Desarrollo Android

Arkaitz Garro

Master en desarrollo de Apps Móviles

Persistencia de datos

Introducción

Introducción

Guardar y cargar los datos de una aplicación es esencial en la mayoría de los casos

Más en el caso de Android, en el que una app puede ser terminada y reinicida en cualquier momento

Android nos ofrece varias maneras de guardar información, optimizadas para cumplir funciones concretas

Persistencia de datos

Técnicas para la persistencia de datos

Preferencias: conjunto de datos en formato clave/valor (NVP) con el objetivo de almacenar estados de la UI, preferencias de usuario y aplicación.

Estado de la UI: las actividades y los fragmentos disponen de eventos y tipos de datos (Bundle) especializados para almacenar y restaurar el estado de la aplicación.

Ficheros: la menos recomendada, pero inevitable en algunos casos. Android nos permite crear ficheros locales (memoria interna/externa), crear caches temporales...

Persistencia de datos

Preferencias

Preferencias

Gracias a la clase `SharedPreferences` podemos almacenar preferencias persistentes a lo largo de todos los componentes de la aplicación

```
SharedPreferences mySharedPreferences = getSharedPreferences(MY_PREFS,
                                        Activity.MODE_PRIVATE);
```

Las preferencias son almacenadas dentro del sandbox de la aplicación, por lo que no están disponibles para otras aplicaciones

Preferencias

Para modificar las preferencias, hay realizarlo a través de la clase `SharedPreferences.Editor`. El objeto se obtiene llamando al método `edit`

```
SharedPreferences.Editor editor = mySharedPreferences.edit();

// Store new primitive types in the shared preferences object.
editor.putBoolean(“isTrue”, true);
editor.putFloat(“lastFloat”, 1f);
editor.putInt(“wholeNumber”, 2);
editor.putLong(“aNumber”, 3l);
editor.putString(“textEntryValue”, “Not Empty”);

// Commit the changes.
editor.apply();
```

Preferencias

Una vez almacenados los valores, es muy sencillo acceder a ellos

```
// Retrieve the saved values.
boolean isTrue = mySharedPreferences.getBoolean(“isTrue”, false);
float lastFloat = mySharedPreferences.getFloat(“lastFloat”, 0f);
int wholeNumber = mySharedPreferences.getInt(“wholeNumber”, 1);
long aNumber = mySharedPreferences.getLong(“aNumber”, 0);
String stringPreference =
mySharedPreferences.getString(“textEntryValue”, “”);

// Retrieve all values
Map<String, ?> allPreferences = mySharedPreferences.getAll();
boolean containsLastFloat = mySharedPreferences.contains(“lastFloat”);
```

Ejercicio

Persistencia de datos

Framework de preferencias

Framework de preferencias

Android ofrece un framework basado en XML para crear pantallas de preferencias en nuestras aplicaciones

Estas preferencias son consistentes tanto en el sistema como en las aplicaciones de terceros, por lo que el usuario está familiarizado con ello

Incluso es posible incluir preferencias de otras aplicaciones o propias del sistema

Framework de preferencias

El framework se compone de cuatro partes

Preference Activity: clase que muestra la definición de cabeceras.

