Interfaz de usuario

Índice

En esta sesión vamos a introducir el diseño y programación de interfaces de usuario básicas para Android. La API de Android proporciona el acceso a una serie de componentes de alto nivel que nos ahorran tener que programarlos desde cero. Por otro lado su uso nos permitirá dar a nuestras aplicaciones el mismo aspecto que el resto de aplicaciones del sistema.

1. Views

Todos los componentes visuales de la interfaz de usuario, tales como botones, campos de texto, selectores, etc, se denominan `Views` en Android. Los views se pueden agrupar en `ViewGroups` que sirven para reutilizar componentes que siempre vayan a utilizarse juntos.

Los views se distribuyen sobre `Layouts`. Hay distintos tipos de layout, según la distribución de componentes que queramos tener en la pantalla. El layout que más se utiliza es el `LinearLayout` que puede disponer los componentes uno después del otro, o bien horizontalmente, o bien verticalmente. Para hacer combinaciones se pueden incluir layouts más pequeños dentro de otros.

Cualquier view o layout puede ocupar, o bien el tamaño completo que su contenedor le permita: `fill_parent`, o bien el tamaño mínimo que necesite para dar cabida a los componentes y contenidos que haya en él: `wrap_content`. Estos dos valores pueden ser aplicados tanto en horizontal como en vertical, mediante los atributos `layout_width` y `layout_height`.

Aunque cualquier interfaz gráfico se podría crear programáticamente, sin hacer uso de ningún recurso XML, lo normal es diseñar nuestros layouts en formato XML y con la ayuda del diseñador de interfaces disponible en el plugin de Android para Eclipse. Así, podemos introducir un componente `TextView` en un layout llamado `main.xml` y mostrarlo en nuestra actividad principal:

```
public class Interfaces extends Activity {
    /** Called when the activity is first created. */

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ((TextView)findViewByID(R.id.TextView01)).setText("Hola
Android");
        }
}
```

donde el XML de `main.xml` sería:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
```

```
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="wrap_content">
<TextView
        android:text="Hola Android"
        android:id="@+id/TextView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

O bien podemos prescindir de recursos XML y añadir los views desde el código:

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        TextView miTextView = new TextView(this);
        setContentView(miTextView);
        miTextView.setText("Hola Android");
    }
```

Por supuesto, es preferible trabajar con los recursos XML cuando sea posible, ya que facilitan la mantenibilidad del programa, así como su diseño, que viene apoyado por la herramienta diseñadora del plugin para Eclipse.

Algunos views estándar que Android proporciona son los siguientes:

- `TextView`, etiqueta de texto.
- `EditText`, campo de texto.
- `Button`, botón pulsable con etiqueta de texto.
- `ListView`, grupo de views que los visualiza en forma de lista vertical.
- `Spinner`, lista desplegable, internamente es una composición de TextView y de List View.
- `CheckBox`, casilla marcable de dos estados.
- `RadioButton`, casilla seleccionable de dos estados, donde un grupo de RadioButtons sólo permitiría seleccionar uno de ellos al mismo tiempo.
- `ViewFlipper`, un grupo de Views que nos permite seleccionar qué view visualizar en este momento.
- `ScrollView`, permite usar barras de desplazamiento. Sólo puede contener un elemento, que puede ser un Layout (con otros muchos elementos dentro).
- `DatePicker`, permite escoger una fecha.
- `TimePicker`, permite escoger una hora.
- Otros más avanzados como `MapView` (vista de Google Maps) y `WebView` (vista de navegador web), etc.

Una buena práctica de programación es extender el comportamiento de los componentes por medio de herencia. Así crearemos nuestros propios componentes personalizados. Más

información sobre Views se puede obtener en el tutorial oficial: http://developer.android.com/resources/tutorials/views/index.html.

1.1. Algunas clases útiles

En la API para interfaces gráficos hay otras clases útiles para la interacción con el usuario. Veamos algunas de ellas.

1.1.1. Toast

Los `Toast` sirven para mostrar al usuario algún tipo de información de la manera menos intrusiva posible, sin robar el foco a la actividad y sin pedir ningún tipo de interacción, desapareciendo automáticamente. El tiempo que permanecerá en pantalla puede ser, o bien `Toast.LENGTH_SHORT`, o bien `Toast.LENGTH_LONG`. Se crean y muestran así:

