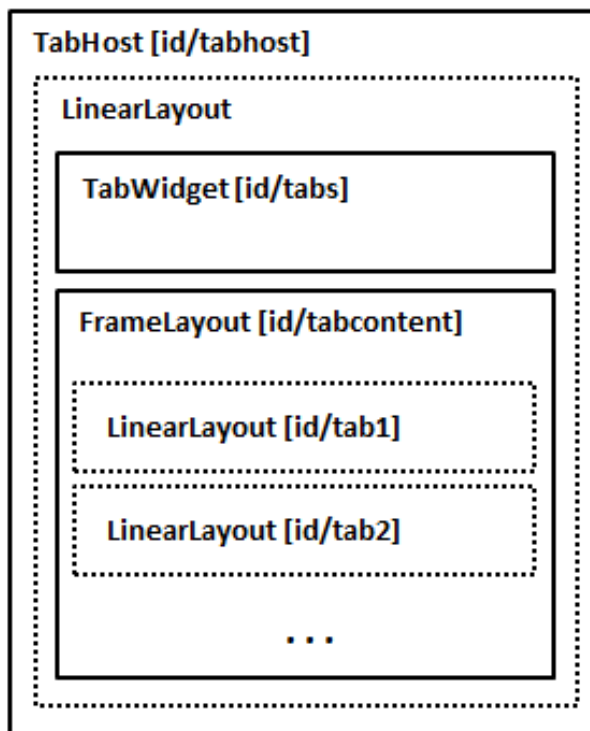


22.- Pestañas (Tabs)

Dado el poco espacio con el que contamos en las pantallas de muchos dispositivos, o simplemente por cuestiones de organización, a veces es necesario/interesante dividir nuestros controles en varias pantallas. Y una de las formas más clásicas de conseguir esto consiste en la distribución de los elementos por pestañas o *tabs*. Android por supuesto permite utilizar este tipo de interfaces, aunque lo hace de una forma un tanto peculiar, ya que la implementación no va a depender de un sólo elemento sino de varios, que además deben estar distribuidos y estructurados de una forma determinada nada arbitraria. Adicionalmente no nos bastará simplemente con definir la interfaz en XML como hemos hecho en otras ocasiones, sino que también necesitaremos completar el conjunto con algunas líneas de código. Desarrollemos esto poco a poco.

En Android, el elemento principal de un conjunto de pestañas será el control `TabHost`. Éste va a ser el contenedor principal de nuestro conjunto de pestañas y deberá tener obligatoriamente como id el valor `@android:id/tabhost`. Dentro de éste vamos a incluir un `LinearLayout` que nos servirá para distribuir verticalmente las secciones principales del layout: la sección de **pestañas en la parte superior** y la sección de **contenido en la parte inferior**. La sección de pestañas se representará mediante un elemento `TabWidget`, que deberá tener como id el valor `@android:id/tabs`, y como contenedor para el contenido de las pestañas añadiremos un `FrameLayout` con el id obligatorio `@android:id/tabcontent`. Por último, dentro del `FrameLayout` incluiremos el contenido de cada pestaña, normalmente cada uno dentro de su propio layout principal (como ejemplo utilizamos `LinearLayout`) y con un id único que nos permita posteriormente hacer referencia a ellos fácilmente (en el ejemplo se han utilizado los ids `"tab1"`, `"tab2"`, ...). A continuación vemos de forma gráfica toda la estructura descrita.



