

Desarrollo Android

Arkaitz Garro



Interfaz de usuario

Fundamentos

As if a device can function if it has no style. As if a device can be called stylish that does not function superbly.... Yes, beauty matters. Boy, does it matter. It is not surface, it is not an extra, it is the thing itself.

—STEPHEN FRY, THE GUARDIAN (OCTOBER 27, 2007)

Para entendernos...

Views (vistas): es la clase padre para todos los elementos visuales, incluyendo controles, layouts, widgets...

View Groups: son una extensión de las vistas, que contienen un grupo de vistas. La clase **ViewGroup** a su vez es extendida para crear diferentes tipos de Layouts.

Fragments: introducidos en Android 3.0, son utilizados para encapsular partes de la UI. Muy útiles a la hora de crear interfaces para distintas resoluciones.

Activity: representa una ventana. Para mostrar una interfaz, es necesario asignar una vista a la actividad.

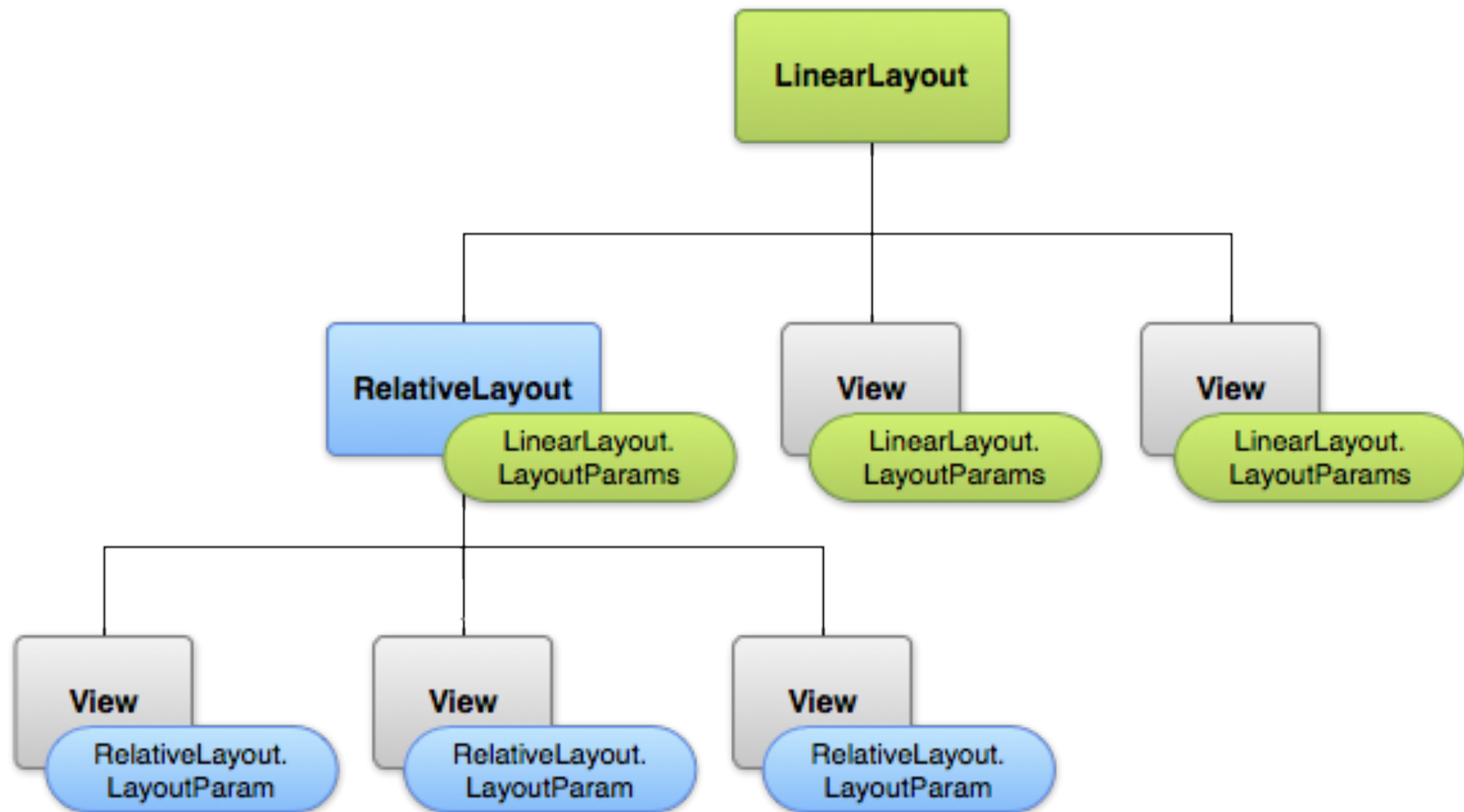
Introduciendo Layouts

Los Layout son contenedores de vistas utilizados para posicionar los elementos de la interfaz

Los Layout pueden anidarse para conseguir interfaces más complejas

Android nos proporciona varios tipos de Layout, que podemos utilizar, modificar e incluso crear nuevos

