Notificaciones en Android

En Android existen varias formas de notificar mensajes al usuario, como por ejemplo los cuadros de diálogo modales o las notificaciones de la bandeja del sistema (o barra de estado).

TOAST

Un *toast* es un mensaje que se muestra en pantalla durante unos segundos al usuario para luego volver a desaparecer automáticamente sin requerir ningún tipo de actuación por su parte, y sin recibir el foco en ningún momento (o dicho de otra forma, sin interferir en las acciones que esté realizando el usuario en ese momento). Aunque son personalizables, aparecen por defecto en la parte inferior de la pantalla, sobre un rectángulo gris ligeramente translúcido. Por sus propias características, este tipo de notificaciones son ideales para mostrar mensajes rápidos y sencillos al usuario, pero por el contrario, al no requerir confirmación por su parte, no deberían utilizarse para hacer notificaciones demasiado importantes.

Su utilización es muy sencilla, concentrándose toda la funcionalidad en la clase `Toast`. Esta clase dispone de un método estático `makeText()` al que deberemos pasar como parámetro el contexto de la actividad, el texto a mostrar, y la duración del mensaje, que puede tomar los valores `LENGTH_LONG` o `LENGTH_SHORT`, dependiendo del tiempo que queramos que la notificación aparezca en pantalla. Tras obtener una referencia al objeto `Toast` a través de este método, ya sólo nos quedaría mostrarlo en pantalla mediante el método `show()`.

Vamos a construir una aplicación de ejemplo para demostrar el funcionamiento de este tipo de notificaciones. Y para empezar vamos

a incluir un botón que muestre un toast básico de la forma que acabamos de describir:

```
1
2       btnDefecto.setOnClickListener(new OnClickListener() {
3           @Override
4           public void onClick(View arg0) {
5               Toast toast1 =
6                   Toast.makeText(getApplicationContext(),
7                           "Toast por defecto", Toast.LENGTH_SHORT);
8
9               toast1.show();
10          }
        });
```

Si ejecutamos esta sencilla aplicación en el emulador y pulsamos el botón que acabamos de añadir veremos como en la parte inferior de la pantalla aparece el mensaje “Toast por defecto”, que tras varios segundos desaparecerá automáticamente.

Como hemos comentado, éste es el comportamiento por defecto de las notificaciones toast, sin embargo también podemos personalizarlo un poco cambiando su posición en la pantalla. Para esto utilizaremos el método `setGravity()`, al que podremos indicar en qué zona deseamos que aparezca la notificación. La zona deberemos indicarla

con alguna de las constantes definidas en la clase `Gravity`: `CENTER`, `LEFT`, `BOTTOM`, … o con alguna combinación de éstas.

Para nuestro ejemplo vamos a colocar la notificación en la zona central izquierda de la pantalla. Para ello, añadamos un segundo botón a la aplicación de ejemplo que muestre un toast con estas características:

```
1
2       btnGravity.setOnClickListener(new OnClickListener() {
3           @Override
4           public void onClick(View arg0) {
5               Toast toast2 =
6                   Toast.makeText(getApplicationContext(),
7                           "Toast con gravity", Toast.LENGTH_SHORT);
8
9               toast2.setGravity(Gravity.CENTER|Gravity.LEFT,0,0);
10
11              toast2.show();
12          }
      });
```

Si volvemos a ejecutar la aplicación y pulsamos el nuevo botón veremos como el toast aparece en la zona indicada de la pantalla:

Si esto no es suficiente y necesitamos personalizar por completo el aspecto de la notificación, Android nos ofrece la posibilidad de definir

un layout XML propio para toast, donde podremos incluir todos los elementos necesarios para adaptar la notificación a nuestras necesidades. para nuestro ejemplo vamos a definir un layout sencillo, con una imagen y una etiqueta de texto sobre un rectángulo gris:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/lytLayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="horizontal"
  android:background="#555555"
  android:padding="5dip" >

    <ImageView android:id="@+id/imgIcono"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/marcador" />

    <TextView android:id="@+id/txtMensaje"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:textColor="#FFFFFF"
        android:paddingLeft="10dip" />