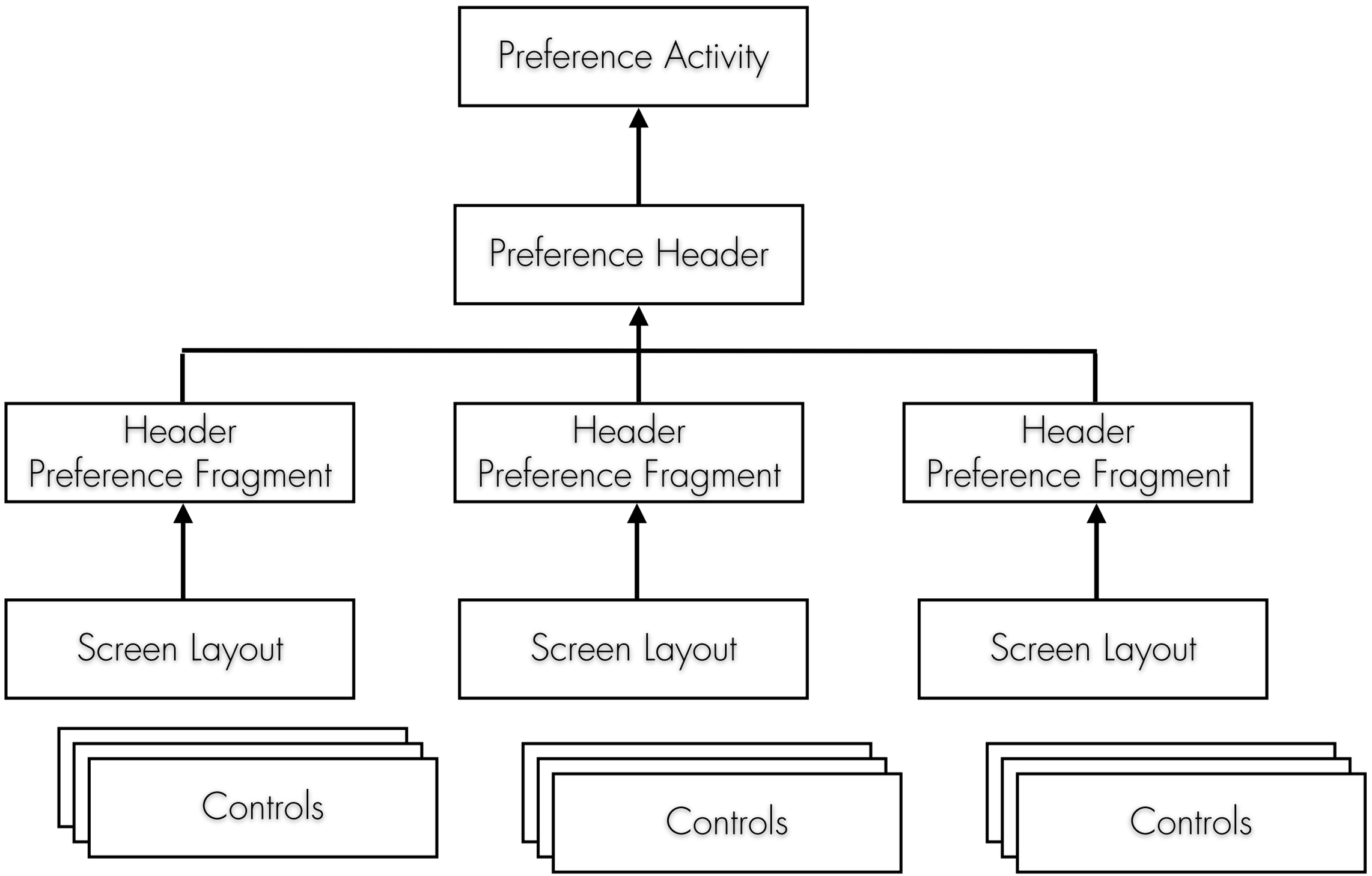
Header definition: un XML que define los fragmentos a utilizar en las preferencias y cómo deben ser organizados.

Preference Fragments: clase de java que muestra el Screen Layout.

Screen Layout: un XML que define la estructura de los elementos a mostrar. Se especifica el tipo de control, los valores, y la clave a utilizar.

Change listener: escucha los cambios producidos en las preferencias.

Framework de preferencias



Preference Activity

Contiene y muestra toda la jerarquía de preferencias

```
public class MyFragmentPreferenceActivity extends PreferenceActivity {
   public void onBuildHeaders(List<Header> target) {
      loadHeadersFromResource(R.xml.userpreferenceheaders, target);
   }
}
```

Como todas las actividades, es necesario registrarla en el manifiesto

```
<activity android:name=”.MyPreferenceActivity”
          android:label=”My Preferences”>
</activity>
```

Iniciar la actividad de manera normal

```
Intent i = new Intent(this, MyPreferenceActivity.class);
startActivityForResult(i, SHOW_PREFERENCES);
```

Preference Headers

Definen como los fragmentos deben agruparse y mostrarse en las preferencias

La manera en la que se muestran, dependerá del dispositivo

Cada `header` tiene asociado un `PreferenceFragment` que se mostrará cuando el `header` haya sido seleccionado