```
Toast.makeText(MiActividad.this,
               "Preferencia de validación actualizada",
               Toast.LENGTH_SHORT).show();
```

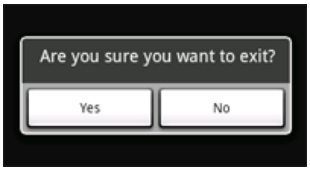
No hay que abusar de ellos pues, si se acumulan varios, irán apareciendo uno después de otro, esperando a que acabe el anterior y quizás a destiempo. Son útiles para confirmar algún tipo de información al usuario, que le dará seguridad de que está haciendo lo correcto. No son útiles para mostrar información importante, ni información extensa. Por último, si se van a utilizar como mensajes de Debug, aunque son útiles es mucho mejor utilizar la instrucción `LOG.d("TAG","Mensaje a mostrar")` y seguir el LogCat en el nivel de debugging.

1.1.2. AlertDialog

Los `AlertDialog` son útiles para pedir confirmaciones, o bien formular preguntas que requieran pulsar "aceptar" o "cancelar". A continuación se muestra un ejemplo de la documentación oficial de Android:

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Are you sure you want to exit?")
       .setCancelable(false)
       .setPositiveButton("Yes", new
DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
                MyActivity.this.finish();
           }
       })
       .setNegativeButton("No", new
DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
                dialog.cancel();
```

```
            }
        });
AlertDialog alert = builder.create();
```

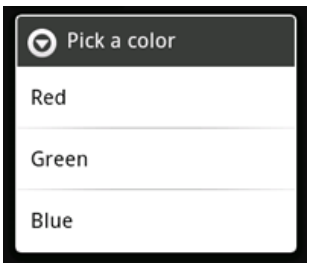


Alert Dialog (de developer.android.com)

Si queremos una lista de la que seleccionar, podemos conseguirlo de la siguiente manera:

```
final CharSequence[] items = {"Red", "Green", "Blue"};

AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Pick a color");
builder.setItems(items, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int item) {
        Toast.makeText(getApplicationContext(), items[item],
Toast.LENGTH_SHORT).show();
    }
});
AlertDialog alert = builder.create();
```



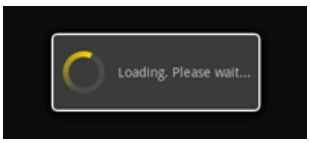
AlertDialog con lista (de developer.android.com)

1.1.3. ProgressDialog

Los `ProgressDialog` sirven para indicar progreso. Por ejemplo,

```
ProgressDialog dialog = ProgressDialog.show(MyActivity.this, "",
                "Loading. Please wait...", true);
```

genera un diálogo con indicador de progreso indefinido:

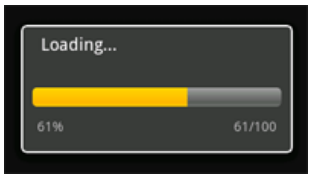


Diálogo de progreso indefinido (de developer.android.com)

mientras que

```
ProgressDialog progressDialog;
progressDialog = new ProgressDialog(mContext);
progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
progressDialog.setMessage("Loading...");
progressDialog.setCancelable(false);
```

genera un diálogo con una barra de progreso horizontal:



Diálogo con barra de progreso (de developer.android.com)

En este caso, para indicar el progreso se utiliza el método `progressDialog.setProgress(int)` indicando el porcentaje total de progreso, y lo típico sería que esta llamada se hiciera desde otro hilo con un `Handler`, o bien desde el método `AsyncTask.onProgressUpdate(String)`.

