANDROID

- La pila de software es un reflejo gráfico de cómo organiza Android sus diferentes partes software.
- La capa de Aplicación muestra las Apps que se ejecutan en el SSOO.
- La capa de Application Framework es la que garantiza que las aplicaciones creadas para Android sigan el esquema que dicta Android.
- La capa de Librerías muestra todos los paquetes de librerías de las que podemos hacer uso para elaborar nuestras Apps.
- La capa del Kernel muestra lo que es el propio núcleo de Linux, con sus controladores hardware incluidos.

# LA PILA DE SOFTWARE DE ANDROID



## Android Framework

### APPLICATIONS

ALARM • BROWSER • CALCULATOR • CALENDAR •  
CAMERA • CLOCK • CONTACTS • DIALER • EMAIL •  
HOME • IM • MEDIA PLAYER • PHOTO ALBUM •  
SMS/MMS • VOICE DIAL

### ANDROID FRAMEWORK

CONTENT PROVIDERS • MANAGERS (ACTIVITY,  
LOCATION, PACKAGE, NOTIFICATION, RESOURCE,  
TELEPHONY, WINDOW) • VIEW SYSTEM

### NATIVE LIBRARIES

AUDIO MANAGER • FREETYPE • LIBC •  
MEDIA FRAMEWORK • OPENGLES • SQLITE  
• SSL • SURFACE MANAGER • WEBKIT

### ANDROID RUNTIME

CORE LIBRARIES •  
DALVIK VM

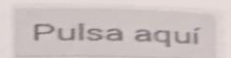






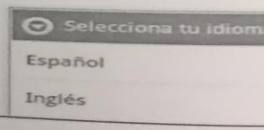
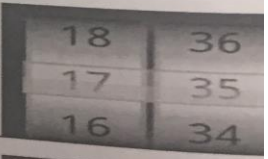

### HAL

AUDIO • BLUETOOTH • CAMERA • DRM •  
EXTERNAL STORAGE • GRAPHICS • INPUT •  
MEDIA • SENSORS • TV

### LINUX KERNEL

DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH,  
CAMERA, DISPLAY, KEYPAD, SHARED MEMORY,  
USB, WIFI) • POWER MANAGEMENT

# CONTROLES DE ENTRADA

44 Programación en Android		Clases	Imagen
Tipo de Control	Descripción		
Botón	Un botón que puede ser “apretado” por el usuario para realizar una acción	Button	
Texto	El texto puede ser no editable (etiqueta o TextView*), o editable (Campo de Texto o EditText). Este último, se puede filtrar tipo y longitud del valor que introduce el usuario	TextView, EditText, AutoCompleteTextView	
Casilla de verificación	Un campo que puede ser encendido o apagado por el usuario. Se presentan al usuario como opciones no mutuamente excluyentes entre sí	CheckBox	
Botón de Opción	Igual que los checkboxes, pero redondos y representan opciones mutuamente excluyentes	RadioButton, RadioGroup	 
Switches	Representan una opción booleana, con valores de encendido/apagado	ToggleButton	 
Spinners	En otros entornos llamados “Combo boxes” o listas desplegables. Permiten al usuario seleccionar un valor de entre varios	Spinner	
Selectores	Diálogos de usuario que permiten al usuario seleccionar un valor (generalmente fecha/hora) mediante flechas o con gestos con la mano	DatePicker, TimePicker	
Imágenes	Se pueden utilizar imágenes para producir acciones al igual que con los botones	ImageView, ImageButton	

\*TextView se puede configurar para ser editable.



# CONTROLES O WIDGETS

- Layouts: Son agrupaciones de controles para organizarlos en algún tipo de formación.
- Tabuladores o Tabs: Son contenedores que organizan la información por diversos criterios.
- Menús y Submenús: Permiten seleccionar al usuario varias opciones para producir diferentes acciones.
- Diálogos de alertas: Permite al usuario responder a una pregunta espontánea de la aplicación.
- Barras de Desplazamiento: Permiten desplazar una parte de la IU para examinar una parte que no estaba visible.
- En el cuadro “Palette” de nuestro proyecto podemos encontrar los distintos controles que podemos utilizar en nuestras App.



# API's DE ANDROID

- Guía de la API de Android:
  - [Http://developer.android.com/guide/index.html](http://developer.android.com/guide/index.html)
- Tutoriales de entrenamiento:
  - [Http://developer.android.com/training/index.html](http://developer.android.com/training/index.html)
- Referencias de Paquetes:
  - [Http://developer.android.com/reference/packages.html](http://developer.android.com/reference/packages.html)
- Para ver la información de los widgets:
  - <http://developer.android.com/reference/android/widget/package-summary.html>

# TEXT VIEW

- Es el widget más simple, la etiqueta.
- En Java se crea una etiqueta con una instancia de la clase `TextView`, aunque lo más normal es crearlas a través de los ficheros XML para la disposición de los controles de la actividad.
- Algunas propiedades:
  - `android:text` → establece el texto de la etiqueta.
  - `android:typeface` → permite cambiar el tipo de fuente para la etiqueta.
  - `android:textStyle` → permite seleccionar el estilo de la letra (cursiva, negrita, o cursiva y negrita).
  - `android:textSize` → permite especificar el tamaño de la fuente.

# TEXT VIEW

- Más información sobre las propiedades del Text View:

<http://developer.android.com/reference/android/widget/TextView.html>

- Se puede usar cualquier tipo de fuente que no sean las que vienen por defecto con Android (sans, monospace y serif), pero hay que cargar el fichero .tff de la fuente (arial.tff...) en el directorio de Assets (herramientas) y cargarlo desde código:

```
TextView txtHelloWorld = (TextView) findViewById(R.id.txtHelloworld);
```

```
Typeface type = Typeface.createFromAsset(getAssets(), "arial.tff");
```

```
txtHelloWorld.setTypeface(type);
```

# BUTTON

- Es una subclase de la clase Text View.
- Para responder a un evento de tipo Click, hay que implementar un Listener (View.OnClickListener).
- A partir de la versión 1.6 de Android también es posible declarar la respuesta a el evento Click en el fichero XML de la actividad, mediante la propiedad *android:onClick*. Se puede indicar el nombre de cualquier método público que reciba un parámetro de tipo View y que retorne void.

```
public void MeHicieronClick(View v){  
  
    System.out.println("Debug: Me hicieron Click!");  
}
```

- En el fichero XML:

```
<Button
```

```
.....
```

```
    android:onClick= "MeHicieronClick"
```

```
..... />
```

# BUTTON

- Otra posibilidad para responder al evento de Click, es codificar clases Anónimas (anonymous – inner classes).
- En el propio código de establecimiento del Listener OnClickListener del botón, creamos una clase sin nombre como argumento del método `setOnClickListener` de la siguiente forma:

```
Button btnGigante = (Button) findViewById(R.id.button);
```

```
BtnGigante.setOnClickListener(
```

```
new View.OnClickListener(){
```

```
public void onClick(View v){
```

```
System.out.println("Me hicieron click!");}
```

```
});
```

- En la definición del listener es donde a través del operador `new` se define un objeto de una clase anónima derivada de la clase `View.OnClickListener` en tiempo de ejecución. Esa clase sólo tiene un método llamado `onClick` que es el que responde al evento.

# GESTIÓN DE EVENTOS

- Para refrescar la gestión de eventos en java tenéis los siguientes enlaces:
- Cómo gestionar eventos:
  - [Http://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html](http://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html)
- Cómo programar escuchadores de eventos:
  - [Http://docs.oracle.com/javase/tutorial/uiswing/events/](http://docs.oracle.com/javase/tutorial/uiswing/events/)

**USUARIO:**  
Quién provoca el evento



**BOTÓN:**  
Objeto sobre el que se produce el evento



**EVENTO:**  
Al presionar el botón..... RINGGG!!!!!!



# ACTIVIDAD I

- Un botón es totalmente personalizable:
  - Se puede deshabilitar el clic durante un periodo de tiempo.
  - Se puede personalizar la forma del botón para que no rompa el estilo de la interfaz de la App.

[Http://developer.android.com/reference/android/widget/Button.html](http://developer.android.com/reference/android/widget/Button.html)

- Probar la herramienta online <http://angrytools.com/android/button/> para personalizar de forma sencilla los botones de vuestra App. Podréis observar cómo cambian las propiedades de los botones con los diferentes posibles parámetros.

# RADIOGROUP Y RADIOBUTTON

- Permiten al usuario la selección de un conjunto de opciones excluyentes entre sí.
- Para ello los botones de tipo RadioButton se agrupan en un RadioGroup.
- Los RadioGroup son contenedores (containers) mientras que los RadioButton son widgets.
- Algunas de las propiedades que podemos establecer desde nuestro código Java son:
  - RadioGroup:
    - void check(int id) → Activa el RadioButton con identificador id.
    - Void clearCheck() → Quita la selección dejando todos los RadioButton desactivados.
    - int getCheckedRadioButtonId() → Devuelve el Id del RadioButton seleccionado.
  - RadioButton:
    - void toggle() → Cambia el estado de activación del RadioButton.



# RADIOGROUP Y RADIOBUTTON

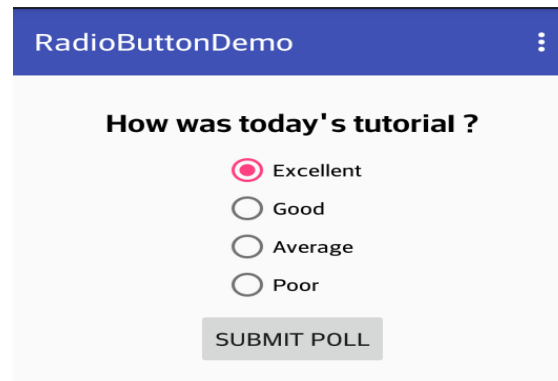
- Eventos:
  - El Click en un RadioGroup y en un RadioButton se puede capturar de la misma forma que en un Button.
  - Se pueden capturar los eventos de cambio de selección en un RadioGroup mediante el siguiente método:

*`void setOnCheckedChangeListener(RadioGroup.OnCheckedChangeListener listener)`*

- *Se hace igual que con el botón desde código, pero no a través de propiedades en el archivo XML de la actividad.*

<http://developer.android.com/reference/android/widget/RadioButton.html>

<http://developer.android.com/reference/android/widget/RadioGroup.html>



RadioButtonDemo

**How was today's tutorial ?**

☒ Excellent

☐ Good

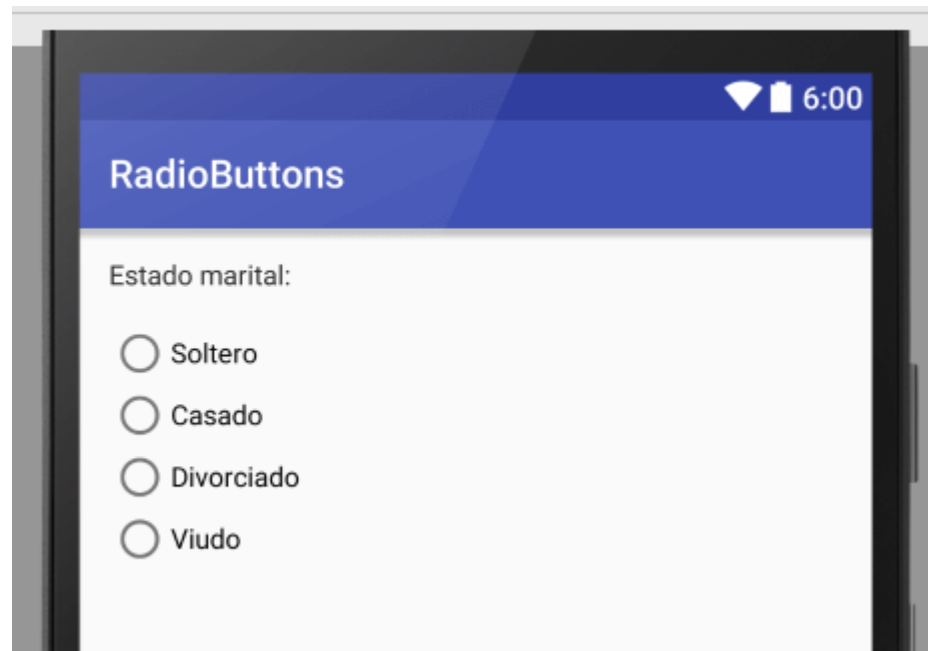
☐ Average

☐ Poor

SUBMIT POLL

# ACTIVIDAD 2

- Crear una App que dé al usuario 4 opciones para elegir (su equipo de fútbol, su luchador de wwf, o su postre favorito...).
- La aplicación responderá con un mensaje distinto para cada elección del usuario.

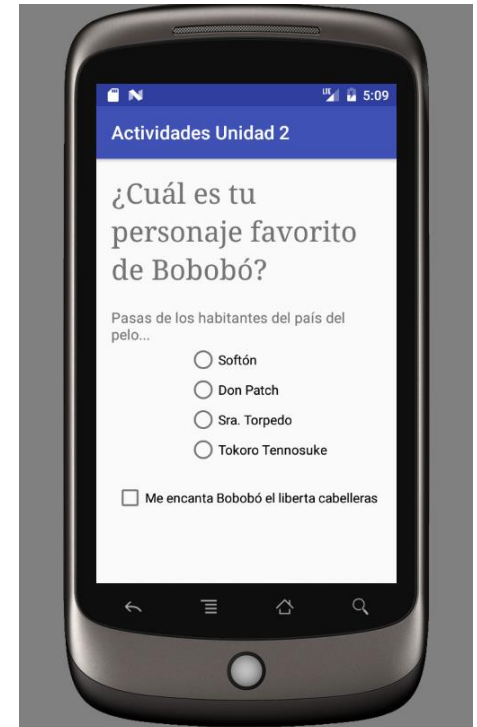
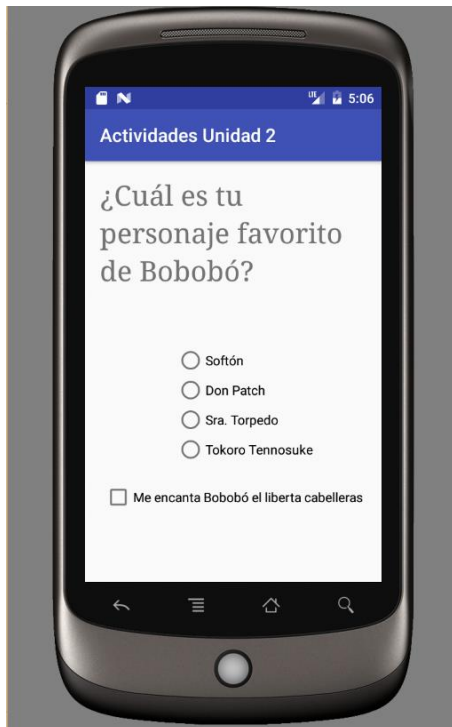


# CHECKBOX

- Se utilizan normalmente para marcar una casilla de verificación.
- Tiene dos estados:
  - Checked (marcado) → `setChecked(true);`
  - Unchecked (desmarcado) → `setChecked(false);`
- Para consultar si un checkbox está marcado tenemos el método `isChecked()` → devuelve `true` si está marcado y `false` si no lo está.
- Para cambiar el estado del checkbox también podemos utilizar el método `toggle()`.
- `CheckBox` también hereda de `TextView`.
- El evento que controla el cambio de estado se captura de igual manera que los vistos hasta ahora:
  - *`onCheckedChanged`* → implementamos la interfaz *`CheckBox.OnCheckedChangeListener`*.

# ACTIVIDAD 3

- Añadir una casilla a la aplicación de la actividad 2 para que el usuario pueda indicar si le gusta o no el tema para el que se le da las opciones.
- La aplicación responderá con un mensaje distinto si la casilla está marcada o no.



