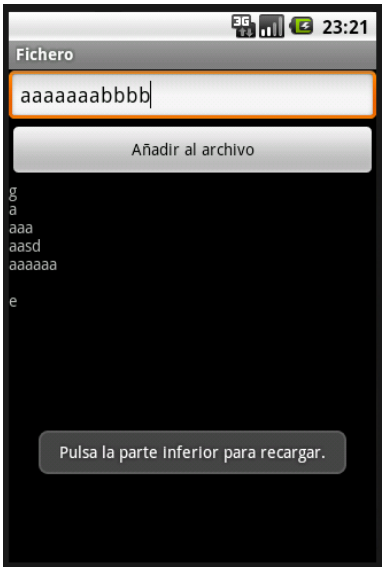
Ficheros y acceso a datos - Ejercicios

Índice

1. Escribir en un archivo de texto

Vamos a leer y escribir en un archivo de texto. La escritura la haremos añadiendo al final del archivo, mientras que la lectura la haremos leyendo línea a línea.

- Crear un proyecto Fichero cuya única actividad, la principal, mostrará un `EditText` cuyo texto será recogido cada vez que se pulse un `Button` que habrá debajo. Dicho texto será añadido a un archivo de texto llamado `mytextfile.txt`. Para añadir texto a un archivo primero lo abriremos con `openFileOutput(FILENAME,Context.MODE_APPEND))` y después utilizaremos el método `append(...)` de `OutputStreamWriter`.
- ¿Cuándo abrimos el archivo? Recordemos que la actividad puede pasar a inactiva en cualquier momento y que puede no volver a recuperarse. ¿Cuándo nos conviene cerrar el archivo?
- Debajo del campo de texto y del botón, vamos a añadir un `TextEdit` que ocupe el resto de la pantalla. Lo haremos pulsable con el método `setClickable(true)` y cada vez que se haga click sobre él, leeremos el archivo línea a línea y lo mostraremos entero.



Programa que escribe y lee un fichero de texto

2. Crear y utilizar un DataHelper para SQLite

En las plantillas de la sesión tenemos el esqueleto de un DataHelper. Se trata de un patrón de diseño que nos ayuda a acceder a nuestros datos encapsulando todo el manejo de la base de datos en el `DataHelper`.

El `DataHelper` utiliza a su vez otro patrón de diseño, el `SQLiteOpenHelper`. Éste nos obliga a definir qué ocurre cuando la base de datos todavía no existe y debe ser creada, y qué ocurre si ya existe pero debe ser actualizada porque ha cambiado de versión. Así el `SQLiteOpenHelper` que implementemos, en este caso `MiOpenHelper`, nos devolverá siempre una base de datos separándonos de la lógica encargada de comprobar si la base de datos existe o no.

Se pide:

- Ejecutar la sentencia de creación de bases de datos (la tenemos declarada como constante de la clase) en el método `MiOpenHelper.onCreate(...)`.
- Implementar también el método onUpgrade. Idealmente éste debería portar las tablas de la versión antigua a la versión nueva, copiando todos los datos. Nosotros vamos a eliminar directamente la talba que tenemos con la sentencia SQL `"DROP TABLE IF EXISTS " + TABLE_NAME` y volveremos a crearla.
- En el constructor del `DataHelper` debemos obtener en el campo `db` la base de datos, utilizando `MiOpenHelper`.

También debemos completar código relacionado con las sentencias de inserción y de borrado.

- En el constructor del `DataHelper`, una vez obtenida la base de datos a través del helper, utilizarla para compilar la sentencia `INSERT` que tenemos como constante String.
- Completar la función `insert()` introduciendo el nombre en la sentencia compilada (con la función `bindString`) y ejecutándola. Devolverá el número de filas afectadas, que a su vez se devolverá con el `return`.
- Completar la función `deleteAll()` que también devolverá el número de filas afectadas.

Ahora podemos utilizar el helper para leer y escribir en la base de datos de forma transparente.

- En el `Main` debemos probar el borrado, inserción y listado de datos.
- Comprobemos por línea de comandos que la base de datos está creada. Serán útiles los siguientes comandos:

```
adb -e shell
#cd /data/data/es.ua.jtech.daa/databases
#sqlite3 misusuarios.db
sqlite> .schema
sqlite> .tables
sqlite> select * from usuarios;
```

- Ahora vamos a cambiar en el `DataHandler` el nombre de la segunda columna, en lugar de nombre, se va a llamar nombres. Ejecutamos la aplicación y comprobamos que sale con una excepción. Se pide comprobar en el Log cuál ha sido el error. ¿Cómo lo podemos solucionar?

Nota:

> Pista: conforme hemos programado el `DataHandler` y siguiendo el patrón de diseño de `SQLiteOpenHelper`, podemos arreglar el problema tocando sólo una tecla.

Opcional:

- Añade al helper la función para eliminar por nombre de usuario.
- Añade al layout un spinner para que se pueda seleccionar qué usuario eliminar, y un botón para eliminarlo. Añade también un campo de texto para la introducción de nuevos, y un botón de inserción de nuevo usuario.