Introduciendo Layouts



Layouts de Android



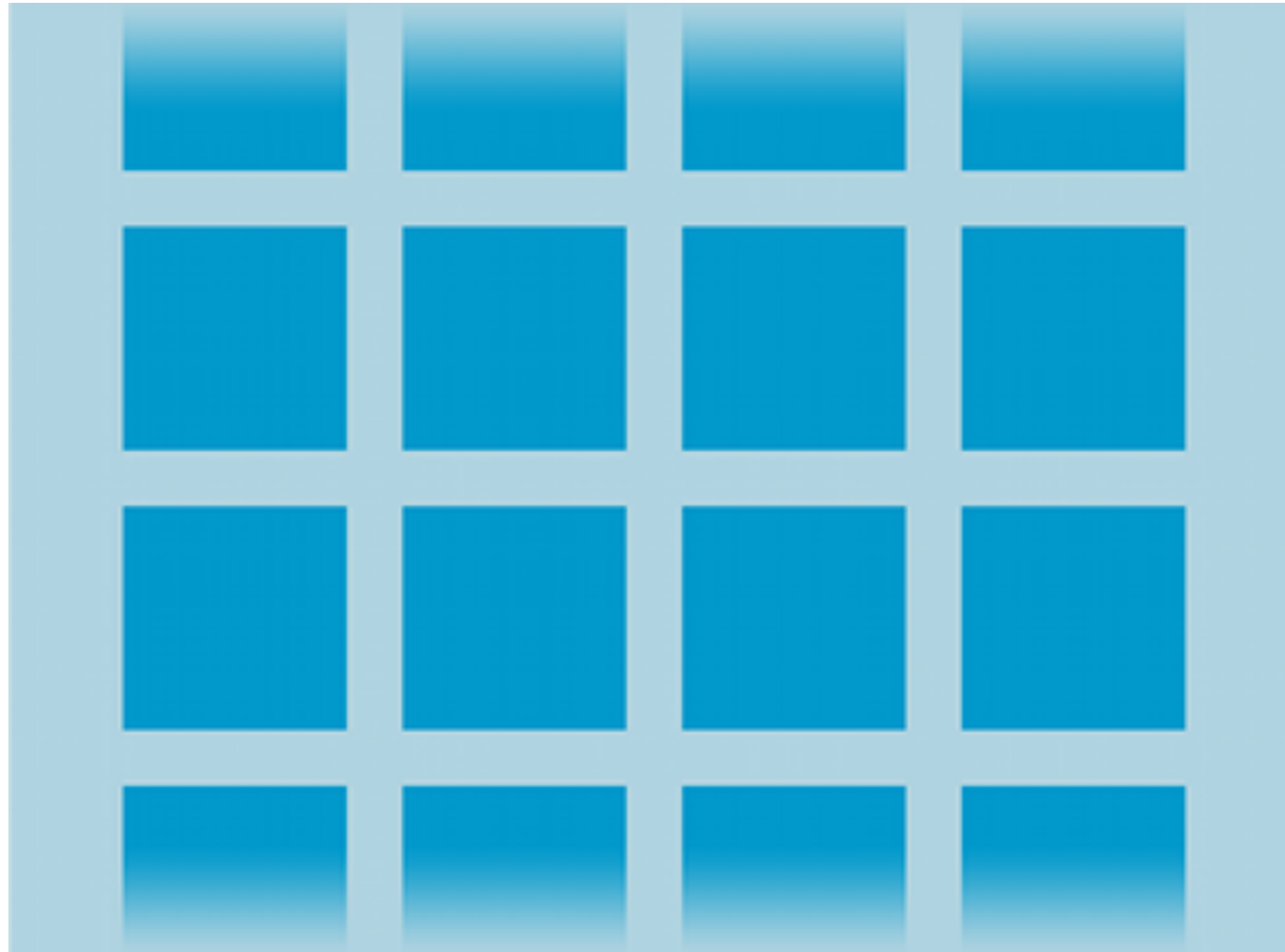
Linear Layout: alinea cada vista, una tras otra, en vertical u horizontal. Cada una de estas filas/columnas puede tener un atributo **weight** que indica su tamaño relativo.

Layouts de Android



Relative Layout: permite definir la posición de cada vista con respecto a otras vistas y los límites de la pantalla

Layouts de Android



Grid Layout: permite representar de una manera muy sencilla elementos de manera bi-dimensional. Los elementos son insertados automáticamente utilizando un adaptador.

Layouts de Android



List View: permite representar de una manera muy sencilla elementos en forma de listado. Los elementos son insertados automáticamente utilizando un adaptador.

Cómo definir un Layout

La “mejor” manera de definir un Layout es a través de XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Interfaces independientes

La manera más sencilla de crear interfaces es utilizando
LinearLayouts

Esta facilidad limita su flexibilidad, ya que sólo es posible
listas horizontales o verticales

Principalmente es utilizado dentro de otros Layout, para
alinear elementos

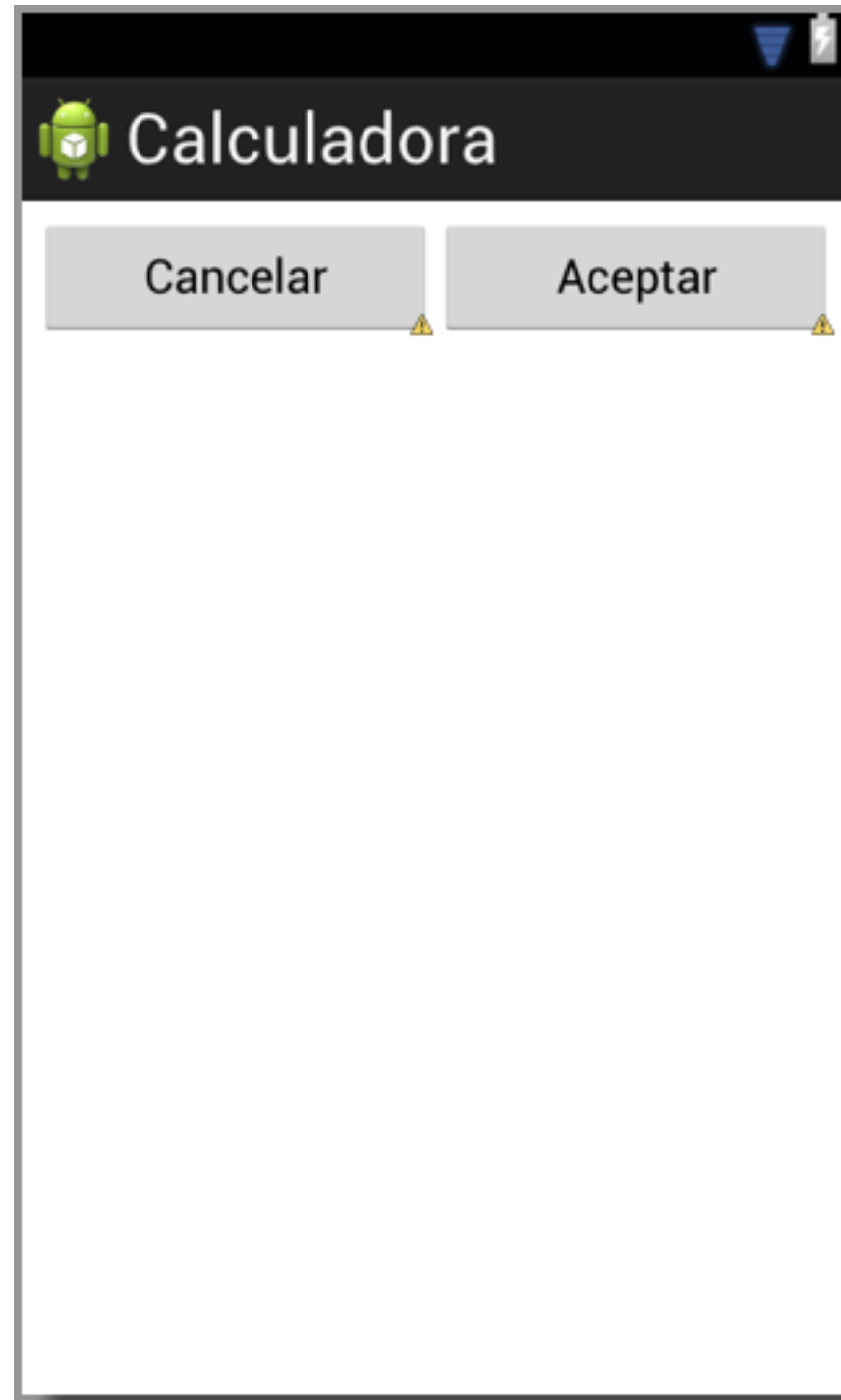
[http://developer.android.com/reference/android/widget/
LinearLayout.html](http://developer.android.com/reference/android/widget/LinearLayout.html)

[http://developer.android.com/reference/android/widget/
LinearLayout.LayoutParams.html](http://developer.android.com/reference/android/widget/LinearLayout.LayoutParams.html)