Si traducimos esta estructura a nuestro fichero de layout XML tendríamos lo siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <TabHost android:id="@android:id/tabhost"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <LinearLayout
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="match_parent" >

            <TabWidget android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:id="@android:id/tabs" />

            <FrameLayout android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:id="@android:id/tabcontent" >

                <LinearLayout android:id="@+id/tab1"
                    android:orientation="vertical"
                    android:layout_width="match_parent"
                    android:layout_height="match_parent" >

                    <TextView android:id="@+id/textView1"
                        android:text="Contenido Tab 1"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content" />
                </LinearLayout>

                <LinearLayout android:id="@+id/tab2"
                    android:orientation="vertical"
                    android:layout_width="match_parent"
                    android:layout_height="match_parent" >

                    <TextView android:id="@+id/textView2"
                        android:text="Contenido Tab 2"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content" />
                </LinearLayout>
            </FrameLayout>
        </LinearLayout>
    </TabHost>
</LinearLayout>
```

Como podemos ver, como contenido de las pestañas tan sólo hemos añadido por simplicidad una etiqueta de texto con el texto “*Contenido Tab NºTab*“. Esto nos permitirá ver que el conjunto de pestañas funciona correctamente cuando ejecutemos la aplicación.

Con esto ya tendríamos montada toda la estructura de controles necesaria para nuestra interfaz de pestañas. Sin embargo, como ya hemos dicho anteriormente, con esto no es suficiente. **Necesitamos asociar de alguna forma cada pestaña con su contenido**, de forma que el control se comporte correctamente cuando cambiamos de pestaña. Y esto tendremos que hacerlo mediante código en nuestra actividad principal.

Empezaremos obteniendo una referencia al control principal `TabHost` y preparándolo para su configuración llamando a su método `setup()`. Tras esto, crearemos un objeto de tipo `TabSpec` para cada una de las pestañas que queramos añadir mediante el método `newTabSpec()`, al que pasaremos como parámetro una etiqueta identificativa de la pestaña (en este caso de ejemplo `"mitab1"`, `"mitab2"`, ...). Además, también le asignaremos el layout de contenido correspondiente a la pestaña llamando al método `setContent()`, e indicaremos el texto y/o el icono que queremos mostrar en la pestaña mediante el método `setIndicator(texto, icono)`. Veamos el código completo.

```
Resources res = getResources();


TabHost tabs=(TabHost)findViewById(android.R.id.tabhost);
tabs.setup();

TabHost.TabSpec spec=tabs.newTabSpec("mitab1");
spec.setContent(R.id.tab1);
spec.setIndicator("TAB1",
    res.getDrawable(android.R.drawable.ic_btn_speak_now));
tabs.addTab(spec);

spec=tabs.newTabSpec("mitab2");
spec.setContent(R.id.tab2);
spec.setIndicator("TAB2",
    res.getDrawable(android.R.drawable.ic_dialog_map));
tabs.addTab(spec);

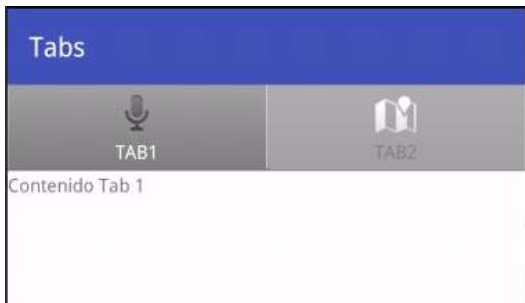
tabs.setCurrentTab(0);