1.1.4. InputFilter

Cuando se introduce texto en un `EditText`, el contenido permitido se puede limitar y/o corregir usando un `InputFilter` o una colección de ellos. Hay dos `InputFilter` ya creados, uno es para obligar a que todo sean mayúsculas y el otro es para limitar la longitud del campo. Además se pueden crear filtros personalizados. En el siguiente ejemplo se asignan tres filtros (uno de cada tipo) a un campo de texto. Los filtros se aplican por el orden en el que estén en el vector.

```
EditText editText = (EditText)findViewById(R.id.EditText01);
InputFilter[] filters = new InputFilter[3];
filters[0] = new InputFilter.LengthFilter(9);
filters[1] = new InputFilter.AllCaps();
filters[2] = new InputFilter() {
    public CharSequence filter(CharSequence source, int start, int
end,
            Spanned dest, int dstart, int dend) {
    if (end > start) {
        String destTxt = dest.toString();
        String resultingTxt = destTxt.substring(0, dstart) +
                source.subSequence(start, end) +
destTxt.substring(dend);
        if (!resultingTxt.matches("^[A-F0-9]*$")) {
         if (source instanceof Spanned) {
             SpannableString sp = new SpannableString("");
             return sp;
         } else {
             return "";
```

```
            }
        }
    }
    return null;
    }
};
dniEditText.setFilters(filters);
```

El último filtro comprueba que se cumpla la expresión regular `^[A-F0-9]*$` (caracteres de número hexadecimal).

2. Layouts

Los Layouts son una extensión de la clase `ViewGroup` y se utilizan para posicionar controles (Views) en la interfaz de usuario. Se pueden anidar unos dentro de otros.

Los layout se pueden definir en formato XML en la carpeta `res/layout`. Por ejemplo, el siguiente layout lineal dispondrá sus elementos (TextView y Button) uno debajo del otro:

```
<LinearLayout
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:gravity="right">

        <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />

        <Button android:text="Siguiente"
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />

</LinearLayout>
```

Para disponerlos uno al lado del otro se utiliza `orientation="horizontal"`. Por otro lado, el atributo `gravity` indica hacia qué lado se van a alinear los componentes.

Algunos de los layouts más utilizados son:

- `LinearLayout`, dispone los elementos uno después del otro.
- `FrameLayout`, dispone cualquier elemento en la esquina superior izquierda.
- `RelativeLayout`, dispone los elementos en posiciones relativas con respecto a otros, y con respecto a las fronteras del layout.
- `TableLayout`, dispone los elementos en forma de filas y columnas.
- `Gallery`, dispone los elementos en una única fila desplazable.

3. Eventos

Para que los views sean usables, hay que asignar manejadores a los eventos que nos interesen. Por ejemplo, para un `Button` podemos asociar un comportamiento asignándole un `onClickListener`:

```
        ImageButton imageButton =
(ImageButton)findViewById(R.id.ImageButton01);
        imageButton.setOnClickListener(new OnClickListener() {

                        @Override
                        public void onClick(View v) {
Toast.makeText(getApplicationContext(),
                                        "Gracias por pulsar.",
Toast.LENGTH_SHORT).show();

                        }
                });
```

Se pueden escuchar los eventos de cualquier otro tipo de view, incluso de los `TextView`.

4. Activities e Intents

Ya estamos familiarizados con las actividades de android: se trata de tareas que muestran un interfaz gráfico al usuario, y sólo podemos ver en pantalla una Activity a la vez. Muchas aplicaciones tienen una actividad principal que puede llevarnos a otras actividades de la aplicación, o incluso a actividades de otras aplicaciones.