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
  <header android:fragment="com.paad.preferences.MyPreferenceFragment"
          android:icon="@drawable/preference_icon"
          android:title="My Preferences"
          android:summary="Description of these preferences" />
  <header android:icon="@drawable/ic_settings_display"
        android:title="Intent"
        android:summary="Launches an Intent.">
        <intent android:action="android.settings.DISPLAY_SETTINGS "/>
  </header>
</preference-headers>
```

Importar preferencias

Es posible incluir preferencias de otras aplicaciones o las propias del sistema

Sólo hay que añadir un Intent al Layout de preferencias, para que Android lo interprete como una llamada e incluya las preferencias

```
<?xml version=”1.0” encoding=”utf-8”?>
<PreferenceScreen xmlns:android=”http://schemas.android.com/apk/res/android”
   android:title=”Intent preference”
   android:summary=”System preference imported using an intent”>
   <intent android:action=”android.settings.DISPLAY_SETTINGS “/>
</PreferenceScreen>
```

Preference Fragment

Es utilizado simplemente para mostrar las pantallas de preferencias

```
public class MyPreferenceFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.userpreferences);
    }
}
```

Nuestra aplicación puede incluir tantos fragmentos como necesitemos

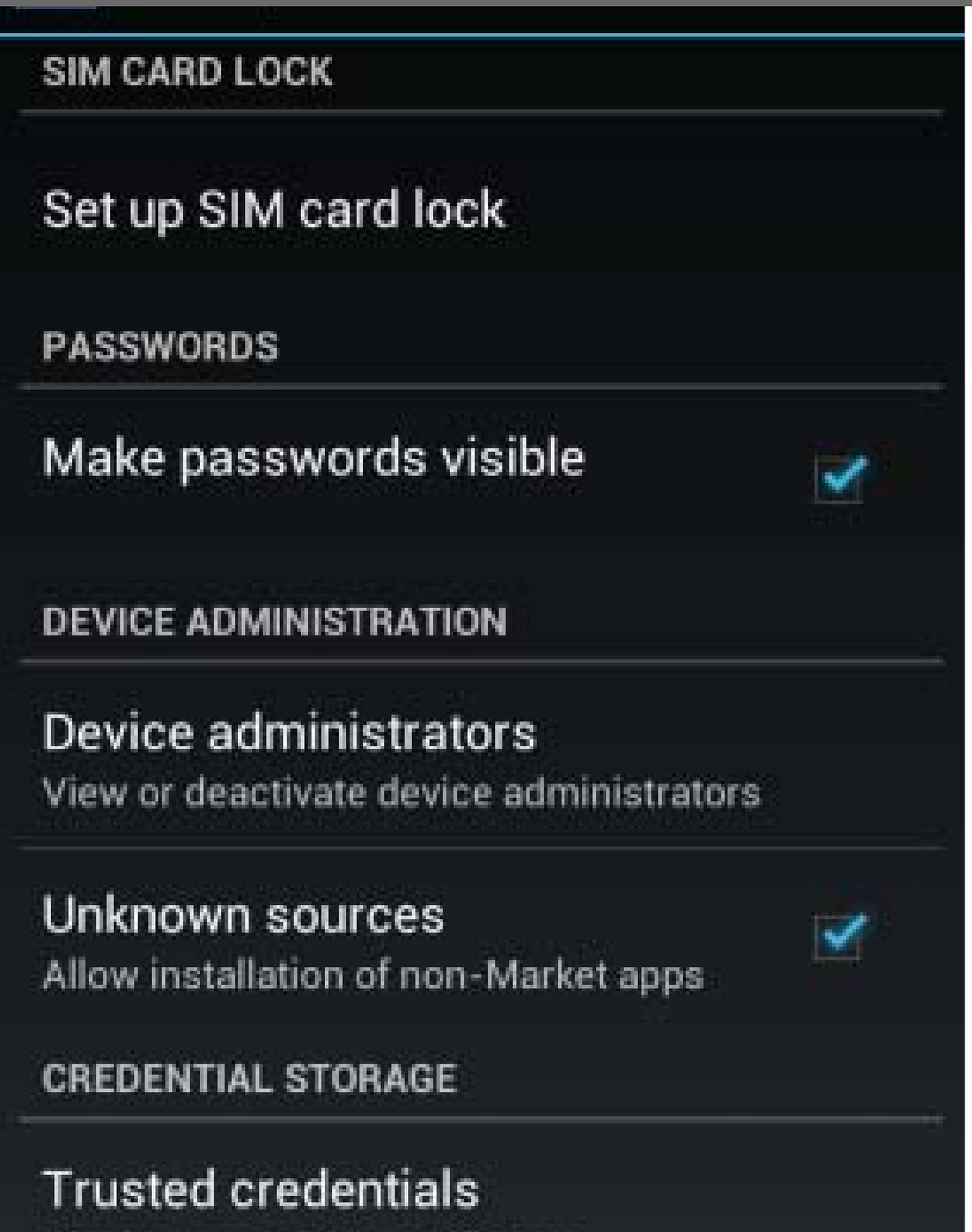
Screen Layout

Conceptualmente similares a los Layout, pero especializados en mostrar controles de preferencias

Definen la estructura de la pantalla de preferencias, pudiendo crear una estructura de varias pantallas en el mismo Layout

Dentro de cada pantalla es posible combinar controles sueltos y categorías de controles

Screen Layout



Acceso a nueva ventana de preferencias

Categorias de controles de preferencias

Control CheckBox

Screen Layout

Una vez definida la estructura, sólo queda añadir los controles necesarios

Cada uno de estos controles, debe especificar al menos estos cuatro atributos

`android:key`: clave única que identifica la preferencia.

`android:title`: texto a mostrar.

`android:summary`: texto descriptivo de ayuda.

`android:defaultValue`: valor por defecto del control.

Screen Layout

res/xml/userpreferences.xml

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory
    android:title="My Preference Category">
    <CheckBoxPreference
      android:key="PREF_CHECK_BOX"
      android:title="Check Box Preference"
      android:summary="Check Box Preference Description"
      android:defaultValue="true"
    />
  </PreferenceCategory>
</PreferenceScreen>
```

