Interfaz de usuario en Android: Controles de selección (III)

En el artículo anterior ya vimos cómo utilizar los controles de tipo `ListView` en Android. Sin embargo, acabamos comentando que existía una forma más eficiente de hacer uso de dicho control, de forma que la respuesta de nuestra aplicación fuera más ágil y se redujese el consumo de batería, algo que en plataformas móviles siempre es importante.

Como base para este artículo vamos a utilizar como código que ya escribimos en el artículo anterior, por lo que si has llegado hasta aquí directamente te recomiendo que leas primero el primer post dedicado al control ListView.

Cuando comentamos cómo crear nuestro propio adaptador, extendiendo de `ArrayAdapter`, para personalizar la forma en que nuestros datos se iban a mostrar en la lista escribimos el siguiente código:

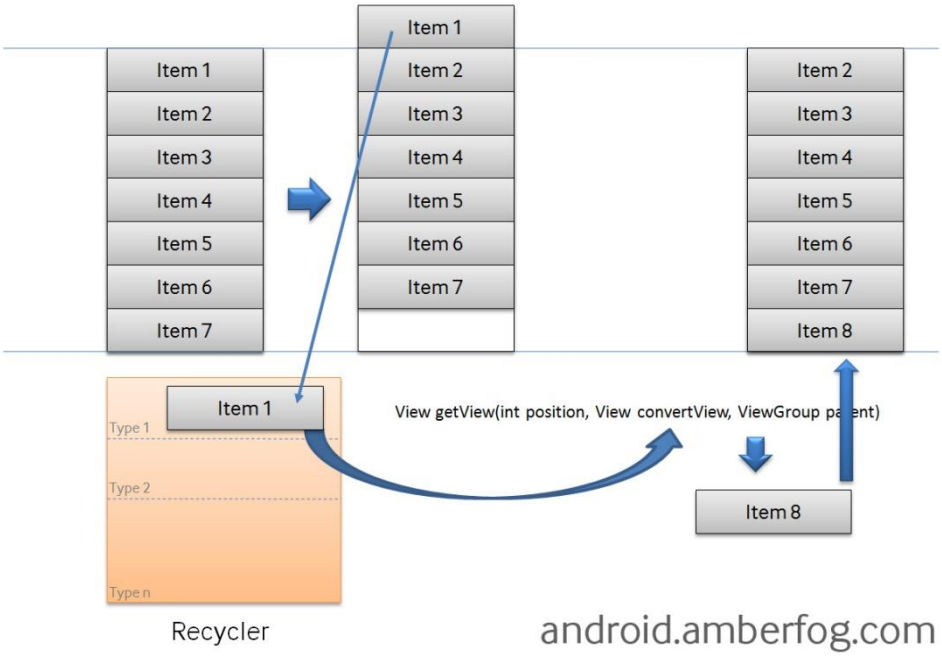
```
class AdaptadorTitulares extends ArrayAdapter<Titular> {


    public AdaptadorTitulares(Context context, Titular[] datitos) {
        super(context, R.layout.listitem_titular, datitos);
        }

    public View getView(int position, View convertView, ViewGroup
parent) {
        LayoutInflater inflater = LayoutInflater.from(getContext());
        View itemView = inflater.inflate(R.layout.listitem_titular,
null);

        TextView lblTitulo =
(TextView)itemView.findViewById(R.id.LblTitulo);
        lblTitulo.setText(getItem(position).getTitulo());

        TextView lblSubtitulo =
(TextView)itemView.findViewById(R.id.LblSubTitulo);
        lblSubtitulo.setText(getItem(position).getSubtitulo());

        return(itemView);
    }
}
```

Centrándonos en la definición del método `getView()` vimos que la forma normal de proceder consistía en primer lugar en "inflar" nuestro layout XML personalizado para crear todos los objetos correspondientes (con la estructura descrita en el XML) y posteriormente acceder a dichos objetos para modificar sus propiedades. Sin embargo, hay que tener en cuenta que esto se hace todas y cada una de las veces que se necesita mostrar un elemento de la lista en pantalla, se haya mostrado ya o no con anterioridad, ya que Android no "guarda" los elementos de la lista que desaparecen de pantalla (por ejemplo al hacer scroll sobre la lista).

El efecto de esto es obvio, dependiendo del tamaño de la lista y sobre todo de la complejidad del layout que hayamos definido esto puede suponer la creación y destrucción de cantidades ingentes de objetos (que puede que ni siquiera nos sean necesarios), es decir, que la acción de inflar un layout XML puede ser bastante costosa, lo que podría aumentar mucho, y sin necesidad, el uso de CPU, de memoria, y de batería.