# TOGGLEBUTTON Y SWITCH

- Son widgets con dos estados → encendido y apagado (on y off).
- Se diferencian del CheckBox en su aspecto gráfico.
- El ToogleButton **no** tiene un texto asociado, el Switch **sí** lo tiene, pero en ambos casos se puede cambiar el título de los dos estados mediante las propiedades *android:textOn* y *android:textOff*.
- Para mantener activado o desactivado cualquiera de los dos componentes se utiliza la propiedad *Checked*.
- Se puede capturar el evento *OnCheckedChangeListener* de la misma forma que hasta ahora.



Switch



ToggleButton

# IMAGEVIEW E IMAGEBUTTON

- Estas clases son análogas a TextView y Button pero en lugar de mostrar texto muestran una imagen.
- Tienen el atributo android:src en el fichero XML de la actividad para especificar la imagen a utilizar mediante una referencia a un **recurso dibujable** (@drawable).
- Antes de añadir un widget de estas clases tenemos que añadir a la carpeta “drawable” de nuestro proyecto los ficheros de las imágenes que queremos utilizar.
- Android recomienda usar ficheros png pero también admite bmp y jpg.
- Cada imagen debe tener un identificador para poder ser referenciada y este identificador será un número entero accesible a través de la clase R → **R.drawable.identificador**.
- Hay varias carpetas drawable con los diferentes tipos de pantalla para los que nuestra App puede funcionar.

[http://developer.Android.com/guide/practices/screens\\_support.html](http://developer.Android.com/guide/practices/screens_support.html)

# IMAGEVIEW E IMAGEBUTTON



ImageButtonTesting



# ACTIVIDAD 4

- Crear una App que muestre un botón con imagen y una imagen simple.
- Al pulsar el botón se cambiará la imagen tanto del ImageView como del ImageButton.
- Podéis utilizar la siguiente utilidad online (Android Asset Studio) para generar imágenes para todas las densidades de pantalla:

<http://romannurik.github.io/AndroidAssetStudio/nine-patches.html>



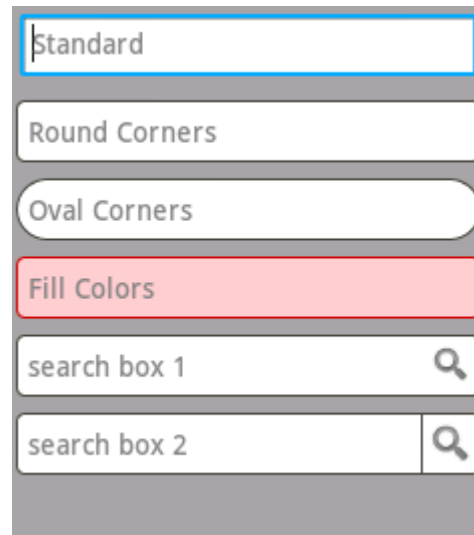
# EDITTEXT

- Es una subclase de TextView.
- Propiedades interesantes:
  - android:autotext → controla si el usuario recibirá asistencia automática al escribir el texto.
  - android:capitalize → controla si el propio campo de texto debe escribir la primera letra de cada palabra en mayúscula.
  - android:digits → configura el campo de texto para que admita solo determinados dígitos.
  - android:singleline → especifica si el campo admite solo una línea y al pulsar enter se va al siguiente control o se crea una línea nueva al pulsar intro (multilínea).
  - android:inputType → permite seleccionar entre varios tipos de teclado que se mostrarán cuando se esté introduciendo texto:
    - “text”, “textEmailAddress”, “textUri”, “number”, “phone”...

<http://developer.Android.com/guide/topics/ui/controls/text.html>

# EDITTEXT

- La propiedad ***android:inputType*** también permite controlar diversos comportamientos como:
  - Que al terminar de escribir una palabra se autocorrija con el diccionario.
  - Que los caracteres aparezcan convertidos en puntos (para passwords...).



# EDITTEXT

## Acciones de Finalización

- Se puede añadir un botón al teclado que aparece cuando se está editando un texto para que el usuario indique que ha finalizado la edición y desencadenar una acción.
- “*actionSend*” (enviar) o “*actionSearch*” (buscar).
- Si no se especifica esta propiedad se establece por defecto “*actionNone*”.
- Se utiliza la propiedad ***android:imeOptions*** en el fichero xml:

*android:imeOptions=“actionSend”*

- Para responder a los eventos se utiliza un evento `EditorAction` mediante el método `setOnEditorActionListener()`.

# EDITTEXT

## Acciones de Finalización

```
EditText editText = (EditText) findViewById(R.id.editText);

editText.setOnEditorActionListener(new

TextView.OnEditorActionListener(){

Public boolean onEditorAction(TextView v, int actionId, KeyEvent event){

    Boolean manejada = false;

    If(actionId == EditorInfo.IME_ACTION_SEND){

        System.out.println("acción enviada");

        manejada = true;

    }//Fin del if

    return manejada;

} //Fin de onEditorAction()

}); //Fin de OnEditorActionListener()
```

# EDITTEXT

## Acciones de Finalización

- El método *onEditorAction()* recibe como parámetros el campo de texto `TextView t`, el identificador de la acción (se puede consultar con la colección de enteros `EditorInfo`), y el evento de tecla `KeyEvent` que en estos casos será *null*.
- Hay que devolver si hemos manejado o no el tipo de acción que se nos ha comunicado, en este caso retornaremos cierto si se invocó el método para la acción de terminar (`IME_ACTION_SEND`).



# EDITTEXT

## Capturar Cambios en el Texto

- Se puede controlar cualquier interacción del usuario con un campo de texto.
- Se puede programar un método que responda al cambio del texto en una determinada posición o realizar algo justo antes de que el usuario se ponga a cambiar el texto, o incluso después de cambiar el texto...
- Para ello necesitamos un objeto que implemente la interfaz TextWatcher y registrarla en el campo de texto mediante el método:

*addTextChangedListener(TextWatcher watcher);*

# EDITTEXT

## Capturar Cambios en el Texto

- La interfaz `TextWatcher` tiene tres métodos:

*`Public void beforeTextChanged(CharSequence s, int start, int count, int after)`*

- Avisa de que en la cadena *s*, los *count* caracteres empezando en *start*, están a punto de ser reemplazados por nuevo texto con longitud *after*.

*`Public void onTextChanged(CharSequence s, int start, int before, int count)`*

- Avisa de que en la cadena *s*, se han reemplazado *count* caracteres comenzando en *start* y han reemplazado *before* número de caracteres.

*`Public void afterTextChanged(Editable s)`*

- Se ejecuta después de haberse cambiado el texto y te avisa de que en algún punto de la cadena *s*, el texto ha cambiado.

# ACTIVIDAD 5

- Crear una App que informe al usuario de cuándo se están cambiando los caracteres del campo de texto ejecutando los distintos métodos vistos hasta ahora.

```
editText.addTextChangedListener(new TextWatcher(){  
  
    public void onTextChanged(CharSequence s, int start, int before, int count){  
  
        System.out.println("Se han reemplazado de '"+s+"'"+count+" caracteres a partir de "+start+"  
que antes ocupaban "+before+" posiciones");  
  
    }// Fin de onTextChanged()  
  
    public void beforeTextChanged(CharSequence s, int start, int count, int after){  
  
        System.out.println("Se van a reemplazar de '"+s+"'"+count+" caracteres a partir de "+start+"  
por texto con longitud "+after);  
  
    }// Fin de beforeTextChanged()  
  
    public void afterTextChanged(Editable s){  
  
        System.out.println("Tu texto tiene "+s.length()+" caracteres");  
  
    }// Fin de afterTextChanged()  
  
});//Fin de TextWatcher
```



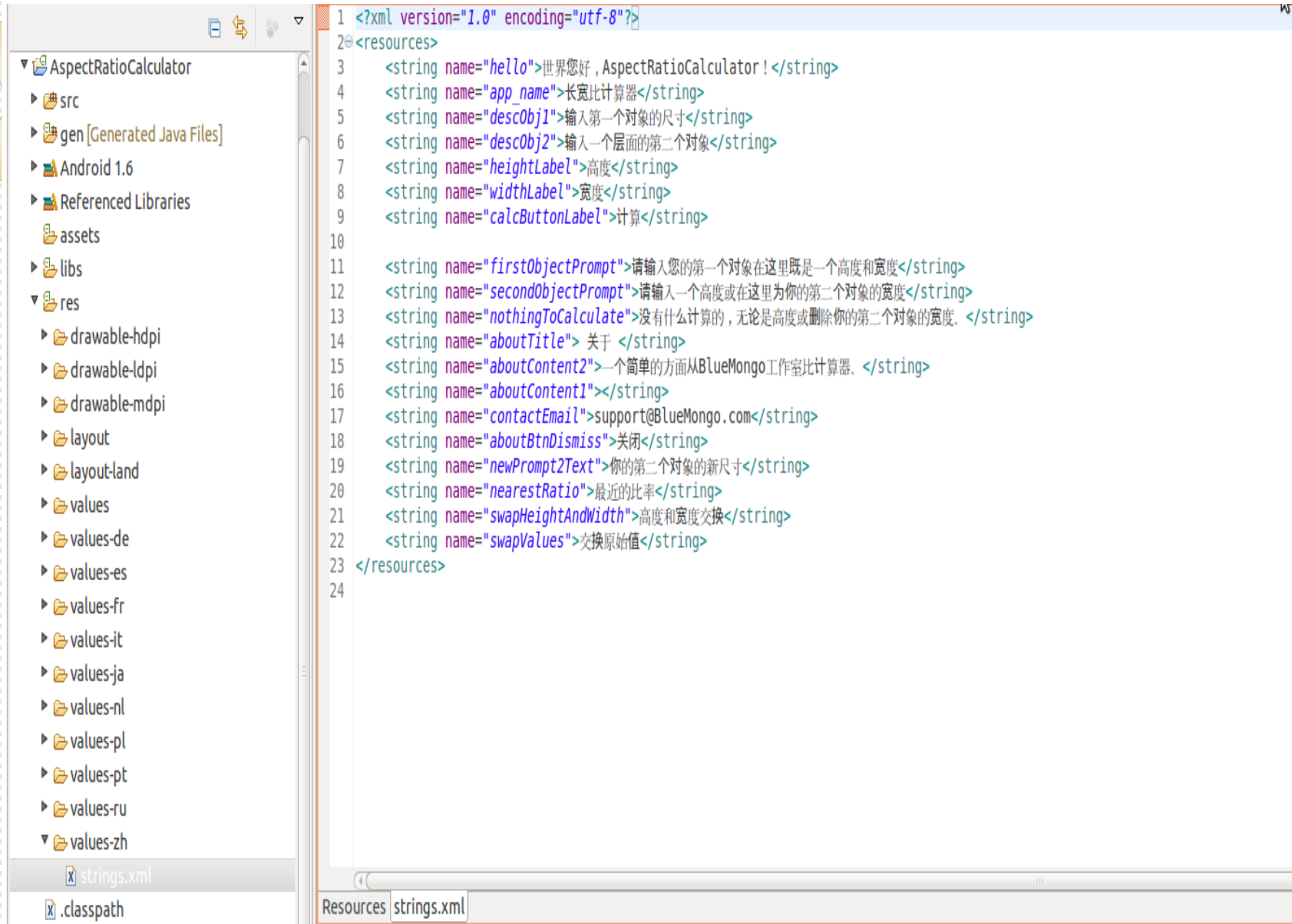
# INTERNACIOLIZACIÓN DE APPS

- Los *resources* son recursos estáticos (datos, imágenes, sonidos, animaciones, layouts...) que nuestra App puede utilizar.
- Una App puede incluir diversos conjuntos de recursos que puede utilizar para funcionar con diversas configuraciones.
- En primer lugar debemos acostumbrarnos a utilizar recursos para almacenar todas las cadenas de caracteres con texto y el resto de recursos de nuestras Apps (ficheros gráficos, animaciones, layouts...) fuera del código fuente de la App.
- El archivo “*res/values/strings.xml*” debe contener las cadenas de caracteres en el idioma que consideremos de uso mayoritario ya que es el recurso por defecto para los strings de nuestra App.
- Desde el código, podemos referenciar estos valores desde el diseño de una actividad mediante `@string/recurso`, o desde código invocando a la función `getString(IDRecurso)`.

*String s = getString(R.string.recurso);*

<http://developer.Android.com/training/basics/supporting-devices/index.html>

# INTERNACIOLIZACIÓN DE APPS





# APLICACIONES MULTIIDIOMA

- Además del recurso por defecto podemos generar **recursos alternativos** (alternative resources).
- Para ello podemos especificar múltiples directorios “/res/<cualificador>”, donde un cualificador especifica un lenguaje o una combinación de lenguaje-región.
- Creamos un conjunto de recursos por defecto y diferentes alternativas para ejecutarse con distintas localizaciones, y cuando un usuario ejecuta la App Android selecciona qué recursos cargar dependiendo de su configuración.
- Si el lenguaje principal de nuestra App es el español podríamos tener los siguientes recursos:
  - res/values → recurso por defecto con los archivos con las cadenas de caracteres en español.
  - res/values-fr → recurso alternativo con las cadenas traducidas a francés.
  - res/values-en → recurso alternativo con las cadenas traducidas a inglés.

# APLICACIONES MULTIIDIOMA

- Se puede cambiar la configuración del emulador mediante la pestaña de aplicaciones.
- También se puede hacer desde la Shell del sistema operativo:
  - Vamos a la carpeta “sdk/platform-tools” dentro del directorio donde hemos instalado Android y ejecutamos el comando *adb -e Shell*.
  - Para cambiar el lenguaje y el país escribimos en la consola:

*setprop persist.sys.language <lenguaje>;*

*setprop persist.sys.country <país>;*

*stop; sleep 5; start;*

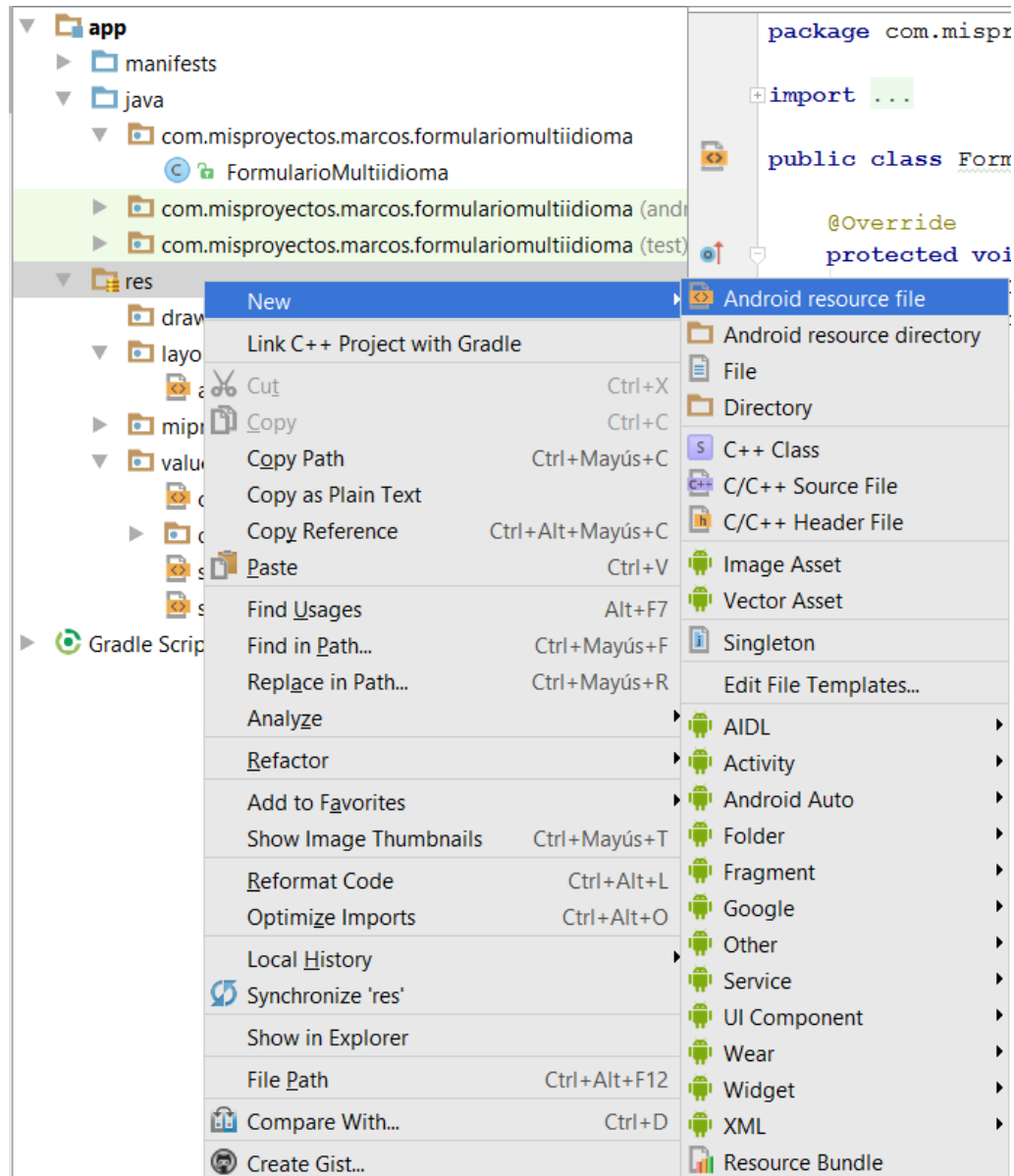
# ACTIVIDAD 6

- Crear una App con una actividad con capacidad de ejecución con idioma español y con idiomas alternativos en inglés y alemán.
- Consistirá en un simple formulario para rellenar datos personales para el registro de un usuario.
- El formulario en los tres idiomas tendrá esta pinta:

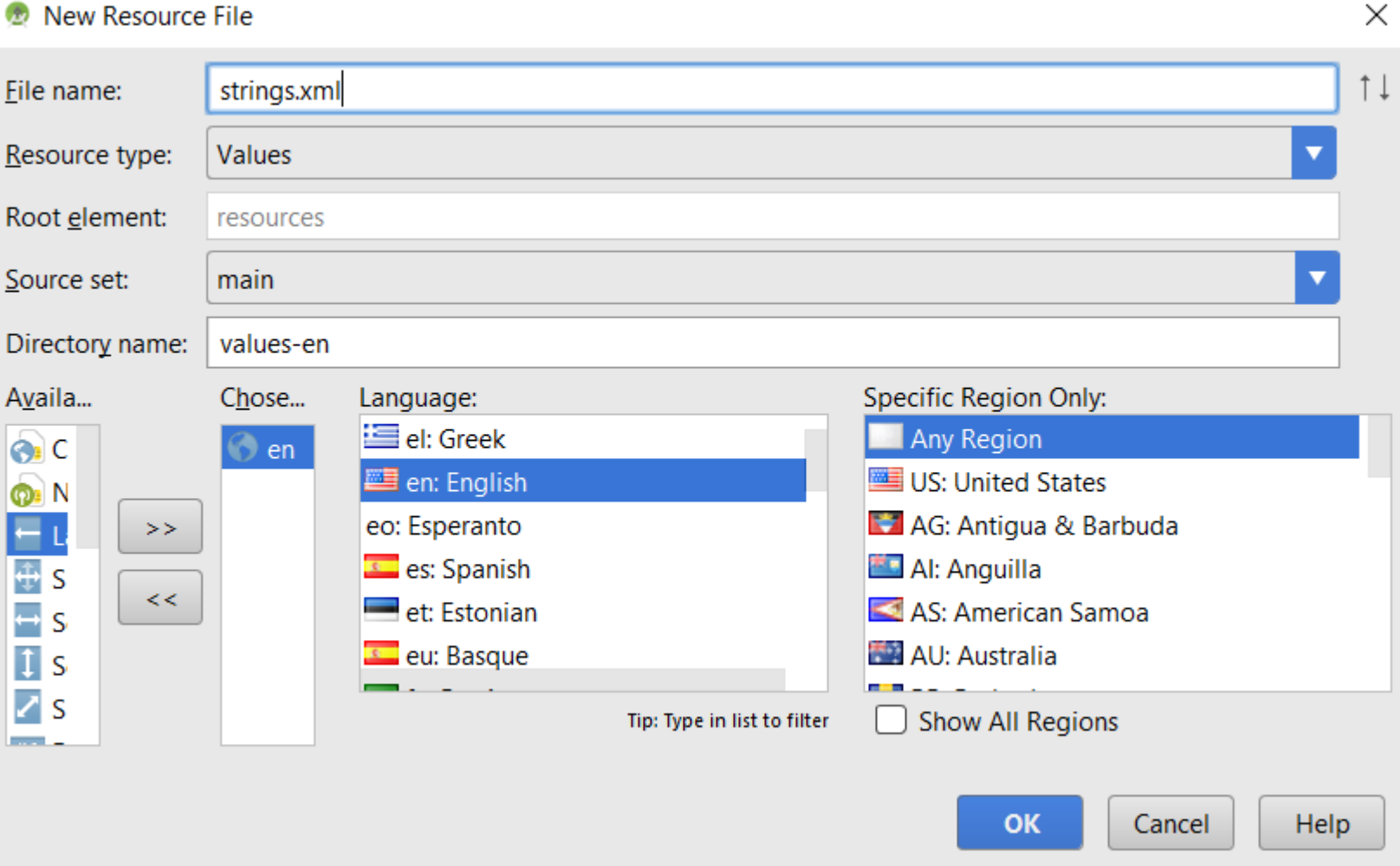
The image displays three side-by-side screenshots of a mobile application's registration form, demonstrating its internationalization capabilities. Each screen shows a different language: Spanish (left), German (middle), and English (right). The app's title bar at the top of each screen reads 'Internacionalización' in Spanish, 'Internacionalization' in German, and 'Internationalization' in English. The registration form includes fields for Name, Surname, Email, and Password, each with a corresponding label in the respective language. The Spanish version uses 'Nombre', 'Apellidos', 'Email', and 'Clave'; the German version uses 'Vorname', 'Nachname', 'Email', and 'Passwort'; and the English version uses 'Name', 'Surname', 'Email', and 'Password'. The welcome message at the top of each form is also localized: 'Bienvenido al formulario de registro!' in Spanish, 'Willkommen zu anmeldung Formular!' in German, and 'Welcome to sign up form' in English. The time '13:03' is visible in the status bar of the German screenshot.

Spanish (Internacionalización)	German (Internacionalization)	English (Internationalization)
Bienvenido al formulario de registro!	Willkommen zu anmeldung Formular!	Welcome to sign up form
Nombre: _____	Vorname: _____	Name: _____
Apellidos: _____	Nachname: _____	Surname: _____
Email: _____	Email: _____	Email: _____
Clave: _____	Passwort: _____	Password: _____

# ACTIVIDAD 6



# ACTIVIDAD 6

A screenshot of the 'New Resource File' dialog box in an IDE. The dialog has a title bar with a close button. It contains several input fields and dropdown menus for configuring a new resource file. The 'File name' field is 'strings.xml'. The 'Resource type' dropdown is 'Values'. The 'Root element' field is 'resources'. The 'Source set' dropdown is 'main'. The 'Directory name' field is 'values-en'. Below these are three sections: 'Available Languages' with a list of language icons and a '>>' button; 'Chose...' with a list containing 'en' and a '<<' button; and 'Language:' with a list of languages where 'en: English' is selected. To the right of the language list is a 'Specific Region Only:' section with a list of regions where 'Any Region' is selected, and a 'Show All Regions' checkbox. A tip 'Tip: Type in list to filter' is at the bottom of the language list. At the bottom right are 'OK', 'Cancel', and 'Help' buttons.

New Resource File

File name: strings.xml

Resource type: Values

Root element: resources

Source set: main

Directory name: values-en

Available Languages: C, N, L, S, S, S, S

Chose...: en

Language:

- el: Greek
- en: English
- eo: Esperanto
- es: Spanish
- et: Estonian
- eu: Basque

Specific Region Only:

- Any Region
- US: United States
- AG: Antigua & Barbuda
- AI: Anguilla
- AS: American Samoa
- AU: Australia

Tip: Type in list to filter

Show All Regions

OK Cancel Help



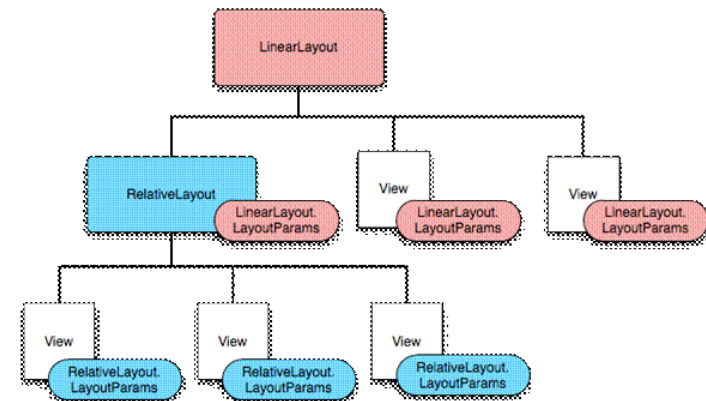
# LAYOUTS Y CONTENEDORES

- Los contenedores permiten organizar un conjunto de widgets en una estructura específica definida a nuestro gusto.
- Siempre que necesitemos utilizar múltiples widgets necesitaremos un contenedor para alojarlos y tener un elemento de referencia o raíz.
- Los layouts se pueden manejar de dos formas:
  - Declarando los ficheros en XML.
  - Instanciando objetos programáticamente en tiempo de ejecución.
- Declararlos a través de la definición en XML nos permite separar la IU de la lógica de nuestro programa. Esto nos da más flexibilidad y nos permite cambiar el diseño de la interfaz sin tener que tocar código ni recompilar.
- Los `LinearLayout`, `RelativeLayout` y `TableLayout` son contenedores que gestionan la distribución de los controles de una determinada manera en una pantalla.
- El contenedor agrupa controles (hijos) y el layout los ordena dentro del contenedor.
- Podemos anidar contenedores unos dentro de otros para hacer más flexible y potente nuestra interfaz de usuario.

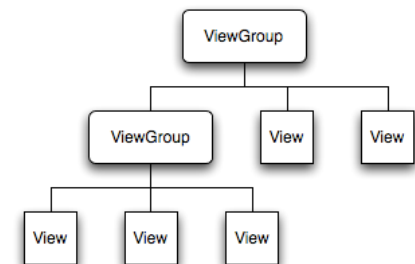


# LAYOUTS - PROPIEDADES

- Los Layout son subclases de ViewGroup que a su vez es una subclase de View.
- Todos los widgets son Views e implementan métodos y propiedades comunes para todos los elementos gráficos de nuestra IU.
- Cada clase ViewGroup implementa una clase anidada que hereda de ViewGroup.LayoutParams, donde se implementan propiedades como la posición y el tamaño de los elementos hijos.
  - Android:layout\_height
  - Android:layout\_width
  - FILL\_PARENT
  - MATCH\_PARENT
  - WRAP\_CONTENT



<http://developer.Android.com/reference/android/view/ViewGroup.LayoutParams.html>



# RELATIVE LAYOUTS

- Este layout permite a los hijos especificar su posición con relación al padre o a uno de sus hermanos.
- Se pueden alinear elementos al borde, poner uno debajo de otro, centrarlo en la pantalla, a la izquierda...
- Esto se puede hacer desde el propio entorno de desarrollo o desde el fichero XML de Layout.
  - `android:layout_alignParentTop="true"`
  - `android:layout_centerVertical="true"`
  - `android:layout_below="@+id/xxxx"`
  - `android:layout_toRightOf="@+id/xxxx"`
  - `android:layout_alignParentLeft="true"`

<http://developer.Android.com/reference/Android/widget/RelativeLayout.LayoutParams.html>

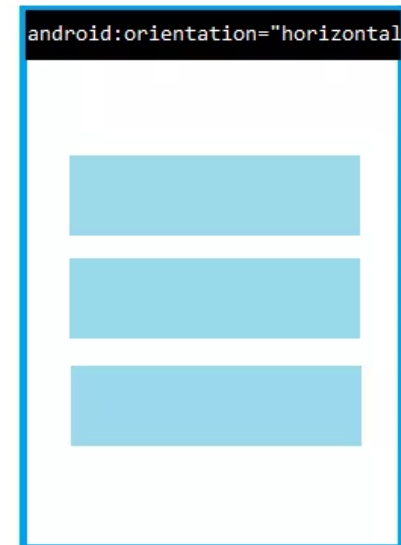
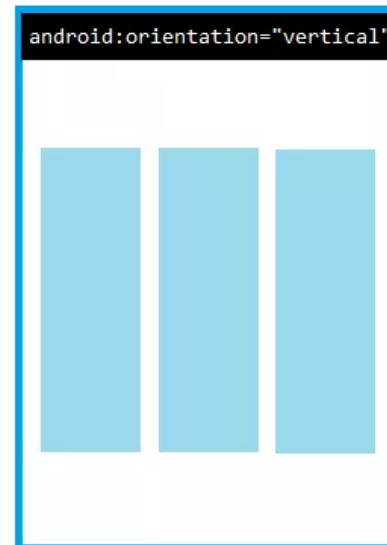
[Demo LinearLayout](#)

# LINEAR LAYOUTS

- Este layout es un modelo de cajas, es decir, alinea los controles horizontalmente en una fila o verticalmente en una columna.
- Para indicar si los controles irán agrupados en fila o en columna se utiliza la propiedad orientation:
  - `android:orientation="vertical"`
  - `android:orientation="horizontal"`
- `Android:layout_weight` → Esta propiedad permite otorgar preferencia a la hora de rellenar el espacio del contenedor en caso de que queramos que todos los controles ocupen el espacio entero del contenedor.

[Demo de LinearLayout](#)

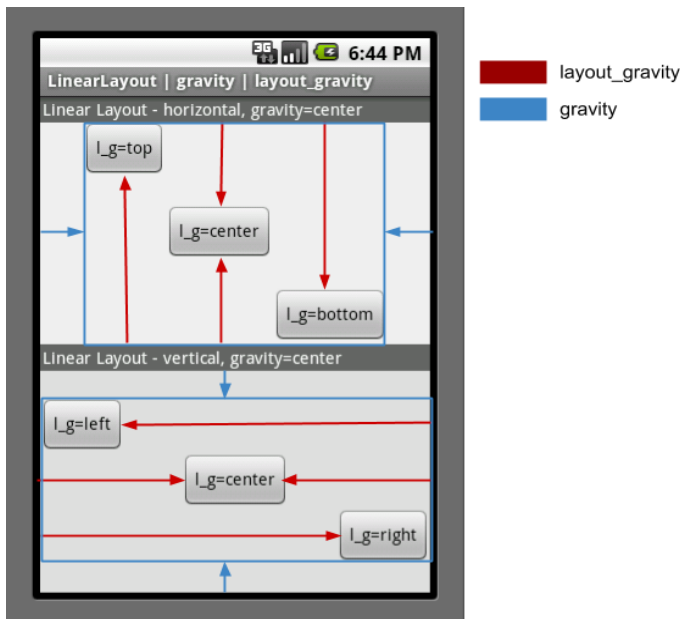
[Demo de layout\\_weight](#)



# LINEAR LAYOUTS

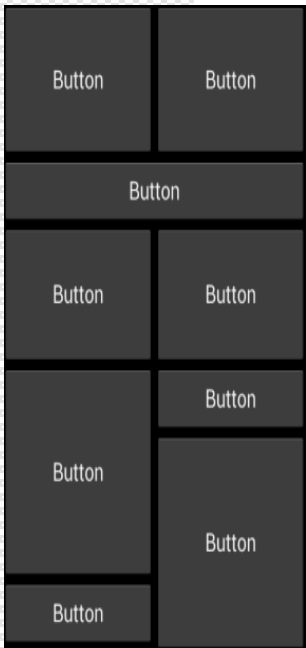
- Otra propiedad importante de un layout linear es la propiedad *layout\_gravity*.
- Por defecto todo se alinea de arriba abajo y de izquierda a derecha. Si queremos hacerlo de otra forma tenemos que utilizar la gravedad.