Linear Layout anidados

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <LinearLayout android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="5dp">
        <Button
            android:text="@android:string/cancel"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"/>
        <Button android:text="@android:string/ok"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"/>
    </LinearLayout>
    <ListView
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

Linear Layout anidados



Utilizando RelativeLayout

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <LinearLayout android:layout_width="fill_parent"
        android:id="@+id/button_bar"
        android:layout_alignParentBottom="true"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="5dp">
        <Button
            android:text="@android:string/cancel"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"/>
        <Button android:text="@android:string/ok"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"/>
    </LinearLayout>
    <ListView
        android:layout_above="@id/button_bar"
        android:layout_alignParentLeft="true"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</RelativeLayout>
```

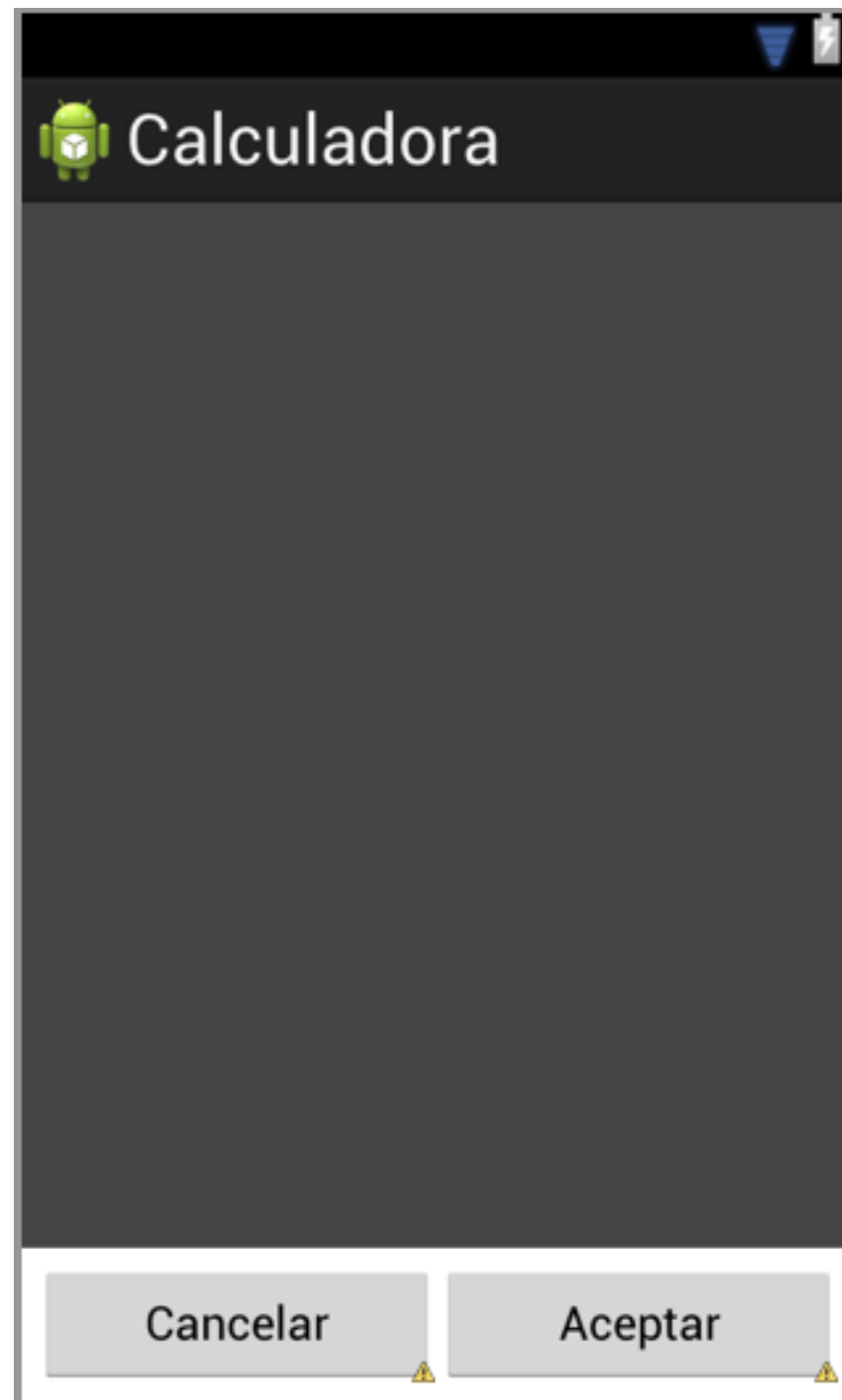
Utilizando RelativeLayout



Utilizando GridLayout

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <ListView
        android:background="#FF444444"
        android:layout_gravity="fill"/>
    <LinearLayout
        android:layout_width="fill_parent"
        android:orientation="horizontal"
        android:padding="5dp">
        <Button
            android:text="@android:string/cancel"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"/>
        <Button android:text="@android:string/ok"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"/>
    </LinearLayout>
</GridLayout>
```

