Utilización de preferencias

|  |
| --- |
|  |
| UD 4: Acceso al sistema de ficheros |
|  |
| Desarrollo de aplicaciones para Dispositivos Móviles |



**cENTRO DE ESTUDIOS SEIM**

1 de julio de 2012

Autor: Daniel Miguel

En la unidad 2 utilizábamos las preferencias para almacenar la configuración de la aplicación. También podemos utilizar las *SharedPreferences* para almacenar pequeñas cantidades de información para que más tarde las utilicen las aplicaciones.

Cada preferencia se almacena en forma de clave-valor, es decir, cada una de ellas Cada preferencia se almacenará en forma de clave-valor, es decir, cada una de ellas estará compuesta por un identificador único (p.e. “email”) y un valor asociado a dicho identificador (p.e. “prueba@email.com”). Además, los datos se guardan en ficheros XML como veremos más adelante.

Toda la gestión se centraliza en la clase SharedPrefences, que representará a una colección de preferencias. Una aplicación Android puede gestionar varias colecciones de preferencias, que se diferenciarán mediante un identificador único. Para obtener una referencia a una colección determinada utilizaremos el método getSharedPrefences() al que pasaremos el identificador de la colección y un modo de acceso. El modo de acceso indicará qué aplicaciones tendrán acceso a la colección de preferencias y qué operaciones tendrán permitido realizar sobre ellas. Así, tendremos tres posibilidades principales:

•MODE\_PRIVATE. Sólo nuestra aplicación tiene acceso a estas preferencias.

|  |
| --- |
| Ejemplo: Acceso a una colección de preferencias llamada “Mis Preferencias” y como modo de acceso exclusivo para nuestra aplicación  SharedPreferences prefs =  getSharedPreferences("MisPreferencias",Context.MODE\_PRIVATE);  String correo= prefs.getString(“email”,[xxx@servidor.com](mailto:xxx@servidor.com));  Como vemos primero obtenemos un puntero a la colección de referencias deseada, con el modo de acceso que nos interese y luego utilizamos el método *getString* para acceder al valor cuya clave es “email”. El segundo parámetro es un valor por defecto, que será devuelto si la preferencia solicitada no existe en la colección.  Por supuesto existen tipos análogos para recuperar preferencias cuyo tipo sea cualquier otro tipo de datos básico: *getInt, getLong,* etc. |

# ¿Donde se almacenan estas preferencias compartidas?

El almacenamiento de las preferencias se realiza en ficheros XML. Estos ficheros XML se almacenan en una ruta con el siguiente patrón:

/data/data/paquetejava/shared\_prefs/nombre\_coleccion.xml

Si descargamos este fichero desde el DDMS y lo abrimos con cualquier editor de texto veremos un contenido como el siguiente:

|  |
| --- |
| <?xml version='1.0' encoding='utf-8' standalone='yes' ?>  <map>  <string name="nombre">prueba</string>  <string name="email">modificado@email.com</string>  </map> |

En este XML podemos observar cómo se han almacenado las dos preferencias de ejemplo que insertamos anteriormente, con sus claves y valores correspondientes.

Accediendo a la memoria interna de Android

En Android también podremos manipular ficheros tradicionales de una forma muy similar a como se realiza en Java.

Lo primero que hay que tener en cuenta es dónde queremos almacenar los ficheros y el tipo de acceso que queremos tener a ellos. Así, podremos leer y escribir ficheros localizados en:

* La **memoria interna** del dispositivo.
* La propia aplicación, en forma de **recurso**.
* La **tarjeta SD** externa, si existe.

Para trabajar con la memoria interna lo primero que tenemos que tener en cuenta son las limitaciones de capacidad de la misma. Habitualmente la memoria interna está limitada y más teniendo en cuenta que parte de las aplicaciones que instalamos y del propio sistema operativo tienen que estar ubicadas en dicha memoria.