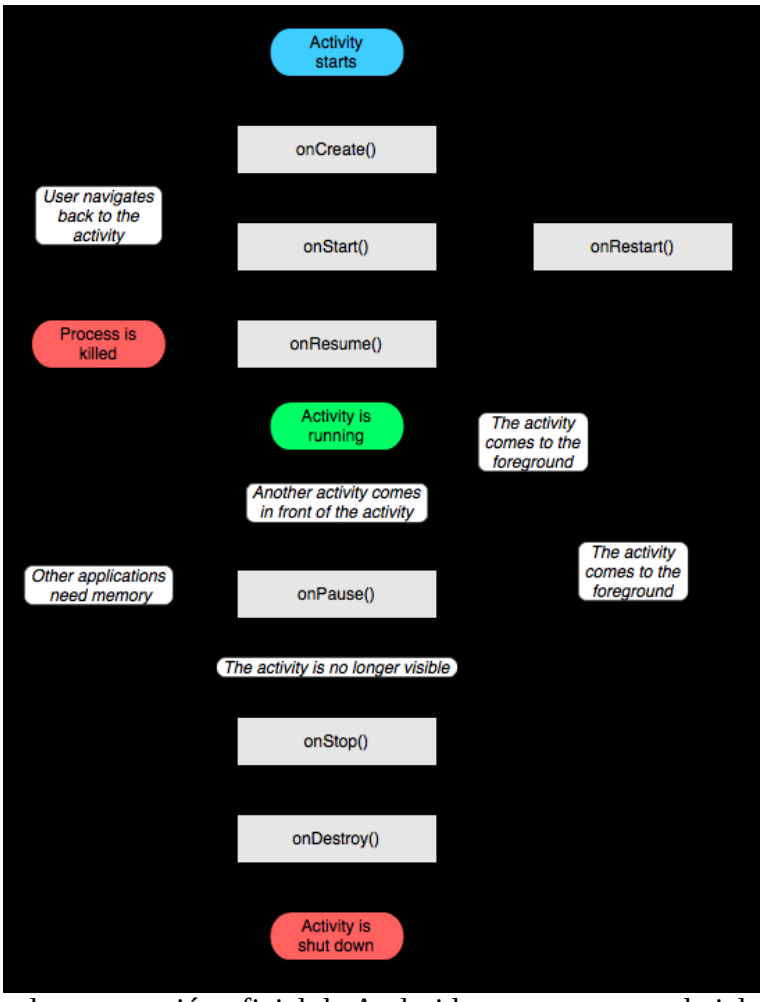


Diagrama de la documentación oficial de Android, que representa el ciclo de vida de las actividades.

Este ciclo de vida puede definirse por medio de los siguientes métodos:

```
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle savedInstanceState);

    protected void onStart();

    protected void onRestart();

    protected void onResume();

    protected void onPause();

    protected void onStop();

    protected void onDestroy();
}
```

Hay tres subciclos típicos en la vida de una actividad:

- Todo el ciclo de vida ocurre desde la primera llamada a `onCreate(Bundle)` hasta la llamada (única) al método `onDestroy()`. Por tanto en esta última se deben liberar los recursos que queden por liberar.
- El tiempo de vida visible ocurre entre `onStart()` y `onStop()`. Durante este tiempo el usuario verá la actividad en pantalla, incluso aunque ésta no tenga el foco en este momento o esté en segundo plano (pero visible). Los métodos `onStart()` y `onStop()` podrán ser llamados múltiples veces, según la actividad se haga visible o se oculte.
- El tiempo de vida en primer plano ocurre entre los métodos `onResume` y `onPause`. La actividad puede pasar con frecuencia entre el estado pausado y primer plano, de manera que el código de estos métodos debe ser rápido.

Nota:

Cuando una actividad se pausa, ésta puede no volver nunca a primer plano sino ser matada debido a que el sistema operativo lo decida así, por falta de recursos de memoria. Por tanto tendremos que intentar guardar los estados de nuestras actividades de tal manera que si la actividad se retoma con `onResume()`, el usuario tenga la misma experiencia que si arranca la aplicación de nuevo. Para ello nos ayuda el parámetro de `onCreate(Bundle)` que guarda el estado de los formularios y los rellena de nuevo "automágicamente", sin tener que programarlo nosotros.