## [Demo de la propiedad gravedad](#)



# LAYOUTS TABULARES

- Estos layouts crean una distribución basada en filas y columnas, tal y como lo hacemos con una tabla html o en una hoja de cálculo.
- Tenemos dos layouts tabulares similares: el GridLayout y el TableLayout.
- Primero tenemos que especificar el número de filas y columnas que va a tener nuestro Grid:
  - `android:rowCount`
  - `android:columnCount`
- Una vez hecho esto podemos incluir widgets dentro de cada fila y cada columna de forma sencilla.
- Nos basta con especificar en cada widget a qué columna (`android:layout_column`) y a qué fila (`android:layout_row`) va a pertenecer, teniendo en cuenta que los índices para las filas y columnas empiezan desde el 0.
- Si queremos modificar el espacio entre los controles podemos utilizar la propiedad *layout\_margin*.
- Se puede dejar una posición vacía en el Grid si no incluimos ningún control en dicha posición.



# LAYOUTS TABULARES

- Se utiliza el concepto de row spanning y col spanning para hacer que un elemento ocupe más de una celda.
- Para ello utilizamos las propiedades *android:layout\_columnSpan* y *android:layout\_rowSpan* y las activamos asignando el valor “fill” a la propiedad *android:layout\_gravity* del componente.

*<Button*

*android:layout\_width=“wrap\_content”*

*android:layout\_height=“wrap\_content”*

*android:text=“Nuevo Botón”*

*android:id=“@+id/boton2”*

*android:layout\_columnSpan=“2”*

*android:layout\_gravity=“fill”*

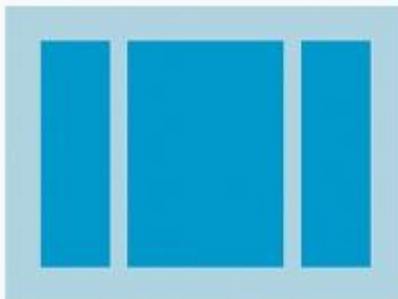
*android:layout\_row=“1”*

*android:layout\_column=“0” />*

# PROGRAMANDO CONTENEDORES

- También se puede trabajar con Layouts desde el código fuente, y necesitaremos hacerlo cuando queramos:
  - Recorrer todos los hijos del Layout.
  - Añadir o eliminar elementos hijos de un Layout en tiempo de ejecución.
  - Responder a eventos de los widgets hijos de un Layout.

Linear Layout



Relative Layout



Web View



# RECORRIDO DEL CONTENEDOR

- Recorrer significa obtener una referencia a cada uno de los widgets de un Layout.
- Esto se puede hacer fácilmente con un bucle for.
- `getChildAt(int index)` → devuelve una referencia al widget con identificador `index` dentro del contenedor.
- `getChildCount()` → devuelve un entero que representa el número de elementos que hay en el contenedor.
- Ejemplo de recorrido de los elementos de un Layout:

```
public void Recorrer(){
```

```
View v;
```

```
GridLayout g = (GridLayout) findViewById(R.id.grid1);
```

```
for (int i = 0; i < g.getChildCount(); i++){
```

```
    v = g.getChildAt(i);
```

```
    System.out.println("objeto: "+ v.toString());
```

```
    }
```

```
}
```



# DIFERENCIANDO TIPOS

- Para conocer el tipo de widget de cada control contenido en un layout se puede hacer de la siguiente forma:
  - Utilizando el método `getClass` de la clase `View` e invocando a su método `getSimpleName` para conocer el nombre de la clase a la que pertenece el objeto `View`.
  - Para tratar sus propiedades específicas hacemos un cast a un objeto de la clase en cuestión.
- Ejemplo:

*Button b;*

*If (v.getClass().getSimpleName().equals("Button")){*

*b = (Button) v;*

*b.setOnClickListener(...);*

*//o cualquier otro método/propiedad de la clase Button*

*}*

# AÑADIR ELEMENTOS

- Se pueden añadir los elementos que queramos a un contenedor, teniendo en cuenta lo siguiente:
  - Cada widget que incluyamos debe tener definido sus parámetros de Layout → los widgets tienen un método llamado `setLayoutParams` para establecer las propiedades en tiempo de ejecución.
  - Cada widget debe tener un identificador → en el XML se indica la propiedad `id` mediante un recurso y en código fuente se hace con un entero. Para evitar conflictos existe el método `generateViewId` que devuelve un id único no utilizado hasta el momento.
- La clase `GroupView` incorpora el método `addView(View v, int index)` para incorporar todos los elementos que deseemos.



# AÑADIR ELEMENTOS

```
public void añadeHijos(){  
    GridLayout g = (GridLayout) findViewById(R.id.grid1);  
    Button b;  
    for (int i=0; i<18; i++){  
        b= new Button(this);  
        b.setLayoutParams(new ViewGroup.LayoutParams(  
            ViewGroup.LayoutParams.WRAP_CONTENT,  
            ViewGroup.LayoutParams.WRAP_CONTENT));  
        b.setText("btn"+ i);  
        b.setId(View.generateViewId());  
        g.addView(b,i);  
    }  
}
```

# AÑADIR ELEMENTOS

- Para responder a eventos generales de los widgets hijos necesitamos un objeto Listener que escuche por todos (o por algunos de ellos).

```
public class MyActivity extends Activity implements View.OnClickListener{
```

```
...
```

```
protected void onCreate(Bundle savedInstanceState){
```

```
    super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_my);
```

```
    añadeHijos();
```

```
}
```

# AÑADIR ELEMENTOS

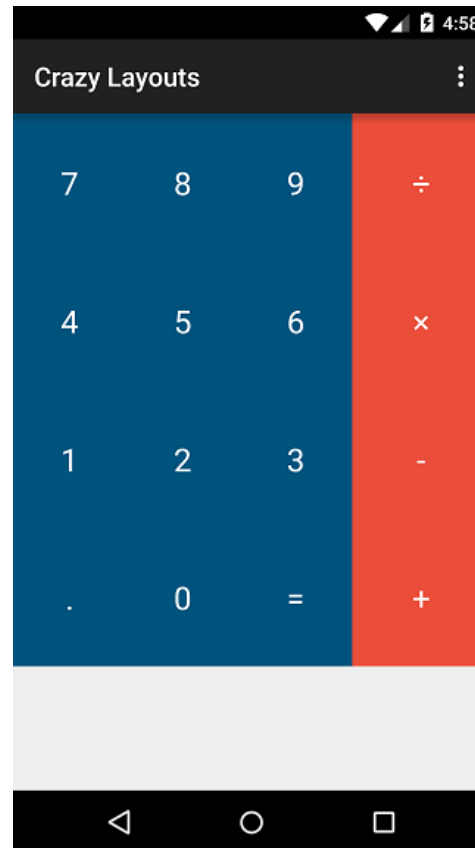
```
public void añadeHijos(){  
    GridLayout g = (GridLayout) findViewById(R.id.grid1);  
    Button b;  
    for (int i=0; i<18; i++){  
        b= new Button(this);  
        b.setLayoutParams(new ViewGroup.LayoutParams(  
            ViewGroup.LayoutParams.WRAP_CONTENT,  
            ViewGroup.LayoutParams.WRAP_CONTENT));  
        b.setText("btn"+ i);  
        b.setId(View.generateViewId());  
        b.setOnClickListener(this);  
        g.addView(b,i);  
    }  
}
```

# AÑADIR ELEMENTOS

```
public void onClick(View v){  
    if(v.getClass().getSimpleName().equals("Button")){  
        Button b = (Button) v;  
        accion(b);  
    }  
}  
  
public void acción(Button b){  
    //programamos aquí la acción con el botón b  
}  
}
```

# AÑADIR ELEMENTOS

- En cada iteración del *for* en *añadeHijos* se establece el listener a la propia actividad para que responda mediante el método *onClick*.
- Cuando se presiona uno de los botones del contenedor *GridLayout* se invoca al método *accion* pasando como parámetro el botón pulsado.



# ACTIVIDAD 7

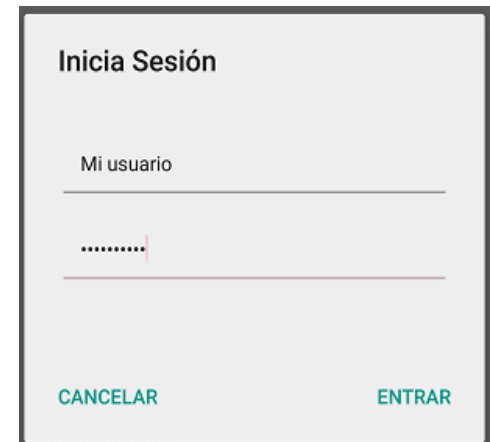
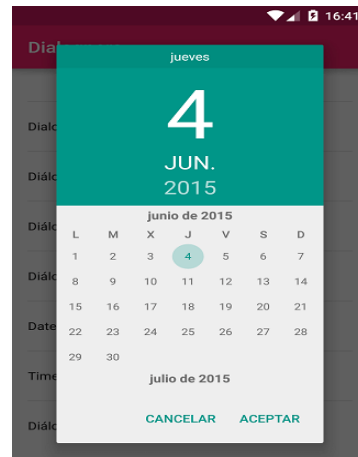
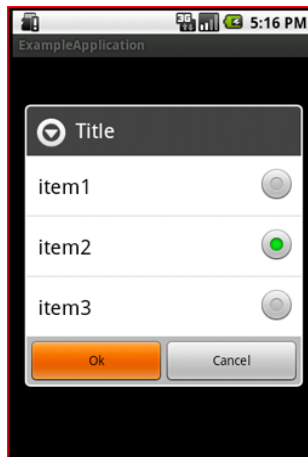
- Realiza una aplicación que programáticamente maneje controles dentro de un GridLayout. El programa deberá crear 18 botones automáticamente con diferentes colores, al pulsar cada uno de ellos, se volverá blanco. El último botón será un botón de RESET, de tal manera que cuando se pulse, se vuelva a la configuración original.



# DIÁLOGOS Y FRAGMENTOS

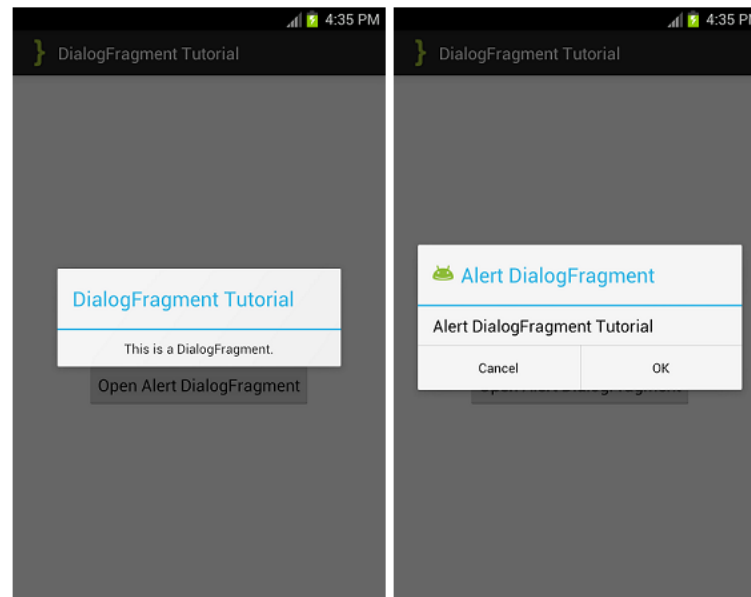
- Los diálogos son ventanas que aparecen en la pantalla de tu dispositivo móvil de manera espontánea para hacer al usuario alguna pregunta sobre alguna decisión que deba tomar para poder proceder con el funcionamiento de nuestra App.
- Existen diversos tipos de diálogos:
  - Diálogos con listas de elementos.
  - Diálogos con botones de tipo Radio o checkboxes.
  - Etc...
- Se puede utilizar un fichero de recursos XML para definir un Layout en el que diseñemos nuestro diálogo personalizado.

<http://developer.android.com/guide/topics/ui/dialogs.html#CustomLayout>



# DIÁLOGOS Y FRAGMENTOS

- Se recomienda usar la clase DialogFragment para contener diálogos.
- Un fragmento de una actividad es una parte de la App que controla una parte de la interfaz de usuario de la Activity.
- Los fragmentos tienen su propio ciclo de vida, reciben sus propios eventos y se pueden eliminar o añadir a la Activity mientras esta se ejecuta.
- Un DialogFragment es un tipo especial de fragmento usado para crear diálogos.

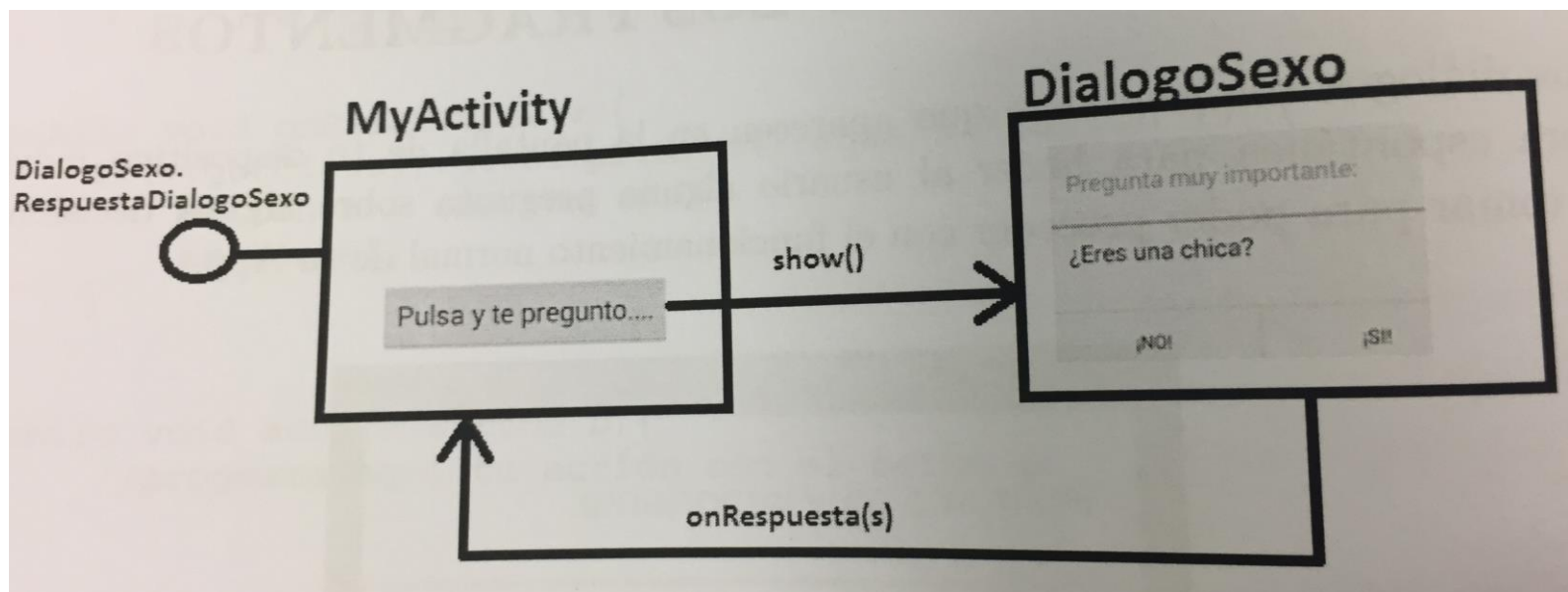


# DIÁLOGOS Y FRAGMENTOS

**CASO PRÁCTICO:** Tenemos una App que en base a la actuación de un usuario necesitamos enseñarle un cuadro de diálogo para hacerle una pregunta. En base a la respuesta la aplicación tomará un curso de acción u otro.