Para acceder a la memoria utilizamos openFileOutput(), que recibe como parámetros el nombre del fichero y el modo de acceso con el que queremos abrir el fichero. Este modo de acceso puede variar entre MODE\_PRIVATE (por defecto) para acceso privado desde nuestra aplicación, MODE\_APPEND para añadir datos a un fichero ya existente, MODE\_WORLD\_READABLE para permitir a otras aplicaciones leer el fichero, o MODE\_WORLD\_WRITABLE para permitir a otras aplicaciones escribir sobre el fichero.

Devuelve una referencia al *stream* de salida asociado al fichero (en forma de objeto FileOutputStream), a partir del cual ya podremos utilizar los métodos de manipulación de ficheros tradicionales del lenguaje java (api java.io). Como ejemplo, convertiremos este stream a un OutputStreamWriter para escribir una cadena de texto al fichero.

|  |
| --- |
| try  {  OutputStreamWriter fout=new OutputStreamWriter(  openFileOutput("prueba\_int.txt", Context.MODE\_PRIVATE));  fout.write("Texto de prueba.");  fout.close();  }  catch (Exception ex)  {  Log.e("Ficheros", "Error al escribir fichero a memoria interna");  } |

Los ficheros por defecto se almacenan en la ruta /data/data/paquete\_java/files/nombre\_fichero

Para leer ficheros, utilizaremos el método el método openFileInput() para abrir el fichero, y los métodos de lectura de java.io para leer el contenido.

|  |
| --- |
| try  {  BufferedReader fin = new BufferedReader( new InputStreamReader(openFileInput("prueba\_int.txt")));  String texto = fin.readLine();  fin.close();  }  catch (Exception ex)  {  Log.e("Ficheros", "Error al leer fichero desde memoria interna");  } |

La segunda forma de almacenar ficheros en la memoria interna del dispositivo es incluirlos como recurso en la propia aplicación. Aunque este método es útil en muchos casos, sólo debemos utilizarlo cuando no necesitemos realizar modificaciones sobre los ficheros, ya que tendremos limitado el acceso a sólo lectura.

Para incluir un fichero como recurso de la aplicación debemos colocarlo en la carpeta “/res/raw” de nuestro proyecto de Eclipse. Esta carpeta no suele estar creada por defecto, por lo que deberemos crearla manualmente en Eclipse desde el menú contextual con la opción “*New / Folder*“.

Una vez creada la carpeta /raw podremos colocar en ella cualquier fichero que queramos que se incluya con la aplicación en tiempo de compilación en forma de recurso. Nosotros incluiremos como ejemplo un fichero de texto llamado “*prueba\_raw.txt*“. Ya en tiempo de ejecución podremos acceder a este fichero, sólo en modo de lectura, de una forma similar a la que ya hemos visto para el resto de ficheros en memoria interna.

Para acceder al fichero, accederemos en primer lugar a los recursos de la aplicación con el método getResources() y sobre éstos utilizaremos el método openRawResource(*id\_del\_recurso*) para abrir el fichero en modo lectura. Este método devuelve un objeto InputStream, que ya podremos manipular como queramos mediante los métodos de la API java.io. Como ejemplo, nosotros convertiremos el stream en un objeto BufferedReader para leer el texto contenido en el fichero de ejemplo (por supuesto los ficheros de recurso también pueden ser binarios, como por ejemplo ficheros de imagen, video, etc). Veamos cómo quedaría el código:

|  |
| --- |
| try  {  InputStream fraw =  getResources().openRawResource(R.raw.prueba\_raw);  BufferedReader brin =  new BufferedReader(new InputStreamReader(fraw));  String linea = brin.readLine();  fraw.close();  }  catch (Exception ex)  {  Log.e("Ficheros", "Error al leer fichero desde recurso raw");  } |

Como puede verse en el código anterior, al método openRawResource() le pasamos como parámetro el ID del fichero incluido como recurso, que seguirá el patrón “***R.raw.****nombre\_del\_fichero*“, por lo que en nuestro caso particular será R.raw.prueba\_raw.