tabs.setOnTabChangeListener(new TabHost.OnTabChangeListener() {
    @Override
    public void onTabChanged(String s) {
        Log.i("AndroidTabsDemo", "Pulsada pestaña: "+s);
    }
});
```

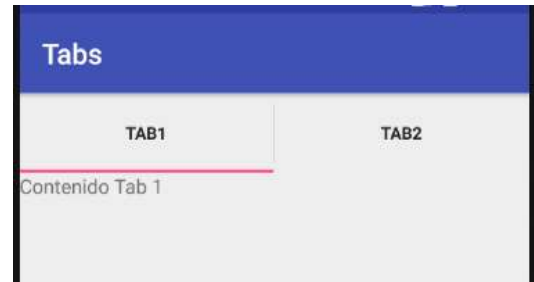
Si vemos el código, vemos por ejemplo como para la primera pestaña creamos un objeto `TabSpec` con la etiqueta `"mitab1"`, le asignamos como contenido uno de los `LinearLayout` que incluimos en la sección de contenido (en este caso `R.id.tab1`) y finalmente le asignamos el texto `"TAB1"` y el icono `android.R.drawable.ic_btn_speak_now` ( Éste es un icono incluido con la propia plataforma Android. Se puede sustituir por cualquier otro icono). Finalmente añadimos la nueva pestaña al control mediante el método `addTab()`.

Si ejecutamos ahora la aplicación tendremos algo como lo que muestra la siguiente imagen, donde podremos cambiar de pestaña y comprobar cómo se muestra correctamente el contenido de la misma.

En Android 2.x



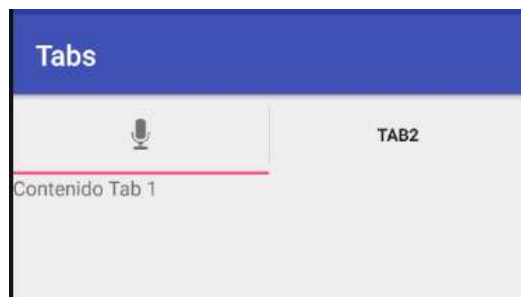
Y en Android 4.x



Una pequeña sorpresa. Como se puede comprobar, aunque estamos indicando en todos los casos un texto y un icono para cada pestaña, el comportamiento difiere entre las distintas versiones de Android. En Android 4, el comportamiento por defecto del control `TabHost` es mostrar sólo el texto, o solo el icono, pero no ambos. Si eliminamos el texto de la primera pestaña y volvemos a ejecutar veremos como el icono sí aparece.

```
TabHost.TabSpec spec=tabs.newTabSpec("mitab1");
spec.setContent(R.id.tab1);
spec.setIndicator("",
    res.getDrawable(android.R.drawable.ic_btn_speak_now));
tabs.addTab(spec);
```

Con esta pequeña modificación la aplicación se vería así en Android 4.x



En cuanto a los eventos disponibles del control `TabHost`, aunque **no suele ser necesario capturarlos**, podemos ver a modo de ejemplo el más interesante de ellos, `OnTabChanged`, que se lanza cada vez que se cambia de pestaña y que nos informa de la nueva pestaña visualizada. Este evento podemos implementarlo y asignarlo mediante el método `setOnTabChangeListener()` de la siguiente forma:

```
tabs.setOnTabChangeListener(new TabHost.OnTabChangeListener() {
    @Override
    public void onTabChanged(String tabId) {
        Log.i("AndroidTabsDemo", "Pulsada pestaña: "+tabId);
    }
});
```

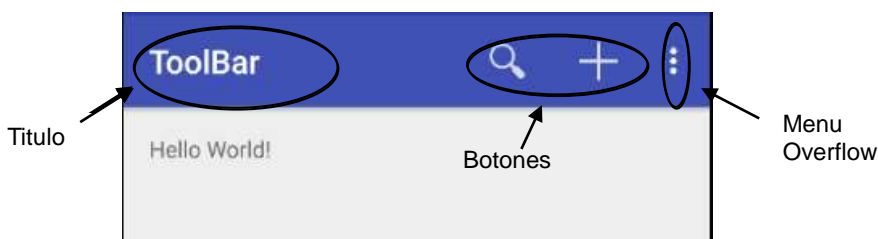
En el método `onTabChanged()` recibimos como parámetro la etiqueta identificativa de la pestaña (no su ID), que debemos asignar cuando creamos su objeto `TabSpec` correspondiente. Para este ejemplo, lo único que haremos al detectar un cambio de pestaña será escribir en el log de la

aplicación un mensaje informativo con la etiqueta de la nueva pestaña visualizada. Así por ejemplo, al cambiar a la segunda pestaña recibiremos el mensaje de log: “*Pulsada pestaña: mitab2*”.

23.-Toolbar en Android (Action bar/App bar)

La *ActionBar*, o *AppBar* como se la ha rebautizado con la llegada de Material Design y Android 5.0, es la barra de título y herramientas que aparece en la parte superior de la gran mayoría de aplicaciones actuales de la plataforma Android.

La **App bar** normalmente muestra el título de la aplicación o la actividad en la que nos encontramos, una serie de botones de acción, y un menú desplegable (menú de *overflow*) donde se incluyen más acciones que no tienen espacio para mostrarse como botón o simplemente no se quieren mostrar como tal.



Antes de la llegada de *Material Design*, a la izquierda del título también podía (y solía) aparecer el icono de la aplicación, aunque esto último ya no se recomienda en las últimas guías de diseño de la plataforma. Lo que sí puede aparecer a la izquierda del título son los iconos indicativos de la existencia de menú lateral deslizante (*navigation drawer*) o el botón de navegación hacia atrás/arriba, pero esto ya lo veremos más adelante.

En la actualidad, Android proporciona este componente a través de la librería de soporte *appcompat-v7*, que podemos incluir en nuestro proyecto añadiendo su referencia en la sección *dependencies* del fichero *build.gradle*:

```
dependencies {  
    . . .  
    implementation 'com.android.support:appcompat-v7:28.0.0'  
}
```

De cualquier forma, en versiones actuales de Android Studio, esta referencia viene incluida por defecto al crear un nuevo proyecto en blanco, en este caso hemos elegido la opción **Basic activity**, que incluye el *Action bar* y un *botón Flotante* que en este ejemplo eliminaremos.

A partir de aquí, podemos hacer uso de la *actionBar* de dos formas diferentes. La primera de ellas, más sencilla aunque menos flexible, consiste en utilizar la *ActionBar* por defecto que se añade a nuestras actividades por tan sólo utilizar un tema (*theme*) determinado en la aplicación y extender nuestras actividades de una clase específica. La segunda, más flexible y personalizable aunque requiere más código, consiste en utilizar el nuevo componente *ToolBar* disponible desde la llegada de Android 5.0 (aunque compatible con versiones anteriores). Ahora, en un principio, nos centraremos en la primera de las alternativas indicadas, y a continuación, hablaremos de *ToolBar*.

Vamos a comprobar por tanto en primer lugar, aunque también debe venir configurado por defecto con un nuevo proyecto de Android Studio, que el tema utilizado por nuestra aplicación sea

uno de los proporcionados por la librería *appcompat-v7*. Para ello abrimos el fichero *styles.xml* situado en la carpeta */res/values* y nos aseguramos que el tema de nuestra aplicación extiende de alguno de los temas *Theme.AppCompat* disponibles, en nuestro ejemplo *Theme.AppCompat.Light.DarkActionBar* (fondo claro con action bar oscura):

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        . . .
    </style>
</resources>
```

Revisaremos en segundo lugar que nuestras actividades extienden de la clase *AppCompatActivity*, de forma que puedan hacer uso de toda la funcionalidad de la *Action bar*. Hasta la versión 21 de la librería *appcompat-v7* la clase base de la que debían heredar nuestras actividades era *ActionBarActivity*, pero esto cambió con la versión 22, donde la nueva clase a utilizar como base será *AppCompatActivity*. En nuestro caso, la actividad principal quedaría definida así:

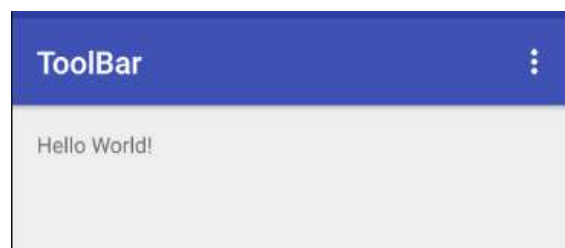
```
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;

public class MainActivity extends AppCompatActivity {

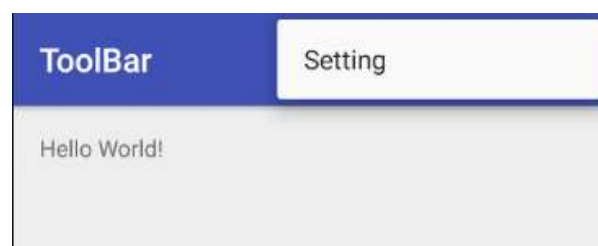
    . . .
    . . .
}

}
```