Si lo hacemos con las recomendaciones de Android y programamos el diálogo como un DialogFragment nos servirá para futuras ocasiones.

Suponed que tenemos una Actividad con un botón, al pulsar el botón aparecerá un cuadro de diálogo preguntándonos por nuestro sexo. Al terminar el diálogo, este retornará devolviendo a la actividad el resultado mediante un *callback*.



# DIÁLOGOS Y FRAGMENTOS

## CASO PRÁCTICO:

Nuestra App esta vez tendrá dos clases, una con nuestra actividad principal “MyActivity” y otra con una clase llamada “DialogoSexo” que extenderá la clase *DialogFragment*.

El código de la actividad sería:

```
public class MyActivity extends Activity implements DialogoSexo.RespuestaDialogoSexo{  
    @Override  
    public void onRespuesta(String s){  
        Toast.makeText(getApplicationContext(), s, Toast.LENGTH_LONG).show();  
    }  
    public void click(View v){  
        DialogoSexo ds = new DialogoSexo();  
        ds.show(getFragmentManager(), “Mi diálogo”);  
    }  
    @Override  
    protected void onCreate(Bundle savedInstanceState){...}  
}
```



# DIÁLOGOS Y FRAGMENTOS

## CASO PRÁCTICO:

Programamos una función que responda al click del botón y creamos un objeto de clase `DialogoSexo`, invocando al método `show` de un `Dialog`, que recibe como parámetro un objeto llamado `FragmentManager` que se obtiene con la llamada al método `getFragmentManager()` de la `Activity`, y una etiqueta con el texto identificador del diálogo.

Tenemos que recibir la respuesta que nos envía el diálogo, esto lo haremos registrando una función de callback llamada *RespuestaDialogo* y que programaremos nosotros en la clase `DialogoSexo`. Este método `onRespuesta` crea un pequeño mensaje a modo de notificación que le aparece al usuario en el dispositivo móvil a través de la clase `Toast`.

<http://developer.android.com/guide/topics/ui/notifiers/toast.html>

# DIÁLOGOS Y FRAGMENTOS

## CASO PRÁCTICO:

El código de la clase DialogoSexo sería:

```
public class DialogoSexo extends DialogFragment {  
    RespuestaDialogoSexo respuesta;  
  
    //Este método es llamado al hacer el show() de la clase DialogFragmet()  
    @Override  
    public Dialog onCreateDialog(Bundle savedInstanceState) {  
  
        // Usamos la clase Builder para construir el diálogo  
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());  
  
        //Escribimos el título  
        builder.setTitle("Pregunta muy importante:");  
  
        //Escribimos la pregunta  
        builder.setMessage("¿Eres una chica?");  
    }  
}
```

# DIÁLOGOS Y FRAGMENTOS

## CASO PRÁCTICO:

*//añadimos el botón de Si y su acción asociada*

```
builder.setPositiveButton("¡SI!", new DialogInterface.OnClickListener() {  
    public void onClick(DialogInterface dialog, int id) {  
        respuesta.onRespuesta("Es una chica!");  
    }  
});
```

*//añadimos el botón de No y su acción asociada*

```
builder.setNegativeButton("¡NO!", new DialogInterface.OnClickListener() {  
    public void onClick(DialogInterface dialog, int id) {  
        respuesta.onRespuesta("Es un chico!");  
    }  
});
```

*// Crear el AlertDialog y devolverlo*

```
return builder.create();  
}
```

# DIÁLOGOS Y FRAGMENTOS

## CASO PRÁCTICO:

*//Interfaz para la comunicación entre la Actividad y el Fragmento*

```
public interface RespuestaDialogoSexo{  
    public void onRespuesta(String s);  
}
```

*//Se invoca cuando el fragmento se añade a la actividad*

*@Override*

```
public void onAttach(Activity activity) {  
    super.onAttach(activity);  
    respuesta=(RespuestaDialogoSexo)activity;  
}  
}
```



# DIÁLOGOS Y FRAGMENTOS

## CASO PRÁCTICO:

Los diálogos se crean mediante la clase `AlertDialog.Builder`, estableciendo el título con `setTitle()`, el mensaje con `setMessage()`, y estableciendo los botones de sí o no con su correspondiente respuesta a través de una función de callback de la interfaz `DialogInterface.OnClickListener`.

Todo esto se hace reprogramando el método `onCreateDialog` de la clase `DialogFragment`, método que se invoca cuando la actividad principal ejecuta el `show` de nuestra clase `DialogoSexo`.

Cuando el usuario pulsa uno de los botones del diálogo (sí o no), se desencadena la llamada a la función de callback `onRespuesta` que se ha programado en la actividad principal. Para poder hacer esto, se recurre a el siguiente mecanismo:

- Cuando el diálogo se crea, el fragmento se une “Attach” a la actividad principal. En ese momento el método `onAttach()` se ejecuta y podemos quedarnos con una referencia a la actividad.

```
respuesta = (RespuestaDialogoSexo)activity;
```

- Con esta referencia a la actividad podremos invocar al método `onRespuesta` de la interfaz que obligamos a la actividad principal a implementar.

# DIÁLOGOS Y FRAGMENTOS

## CASO PRÁCTICO:

```
builder.setNegativeButton("¡NO!", new DialogInterface.OnClickListener() {  
    public void onClick(DialogInterface dialog, int id) {  
        respuesta.onRespuesta("Es un chico!");  
    }  
});
```

La clase Builder de AlertDialog tiene, además de los métodos que hemos visto (setTitle, setMessage, setPositiveButton, setNegativeButton) otros métodos que pueden resultar muy útiles:

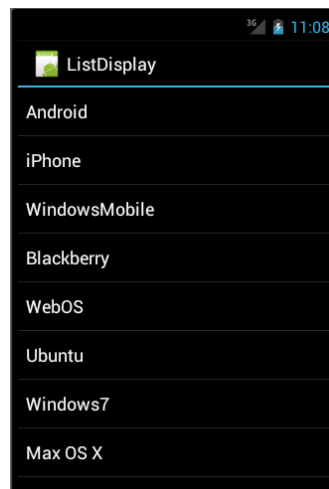
- setItems() → coge de un array una lista de elementos a seleccionar.
- setSingleChoiceItems() → para mostrar una lista con RadioButtons.
- setMultiChoiceItems() → para mostrar una lista de elementos con CheckBoxes.

# ACTIVIDAD 8

- Realiza una aplicación que tenga una Actividad con un botón y al pulsarlo aparezca un cuadro de diálogo haciéndole una pregunta al usuario. Una vez que el usuario responda volver a la actividad y mostrar un texto en respuesta a su elección.

# WIDGETS DE SELECCIÓN

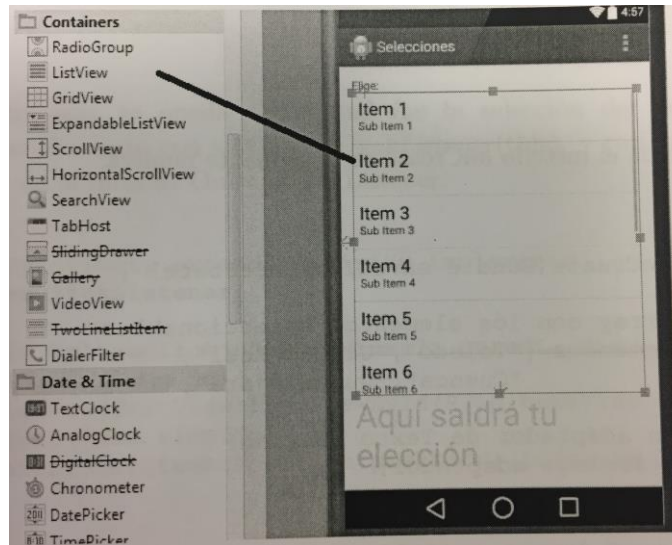
- Permiten al usuario elegir un valor de una lista de posibles valores.
- El widget de selección más típico es el ListView.
- ListView es una subclase de ViewGroup por lo que es un contenedor pero con **adaptadores**.
- Los adaptadores son un “puente” entre un widget y sus datos.
- Para poder llenar una lista de contenido necesitamos dos cosas:
  - Un adaptador.
  - Un nuevo recurso a modo de layout file con la definición de un TextView que se usará como elemento simple del ListView.



# WIDGETS DE SELECCIÓN

**Ejemplo:** Crear una actividad con una lista de elementos seleccionables.

- Creamos un nuevo proyecto e insertamos un ListView y dos etiquetas para la IU.



- Declaramos en el código fuente un objeto de esta clase (ListView) y lo rellenamos con los elementos de un array. Estos elementos serán canalizados hasta los elementos individuales que contenga la lista a través de un adaptador.

# WIDGETS DE SELECCIÓN

## Ejemplo:

- Insertamos un nuevo fichero de recursos con el nombre “fila.xml”.
- Abrimos el fichero fila.xml e insertamos el código para crear un TextView con el formato que deseemos, por ejemplo:

```
<TextView xmlns:android=http://schemas.android.com/apk/res/android
```

```
    android:orientation="vertical"
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
```

```
    android:textStyle="italic"
```

```
    android:textSize="20dp"
```

```
    android:textColor="FF0000"
```

```
>
```

```
</TextView>
```

# WIDGETS DE SELECCIÓN

## Ejemplo:

- Después programamos el método onCreate de la siguiente forma:

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    //Crear un array con los elementos seleccionables
    String[] elementos={"Toledo", "Ciudad Real", "Cuenca", "Guadalajara",
                        "Albacete"}

    //Declarar un adaptador de Texto (String)
    ArrayAdapter<String> adaptador;

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_my);

    //Obtenemos una referencia a la lista
    ListView l = (ListView) findViewById(R.id.listView);

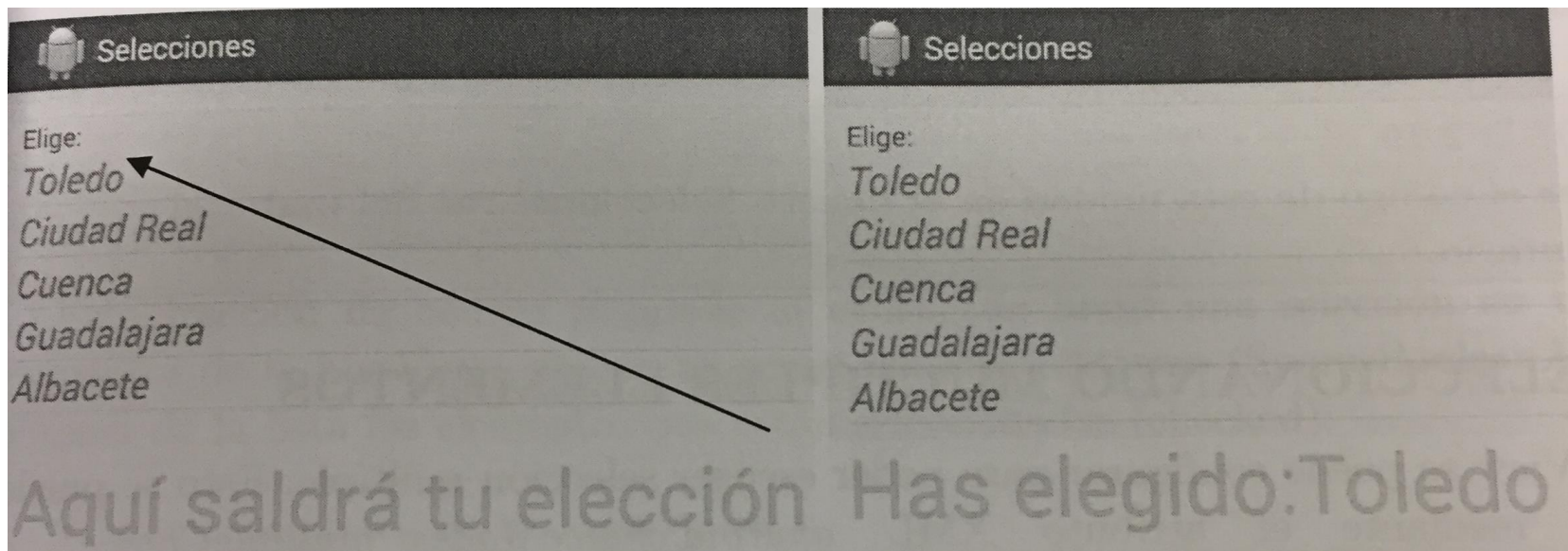
    //Creamos el adaptador
    adaptador = new ArrayAdapter<String>(this, R.layout.fila, elementos);

    //Le damos el adaptador a la lista
    l.setAdapter(adaptador);
}
```

# WIDGETS DE SELECCIÓN

## Ejemplo:

- Para crear el adaptador, el constructor de la clase ArrayAdapter recibe tres parámetros:
  - La referencia a la propia actividad (this).
  - La referencia al nuevo fichero de recurso que hemos creado (R.layout.fila).
  - Los elementos del array con el texto que se escribirá en cada TextView que forme la lista de selección.





# WIDGETS DE SELECCIÓN

## Ejemplo:

- Para saber cuál fue la selección del usuario necesitamos registrar el evento con `setOnItemClickListener(this)`, y recibirlo con el método `OnItemClickListener` de la interfaz `OnItemClickListener`.

```
public class MyActivity extends Activity implements  
    ListView.OnItemClickListener{
```

```
    public void onItemClick(AdapterView<?> parent, View view, int position,  
        long id){
```

```
        TextView t = (TextView) findViewById(R.id.textView);  
        t.setText("Has elegido:" +  
parent.getItemAtPosition(position).toString());
```

```
    }
```

```
    ...
```

```
}
```

# WIDGETS DE SELECCIÓN

## Ejemplo:

- En el método `onItemClick` se pasa como parámetro el componente padre donde se hizo la selección (`AdapterView<?>`), la `View` (el widget) que se pulsó, la posición que ocupa en la lista y el identificador de la fila que se seleccionó.
- Este es el método *callback* que se invoca cuando se hace clic en alguno de los elementos de la lista.
- Gracias a la posición que se pasa como parámetro podemos elegir el elemento que se ha seleccionado con el método *`getItemAtPosition`*.
- Otra alternativa sería, como también se pasa la vista que se ha pulsado, convertir la vista a `TextView` y obtenerlo con el método `getText`:

```
t.setText("Has elegido:" + ((TextView) view).getText());
```

- Tenemos que acordarnos de registrar el listener de la lista, de lo contrario, no hará el *callback*.