</LinearLayout>
```

Guardaremos este layout con el nombre "*toast_layout.xml*", y como siempre lo colocaremos en la carpeta "*res\layout*" de nuestro proyecto.

Para asignar este layout a nuestro toast tendremos que actuar de una forma algo diferente a las anteriores. En primer lugar deberemos *inflar* el layout mediante un objeto `LayoutInflater`, como ya vimos en varias ocasiones al tratar los artículos de interfaz gráfica. Una vez construido el layout modificaremos los valores de los distintos controles para mostrar la información que queramos. En nuestro caso, tan sólo modificaremos el mensaje de la etiqueta de texto, ya que la imagen ya la asignamos de forma estática en el layout XML mediante el atributo `android:src`. Tras esto, sólo nos quedará establecer la

duración de la notificación con `setDuration()` y asignar el layout personalizado al toast mediante el método `setView()`. Veamos cómo quedaría todo el código incluido en un tercer botón de ejemplo:

```
1
2       btnLayout.setOnClickListener(new OnClickListener() {
3           @Override
4           public void onClick(View arg0) {
5               Toast toast3 = new Toast(getApplicationContext());
6
7               LayoutInflater inflater = getLayoutInflater();
8               View layout = inflater.inflate(R.layout.toast_layout,
9                                 (ViewGroup) findViewById(R.id.lytLayout));
10
11              TextView txtMsg = (TextView)layout.findViewById(R.id.txtMensaje);
12              txtMsg.setText("Toast Personalizado");
13
14              toast3.setDuration(Toast.LENGTH_SHORT);
15              toast3.setView(layout);
16              toast3.show();
17          }
      });
```

Si ejecutamos ahora la aplicación de ejemplo y pulsamos el nuevo botón, veremos como nuestro toast aparece con la estructura definida en nuestro layout personalizado.

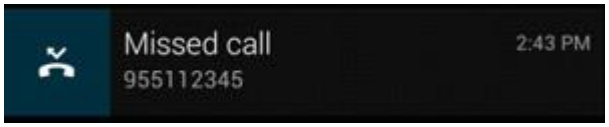
Como podéis comprobar, mostrar notificaciones de tipo Toast en nuestras aplicaciones Android es algo de lo más sencillo, y a veces

resulta un elemento de lo más interesante para avisar al usuario de determinados eventos.

Barra de Estado

vamos a tratar otro tipo de notificaciones algo más persistentes, las notificaciones de la barra de estado de Android. Estas notificaciones son las que se muestran en nuestro dispositivo por ejemplo cuando recibimos un mensaje SMS, cuando tenemos actualizaciones disponibles, cuando tenemos el reproductor de música abierto en segundo plano, … Estas notificaciones constan de un icono y un texto mostrado en la barra de estado superior, y adicionalmente un mensaje algo más descriptivo y una marca de fecha/hora que podemos consultar desplegando la bandeja del sistema. A modo de ejemplo, cuando tenemos una llamada perdida en nuestro terminal, se nos muestra por un lado un icono en la barra de estado superior (más un texto que aparece durante unos segundos).

… y un mensaje con más información al desplegar la bandeja del sistema, donde en este caso concreto se nos informa del evento que se ha producido (“*Missed call*“), el número de teléfono asociado, y la fecha/hora del evento. Además, al pulsar sobre la notificación se nos dirige automáticamente al historial de llamadas.



Pues bien, aprendamos a utilizar este tipo de notificaciones en nuestras aplicaciones. Vamos a construir para ello una aplicación de ejemplo, como siempre lo más sencilla posible para centrar la atención

en lo realmente importante. En este caso, el ejemplo va a consistir en un único botón que genere una notificación de ejemplo en la barra de estado, con todos los elementos comentados y con la posibilidad de dirigirnos a la propia aplicación de ejemplo cuando se pulse sobre ella.

Para generar notificaciones en la barra de estado del sistema vamos a utilizar una clase incluida en la librería de compatibilidad *android-support-v7.jar* que ya hemos utilizado en otras ocasiones y que debe estar incluida por defecto en vuestro proyecto. La clase en cuestión se llama `NotificationCompat.Builder` y lo que tendremos que hacer será crear un nuevo objeto de este tipo pasándole el contexto de la aplicación y asignar todas las propiedades que queramos mediante sus métodos *set()*.

En primer lugar estableceremos los iconos a mostrar mediante los métodos `setSmallIcon()` que se corresponde al icono mostrados a a la izquierda del contenido de la notificación.

A continuación estableceremos el título y el texto de la notificación, utilizando para ello los métodos `setContentTitle()` y `setContentText()`.

Por último, estableceremos opcionalmente el *ticker* (texto que será leído en móviles que tengan servicios de accesibilidad habilitados) mediante `setTicker()` y el texto auxiliar que aparecerá a la derecha, debajo de la hora mediante `setContentInfo()`.

La fecha/hora asociada a nuestra notificación se tomará automáticamente de la fecha/hora actual si no se establece nada, o bien puede utilizarse el método `setWhen()` para indicar otra marca de tiempo. Veamos cómo quedaría nuestro código por el momento:

```
NotificationCompat.Builder notificationBuilder =
(NotificationCompat.Builder) new
NotificationCompat.Builder(MainActivity.this)
        .setSmallIcon(R.mipmap.ic_launcher)
        .setContentTitle("Simple Notification")
        .setContentText("This is a normal notification.")
        .setContentInfo("Cuatro")
        .setTicker("Alerta!");
```

El segundo paso será establecer la actividad a la cual debemos dirigir al usuario automáticamente si éste pulsa sobre la notificación. Para ello debemos construir un objeto `PendingIntent`, que será el que contenga la información de la actividad asociada a la notificación y que será lanzado al pulsar sobre ella. Para ello definiremos en primer lugar un objeto `Intent`, indicando la clase de la actividad concreta a lanzar, que en nuestro caso será la propia actividad principal de ejemplo (`MainActivity.class`). Este *intent* lo utilizaremos para construir el `PendingIntent` final mediante el método `PendingIntent.getActivity()`. Por último asociaremos este objeto a la notificación mediante el método `setContentIntent()` del `Builder`. Veamos cómo quedaría esta última parte comentada:

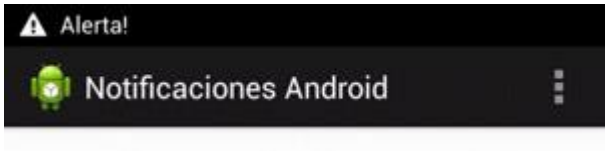
```
Intent i= new Intent(MainActivity.this,MainActivity.class);
PendingIntent contIntent =
PendingIntent.getActivity(MainActivity.this,0,i,0);
notificationBuilder.setContentIntent(contIntent);
```

Por último, una vez tenemos completamente configuradas las opciones de nuestra notificación podemos generarla llamando al método `notify()` del *Notification Manager*,al cual podemos acceder mediante una llamada a `getSystemService()` con la constante `Context.NOTIFICATION_SERVICE`. Por su parte al método `notify()` le pasaremos como parámetro un identificador único definido por nosotros que identifique nuestra notificación y el resultado del builder que hemos construido antes, que obtenemos llamando a su método `build()`.