Tan sólo con esto, si ejecutamos el proyecto podremos comprobar como nuestra actividad muestra la *action bar* en la parte superior, con el siguiente aspecto:



Si desplegamos el menú de overflow veremos que también aparece una opción por defecto llamada “Settings”:



Como vemos, la *ActionBar* por defecto tan sólo contiene el título y el menú de overflow. El título mostrado por defecto corresponde al título de la actividad, definido en el fichero

AndroidManifest.xml, en el atributo `label` del elemento `activity` correspondiente a dicha actividad:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".ActionBarActivity"
            android:label="ActionBar"
            android:theme="@style/AppTheme.NoActionBar">
        </activity>
        <activity android:name=".PestanasActivity" />
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Por su parte, los botones de acción y las opciones del menú de overflow se definen de la misma forma que los antiguos menús de aplicación que se utilizaban en versiones de Android 2.x e inferiores. De hecho, es exactamente la misma implementación. Nosotros definiremos un menú, y si la aplicación se ejecuta sobre Android 2.x las acciones se mostrarán como elementos del menú como tal (ya que la actionBar no se verá al no ser compatible) y si se ejecuta sobre Android 3.0 o superior aparecerán como acciones de la actionBar, ya sea en forma de botón de acción o incluidas en el menú de overflow. En definitiva, una forma de mantener cierta compatibilidad con versiones anteriores de Android, aunque en unas ocasiones se muestre la actionBar y en otras no.

¿Y cómo se define un menú de aplicación? Lo estudiaremos más adelante, en un apartado dedicado exclusivamente a ello donde poder profundizar, pero de cualquier forma, aquí veremos las directrices generales para definir uno sin mucha dificultad.

Un **menú** se define, como la mayoría de los recursos de Android, mediante un fichero XML, y se colocará en la carpeta `/res/menu`. El menú se definirá mediante un elemento raíz `<menu>` y contendrá una serie de elementos `<item>` que representarán cada una de las opciones. Los elementos `<item>` por su parte podrán incluir varios atributos que lo definan, entre los que destacan los siguientes:

- **android:id**. El ID identificativo del elemento, con el que podremos hacer referencia a dicha opción.

- **android:title.** El texto que se visualizará para la opción.
- **android:icon.** El icono asociado a la acción.
- **android:showAsAction.** Si se está mostrando una action bar, este atributo indica si la opción de menú se mostrará como botón de acción o como parte del menú de overflow. Puede tomar varios valores:
 - **ifRoom.** Se mostrará como botón de acción sólo si hay espacio disponible.
 - **withText.** Se mostrará el texto de la opción junto al icono en el caso de que éste se esté mostrando como botón de acción.
 - **never.** La opción siempre se mostrará como parte del menú de overflow.
 - **always.** La opción siempre se mostrará como botón de acción. Este valor puede provocar que los elementos se solapen si no hay espacio suficiente para ellos.

Al crear un proyecto nuevo en Android Studio, se crea un menú por defecto para la actividad principal, que es el que podemos ver en la imagen anterior con una única opción llamada “Settings”. Si abrimos este menú (llamado normalmente */res/menu/menu_main.xml* si no lo hemos cambiado de nombre durante la creación del proyecto) veremos el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      tools:context=".MainActivity">

    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:orderInCategory="100"
          app:showAsAction="never" />