```
l.setOnItemClickListener(this);
```

# SELECCIONANDO MÚLTIPLES ELEMENTOS

- Podemos configurar un ListView para poder escoger selección múltiple.
- Esto se puede seleccionar mediante el atributo XML *android:choiceMode="multipleChoice"*.
- También podemos hacerlo invocando el método *setChoiceMode(CHOICE\_MODE\_MULTIPLE)* desde el código fuente.
- Debemos utilizar como Layout para la selección el predefinido por Android para la selección múltiple:

*android.R.layout.simple\_list\_item\_multiple\_choice.*

- Al crear el adaptador debemos indicarlo como segundo parámetro:

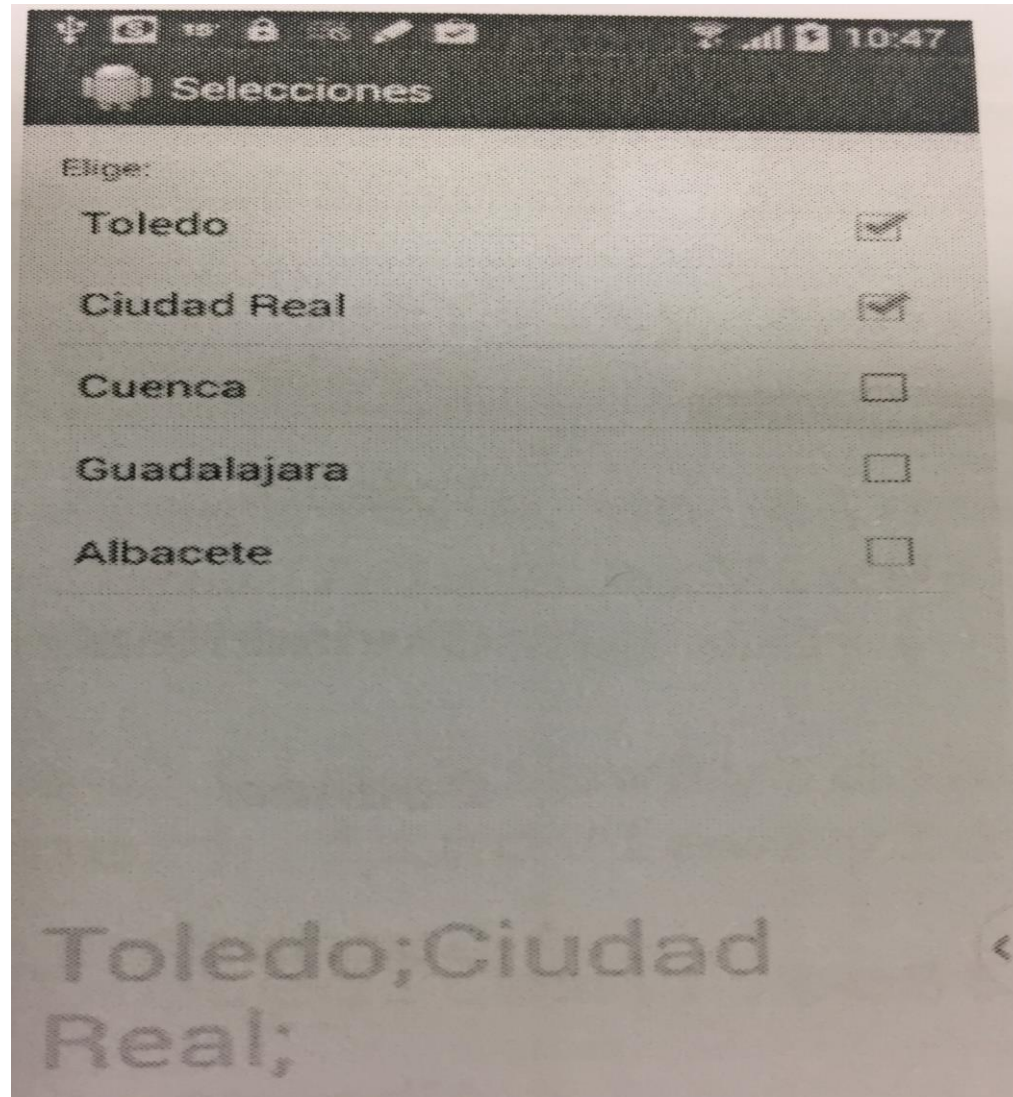
```
adaptador = new ArrayAdapter<String>(this,  
android.R.layout.simple_list_item_multiple_choice, elementos);
```

# SELECCIONANDO MÚLTIPLES ELEMENTOS

- Los métodos para capturar los eventos de selección funcionan exactamente igual que un ListView de selección simple.
- Si queremos explorar todos los elementos seleccionados, por ejemplo mediante un bucle, podemos usar el siguiente código:

```
public void onItemClickListener(AdapterView<?> a, View view, int position, long id){  
    TextView t = (TextView) findViewById(R.id.textView);  
    ListView l = (ListView) findViewById(R.id.listView);  
    String seleccionado = new String();  
    SparseBooleanArray checked = l.getCheckedItemPositions();  
  
    for(int i=0; i<checked.size; i++){  
        if(checked.valueAt(i)){  
            seleccionado = seleccionado +  
                l.getItemAtPosition(checked.keyAt(i)).toString() + “.”;  
        }  
    }  
    t.setText(seleccionado);  
}
```

# SELECCIONANDO MÚLTIPLES ELEMENTOS





# SELECCIONANDO MÚLTIPLES ELEMENTOS

- El código anterior responde al evento de hacer una selección en uno de los elementos de la ListView.
- Hace uso de un array booleano disperso (`SparseBooleanArray`) para obtener de la lista los elementos que están seleccionados (`checked`):

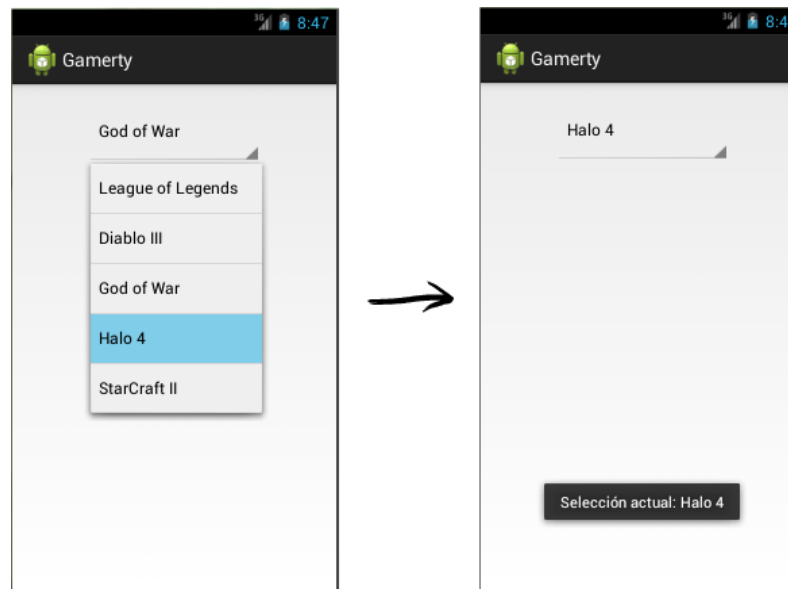
*`SparseBooleanArray checked = l.getCheckedItemPositions();`*

- El método `getCheckedItemPositions` de la ListView devuelve este tipo de array para que podamos consultar qué valores se han seleccionado.
- El método `valueAt(i)` devuelve un booleano indicando si el elemento `i` del array fue seleccionado o no.
- Con el método `keyAt(i)` del array obtenemos un número que indica la posición de un elemento seleccionado dentro de la ListView.
- De esta forma podemos hacer un bucle desde 0 hasta el tamaño del array, así como extraer los valores múltiples seleccionados.

<http://developer.android.com/reference/android/útil/SparseBooleanArray.html>

# SPINNERS

- Los Spinners en Android son el equivalente a los ComboBox de otros sistemas, son listas desplegables.
- Funcionan de forma similar a los ListView, con lo que hay que utilizar un adaptador para enchufar los datos y capturar los eventos de selección.
- En el caso de los Spinners esto se hace a través de *setOnItemSelectedListener()* de la interfaz *OnItemSelectedListener* de la clase Spinner.



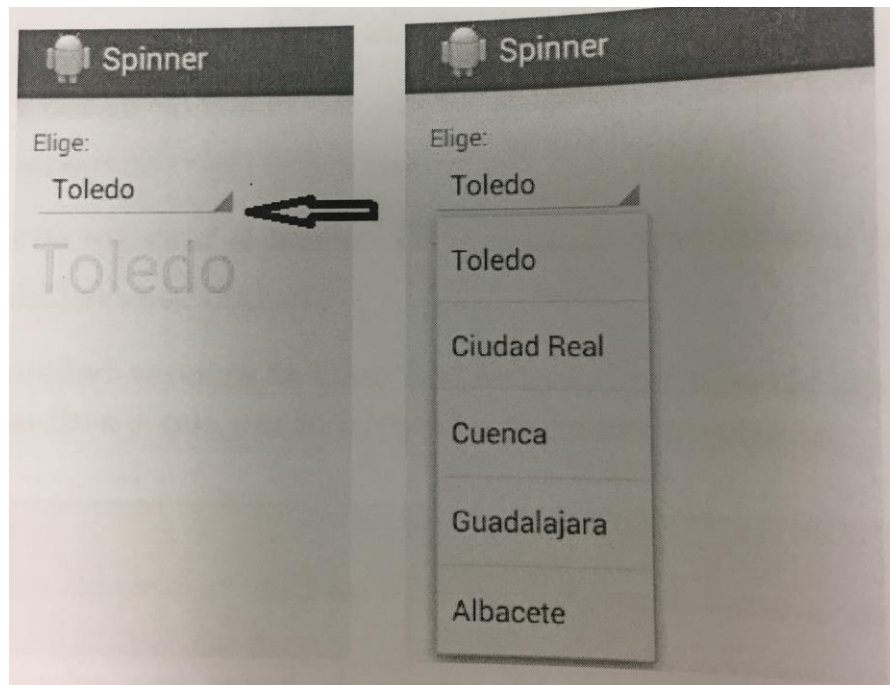
# SPINNERS

- El adaptador se programa de la siguiente forma:

```
adaptador = new ArrayAdapter<String>(this,  
    android.R.layout.simple_spinner_item, elementos);
```

```
adaptador.setDropDownViewResource(  
    android.R.layout.simple_spinner_dropdown_item);
```

- En este caso hay dos layouts, uno para mostrar el elemento seleccionado y otro para mostrar los elementos desplegables.





# SPINNERS

```
protected void onCreate(Bundle savedInstanceState) {  
    String[] elementos = {"Toledo", "Ciudad Real", "Cuenca",  
        "Guadalajara", "Albacete"};  
  
    ArrayAdapter<String> adaptador;  
  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_my);  
  
    Spinner sp = (Spinner) findViewById(R.id.spinner);  
    adaptador = new ArrayAdapter<String>(this,  
        android.R.layout.simple_spinner_item, elementos);  
  
    adaptador.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);  
    sp.setAdapter(adaptador);  
  
    sp.setOnItemSelectedListener(this);  
}
```

# SPINNERS

- Para capturar el evento de selección de un elemento hay que implementar dos métodos, puesto que la interfaz `OnItemSelectedListener` exige los dos.

```
public class MyActivity extends Activity implements  
Spinner.OnItemSelectedListener{
```

```
    public void onItemSelected(AdapterView<?> a, View view, int position,  
    long id){
```

```
        TextView t=(TextView)findViewById(R.id.textView);  
        Spinner sp = (Spinner) findViewById(R.id.spinner);
```

```
        t.setText(sp.getSelectedItem().toString());
```

```
    }
```

```
    public void onNothingSelected(AdapterView<?> a){  
        TextView t=(TextView)findViewById(R.id.textView);  
        t.setText("No se ha seleccionado nada");
```

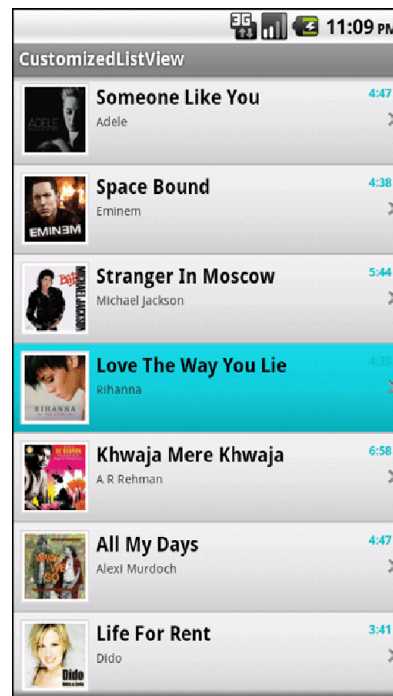
```
    }
```

```
    ...
```

```
}
```

# SELECCIONES PERSONALIZABLES

- Podemos personalizar cualquier tipo de control de selección.
- Podemos definir la estructura de cada fila de la lista de selección como nosotros queramos.
- Para integrarlo con un Spinner o una ListView tenemos que reescribir el funcionamiento del adaptador (`ArrayAdapter<T>`) de manera que cuando el selector pida los datos de las filas, devuelva una vista que sea la que queremos en cada momento.



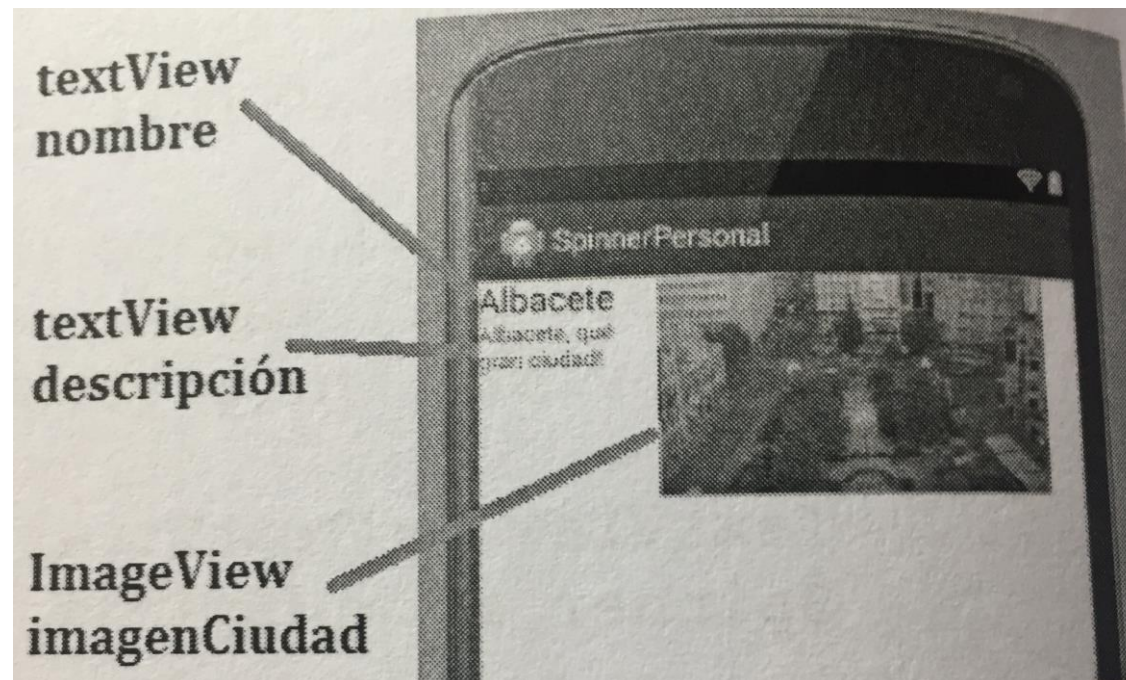
# SELECCIONES PERSONALIZABLES

- El inflador: Se pueden crear vistas (View) en tiempo de ejecución a partir de un fichero de recursos XML. De esta manera, podemos crear un widget personalizado e “inflarlo” (crearlo) habiendo definido su estructura mediante un Layout definido en XML.
- Hay que redefinir algunos de los métodos del adaptador:
  - `getDropDownView(int position, View convertView, ViewGroup parent)`
  - `getView(int position, View convertView, ViewGroup parent)`
- Estos dos métodos devuelven la vista personalizada de una determinada fila.
- Cuando el Spinner invoque a estos métodos nosotros debemos pasarle la fila personalizada de la posición que nos indique el parámetro posición.

# SELECCIONES PERSONALIZABLES

## Ejemplo:

- Vamos a mostrar un Spinner personalizado, de tal forma que nos muestre las ciudades de Castilla – La Mancha con un formato personalizado.
- La App constará de dos ficheros XML, el propio de la actividad, con un Spinner y una caja de texto donde se escribirá el resultado de la elección, y el fichero XML con el diseño de una fila del Spinner.



# SELECCIONES PERSONALIZABLES

## Ejemplo:

- El fichero de la fila, llamado *lineaspinex.xml* tiene el siguiente código:

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

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Albacete, qué gran ciudad!"
    android:id="@+id/descripcion"
    android:layout_alignParentTop="true"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:ellipsize="none"
    android:scrollHorizontally="false"
    android:layout_marginTop="26dp"
    android:layout_alignBottom="@+id/imagenCiudad"
    android:layout_alignRight="@+id/nombre"
    android:layout_alignEnd="@+id/nombre" />
```