Por último vamos a ver como acceder a la memoria externa SD. A diferencia de la memoria externa, es posible que no haya ninguna insertada o que pese a estarlo el sistema no la reconozca. Por ello lo primero que hay que hacer es comprobar si podemos utilizar el almacenamiento extraíble mediante el método *getExternalStorageStatus().*

Los valores devueltos por esta función son:

* MEDIA\_MOUNTED, que indica que la memoria externa está disponible y podemos tanto leer como escribir en ella.
* MEDIA\_MOUNTED\_READ\_ONLY, que indica que la memoria externa está disponible pero sólo podemos leer de ella.
* Otra serie de valores que indicarán que existe algún problema y que por tanto no podemos ni leer ni escribir en la memoria externa (MEDIA\_UNMOUNTED, MEDIA\_REMOVED, …). Podemos ver información al respecto en la [documentación oficial de la clase Evironment](http://developer.android.com/reference/android/os/Environment.html)

|  |
| --- |
| Ejemplo: En el siguiente ejemplo realizamos la comprobación acerca del estado del dispositivo de almacenamiento externo SD.  boolean sdDisponible = false;  boolean sdAccesoEscritura = false;  //Comprobamos el estado de la memoria externa (tarjeta SD)  String estado = Environment.getExternalStorageState();  if (estado.equals(Environment.MEDIA\_MOUNTED))  {  sdDisponible = true;  sdAccesoEscritura = true;  }  else if (estado.equals(Environment.MEDIA\_MOUNTED\_READ\_ONLY))  {  sdDisponible = true;  sdAccesoEscritura = false;  }  else  {  sdDisponible = false;  sdAccesoEscritura = false;  } |

Para escribir un fichero a la tarjeta SD tenemos que:

* Obtener en primer lugar la ruta al directorio raíz de esta memoria. Para ello utilizaremos el método getExternalStorageDirectory() de la clase Environment, que nos devolverá un objeto File con la ruta de dicho directorio.
* A partir de este objeto, podremos construir otro con el nombre elegido para nuestro fichero (como ejemplo “prueba\_sd.txt“), creando un nuevo objeto File que combine ambos elementos.
* Por último encapsularlo en algún objeto de escritura de ficheros de la API de java y escribir algún dato. En nuestro caso lo convertiremos a un objeto OutputStreamWriter para escribir al fichero un mensaje de texto.

|  |
| --- |
| Ejemplo: Una vez comprobada la existencia de un dispositivo SD correctamente montado, escribimos un dato de prueba.  try  {  File ruta\_sd = Environment.getExternalStorageDirectory();  File f = new File(ruta\_sd.getAbsolutePath(), "prueba\_sd.txt");  OutputStreamWriter fout =new OutputStreamWriter(new FileOutputStream(f));  fout.write("Texto de prueba.");  fout.close();  }  catch (Exception ex)  {  Log.e("Ficheros", "Error al escribir fichero a tarjeta SD");  } |

Para que esto funcione, hay que incluir la siguiente línea en el AndroidManifiest.xml:

<uses-permission

android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

</uses-permission>

Solicitando permiso para acceder a la tarjeta de almacenamiento SD.

Para leer el archivo escrito anteriormente haríamos un proceso similar:

Obtenemos el directorio raiz de la memoria externa con getExternalStorageDirectory(), creamos un objeto File que combine esa ruta con el nombre del fichero a leer y lo encapsulamos dentro de algún objeto que facilite la lectura. Utilizaremos un BufferedReader.

|  |
| --- |
| Ejemplo: Recuperamos la información para leer del fichero creado anteriormente. Habría que hacerlo tras comprobar el montaje de la tarjeta.  try  {  File ruta\_sd = Environment.getExternalStorageDirectory();  File f = new File(ruta\_sd.getAbsolutePath(), "prueba\_sd.txt");  BufferedReader fin =new BufferedReader(new InputStreamReader(new FileInputStream(f)));  String texto = fin.readLine();  fin.close();  }  catch (Exception ex)  {  Log.e("Ficheros", "Error al leer fichero desde tarjeta SD");  } |