</menu>
```

Como vemos se define un menú con una única opción, con el texto “Settings” y con el atributo `showAsAction="never"` de forma que ésta siempre aparezca en el menú de overflow.

Esta opción por defecto se incluye solo a modo de ejemplo, por lo que podríamos eliminarla sin problemas para incluir las nuestras propias. En el ejemplo, por el momento, la vamos a conservar pero vamos a añadir dos más de ejemplo: “Buscar” y “Nuevo”, la primera de ellas para que se muestre, si hay espacio, como botón con su icono correspondiente, y la segunda igual pero además acompañada de su título de acción:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      tools:context=".MainActivity">

    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:orderInCategory="100"
          app:showAsAction="never" />

    <item android:id="@+id/action_buscar"
          android:title="@string/action_buscar"
          android:icon="@drawable/ic_buscar"
          android:orderInCategory="100"
          app:showAsAction="ifRoom" />

    <item android:id="@+id/action_nuevo"
          android:title="@string/action_nuevo"
          android:orderInCategory="100"
          app:showAsAction="never" />

</menu>
```



```

        <item android:id="@+id/action_nuevo"
            android:title="@string/action_nuevo"
            android:icon="@drawable/ic_nuevo"
            android:orderInCategory="100"
            app:showAsAction="ifRoom|withText" />

    </menu>

```

Los iconos `ic_buscar` e `ic_nuevo` los hemos añadido al proyecto de igual forma que lo hemos hecho anteriormente, por ejemplo como vimos en el punto correspondiente a los botones.

Como vemos en la segunda opción (`action_nuevo`), se pueden combinar varios valores de `showAsAction` utilizando el caracter “|”.

Una vez definido el menú en su fichero XML correspondiente tan sólo queda asociarlo a nuestra actividad principal. Esto se realiza sobrescribiendo el método `onCreateOptionsMenu()` de la actividad, dentro del cual lo único que tenemos que hacer normalmente es inflar el menú llamando al método `inflate()` pasándole como parámetro el ID del fichero XML donde se ha definido. Este trabajo también suele venir hecho ya al crear un proyecto nuevo desde Android Studio:

```

public class EjemploAppBar extends AppCompatActivity {

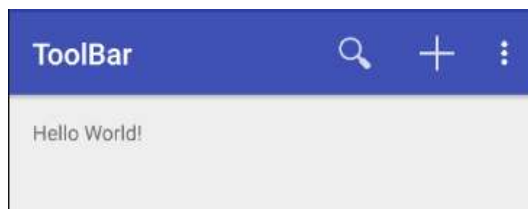
    . . .

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

}

```

Ejecutemos de nuevo la aplicación a ver qué ocurre:



Como podemos observar, la opción “Settings” sigue estando dentro del menú de overflow, y ahora aparecen como botones de acción las dos opciones que hemos marcado como `showAsAction="ifRoom"`, pero para la segunda no aparece el texto. ¿Y por qué? Porque no hay espacio disponible suficiente con la pantalla en vertical. Pero si rotamos el emulador para ver qué ocurre con la pantalla en horizontal vemos lo siguiente:



Con la pantalla en horizontal sí se muestra el texto de la segunda opción, tal como habíamos solicitado con el valor `withText` del atributo `showAsAction`.

Podemos seguir personalizando nuestra action bar definiendo los colores principales del tema seleccionado. Esto podemos hacerlo dentro del fichero `/res/values/styles.xml`, y podemos configurar tres colores principales:

- `colorPrimary`. Es el color principal de la aplicación, y se utilizará entre otras cosas como color de fondo de la action bar.
- `colorPrimaryDark`. Es una variante más oscura del color principal, que por ejemplo en Android 5.0 se utilizará como color de la barra de estado (la barra superior que contiene los iconos de notificación, batería, reloj, ...). Nota: en Android 4.x e inferiores la barra de estado permanecerá negra.
- `colorAccent`. Suele ser el color utilizado para destacar el botón de acción principal de la aplicación (por ejemplo un botón flotante como el que vimos en el punto sobre botones y para otros pequeños detalles de la interfaz, como por ejemplo algunos controles, cuadros de texto o checkboxes).