# SELECCIONES PERSONALIZABLES

## Ejemplo:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:text="Albacete"
    android:id="@+id/nombre"
    android:layout_alignParentTop="true"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true" />
```

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/albacete"
    android:id="@+id/imagenCiudad"
    android:layout_toEndOf="@+id/descripcion"
    android:layout_alignParentTop="true"
    android:layout_alignParentRight="true"
    android:layout_alignParentEnd="true" />
```

```
</RelativeLayout>
```

# SELECCIONES PERSONALIZABLES

## Ejemplo:

- En el fichero Java de la actividad, vamos a crear una subclase de la clase ArrayAdapter, con los métodos sobrecargados, esta clase se apoyará en tres arrays de datos para crear el objeto View que se mandará al Spinner.
- Cada array tendrá una información para los diferentes campos que componen la fila del spinner:

```
String[] ciudades = { "Toledo", "Ciudad Real", "Albacete", "Cuenca", "Guadalajara" };  
String[] descripciones = { "La ciudad Imperial", "Qué gran ciudad",  
    "Ciudad gastronómica", "Ciudad encantada", "Ciudad colgante" };
```

```
int imagenes[] = { R.drawable.toledo, R.drawable.ciudadreal, R.drawable.albacete,  
    R.drawable.cuenca, R.drawable.guadalajara};
```

```
public class AdaptadorPersonalizado extends ArrayAdapter<String> {  
    public AdaptadorPersonalizado(Context ctx, int textViewResourceId, String[] objects){  
        super(ctx, textViewResourceId, objects);  
    }  
  
    @Override  
    public View getDropDownView(int position, View convertView, ViewGroup parent){  
        return crearFilaPersonalizada(position, convertView, parent);  
    }  
}
```



# SELECCIONES PERSONALIZABLES

## Ejemplo:

```
@Override
public View getView(int pos, View convertView, ViewGroup prnt){
    return crearFilaPersonalizada(pos, convertView, prnt);
}

public View crearFilaPersonalizada(int position, View convertView, ViewGroup parent){

    LayoutInflater inflater = getLayoutInflater();
    View miFila = inflater.inflate(R.layout.lineaspiner, parent, false);

    TextView nombre = (TextView) miFila.findViewById(R.id.nombre);
    nombre.setText(ciudades[position]);

    TextView descripcion = (TextView) miFila.findViewById(R.id.descripcion);
    descripcion.setText(descripciones[position]);

    ImageView imagen = (ImageView) miFila.findViewById(R.id.imagenCiudad);
    imagen.setImageResource(imagenes[position]);
    return miFila;
}
}
```

# SELECCIONES PERSONALIZABLES

## Ejemplo:

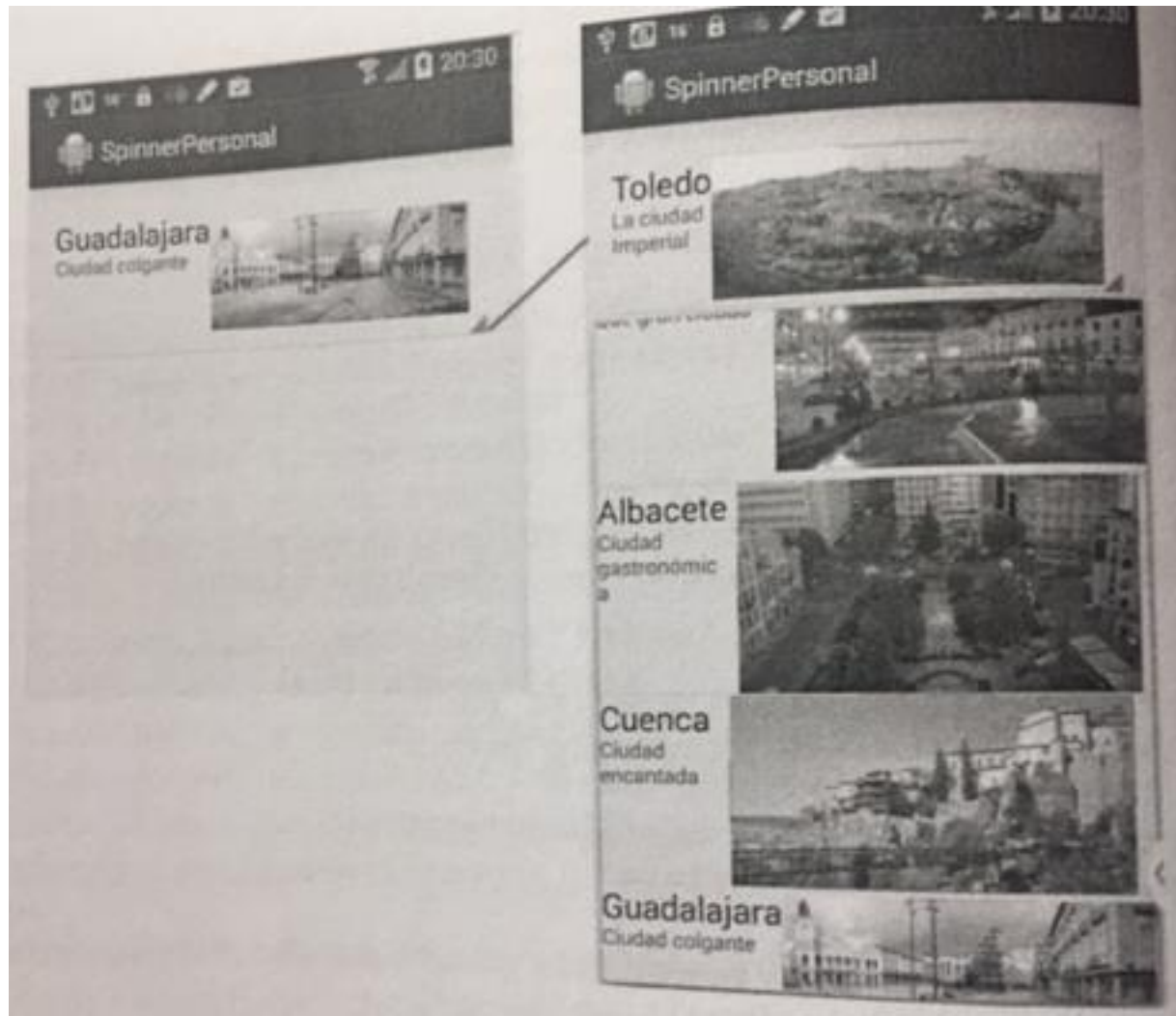
- Los dos métodos reescritos (getDropDownView y getView) invocan a una tercera función crearFilaPersonalizada() que “infla” la vista del fichero xml *lineaspiner.xml* para crear un objeto View con los datos de los arrays y lo devuelve.
- Cuando creamos el Spinner debemos acordarnos de enviarle el adaptador de esta subclase:

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_my);  
  
    Spinner selectorCiudades = (Spinner) findViewById(R.id.spinner);  
    AdaptadorPersonalizado a=new AdaptadorPersonalizado(this,  
R.layout.lineaspiner, ciudades);  
    selectorCiudades.setAdapter(a);  
    selectorCiudades.setOnItemSelectedListener(this);  
}
```

# SELECCIONES PERSONALIZABLES

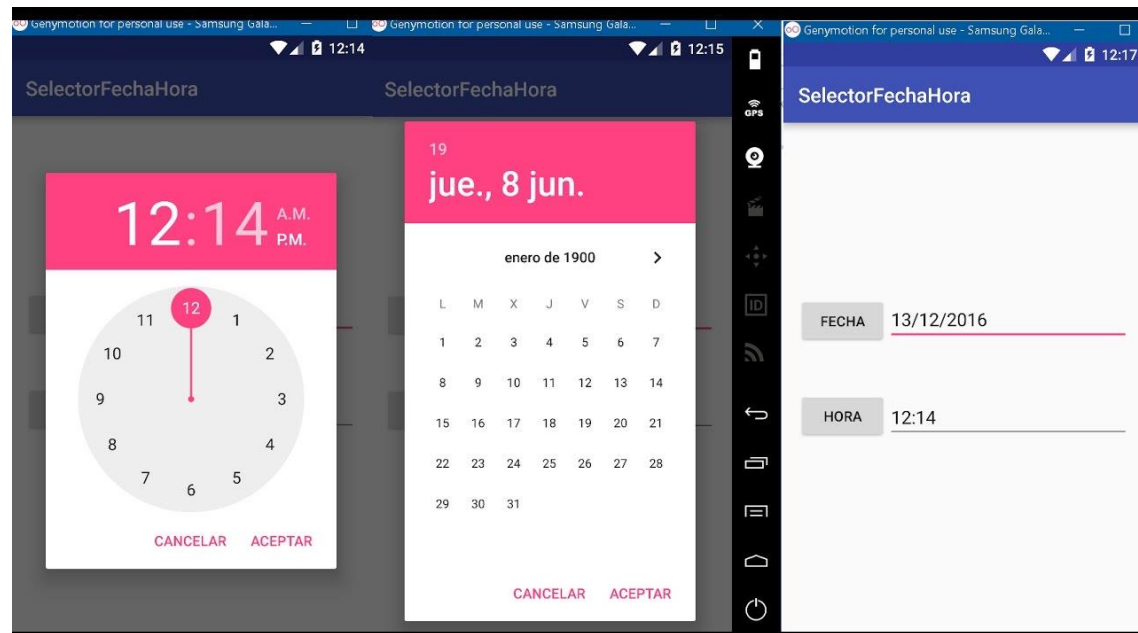
## Ejemplo:

- De esta manera el resultado nos quedaría de la siguiente forma:



# SELECTORES DE FECHA/HORA

- En inglés se llama a este tipo de selectores con los nombres *Pickers*, *DatePicker* o *TimePicker*.
- Estos selectores nos ahorrarán muchas comprobaciones en código sobre si el usuario ha escrito correctamente el formato de la hora/fecha, si se ajustan a las configuraciones locales del dispositivo móvil, o si es una fecha válida.
- Al igual que los Diálogos, Android recomienda utilizar la clase `DialogFragment` para implementar este tipo de selectores.



# SELECTORES DE FECHA/HORA

## Ejemplo:

- Vamos a crear un sencillo formulario que muestra al usuario la posibilidad de introducir su fecha de nacimiento y una hora a través de un campo de texto.
- Los campos de fecha y hora estarán inhabilitados para forzar al usuario a pulsar un botón para poder seleccionar de un *Picker* tanto la fecha como la hora.
- Al pulsar en cada respectivo botón se mostrará un diálogo con su *Picker* correspondiente.
- Para cada diálogo creamos una clase que herede de `DialogFragment` y que poseerá una interfaz para comunicarse con la actividad principal, devolviendo esta un objeto `Fecha` cuando se haya seleccionado tanto la fecha como la hora.

# SELECTORES DE FECHA/HORA

## Ejemplo:

- Para la devolución utilizaremos la clase `GregorianCalendar`, que sustituye a la clase `Date` de Java ya que está depreciada (`deprecated`) desde la API 1.
- Para crear el diálogo se puede usar un Layout XML que contenga un *Picker* o directamente utilizar las clases *DatePickerDialog* y *TimePickerDialog* que directamente construyen un *Dialog* con un *Picker*.
- Para este ejemplo vamos a usar la segunda opción, y además al crear el fragmento se ejecutará el método *onAttach*, momento que aprovecharemos para quedarnos con una referencia a la actividad y así poder invocar al método *onResultadoFecha/onResultadoHora* de la interfaz y pasar el resultado de la elección del usuario a la actividad principal.

# SELECTORES DE FECHA/HORA

## Ejemplo:

- La clase DialogoFecha consistirá en:

```
public class DialogoFecha extends DialogFragment implements  
DatePickerDialog.OnDateSetListener{
```

```
    OnFechaSeleccionada f;
```

```
    @Override
```

```
    public void onAttach(Activity activity) {  
        f=(OnFechaSeleccionada)activity;  
        super.onAttach(activity);  
    }
```

```
    @Override
```

```
    public Dialog onCreateDialog(Bundle savedInstanceState) {
```

```
        Calendar c=Calendar.getInstance();
```

```
        int año=c.get(Calendar.YEAR);
```

```
        int mes=c.get(Calendar.MONTH);
```

```
        int dia=c.get(Calendar.DAY_OF_MONTH);
```

```
        return new DatePickerDialog(getActivity(),this,año,mes,dia);
```

```
    }
```

# SELECTORES DE FECHA/HORA

## Ejemplo:

```
@Override
public void onDateSet(DatePicker datePicker, int i, int i2, int i3) {
    GregorianCalendar g=new GregorianCalendar(i,i2,i3);
    f.onResultadoFecha(g);
}

public interface OnFechaSeleccionada{
    public void onResultadoFecha(GregorianCalendar fecha);
}

}
```



# SELECTORES DE FECHA/HORA

## Ejemplo:

- La clase DialogoHora consistirá en:

```
public class DialogoHora extends DialogFragment implements  
TimePickerDialog.OnTimeSetListener{
```

```
    OnHoraSeleccionada f;
```

```
    @Override
```

```
    public void onAttach(Activity activity) {  
        f=(OnHoraSeleccionada)activity;  
        super.onAttach(activity);  
    }
```

```
    @Override
```

```
    public Dialog onCreateDialog(Bundle savedInstanceState) {
```

```
        Calendar c=Calendar.getInstance();
```

```
        int hora=c.get(Calendar.HOUR);
```

```
        int minutos=c.get(Calendar.MINUTE);
```

```
        return new TimePickerDialog(getActivity(),this,hora,minutos,true);
```

```
    }
```

# SELECTORES DE FECHA/HORA

## Ejemplo:

```
@Override
public void onTimeSet(TimePicker timePicker, int i, int i2) {
    GregorianCalendar g=new GregorianCalendar();
    g.set(Calendar.HOUR,i);
    g.set(Calendar.MINUTE,i2);
    f.onResultadoHora(g);
}

public interface OnHoraSeleccionada{
    public void onResultadoHora(GregorianCalendar hora);
}