Utilizando GridLayout



Utilizando GridLayout

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:columnWidth="90dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
    android:gravity="center"
/>
```

Optimizar Layouts

Mostrar Layouts en pantalla es costoso, cada uno de ellos incide negativamente en el rendimiento de la aplicación

Es importante que sean lo más simple posible, y que evitemos recargar layouts completos en cambios en la UI

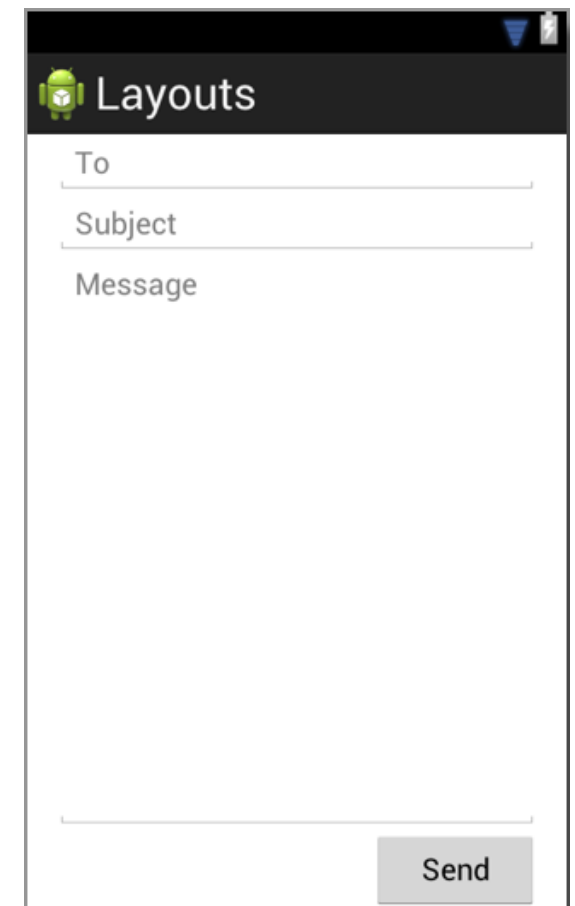
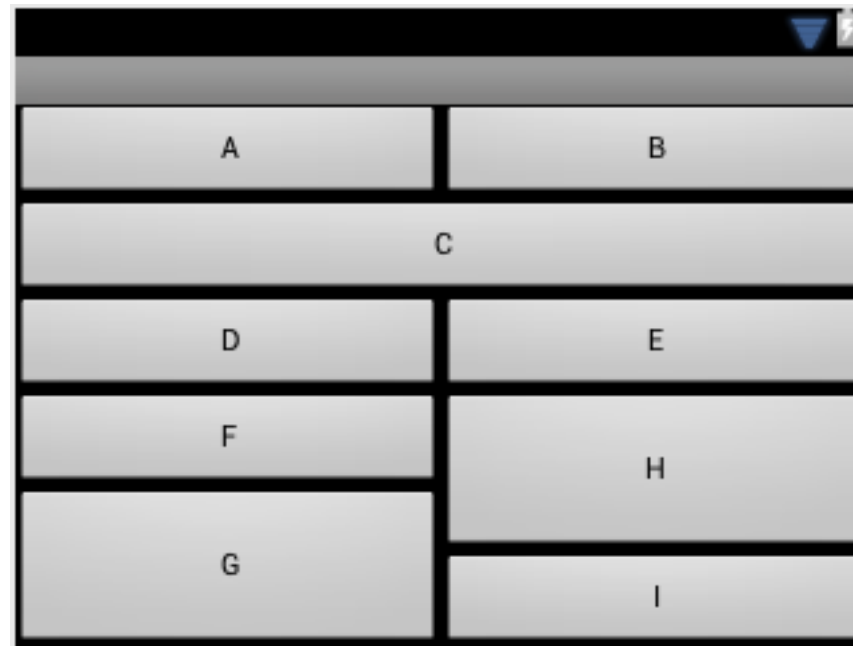
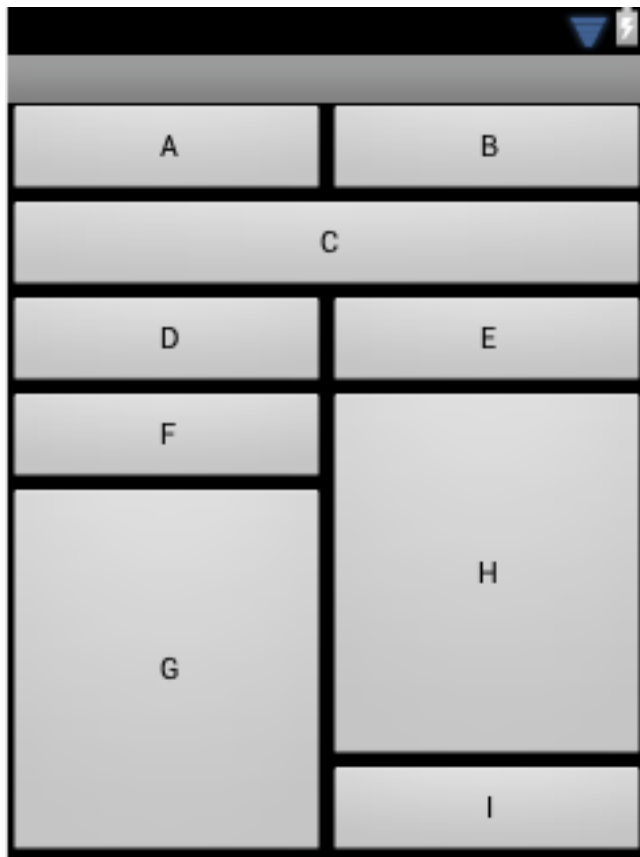
Los Layout pueden anidarse sin restricciones, por lo que es sencillo crear estructuras complejas

A pesar de no existir límites, es recomendable no sobrepasar los 10 niveles de profundidad :S

Ejercicio

Ejercicio

Crear las siguientes interfaces utilizando LinearLayout y RelativeLayout



Interfaz de usuario

Adapters

Adapters: introducción

Los adaptadores son utilizados para relacionar datos con vistas que implementan la interfaz AdapterView (como las listas o galerías)

Android provee de una serie de adaptadores, suficientes para la gran mayoría de los casos

Los adaptadores son los responsables de gestionar los datos y de crear las vistas para mostrar estos datos

Esto hace que podamos modificar la vista y el comportamiento por defecto de los controles

Adapters: introducción

Dos de las clases más utilizadas en Android:

ArrayAdapter: este adaptador utiliza genéricos para relacionar los datos de una determinada clase con los controles. Por defecto, este adaptador utiliza el método **toString** del objeto para mostrar los datos.

SimpleCursorAdapter: permite relacionar las **Views** de un **Layout** con las columnas específicas definidas en el cursor. Se crea un **Layout** por cada entrada del cursor, y se completa cada **View** con el valor correspondiente de la columna.

Ejercicio: TO-DO List

Interfaz de usuario

Fragmentos

Introduciendo Fragments

Los fragmentos permiten dividir una actividad en componentes encapsulados y reutilizables, cada uno de ellos con su propio ciclo de vida y UI

La principal ventaja es la facilidad de crear interfaces dinámicas y flexibles que se adaptan todo tipo de pantallas

Cada fragmento es independiente, pero va asociado a la actividad donde es insertada

Pueden utilizarse en diferentes actividades, combinándolos de la manera necesaria para presentar la UI

A pesar de no ser necesario, es recomendable dividir la actividad en fragmentos

Fragment: creación

```
package com.paad.fragments;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class MySkeletonFragment extends Fragment {

    // Called once the Fragment has been created in order for it to
    // create its user interface.
    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {
        // Create, or inflate the Fragment's UI, and return it.
        // If this Fragment has no UI then return null.
        return inflater.inflate(R.layout.my_fragment, container, false);
    }
}
```

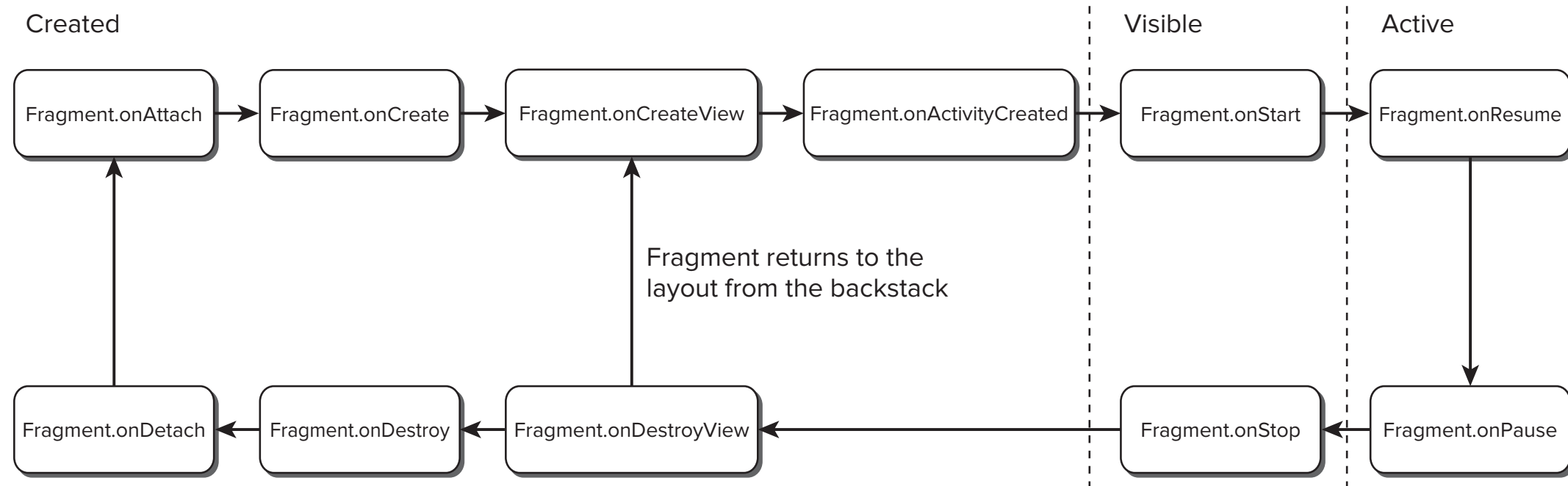


No es necesario registrar el fragmento en el manifiesto

Fragment: ciclo de vida

El ciclo de vida de un fragmento es muy similar al de las actividades

Incluye eventos propios de añadir y eliminar fragmentos de una actividad



Fragment: ciclo de vida