```
NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
notificationManager.notify(NOTIFICATION_ID++,
notificationBuilder.build());
```

Ya estamos en disposición de probar nuestra aplicación de ejemplo. Pulsamos el botón que hemos implementado con todo el código anterior. Si todo va bien, debería aparecer en ese mismo momento nuestra notificación en la barra de estado, con el icono y texto definidos.

Nota: si usamos una versión de Android anterior a la 5, el mensaje de ticker se mostrará en la barra de notificaciones como muestra la siguiente imagen:



Dialogos

En principio, los diálogos de Alerta de Android los podremos utilizar con distintos fines, en general:

- Mostrar un mensaje.
- Pedir una confirmación rápida.
- Solicitar al usuario una elección (simple o múltiple) entre varias alternativas.
- Solicitar datos al usuario información que requiere más espacio del que queremos darle en la pantalla principal(login, fechas, horas)

Para generar Dialogos vamos a utilizar una clase incluida en la librería de compatibilidad *android-support-v7.jar* que ya hemos utilizado en otras ocasiones y que debe estar incluida por defecto en vuestro proyecto. En este caso nos vamos a basar en la clase `DialogFragment`. Para crear un diálogo lo primero que haremos será crear una nueva clase que herede de `DialogFragment` y sobrescribiremos uno de sus métodos `onCreateDialog()`, que será el encargado de construir el diálogo con las opciones que necesitemos.

La forma de construir cada diálogo dependerá de la información y funcionalidad que necesitemos. A continuación mostraré algunas de las formas más habituales.

Diálogo de Alerta

Este tipo de diálogo se limita a mostrar un mensaje sencillo al usuario, y un único botón de OK para confirmar su lectura. Lo construiremos mediante la clase `AlertDialog`, y más concretamente su subclase

`AlertDialog.Builder`, de forma similar a las notificaciones de barra de estado que ya hemos comentado en el capítulo anterior. Su utilización es muy sencilla, bastará con crear un objeto de tipo `AlertDialog.Builder` y establecer las propiedades del diálogo mediante sus métodos correspondientes: título [`setTitle()`], mensaje [`setMessage()`] y el texto y comportamiento del botón [`setPositiveButton()`]. Veamos un ejemplo:

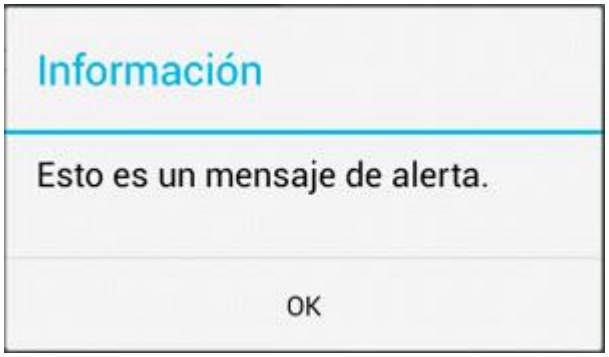
```
1
2     public class DialogoAlerta extends DialogFragment {
3         @Override
4         public Dialog onCreateDialog(Bundle savedInstanceState) {
5
6             AlertDialog.Builder builder =
7                     new AlertDialog.Builder(getActivity());
8
9             builder.setMessage("Esto es un mensaje de alerta.")
10                    .setTitle("Información")
11                    .setPositiveButton("OK", new DialogInterface.OnClickListener() {
12                        public void onClick(DialogInterface dialog, int id) {
13                            dialog.cancel();
14                        }
15                    });
16
17            return builder.create();
18        }
    }
```

Como vemos, al método `setPositiveButton()` le pasamos como argumentos el texto a mostrar en el botón, y la implementación del evento onClick en forma de objeto `OnClickListener`. Dentro de este evento, nos limitamos a cerrar el diálogo mediante su método `cancel()`, aunque podríamos realizar cualquier otra acción.

Para lanzar este diálogo por ejemplo desde nuestra actividad principal, obtendríamos una referencia al *Fragment Manager* mediante una llamada a `getSupportFragmentManager()`, creamos un nuevo objeto de tipo `DialogoAlerta`y por último mostramos el diálogo mediante el método `show()` pasándole la referencia al fragment manager y una etiqueta identificativa del diálogo.

```
    btnAlerta.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            FragmentManager fragmentManager = getFragmentManager();
               DialogoAlerta dialogo = new DialogoAlerta();
            dialogo.show(fragmentManager, "tagAlerta");
        }
    });
```

El aspecto de nuestro diálogo de alerta sería el siguiente:



Diálogo de Confirmación

Un diálogo de confirmación es muy similar al anterior, con la diferencia de que lo utilizaremos para solicitar al usuario que nos confirme una determinada acción, por lo que las posibles respuestas serán del tipo Sí/No.

La implementación de estos diálogos será prácticamente igual a la ya comentada para las alertas, salvo que en esta ocasión añadiremos dos botones, uno de ellos para la respuesta afirmativa (`setPositiveButton()`), y el segundo para la respuesta negativa (`setNegativeButton()`). Veamos un ejemplo:

```
public class DialogoMisiles extends DialogFragment{
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the Builder class for convenient dialog construction
        AlertDialog.Builder builder = new
AlertDialog.Builder(getActivity());
        builder.setMessage("¿Estás seguro de que quieres lanzar los
misiles?")
                .setPositiveButton("Lanzar", new
DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int
```

```
id) {
                        // FIRE ZE MISSILES!
                    }
                })
                .setNegativeButton("No lanzar", new
DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int
id) {
                        // User cancelled the dialog
                    }
                });
        // Create the AlertDialog object and return it
        return builder.create();

    }
}
```

En este caso, generamos a modo de ejemplo dos mensajes de log para poder verificar qué botón pulsamos en el diálogo. El aspecto visual de nuestro diálogo de confirmación sería el siguiente:

Diálogo de Selección

Cuando las opciones a seleccionar por el usuario no son sólo dos, como en los diálogos de confirmación, sino que el conjunto es mayor podemos utilizar los diálogos de selección para mostrar una lista de opciones entre las que el usuario pueda elegir.

Para ello también utilizaremos la clase `AlertDialog`, pero esta vez no asignaremos ningún mensaje ni definiremos las acciones a realizar por cada botón individual, sino que directamente indicaremos la lista de opciones a mostrar (mediante el método `setItems()`) y proporcionaremos la implementación del evento `onClick()` sobre dicha lista (mediante un listener de tipo `DialogInterface.OnClickListener`), evento en el que realizaremos las acciones oportunas según la opción elegida. La lista de opciones la definiremos como un array tradicional. Veamos cómo:

```
1    public class DialogoSeleccion extends DialogFragment {
2        @Override
3        public Dialog onCreateDialog(Bundle savedInstanceState) {
4
         final String[] items = {"Español", "Inglés", "Francés"};
```

```
5
6               AlertDialog.Builder builder =
7                       new AlertDialog.Builder(getActivity());
8
9               builder.setTitle("Selección")
10                  .setItems(items, new DialogInterface.OnClickListener() {
11                      public void onClick(DialogInterface dialog, int item) {
12                          Log.i("Dialogos", "Opción elegida: " + items[item]);
13                      }
14                  });
15
16              return builder.create();
17          }
18      }
19
```

En este caso el diálogo tendrá un aspecto similar a la interfaz mostrada para los controles`Spinner`.

Este diálogo permite al usuario elegir entre las opciones disponibles cada vez que se muestra en pantalla. Pero, ¿y si quisiéramos recordar cuál es la opción u opciones seleccionadas por el usuario para que aparezcan marcadas al visualizar de nuevo el cuadro de diálogo? Para ello podemos utilizar los métodos `setSingleChoiceItems()` o `setMultiChiceItems()`, en vez de el `setItems()` utilizado anteriormente. La diferencia entre ambos métodos, tal como se puede suponer por su nombre, es que el primero de ellos permitirá una selección simple y el segundo una selección múltiple, es decir, de varias opciones al mismo tiempo, mediante controles `CheckBox`.

La forma de utilizarlos es muy similar a la ya comentada. En el caso de `setSingleChoiceItems()`, el método tan sólo se diferencia de `setItems()` en que recibe un segundo parámetro adicional que indica el índice de la opción marcada por defecto. Si no queremos tener ninguna de ellas marcadas inicialmente pasaremos el valor -1.

```
builder.setTitle("Selección")
       .setSingleChoiceItems(items, -1,
                  new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int item) {
                    Log.i("Dialogos", "Opción elegida: " + items[item]);
                }
          });
```

De esta forma conseguiríamos un diálogo como el de la siguiente imagen:

Si por el contrario optamos por la opción de selección múltiple, la diferencia principal estará en que tendremos que implementar un listener del tipo `DialogInterface.OnMultiChoiceClickListener`. En este caso, en el evento `onClick` recibiremos tanto la opción seleccionada (`item`) como el estado en el que ha quedado (`isChecked`). Además, en esta ocasión, el segundo parámetro adicional que indica el estado por defecto de las opciones ya no será un simple número entero, sino que tendrá que ser un array de booleanos. En caso de no querer ninguna opción seleccionada por defecto pasaremos el valor `null`.