En el siguiente link (<https://material.google.com/style/color.html>) se puede consultar la paleta de colores estandar definida en las especificaciones de Material Design. En nuestro caso utilizaremos el Indigo intensidad 500 como color principal (#3F51B5), la intensidad 700 de esa misma tonalidad como variante más oscura (#303F9F), y como color destacado el Pink (#FF4081). Los definimos dentro del tema de la aplicación de la siguiente forma:

```
<resources>

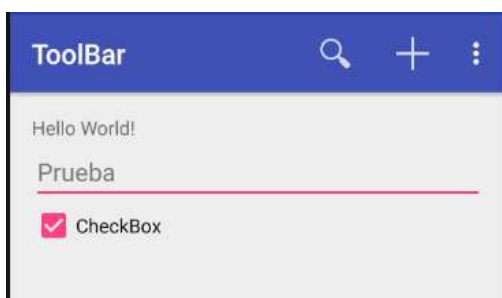
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

</resources>
```

La definición de los colores como tal se realiza en un fichero llamado `colors.xml` dentro de la carpeta `/res/values` (si no existe el fichero podemos crearlo utilizando el menú contextual sobre la carpeta `/res/values`, mediante la opción *New / Values resource file*):

```
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

Si ejecutamos de nuevo la aplicación, añadiendo previamente a la interfaz algunos controles, podemos ver sobre ellos los efectos del color accent.



Por último, ahora que ya sabemos definir y personalizar los elementos de nuestra action bar queda saber cómo responder a las pulsaciones que haga el usuario sobre ellos. Para esto, al igual que se hace con los menús tradicionales (ver punto sobre menús para más detalles), sobrescribiremos el método `onOptionsItemSelected()` de la actividad, donde consultaremos la opción de menú que se ha pulsado mediante el método `getItemId()` de la opción de menú recibida como parámetro y actuaremos en consecuencia. En nuestro caso de ejemplo tan sólo escribiremos un mensaje al log.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    switch (id){
        case R.id.action_nuevo:
            Log.i("ActionBar", "Nuevo!");
            return true;

        case R.id.action_buscar:
            Log.i("ActionBar", "Buscar!");
            return true;

        case R.id.action_settings:
            Log.i("ActionBar", "Settings!");
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Si ejecutamos ahora la aplicación y miramos el log mientras pulsamos las distintas opciones de la action bar veremos como se muestran los mensajes definidos.

Actividad propuesta: Tabs y App bar

Realizar el diseño de **WhatsApp**. Con las siguientes características:

- Contiene 3 pestaña: *Llamadas*, *Chats*, *Contactos*.
- En el Toolbar: Aparece la palabra WhatsApp y tiene los iconos *buscar* y “*otro*”.
- El icono “*otro*” va cambiando dependiendo en la pestaña que nos encontremos.
- Añadir ***ListView*** al contenido de las diferentes pestañas.

24.- Recursos alternativos

Una aplicación Android va a poder ser ejecutada en una gran variedad de dispositivos. El tamaño de pantalla, la resolución o el tipo de entradas puede variar mucho de un dispositivo a otro. Por otra parte, nuestra aplicación ha de estar preparada para diferentes modos de funcionamiento, como el modo “automóvil” o el modo “noche”, y para poder ejecutarse en diferentes idiomas.