```
// Called when the Fragment is attached to its parent Activity.  
@Override  
public void onAttach(Activity activity) {  
    super.onAttach(activity);  
    // Get a reference to the parent Activity.  
}
```

```
// Called to do the initial creation of the Fragment.  
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    // Initialize the Fragment.  
}
```

Fragment: ciclo de vida

```
// Called once the Fragment has been created in order for it to
// create its user interface.
@Override
public View onCreateView(LayoutInflater inflater,
                        ViewGroup container,
                        Bundle savedInstanceState) {
    // Create, or inflate the Fragment's UI, and return it.
    // If this Fragment has no UI then return null.
    return inflater.inflate(R.layout.my_fragment, container, false);
}
```

```
// Called once the parent Activity and the Fragment's UI have
// been created.
@Override
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    // Complete the Fragment initialization - particularly anything
    // that requires the parent Activity to be initialized or the
    // Fragment's view to be fully inflated.
}
```


Fragment: ciclo de vida

```
// Called at the start of the visible lifetime.  
@Override  
public void onStart(){  
    super.onStart();  
    // Apply any required UI change now that the Fragment is visible.  
}
```

```
// Called at the start of the active lifetime.  
@Override  
public void onResume(){  
    super.onResume();  
    // Resume any paused UI updates, threads, or processes required  
    // by the Fragment but suspended when it became inactive.  
}
```

Fragment: ciclo de vida

```
// Called at the end of the active lifetime.
@Override
public void onPause(){
    // Suspend UI updates, threads, or CPU intensive processes
    // that don't need to be updated when the Activity isn't
    // the active foreground activity.
    // Persist all edits or state changes
    // as after this call the process is likely to be killed.
    super.onPause();
}

// Called to save UI state changes at the
// end of the active lifecycle.
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save UI state changes to the savedInstanceState.
    // This bundle will be passed to onCreate, onCreateView, and
    // onCreateView if the parent Activity is killed and restarted.
    super.onSaveInstanceState(savedInstanceState);
}
```

Fragment: ciclo de vida

```
// Called at the end of the visible lifetime.  
@Override  
public void onStop(){  
    // Suspend remaining UI updates, threads, or processing  
    // that aren't required when the Fragment isn't visible.  
    super.onStop();  
}
```

```
// Called when the Fragment's View has been detached.  
@Override  
public void onDestroyView() {  
    // Clean up resources related to the View.  
    super.onDestroyView();  
}
```

Fragment: ciclo de vida

```
// Called at the end of the full lifetime.
```

```
@Override
```

```
public void onDestroy(){
```

```
    // Clean up any resources including ending threads,
```

```
    // closing database connections etc.
```

```
    super.onDestroy();
```

```
}
```

```
// Called when the Fragment has been detached from its parent Activity.
```

```
@Override
```

```
public void onDetach() {
```

```
    super.onDetach();
```

```
}
```

Fragment: eventos específicos

- onAttach:** es lanzado antes de la propia inicialización del fragmento y de la actividad. Tareas previas a la propia inicialización.
- onDetach:** llamado cuando el fragmento se elimina de la actividad.
- onCreate:** al igual que en las actividades, es utilizado para iniciar variables. En este caso, la vista **no** se asocia aquí.
- onCreateView:** es aquí donde la vista es cargada e iniciada con los datos correspondientes.
- onActivityCreated:** si es necesario interactuar con la actividad y su UI, este es el método adecuado.

Fragment: estados

Como los fragmentos “están incluidos” en las actividades, éstos comparten los mismos estados que sus contenedores

Cuando una actividad está activa, los fragmentos que contiene también lo están, al igual que si está en pausa o parada

Si una actividad se termina, sus fragmentos también lo hacen

Una de las ventajas de utilizar fragmentos, es precisamente su ciclo de vida independiente

Nos permite añadir y eliminar fragmentos en tiempo de ejecución, sin afectar al resto de la actividad

Esto añade un grado más de control, ya que los fragmentos deben gestionar, al igual que las actividades, su estado

Añadir fragmentos a la actividad

La manera más simple es añadirlos en el Layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.paad.weatherstation.MyListFragment"
        android:id="@+id/my_list_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
    />
    <fragment android:name="com.paad.weatherstation.DetailsFragment"
        android:id="@+id/details_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="3"
    />
</LinearLayout>
```

Válido para vistas estáticas, pero queremos algo más...

Añadir fragmentos a la actividad

Es mejor definir contenedores, donde posteriormente incluir los fragmentos

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <FrameLayout
        android:id="@+id/ui_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
    />
    <FrameLayout
        android:id="@+id/details_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="3"
    />
</LinearLayout>
```


Gestionar fragmentos

Todas las actividades incluyen un gestor de fragmentos, el cual nos permite acceder, añadir y eliminar fragmentos

```
FragmentManager fragmentManager = getFragmentManager();
```

La manipulación de fragmentos debe realizarse dentro de una transacción

Estas transacciones pueden incluir tantas modificaciones como se desee

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
```

```
// Add, remove, and/or replace Fragments.  
// Specify animations.  
// Add to back stack if required.
```

```
fragmentTransaction.commit();
```

Gestionar fragmentos

Añadir fragmentos

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();  
fragmentTransaction.add(R.id.ui_container, new MyListFragment());  
fragmentTransaction.commit();
```

Eliminar fragmentos

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();  
Fragment fragment = fragmentManager.findFragmentById(R.id.details_fragment);  
fragmentTransaction.remove(fragment);  
fragmentTransaction.commit();
```