Para aliviar este problema, Android nos propone un método que permite reutilizar algún layout que ya hayamos inflado con anterioridad y que ya no nos haga falta por algún motivo, por ejemplo porque el elemento correspondiente de la lista ha desaparecido de la pantalla al hacer scroll. De esta forma evitamos todo el trabajo de crear y estructurar todos los objetos asociados al layout, por lo que tan sólo nos quedaría obtener la referencia a ellos mediante `findViewById()` y modificar sus propiedades.

¿Pero cómo podemos reutilizar estos layouts "obsoletos"? Pues es bien sencillo, siempre que exista algún layout que pueda ser reutilizado éste se va a recibir a través del parámetro `convertView` del método `getView()`.



De esta forma, en los casos en que éste no sea `null` podremos obviar el trabajo de inflar el layout. Veamos cómo quedaría el métod `getView()` tras esta optimización:

```
public View getView(int position, View convertView, ViewGroup parent)

{


View itemView = convertView;
if(itemView == null)
{
        LayoutInflater inflater = LayoutInflater.from(getContext());
        View itemView = inflater.inflate(R.layout.listitem_titular,
null);
}

        TextView lblTitulo =
(TextView)itemView.findViewById(R.id.LblTitulo);
        lblTitulo.setText(getItem(position).getTitulo());

        TextView lblSubtitulo =
(TextView)itemView.findViewById(R.id.LblSubTitulo);
        lblSubtitulo.setText(getItem(position).getSubtitulo());

        return(itemView);
    }
```

Si ejecutamos ahora la aplicación podemos comprobar que al hacer scroll sobre la lista todo sigue funcionando con normalidad, con la diferencia de que le estamos ahorrando gran cantidad de trabajo a la CPU.

Pero vamos a ir un poco más allá. Con la optimización que acabamos de implementar conseguimos ahorrarnos el trabajo de inflar el layout definido cada vez que se muestra un nuevo elemento. Pero aún hay otras dos llamadas relativamente costosas que se siguen ejecutando en todas las llamadas. Me refiero a la obtención de la referencia a cada uno de los objetos a modificar mediante el método `findViewById()`. La búsqueda por ID de un control determinado dentro del árbol de objetos de un layout también puede ser una tarea costosa dependiendo de la complejidad del propio layout. ¿Por qué no aprovechamos que estamos "guardando" un layout anterior para guardar también la referencia a los controles que lo forman de forma que no tengamos que volver a buscarlos? Pues eso es exactamente lo que vamos a hacer mediante lo que suelen llamar patrón `ViewHolder`.
Nuestra clase `ViewHolder` tan sólo va a contener una referencia a cada uno de los controles que tengamos que manipular de nuestro layout, en nuestro caso las dos etiquetas de texto. Definamos por tanto esta clase de la siguiente forma:

```
static class ViewHolder {
    TextView titulo;
    TextView subtitulo;
}
```

La idea será por tanto crear e inicializar el objeto `ViewHolder` la primera vez que inflemos un elemento de la lista y asociarlo a dicho elemento de forma que posteriormente podamos recuperarlo fácilmente. ¿Pero dónde lo guardamos? Fácil, en Android todos los controles tienen una propiedad llamada `Tag`(podemos asignarla y recuperarla mediante los métodos `setTag()` y `getTag()` respectivamente) que puede contener cualquier tipo de objeto, por lo que resulta ideal para guardar nuestro objeto `ViewHolder`. De esta forma, cuando el

parámetro `convertView` llegue informado sabremos que también tendremos disponibles las referencias a sus controles hijos a través de la propiedad `Tag`. Veamos el código modificado de `getView()` para aprovechar esta nueva optimización:

```
public View getView(int position, View convertView, ViewGroup parent)
{

        View itemView = convertView;
        ViewHolder holder;

        if(itemView == null)
        {
            LayoutInflater inflater =
LayoutInflater.from(getContext());
            itemView = inflater.inflate(R.layout.listitem_titular,
null);

            holder = new ViewHolder();
            holder.titulo =
(TextView)itemView.findViewById(R.id.LblTitulo);
            holder.subtitulo =
(TextView)itemView.findViewById(R.id.LblSubTitulo);

            itemView.setTag(holder);
        }
        else
        {
            holder = (ViewHolder)itemView.getTag();
        }

        holder.titulo.setText(getItem(position).getTitulo());
        holder.subtitulo.setText(getItem(position).getSubtitulo());

        return(itemView);
    }
}
```

Con estas dos optimizaciones hemos conseguido que la aplicación sea mucho más respetuosa con los recursos del dispositivo de nuestros usuarios, algo que sin duda nos agradecerán.

Puedes consultar y/o descargar el código completo de los ejemplos desarrollados en este artículo accediendo a la página del curso en GitHub.