}
```

# SELECTORES DE FECHA/HORA

## Ejemplo:

- Al crear un objeto de estos tipos `DialogoFecha/DialogoHora` e invocar a su método `show()`, se ejecutará el método `onCreateDialog()` que debe retornar el diálogo.
- Para crear el diálogo usamos los constructores
  - `New DatePickerDialog(getActivity(), this, año, mes, dia);`
  - `New TimePickerDialog(getActivity(), this, hora, minutos, true);`
- A estos constructores les pasamos los valores de fecha y hora actuales (año, mes, dia) y (hora, minutos) obtenidos de la clase `Calendar`.
- La clase `Calendar` nos proporciona acceso a la fecha y hora actual del dispositivo móvil.
- Para formar una fecha y una hora a partir de la elección del usuario usamos los métodos de callback `on<X>Set` de las interfaces `OnXSetListener`, donde `<X>` puede ser `Time` o `Date`.



# SELECTORES DE FECHA/HORA

## Ejemplo:

- Estos métodos nos avisan de que el usuario ha definido una nueva fecha y hora en el diálogo y ha retornado con éxito de la selección.
- En ese momento se construye un objeto *GregorianCalendar* con los valores seleccionados por el usuario, recibidos en los parámetros de la función *On<X>Set*, y se invoca a la función *onResultadoHora/onResultadoFecha* de la interfaz correspondiente y que debe implementar la actividad.
- Estos dos últimos métodos reciben como parámetro un objeto *GregorianCalendar*, subclase de *Calendar*.

<http://developer.android.com/reference/java/util/Calendar.html>

<http://developer.android.com/reference/java/util/GregorianCalendar.html>

# SELECTORES DE FECHA/HORA

## Ejemplo:

- Por último, la actividad principal tendrá 4 funciones programadas:

```
public void onClickFecha(View view) {
    DialogoFecha d=new DialogoFecha();
    d.show(getFragmentManager(),"Mi diálogo Fecha");
}
public void onClickHora(View view){
    DialogoHora d=new DialogoHora();
    d.show(getFragmentManager(),"Mi diálogo Hora");
}
@Override
public void onResultadoFecha(GregorianCalendar fecha) {
    EditText et=(EditText)findViewById(R.id.etFechaNacimiento);

    et.setText(fecha.get(Calendar.DAY_OF_MONTH)+"/"+(fecha.get(Calendar.MONTH)+1)+"/"+f
echa.get(Calendar.YEAR));
}

@Override
public void onResultadoHora(GregorianCalendar hora) {
    EditText et=(EditText)findViewById(R.id.etHora);
    et.setText(hora.get(Calendar.HOUR)+":"+hora.get(Calendar.MINUTE));
}
```

# SELECTORES DE FECHA/HORA

## Ejemplo:

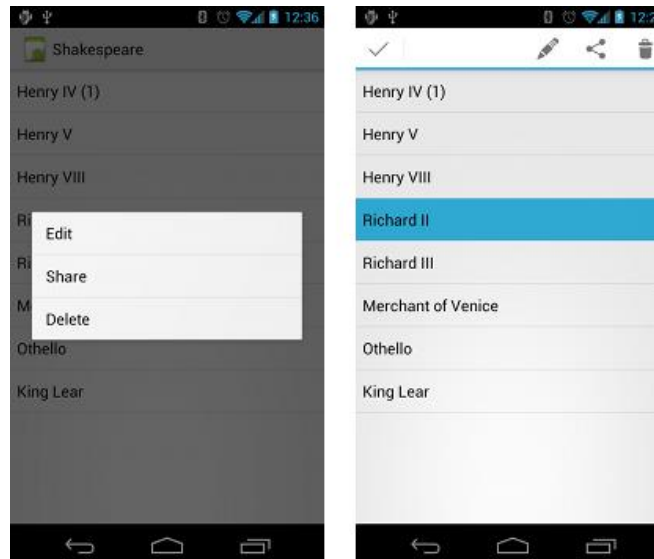
- Las funciones *onClickFecha* y *onClickHora* son los callback de los botones que pulsa el usuario cuando quiere cambiar la fecha, y *onResultadoFecha* y *onResultadoHora* son las funciones de callback que hay que implementar por la necesidad de comunicarse con el *DialogFragment* correspondiente:

```
public class MyActivity extends Activity implements  
DialogoFecha.OnFechaSeleccionada, DialogoHora.OnHoraSeleccionada{  
  
...  
}
```

[Solución Ejemplo](#)

# CONSTRUCCIÓN DE MENÚS

- Para crear elementos de menús en Android tan solo tenemos que definir un archivo de recursos XML dentro de la carpeta “Menu” del proyecto.
- Utilizando solamente las etiquetas “menu” e “item” podemos hacer cualquier organización jerárquica de menús.
- La etiqueta <menu> describe un grupo de elementos (ítem) que será cada entrada del menú.
- Los menús y submenús han caído en desuso desde Android 3.0+, principalmente por la existencia de la ActionBar.





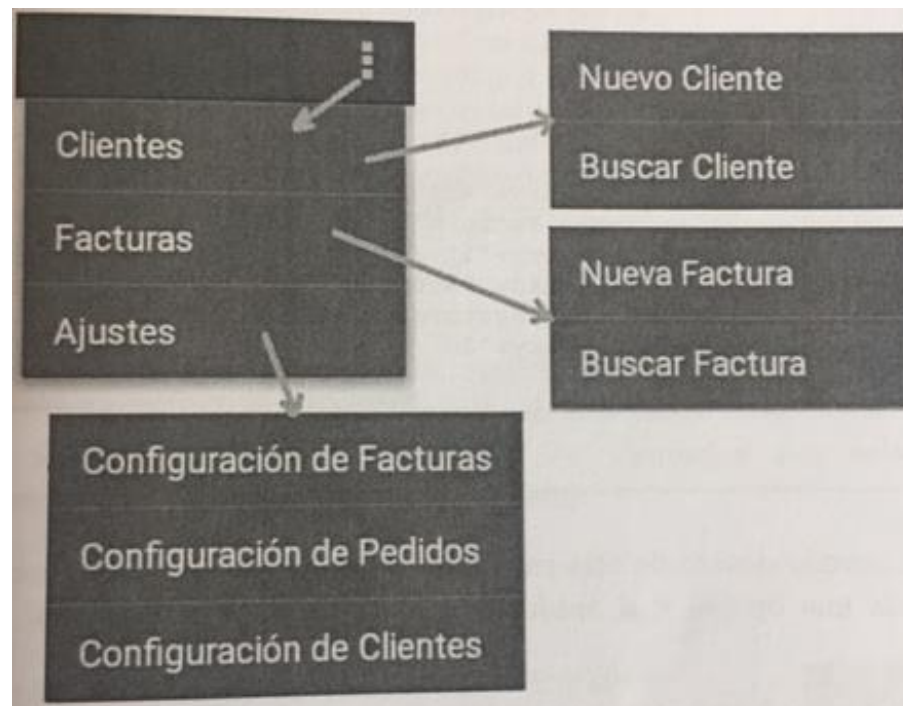
# CONSTRUCCIÓN DE MENÚS

- Hay varios tipos de menús:
  - Menús de opciones o ajustes (settings): Son los menús que salen en la barra de acción de tu App y se controlan a través de un método callback que genera automáticamente Android Studio y que se llama *onOptionsItemSelected*. Este método es invocado cuando el usuario selecciona uno de los elementos de este menú. Android Studio también genera el código para crear el menú *onCreateOptionsMenu* que “infla” un determinado menú diseñado en un fichero de recursos xml.
  - Menús contextuales: Son los menús flotantes que aparecen cuando un usuario hace un clic de manera prolongada en un elemento de la interfaz.
  - Menús de Pop-up: Visualizan elementos de menú en una lista vertical. Estos menús aparecen anclados al elemento de la IU que provocó su aparición.
  - Submenús: Aparecen al seleccionar una opción de menú, reemplazando el menú principal por las opciones del submenú.



# CONSTRUCCIÓN DE MENÚS

- Si utilizamos el recurso *my.xml* estamos definiendo el menú por defecto de la aplicación.
- La función *onCreateOptionsMenu* “infla” este recurso por defecto.
- Imaginaros que queremos hacer una App típica de gestión comercial. Podríamos tener como menú de opciones de nuestra App el siguiente menú:



# CONSTRUCCIÓN DE MENÚS

- Los menús en la barra de acción, a partir de Android 3.0, aparecen también en la parte superior derecha al presionar el botón de *Overflow*, en versiones anteriores aparecen en la esquina inferior izquierda al pulsar la tecla de menú del dispositivo.
- Algunas de las opciones de cada <item> de un menú son las siguientes:
  - android:id → Identificador para el elemento del menú.
  - android:title → Texto que aparece en el elemento de menú.
  - android:orderInCategory → Orden en el que aparece el elemento de menú dentro de su agrupación.
  - android:icon → Icono para el elemento de menú. Los iconos no aparecerán si tu aplicación se ejecuta en Android 3.0 o mayor. Tan solo se mostrarán si forman parte de nuestra ActionBar.
  - android:showAsAction → Indica si el elemento aparecerá como un elemento en la barra de Acción. El valor *never*, hará que el elemento de menú no se muestre en la barra de acción. Si se especifica *always*, siempre aparecerá y si se especifica *ifRoom* solo aparecerá si hay espacio suficiente en la barra de acción.

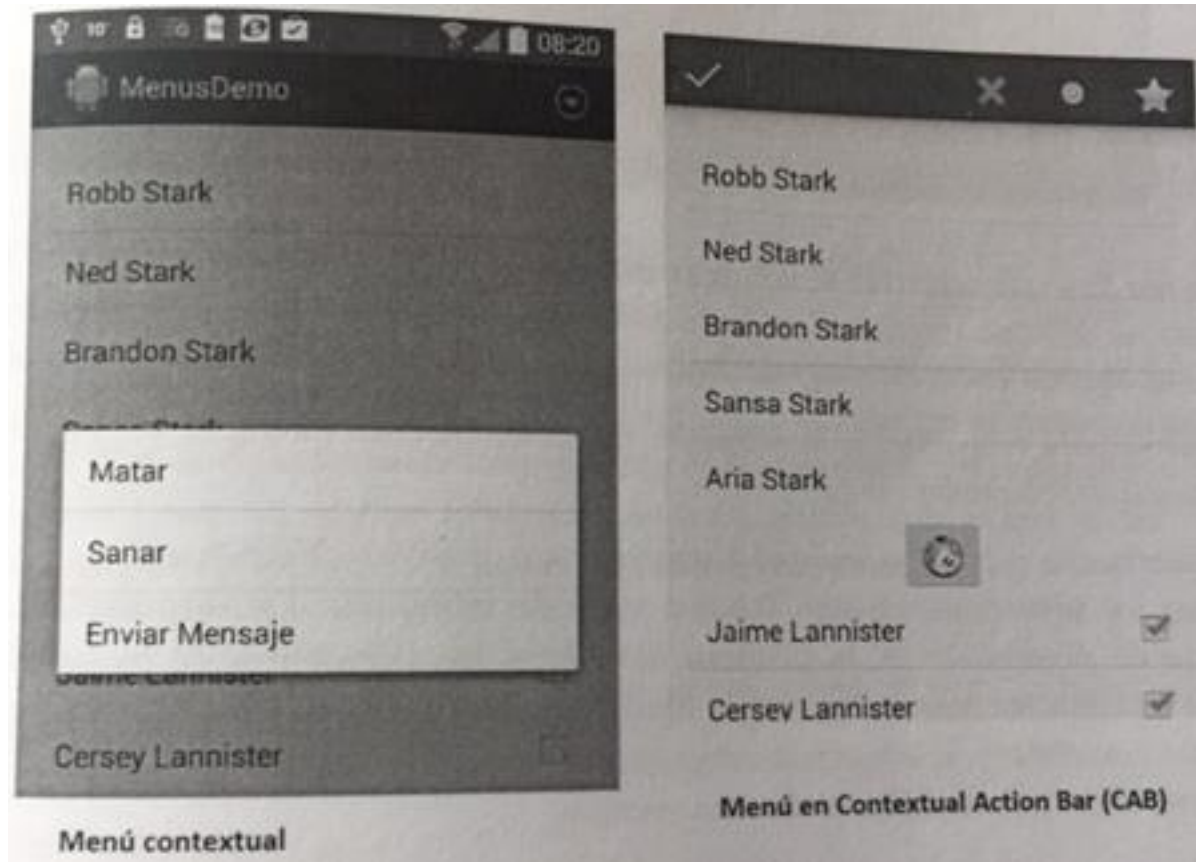
# DOTANDO DE ACCIÓN A LOS ELEMENTOS DE MENÚ

- Para responder al clic en un elemento de menú, hay que modificar el método *onOptionsItemSelected*:

```
@Override
Public boolean onOptionsItemSelected(MenuItem item){
int id = ítem.getItemId();
switch(id){
    case R.id.BuscarCliente:
        Toast.makeText(getApplicationContext(), "Se ha pulsado Bucar Cliente",
Toast.LENGTH_LONG).show();
        return true;
    case R.id.Clientes:
        Toast.makeText(getApplicationContext(), "Se ha pulsado Cliente",
Toast.LENGTH_LONG).show();
        return true;
    case R.id.Facturas:
        Toast.makeText(getApplicationContext(), "Se ha pulsado Facturas",
Toast.LENGTH_LONG).show();
        return true;
    ...
}
return super.onOptionsItemSelected(item);
}
```

# MENÚS CONTEXTUALES

- Un menú contextual ofrece opciones que afectan a un elemento de la interfaz, por ejemplo un elemento de una lista o una imagen. Hay dos formas de hacer menús contextuales:





# MENÚS CONTEXTUALES

- El primero se utiliza cuando el usuario realiza una pulsación larga (long click) en un elemento de la interfaz y el segundo visualiza una barra de acción contextual, en inglés *Contextual Action Bar (CAB)*.
- Esta CAB se puede programar para que aparezca también cuando se produce un clic largo en un elemento de la IU o, por ejemplo, cuando se selecciona uno o varios elementos de una lista.
- Tenéis el código de ejemplo de la creación de un menú contextual en el fichero MenusDemo.rar en la carpeta compartida de google drive.

# ACTIONBAR

- La barra de acción es una de las principales características de nuestras Apps.
- Está visible en todo momento mientras nuestras aplicaciones están visibles y nos da la posibilidad de dotar a nuestras aplicaciones de una identidad.
- También nos permite situar iconos para acciones importantes para nuestras aplicaciones (buscar o crear algo importante).
- Permite al usuario realizar una navegación consistente por toda la aplicación, por ejemplo a través de pestañas o tabs.
- A partir de la API de nivel 11, se incluye en todas las actividades que utilicen el tema Theme.Holo en el diseño de la actividad (este es el tema por defecto).
- Si no queremos incluir un ActionBar en nuestra App tan solo tenemos que cambiar el tema de la actividad a Theme.Holo.NoActionBar en el fichero styles.xml.
- Podemos utilizar y descargar iconos típicos de Android en: <http://developer.android.com/design/style/iconography.html>

# ACTIONBAR

- Para añadir la barra de acción tan solo hay que modificar el método `onCreateOptionsMenu()` e “inflar” el archivo de recursos de menú xml que hayamos creado.
- Cuando vayamos a crear el menú podemos solicitar que aparezca directamente en la barra de acción alguno de los elementos del menú con la opción `showAsAction=“ifRoom”` o `showAsAction=“always”`.

```
public boolean onCreateOptionsMenu(Menu menu){  
    //”Inflamos” los ítems de menú para usar en la barra de acción  
    MenuInflater inflater = getMenuInflater();  
    inflater.inflate(R.menu.menu_action_bar, menu);  
    Return super.onCreateOptionsMenu(menu)  
  
}
```

# ACTIONBAR

- Para responder a los eventos de clic de los eventos de clic de los elementos que hemos colocado en la Action Bar debemos escribir nuestro código en el método *onOptionsItemSelected()*.

```
@Override
Public boolean onOptionsItemSelected(MenuItem item){
int id = ítem.getItemId();
switch(id){
    case R.id.borrar:
        Toast.makeText(getApplicationContext(), "Se ha pulsado borrar",
Toast.LENGTH_LONG).show();
        return true;
    case R.id.edit:
        Toast.makeText(getApplicationContext(), "Se ha pulsado editar",
Toast.LENGTH_LONG).show();
        return true;
    case R.id.llamar:
        Toast.makeText(getApplicationContext(), "Se ha pulsado llamar",
Toast.LENGTH_LONG).show();
        return true;
    default:
        return super.onOptionsItemSelected(ítem);
}
}
```