Reemplazar fragmentos

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();  
fragmentTransaction.replace(R.id.details_fragment,  
                           new DetailFragment(selected_index));  
fragmentTransaction.commit();
```

Fragment: back stack

Vimos que las actividades disponían de una pila que permitía “navegar” entre ellas

En algunos casos, los fragmentos pueden dar la impresión de cambiar a una nueva pantalla

En tal caso, el usuario puede pulsar el botón de volver, pensando en volver a la pantalla anterior

Para ello, es necesario guardar una pila de fragmentos, que nos permita simular esta funcionalidad

Fragment: back stack

```
FragmentTransaction fragmentTransaction =  
fragmentManager.beginTransaction();  
  
fragmentTransaction.add(R.id.ui_container, new MyListFragment());  
  
Fragment fragment =  
fragmentManager.findFragmentById(R.id.details_fragment);  
fragmentTransaction.remove(fragment);  
  
String tag = null;  
fragmentTransaction.addToBackStack(tag);  
  
fragmentTransaction.commit();
```


Comunicación entre fragmentos

Todos los fragmentos implementan el método `getActivity()`, el cual devuelve una referencia a la actividad que las contiene

Es posible acceder al resto de fragmentos a través del `FragmentManager`, pero es mejor utilizar la actividad como intermediario

Esto nos garantiza una independencia y bajo acoplamiento entre fragmentos

Es responsabilidad de la actividad el decidir como actuar

Lo recomendable es crear una interfaz que la actividad deba implementar

```
public class SeasonFragment extends Fragment {

    public interface OnSeasonSelectedListener {
        public void onSeasonSelected(Season season);
    }

    private OnSeasonSelectedListener onSeasonSelectedListener;
    private Season currentSeason;

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);

        try {
            onSeasonSelectedListener = (OnSeasonSelectedListener)activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString() +
                " must implement OnSeasonSelectedListener");
        }
    }

    private void setSeason(Season season) {
        currentSeason = season;
        onSeasonSelectedListener.onSeasonSelected(season);
    }

}
```

Fragment: clases de Android

Android incluye algunas subclases de Fragment que implementan algunas de sus funcionalidades más comunes:

DialogFragment: un fragmento para mostrar ventanas de diálogo sobre la actividad contenedora.

ListFragment: implementa la funcionalidad de lista, pudiendo asociarse con datos de entrada y eventos.

WebViewFragment: encapsula una vista web dentro de un fragmento.

Ejercicio:
Convertir TO-DO List a
fragmentos

Interfaz de usuario

Android Widget Toolbox

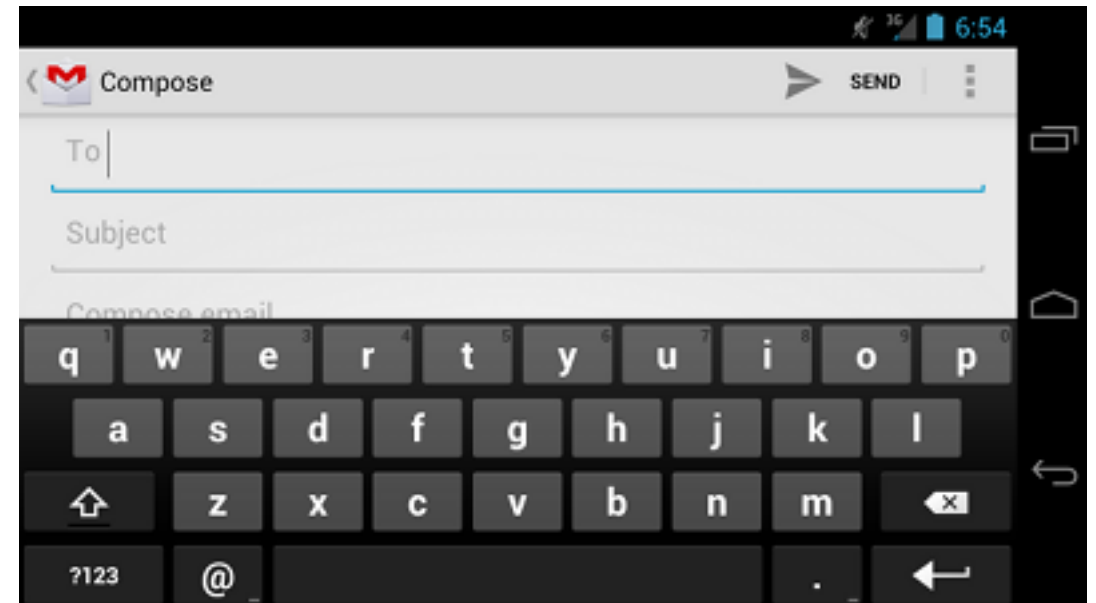
Android Widget Toolbox

TextView: etiquetas de solo lectura.

EditText: Caja de texto editable.

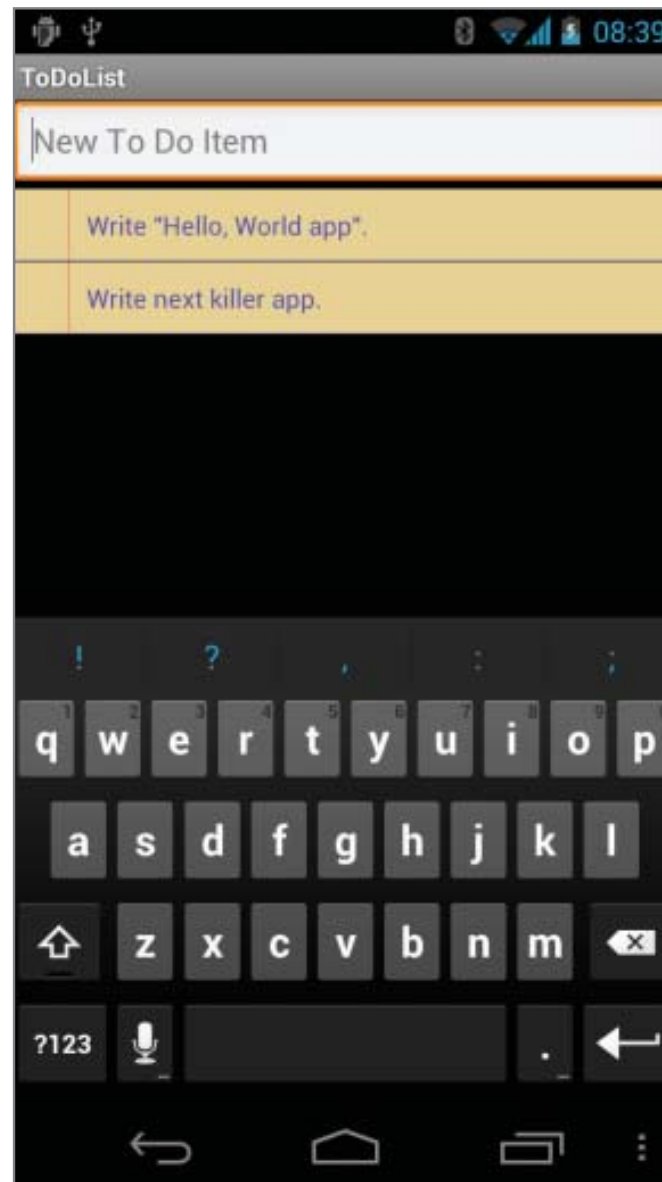
```
<EditText
    android:id="@+id/email_address"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/email_hint"
    android:inputType="textEmailAddress" />
```