A la hora de crear la interfaz de usuario, hemos de tener en cuenta todas estas circunstancias. Afortunadamente, la plataforma Android nos proporciona una herramienta de gran potencia para resolver este problema: el uso de los *recursos alternativos*.

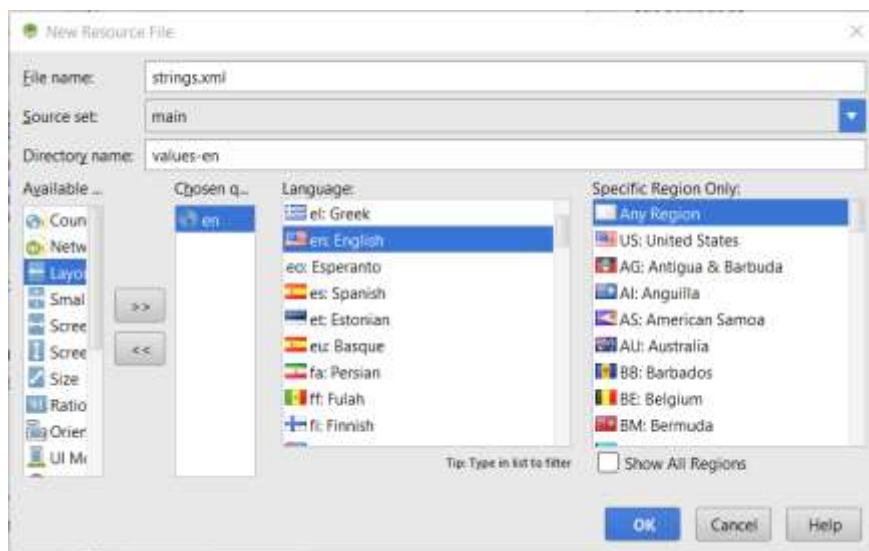
Android utiliza una lista de sufijos para expresar recursos alternativos. Estos sufijos pueden hacer referencia a la orientación del dispositivo, al lenguaje, la región, la densidad de píxeles, la resolución, el método de entrada, ...

Idiomas

Si queremos traducir nuestra aplicación al inglés, español y francés, siendo el primer idioma el usado por defecto, crearíamos tres versiones del fichero **string.xml**, en tres directorios diferentes.

Ejemplo:


- Creamos un recurso alternativo *string.xml*. Sobre la carpeta *res/values* botón derecho y *New* → *Values resource file* ponemos como nombre de fichero *strings.xml*.
- Movemos el modificador *Locale* pulsando >> y elegimos el idioma, por ejemplo: Language: *English* y Region: *Any Region*



- Copiamos el contenido del recurso por defecto, *strings.xml*, al recurso para inglés, *strings.xml (en)* y lo traducimos al inglés, pero no tocamos los identificadores de los recursos (*app_name, ...*) esto es igual en todos los idiomas.
- Ejecutamos la aplicación y para ver el resultado hay que ir a Settings/Ajustes dentro del emulador y cambiar el idioma.

Pantalla Horizontal

Para definir la pantalla en horizontal, haremos lo siguiente y probaremos:

- Creamos un recurso alternativo a nuestra activity, sobre la carpeta `res/layout` botón derecho y *New* → *Layout Resource File* damos el nombre `main_activity.xml` al fichero.
- Seleccionamos el modificador *Orientation* y pulsando  para moverlo.
- En el desplegable *Screen orientation* seleccionamos *Landscape*
Ahora tenemos dos ficheros *activity_main.xml*, uno será el recurso por defecto y el otro el que se usará cuando el dispositivo este en orientación Landscape (horizontal).
- Diseñamos como queremos ver nuestra pantalla horizontal, fichero `activity_main.xml` (`land`).

Tamaño de pantalla

- Podemos crear recursos alternativos para los diferentes tamaños de pantalla, sobre la carpeta `res/layout` botón derecho y *New* → *Layout Resource File*.
- Seleccionamos el modificador *Size* y a continuación elegimos el tipo de *Screen Size* que deseemos (*Small*, *Normal*, *Large* o *X-Large*)
- Diseñamos como queremos ver la pantalla
- Probamos el resultado creando diferentes emuladores de diferentes tamaños.

Combinación

Podemos realizar combinación de los diferentes recursos compartidos.

- Orientación pantalla
 - Portrait (vertical) / Landscape (apaisado u horizontal)
- Tamaño pantalla
 - Small (2-3") / Normal / Large (7") / X-Large (hasta 10")
- Idioma
 - es / fr / en / ...
- Región
 - es / rUS / rUK
- Densidad de pixeles:
 - Low Density / Medium Density / High Density / ...
- Modo nocturno
 - Night / Not Night
- Version (Nivel API):
 - v3 / v4 / v5 ...