Para pasar de una actividad a otra se utilizan los `Intent`. Un Intent es una descripción abstracta de una operación a realizar. Se puede utilizar con el método `startActivity` para lanzar una actividad, con `broadcastIntent` para enviarse a cualquier componente receptor `BroadcastReceiver`, y con `startService` o `bindService` para comunicar con un servicio (`Service`) que corre en segundo plano.

Por ejemplo, para lanzar la actividad llamada `MiActividad`, lo haremos así:

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

Para iniciar una llamada de teléfono también utilizaremos intents:

```
Intent intent = new Intent(Intent.ACTION_DIAL,
Uri.parse("tel:965903400"));
startActivity(intent);
```

Podemos pasar un código de petición a la actividad:

```
startActivityForResult(intent, CODIGO_INT);
```

Y podemos esperar un resultado de la actividad, generándolo así

```
Intent resultado = new Intent(null, Uri.parse("content://datos/1");
resultado.putExtra(1,"aaa");
resultado.putExtra(2,"bbb");

if(ok)
        setResult(RESULT_OK, resultado);
else
        setResult(RESULT_CANCELED, null);

finish();
```

y recogiendo el resultado con el método `onActivityResult()` sobrecargado:

```
@Override
public void onActivityResult(int reqCode, int resultCode, Intent
data){
        switch(reqCode){
        case 1:
                break;
        case 2:
                break;
        default:
        }
}
```

Algunas acciones nativas da Android son:

- ACTION_ANSWER
- ACTION_CALL
- ACTION_DELETE
- ACTION_DIAL
- ACTION_EDIT
- ACTION_INSERT
- ACTION_PICK
- ACTION_SEARCH
- ACTION_SENDTO
- ACTION_VIEW
- ACTION_WEB_SEARCH

5. Menús y preferencias

Hay varios tipos de menús en Android:

- los "icon menu" que aparecen en la parte inferior de la pantalla cuando se pulsa el botón físico de menú
- menús expandidos que aparecen cuando el usuario pulsa el botón "más" del incon menu
- los submenús que aparecen en forma de ventanas flotantes, de las que se sale con la tecla física "atrás" (o "cerrar")

- los menús contextuales que se abren al pulsar prolongadamente sobre un componente

Un icon menú se puede definir en un archivo XML, por ejemplo, en `res/menu/menu.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">
<item android:title="Preferencias" android:id="@+id/item01"></item>
<item android:title="Acerca de..." android:id="@+id/item02"></item>
</menu>
```

Para que el menú se infle al pulsar el botón debemos sobrecargar la función `onCreateOptionsMenu(Menu m)` de nuestra actividad. Esta función deberá desplegar el menú con la función `getMenuInflater().inflate(R.menu.menu, menu);` y devolver `true`:

```
 @Override
public boolean onCreateOptionsMenu(Menu m) {
        getMenuInflater().inflate(R.menu.menu, m);
        return true;
}
```

Para programar las respuestas a las pulsaciones de cada opción del menú tenemos que sobrecargar el método `onOptionsItemSelected()`:

```
 @Override
public boolean onOptionsItemSelected(MenuItem item) {
        switch(item.getItemId()){
        case R.id.item01:
                break;
        case R.id.item02:
                break;
        }
        return true;
}
```

En cuanto a los menús contextuales, para crearlos habrá que sobrecargar la función `onCreateContextMenu(ContextMenu)` del componente correspondiente:

```
@Override
public void onCreateContextMenu(ContextMenu m){
        super.onCreateContextMenu(m);
        m.add("ContextMenuItem1");
}
```

Otra alternativa para los menús contextuales es registrar el view con el método `registerForContextMenu(view)`, para que así el menú contextual para este view sólo

esté disponible en esta actividad, y no en cualquier actividad en la que se incluya el view.