```
<EditText
    android:id="@+id/postal_address"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/postal_address_hint"
    android:inputType="textPostalAddress|
        textCapWords|
        textNoSuggestions"
    android:imeOptions="actionSend" />
```



Android Widget Toolbox

ListView: vista que crea y gestiona listas verticales, mostrando en cada fila el valor del objeto asociado



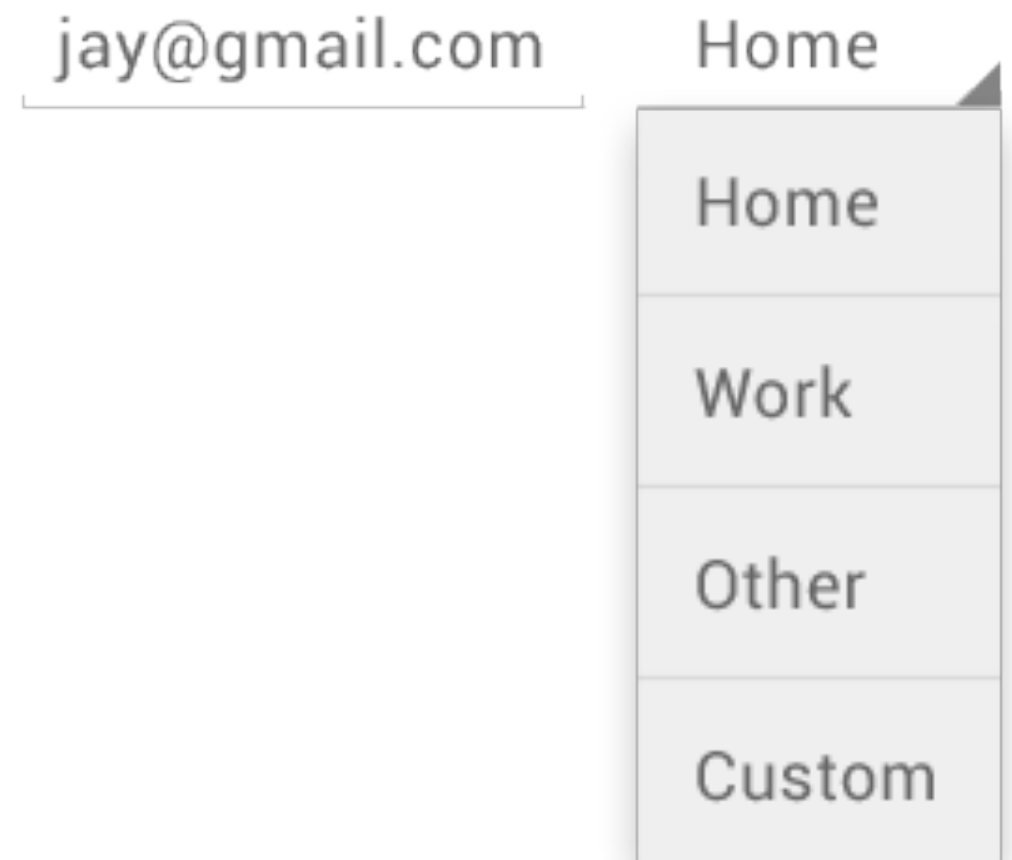
Android Widget Toolbox

Spinner: caja de texto son una lista de valores asociada.
Similar a los combo box o select de HTML.

```
<Spinner
    android:id="@+id/planets_spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="planets_array">
        <item>Mercury</item>
        <item>Venus</item>
        <item>...</item>
    </string-array>
</resources>
```

```
Spinner spinner = (Spinner) findViewById(R.id.spinner);
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
    R.array.planets_array, android.R.layout.simple_spinner_item);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
spinner.setAdapter(adapter);
```



Android Widget Toolbox

Button: clásico botón de acción.

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:drawableLeft="@drawable/button_icon"
    ... />
```

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage" />
```

```
/** Called when the user touches the button */
public void sendMessage(View view) {
    // Do something in response to button click
}
```



Android Widget Toolbox

ToggleButton: botón de dos estados (interruptor).

```
<ToggleButton
    android:id="@+id/togglebutton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Vibrate on"
    android:textOff="Vibrate off"
    android:onClick="onToggleClicked"/>
```



```
public void onToggleClicked(View view) {
    // Is the toggle on?
    boolean on = ((ToggleButton) view).isChecked();

    if (on) {
        // Enable vibrate
    } else {
        // Disable vibrate
    }
}
```

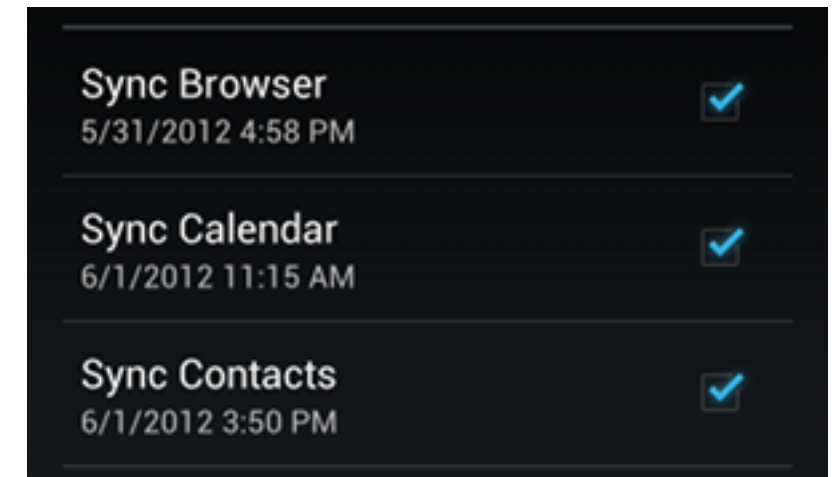
Android Widget Toolbox

CheckBox: control de dos estados.

```
<CheckBox android:id="@+id/checkbox_meat"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/meat"
    android:onClick="onCheckboxClicked"/>
```

```
public void onCheckboxClicked(View view) {
    // Is the view now checked?
    boolean checked = ((CheckBox) view).isChecked();

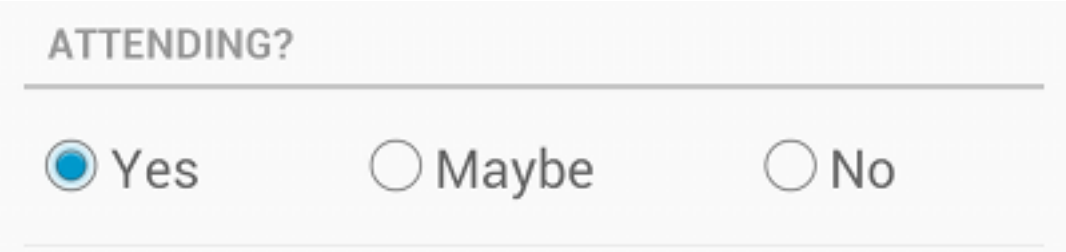
    // Check which checkbox was clicked
    switch(view.getId()) {
        case R.id.checkbox_meat:
            if (checked)
                // Put some meat on the sandwich
            else
                // Remove the meat
            break;
        // TODO: Veggie sandwich
    }
}
```



Android Widget Toolbox

RadioButton: controles agrupados de dos estados.