Controles nativos

`CheckBoxPreference`: control estándar para marcar verdadero o falso.

`EditTextPreference`: permite introducir un texto al usuario.

`ListPreference`: lista desplegable que contiene los diferentes valores disponibles.

`MultiSelectListPreference`: listado de controles CheckBox.

`RingtonePreference`: listado especial que contiene los tonos de llamadas. Especial para notificaciones.

Obtener los valores

Los valores almacenados utilizando el framework de preferencias, son accesibles a través de la clase `SharedPreferences`

```
Context context = getApplicationContext();
SharedPreferences prefs =
    PreferenceManager.getDefaultSharedPreferences(context);
// TODO Retrieve values using get<type> methods.
```

Change Listener

En ocasiones, puede ser interesante realizar acciones puntuales cuando cambia un valor de las preferencias

```
public class MyActivity extends Activity implements
OnSharedPreferenceChangeListener {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      // Register this OnSharedPreferenceChangeListener
      SharedPreferences prefs =
          PreferenceManager.getDefaultSharedPreferences(this);
      prefs.registerOnSharedPreferenceChangeListener(this);
   }

   public void onSharedPreferenceChanged(SharedPreferences prefs, String key) {
      // TODO Check the shared preference and key parameters
      // and change UI or behavior as appropriate.
   }
}
```

Ejercicio

Persistencia de datos

Estado de la aplicación

Guardar el estado

Android nos ofrece dos posibilidades para guardar el estado de la aplicación

A través de preferencias específicas para la actividad

O bien utilizando el Bundle definido en el ciclo de vida

Estado a través de preferencias

Permite guardar valores que no van a ser compartidos con otros componentes, pero se mantienen entre sesiones

```
// Create or retrieve the activity preference object.
SharedPreferences activityPreferences = getPreferences(Activity.MODE_PRIVATE);

// Retrieve an editor to modify the shared preferences.
SharedPreferences.Editor editor = activityPreferences.edit();

// Retrieve the View
TextView myTextView = (TextView)findViewById(R.id.myTextView);

// Store new primitive types in the shared preferences object.
editor.putString(“currentTextValue”, myTextView.getText().toString());

// Commit changes.
editor.apply();
```