```
    builder.setTitle("Selección")
        .setMultiChoiceItems(items, null,
            new DialogInterface.OnMultiChoiceClickListener() {
            public void onClick(DialogInterface dialog, int item, boolean isChecked) {
                Log.i("Dialogos", "Opción elegida: " + items[item]);
            }
    });
```

Y el diálogo nos quedaría de la siguiente forma:

Tanto si utilizamos la opción de selección simple como la de selección múltiple, para salir del diálogo tendremos que pulsar la tecla "*Atrás*" de nuestro dispositivo o añadir un botón positivo igual que vimos en los anteriores ejemplos

Diálogos Personalizados

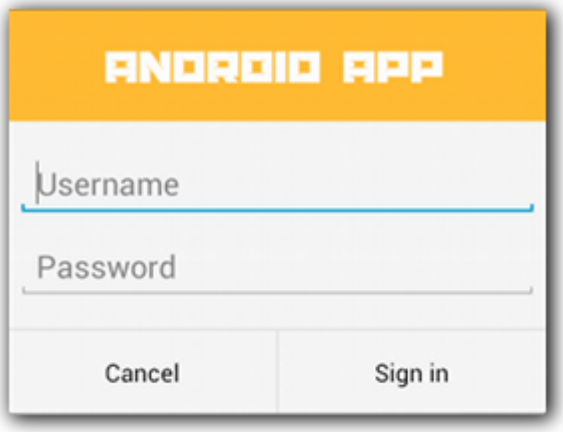
Vamos a comentar cómo podemos establecer completamente el aspecto de un cuadro de diálogo. Para esto vamos a actuar como si estuviéramos definiendo la interfaz de una actividad, es decir, definiremos un layout XML con los elementos a mostrar en el diálogo. En mi caso voy a definir un layout de ejemplo llamado `dialog_personal.xml` que colocaré como siempre en la carpeta `res/layout`. Contendrá por ejemplo una imagen a la izquierda y dos líneas de texto a la derecha:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
        android:src="@drawable/header_logo"
        android:layout_width="match_parent"
        android:layout_height="64dp"
        android:scaleType="center"
        android:background="#FFFFBB33"
        android:contentDescription="@string/app_name" />
    <EditText
        android:id="@+id/username"
        android:inputType="textEmailAddress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="4dp"
        android:layout_marginBottom="4dp"
        android:hint="@string/username" />
    <EditText
        android:id="@+id/password"
        android:inputType="textPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="4dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="4dp"
        android:layout_marginBottom="16dp"
        android:fontFamily="sans-serif"
        android:hint="@string/password"/>
</LinearLayout>
```

Por su parte, en el método `onCreateDialog()` correspondiente utilizaremos el método `setView()` del builder para asociarle nuestro layout personalizado, que previamente tendremos que inflar como ya hemos comentado otras veces utilizando el método `inflate()`. Finalmente podremos incluir botones tal como vimos para los diálogos de alerta o confirmación. En este caso de ejemplo incluiremos un botón de *logear y otro de cancelar.*

```
public class DialogoPersonalizado extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {

    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    LayoutInflater inflater = getActivity().getLayoutInflater();

    builder.setView(inflater.inflate(R.layout.dialog_personal, null))
    builder.setPositiveButton("Sign in", new DialogInterface.OnClickListener()
            public void onClick(DialogInterface dialog, int id) {
                // Loguear
            }
        })
        .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
                // Cancelar            }
        });
    });

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

De esta forma, si ejecutamos de nuevo nuestra aplicación de ejemplo y lanzamos el diálogo personalizado veremos algo como lo siguiente:



Diálogos Preconfigurados: DatePickerDialog y TimePickerDialog

Veamos cómo podemos utilizar estos diálogos ya implementados por Android. En estos casos no es necesario crear el dialogo que herede de DialogFragment ya que Android ya lo ha hecho por nosotros, simplemente deberemos instanciar un objeto del dialogo que queremos usar y mostrarlo.

A la hora de instanciar uno de estos objetos es necesario pasarle por parámetros al constructor, el contexto, el listener a ejecutar cuando el usuario haya introducido la fecha o la hora, y los valores por defecto que se mostrarán en el dialogo. A continuación se muestra un ejemplo con DatePickerDialog:

```
DatePickerDialog dpd= new DatePickerDialog(MainActivity.this, new
DatePickerDialog.OnDateSetListener() {
    @Override
    public void onDateSet(DatePicker view, int year, int monthOfYear,
int dayOfMonth) {
        Log.i("Año", String.valueOf(year));
        Log.i("mes", String.valueOf(monthOfYear));
        Log.i("dia", String.valueOf(dayOfMonth));
    }
},2016,5,18);
dpd.show();
```

Introduciendo este código en el método onClick de un botón, al pulsar el botón veremos:

SnackBars

Un nuevo tipo de notificación, que ha tomado especial relevancia sobre todo a raíz de la aparición de Android 5 Lollipop y *Material Design*, son los llamados *snackbar*. Un snackbar debe utilizarse para mostrar feedback sobre alguna operación realizada por el usuario, y es similar a un *toast* en el sentido de que aparece en pantalla por un corto periodo de tiempo y después desaparece automáticamente, aunque también presenta algunas diferencias importantes, como por ejemplo que puede contener un botón de texto de acción.

Según las especificaciones del componente dentro de Material Design, un snackbar debe mostrar mensajes cortos, debe aparecer desde la parte inferior de la pantalla (con la misma elevación que el *floating action button* si existiera, pero menos que los diálogos o el *navigation drawer*), no debe mostrarse más de uno al mismo tiempo, puede contener un botón de texto para realizar una acción, y normalmente puede descartarse deslizándolo hacia un lateral de la pantalla.

Salvo el último punto, que aclararemos más adelante, el nuevo componente `Snackbar` de la librería de diseño se encargará de asegurar todos los demás, y además mantendrá la facilidad de uso de los toast, siendo su API muy similar.

Entrando ya en detalles, lo primero que tendremos que hacer para utilizar los snackbar será añadir la referencia a la nueva librería de diseño a nuestro proyecto:

```
1    dependencies {
2        //...
3        compile 'com.android.support:design:22.2.0'
4    }
```

Hecho esto, para mostrar un snackbar procederemos de una forma muy similar a cómo lo hacíamos en el caso de los toast. Construiremos el snackbar mediante el método estático

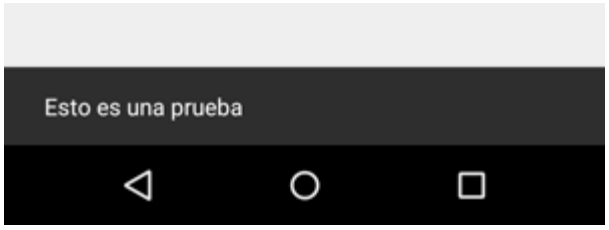
`Snackbar.make()` y posteriormente lo mostraremos llamando al método `show()`. A modo de ejemplo haremos esto desde el evento click de un botón añadido a la aplicación.

En este caso el método `make()` recibe 3 parámetros. El segundo y el tercero son equivalentes a los ya mostrados para los toast, es decir, el texto a mostrar en el snackbar y la duración del mensaje en pantalla (que en este caso podrá ser `Snackbar.LENGTH_SHORT`o `Snackbar.LENGTH_LONG`). El primero requiere más explicación. Como primer parámetro debemos pasar la referencia a una vista que permita al snackbar (navegando hacia arriba por la jerarquía de vistas de nuestro layout) descubrir un "*contenedor adecuado*" donde alojarse. Este contenedor será normalmente el *content view* o vista raíz de la actividad, o un contenedor de tipo `CoordinatorLayout` si se encuentra antes (al final del artículo veremos qué es esto último).

Por tanto, en teoría podríamos pasar en este primer parámetro casi cualquier vista de nuestra actividad (aún no he podido hacer pruebas con layouts complejos). En nuestro caso de ejemplo, aprovechando que creamos el snackbar dentro del evento `onClick` de un botón pasaremos la referencia al view del botón pulsado que recibimos en el evento:

```
1
2     btnSnackbarSimple.setOnClickListener(new View.OnClickListener() {
3         @Override
4         public void onClick(View view) {
5
6             Snackbar.make(view, "Esto es una prueba", Snackbar.LENGTH_LONG)
7                     .show();
8         }
    });
```

Esto es todo lo que hace falta para mostrar un snackbar sencillo, que aparecería en pantalla de la siguiente forma:



Vamos a completar un poco más el ejemplo mostrando también un snackbar con una acción (lanzado desde un segundo botón). Para añadir una acción al snackbar utilizaremos su método `addAction()`, al que pasaremos como parámetros el texto del botón de acción, y un listener para su evento `onClick` (análogo al de un botón normal) donde podremos responder a la pulsación del botón realizando las acciones oportunas (en mi caso una simple un simple mensaje de log):

```
1
2    Snackbar.make(view, "Esto es otra prueba", Snackbar.LENGTH_LONG)
3            .setAction("Acción", new View.OnClickListener() {
4                  @Override
5                  public void onClick(View view) {
6                        Log.i("Snackbar", "Pulsada acción snackbar!");
7                  }
8            })
             .show();
```

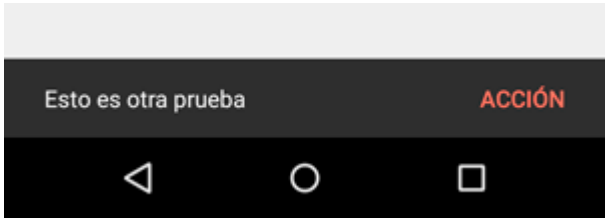
El botón de acción se mostrará por defecto con el *accent color* del tema definido para la aplicación (si se ha definido dicho color) o con el color de selección actual. Sin embargo, también podemos especificarlo en la construcción del snackbar mediante el método `setActionTextColor()`, pasándole directamente un color, o recuperando alguno de los definidos en los recursos de nuestra aplicación mediante `getResources().getColor(id_recurso_color)`:

```
1     Snackbar.make(view, "Esto es otra prueba", Snackbar.LENGTH_LONG)
2            //.setActionTextColor(Color.CYAN)
3            .setActionTextColor(getResources().getColor(R.color.snackbar_action))
4            .setAction("Acción", new View.OnClickListener() {
5                  @Override
6                  public void onClick(View view) {
7                        Log.i("Snackbar", "Pulsada acción snackbar!");
8                  }
             })
             .show();
```

En el código anterior establecemos un color que hemos definido en el fichero*/res/values/colors.xml* de la siguiente forma:

```
<resources>
    <color name="snackbar_action">#ffff7663</color>
</resources>
```

Y con esto, nuestro nuevo snackbar con acción incluida quedaría de la siguiente forma:

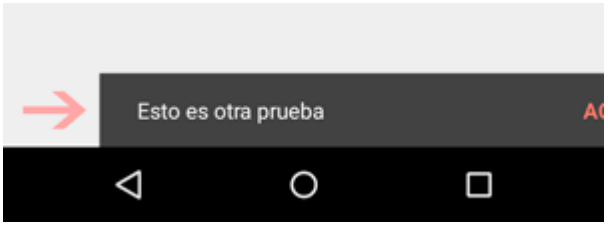


Por último, la posibilidad de descartar un snackbar con el gesto de deslizamiento hacia la derecha. Si añadimos como elemento padre de nuestro layout un `CoordinatorLayout`, nuestros snackbar podrán automáticamente descartarse deslizándolos a la derecha

```
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:orientation="vertical">

        <Button android:id="@+id/BtnSimple"
            android:text="@string/snackbar_simple"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <Button android:id="@+id/BtnAccion"
            android:text="@string/snackbar_accion"
            android:layout_width="match_parent"
```

```
                android:layout_height="wrap_content" />

        </LinearLayout>

    </android.support.design.widget.CoordinatorLayout>
```

Si volvemos a ejecutar ahora la aplicación, veremos que todo sigue funcionando de la misma forma que antes, con el añadido de que podremos descartar los mensajes deslizándolos antes de que desaparezcan.



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