```
<RadioGroup xmlns:android="http://  
schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="vertical">  
    <RadioButton android:id="@+id/radio_pirates"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="@string/pirates"  
        android:onClick="onRadioButtonClicked"/>  
    <RadioButton android:id="@+id/radio_ninjas"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="@string/ninjas"  
        android:onClick="onRadioButtonClicked"/>  
</RadioGroup>
```



ATTENDING?

☒ Yes ☐ Maybe ☐ No

Android Widget Toolbox

ViewFlipper: grupo de vistas en horizontal, donde sólo una vista es visible, y es posible navegar entre ellas.



Android Widget Toolbox

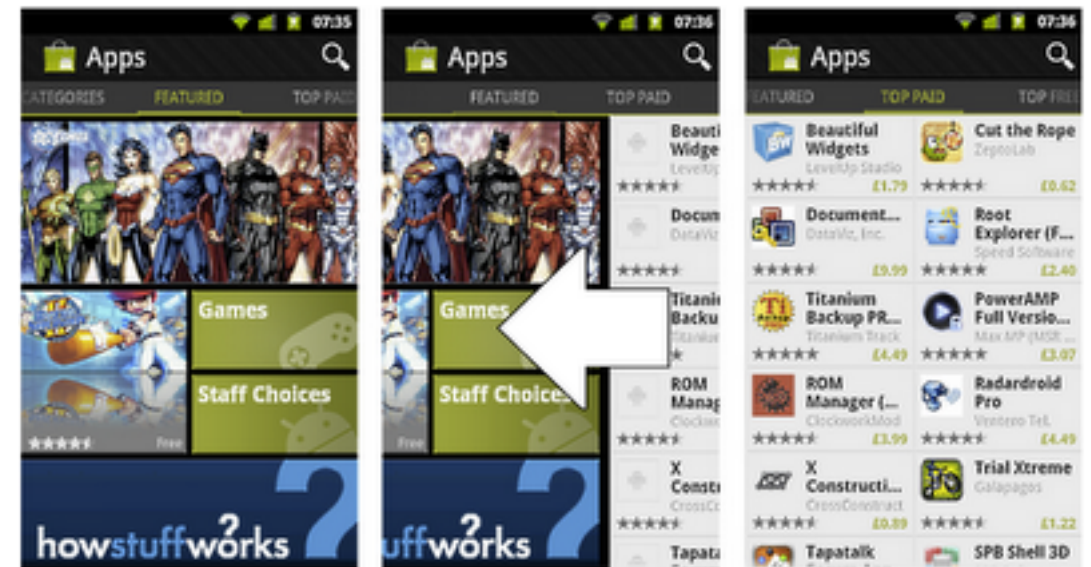
ViewFlipper: grupo de vistas en horizontal, donde sólo una vista es visible, y es posible navegar entre ellas.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <ViewFlipper android:id="@+id/viewFlipper"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
        <include layout="@layout/page_one" />
        <include layout="@layout/page_two" />
    </ViewFlipper>
</RadioGroup>
```

Android Widget Toolbox

ViewPager: similar a ViewFlipper, pero con un mayor control.

```
<android.support.v4.view.ViewPager  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"/>
```



@Override

```
public Object instantiateItem(ViewGroup collection, int position) {  
    View v = inflater.inflate(...);  
    ...  
    ((ViewPager) collection).addView(v,0);  
    return v;  
}
```

@Override

```
public void destroyItem(ViewGroup collection, int position, Object view) {  
    ((ViewPager) collection).removeView((TextView) view);  
}
```

Android Widget Toolbox

QuickContactBadge: muestra un símbolo con la imagen del contacto. Al pulsar en él, muestra una barra con accesos directos a diversas acciones.

```
<QuickContactBadge android:id="@+id/badge_small"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

```
public void onCreate(Bundle savedInstanceState) {
    QuickContactBadge badge =
        (QuickContactBadge) findViewById(R.id.badge_small);

    badge
        .assignContactFromEmail("any@gmail.com", true);
    badge
        .setMode(ContactsContract.QuickContact.MODE_SMALL);
}
```



Crear nuevos controles

Tarde o temprano será necesario crear nuevos controles, o modificar los existentes

Podemos **extender** los controles existentes, modificando cómo se dibujan en pantalla

Podemos **combinar** controles, interconectando la funcionalidad entre ellos

Podemos crear un **nuevo** control desde cero

Extender controles

```
public class MyTextView extends TextView {  
  
    public MyTextView (Context context) {  
        super(context);  
    }  
  
    public MyTextView (Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
  
    public MyTextView (Context context, AttributeSet attrs, int defStyle)  
    {  
        super(context, attrs, defStyle);  
    }  
}
```

Extender controles

```
@Override
public void onDraw(Canvas canvas) {
    [ ... Draw things on the canvas under the text ... ]
    // Render the text as usual using the TextView base class.

    super.onDraw(canvas);
    [ ... Draw things on the canvas over the text ... ]
}
```

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
    [ ... Perform some special processing ... ]
    [ ... based on a particular key press ... ]

    // Use the existing functionality implemented by
    // the base class to respond to a key press event.

    return super.onKeyDown(keyCode, keyEvent);
}
```


Combinar controles

Los controles compuestos son un ViewGroup atómico y autocontenido, cuyos componentes se interconectan

Es necesario definir la interfaz, su apariencia y el comportamiento

Combinar controles

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
    />
    <Button
        android:id="@+id/clearButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Clear"
    />
</LinearLayout>
```

```
public class ClearableEditText extends LinearLayout {

    EditText editText;
    Button clearButton;

    public ClearableEditText(Context context) {
        super(context);

        // Inflate the view from the layout resource.
        String infService = Context.LAYOUT_INFLATER_SERVICE;
        LayoutInflater li;
        li = (LayoutInflater)getContext().getSystemService(infService);
        li.inflate(R.layout.clearable_edit_text, this, true);

        // Get references to the child controls.
        editText = (EditText)findViewById(R.id.editText);
        clearButton = (Button)findViewById(R.id.clearButton);

        // Hook up the functionality
        hookupButton();
    }

    private void hookupButton() {
        clearButton.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                editText.setText("");
            }
        });
    }
}
```

Ejercicio:

TO-DO List personalizado

Demo

A fatal error 0E has occurred at 0028:C0011E36 in VXD VMM(01) + 00010E36. The current application will be terminated.

- * Press any key to terminate the current application.
- * Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all your applications.

Press any key to continue _