Estado a través del ciclo de vida

Las actividades ofrecen la posibilidad de guardar estados de la interfaz en el evento `onSaveInstanceState`

Si una actividad es cerrada por el usuario, o con el método `finish`, este estado no será pasado de nuevo al crear la actividad

Por lo tanto, este método sólo es llamado dentro del ciclo de vida gestionado por Android

Si los datos deben ser persistentes entre sesiones, hay que utilizar las preferencias

Estado a través del ciclo de vida

```
private static final String TEXTVIEW_STATE_KEY = “TEXTVIEW_STATE_KEY”;

@Override
public void onSaveInstanceState(Bundle saveInstanceState) {
   // Retrieve the View
   TextView myTextView = (TextView)findViewById(R.id.myTextView);

   // Save its state
   saveInstanceState.putString(TEXTVIEW_STATE_KEY,
      myTextView.getText().toString());

   super.onSaveInstanceState(saveInstanceState);
}
```

Estado a través del ciclo de vida

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);

   TextView myTextView = (TextView)findViewById(R.id.myTextView);
   String text = “”;

   if (savedInstanceState != null &&
      savedInstanceState.containsKey(TEXTVIEW_STATE_KEY))
      text = savedInstanceState.getString(TEXTVIEW_STATE_KEY);

   myTextView.setText(text);
}
```

Estado en los fragmentos

Los fragmentos utilizados para la construcción de UI, al igual que las actividades, lanzan el evento `onSaveInstanceState`

El estado almacenado es pasado a los eventos `onCreate`, `onCreateView`, `onActivityCreated`

Cuando una actividad es reiniciada, es posible indicar que la instancia de los fragmentos no sea recreada

Esto incrementa notablemente el rendimiento, ya que los fragmentos no tienen que volver a crearse

```
public class MyFragment extends Fragment {

  private static String USER_SELECTION = "USER_SELECTION";
  private int userSelection = 0;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
    if (savedInstanceState != null)
      userSelection = savedInstanceState.getInt(USER_SELECTION);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.mainfragment, container, false);
    setSelection(userSelection);

    return v;
  }

  @Override
  public void onSaveInstanceState(Bundle outState) {
    outState.putInt(USER_SELECTION, userSelection);
    super.onSaveInstanceState(outState);
  }
}
```

Persistencia de datos

Ficheros

Incluir ficheros externos

En algunos casos, puede que sea necesario incluir ficheros externos, de solo lectura

Es una excelente alternativa para grandes ficheros, y que no alteren con el tiempo (diccionarios, listados de países...)

Estos ficheros se almacenan en el directorio `res/raw`

```
Resources myResources = getResources();
InputStream myFile = myResources.openRawResource(R.raw.myfilename);
```

Sistema de ficheros

En algunos casos, como en ficheros multimedia, las preferencias o bases de datos no son suficientes

Android hace uso de la librería `java.io.File` para el manejo de ficheros, aunque la complementa con sus propios métodos

Estos métodos están disponibles desde el contexto de la aplicación

`deleteFile`: elimina un fichero creado por la aplicación.

`fileList`: lista los ficheros creados por la aplicación.

Directorios de almacenamiento

Existen dos opciones para almacenar los ficheros: memoria interna o externa

Cuando nos referimos a memoria externa, hace referencia al sistema de ficheros que puede ser montado y accedido a través de USB. Generalmente es una tarjeta SD, aunque puede ser una partición interna

Los ficheros almacenados en una memoria externa no tienen por qué estar disponibles

La seguridad puede verse comprometida: los ficheros externos pueden ser accedidos por todas las app

Crear ficheros privados

Android ofrece la posibilidad de crear y leer ficheros dentro del sandbox de la aplicación

Estos métodos únicamente acceden al directorio principal de la aplicación, no a subdirectorios

```
String FILE_NAME = “tempfile.tmp”;

// Create a new output file stream that’s private to this application.
FileOutputStream fos = openFileOutput(FILE_NAME, Context.MODE_PRIVATE);

// Create a new file input stream.
FileInputStream fis = openFileInput(FILE_NAME);
```