3. Proveedor de contenidos propio

Vamos a implementar otra forma de acceder a la misma base de datos, siguiento esta vez el patrón de diseño ContentProvider de Android. Seguiremos trabajando con el proyecto del ejercicio anterior.

- Creamos una nueva clase llamada `UsuariosProvider` que herede de `ContentProvider`. Nos obligará a sobrecargar una serie de métodos abstractos. Antes de implementar la `query` vamos a configurar el provider:
- Añadimos algunos campos típicos de los content provider:

```
public static final Uri CONTENT_URI =
        Uri.parse("content://es.ua.jtech.daa/usuarios");
private static final int ALLROWS = 1;
private static final int SINGLE_ROW = 2;
private static final UriMatcher uriMatcher;
static{
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("es.ua.jtech.daa", "usuarios", ALLROWS);
        uriMatcher.addURI("es.ua.jtech.daa", "usuarios/#", SINGLE_ROW);
}
```

- Vamos a acceder a la misma base de datos que el ejercicio anterior, pero no vamos a hacerlo a través del helper que tuvimos que implementar, sino que vamos a copiar de él el código que nos haga falta. Copia los los campos que definen el nombre de la base de datos, de la tabla, de las columnas, la versión, así como la referencia al contexto y a la base de datos. La sentencia compilada del insert ya no va a hacer falta. Inicializa los valores que haga falta en el constructor. Para inicializar la referencia a la base de datos vamos a utilizar, una vez más, `MiOpenHelper`. Podemos copiarlo del helper del ejercicio anterior al `UsuariosProvider`.
- Implementa de forma apropiada el `getType` para devolver un tipo MIME diferente según si se trata de una URI de una fila o de todas las filas. Para ello ayúdate del `uriMatcher`.
- Implementa el método `query`. Simplemente se trata de devolver el cursor que obtenemos al hacer una query a la base de datos SQLite. Algunos de los parámetros que le pasaremos los recibimos como parámetros del método del provider. Los que no tengamos irán con valor `null`.
- Aunque no tenemos todos los métodos del `UsuariosProvider` implementados,

podemos probarlo. Para ello debemos registarlo en el `AndroidManifest.xml`:

```
 ...
                <provider android:name="UsuariosProvider"
                        android:authorities="es.ua.jtech.daa"/>
    </application>
    <uses-sdk android:minSdkVersion="8" />
</manifest>
```

- En `Main` del ejercicio anterior se insertan una serie de valores con el helper y se muestran en el campo de texto. Manteniendo este código, vamos a añadir al campo de texto el resultado obtenido con la `query` del `UsuariosProvider` para comprobar que tanto el helper como el provider nos devuelven el mismo resultado.

4. ¿Por qué conviene crear proveedores de contenidos?

Porque es la forma estándar que establece Android de acceder a contenidos. Además, el content provider nos permitirá notificar al `ContentResolver` de los cambios ocurridos. Así componentes en la pantalla podrán refrescarse de forma automática.

- Descarga las plantillas y utiliza el proyecto `ProveedorContenidos`. Implementa la inserción en el proveedor de contenidos. Pruébala insertando algunos usuarios de ejemplo en el `Main`. Implementa también el `OnClickListener` del botón que inserta nuevos usuarios. El nombre del nuevo usuario irá indicado en el `EditText`.
- Comprueba que la inserción funciona y que, gracias a la línea

  ```
  cursor.setNotificationUri(cr, UsuariosProvider.CONTENT_URI);
  ```

  y gracias a que usamos un `ContentProvider`, la lista se actualiza automáticamente cuando ocurre algún cambio, sin necesidad de pedir explícitamente la actualización al pulsar el botón.
- Implementa el método `delete` del proveedor de contenidos. Pruébalo en el `Main`. Termina de implementar el `onCreateContextMenuListener` que se ejecutará cada vez que se haga una pulsación larga sobre alguna entrada de la lista. Comprueba que funciona (eliminando el usuario correspondiente, y no otro).
- Opcional: Añade un campo más, por ejemplo "permisos" a la tabla y al proveedor de contenidos.

5. Proveedores nativos

Entre los proveedores de contenidos nativos nos encontramos el `Browser`, `CallLog`, `ContactsContract`, `MediaStore`, `Settings`, `UserDictionary`. En este ejercicio vamos a acceder a los contactos. Para reutilizar la tabla, vamos a copiar el proyecto anterior, `ProveedorContenidos` y lo vamos a pegar con nuevo nombre, `ProveedoresPropios`. Para cambiar el nombre de la aplicación tenemos que editar el recurso `strings.xml`.

- Necesitamos permisos para acceder a los contactos. Se añaden en el `AndroidManifest.xml`:

```
    ...
    <uses-sdk android:minSdkVersion="8" />
        <uses-permission android:name="android.permission.READ_CONTACTS"/>
</manifest>
```

- Elimina las acciones de los eventos de inserción y borrado.
- Cambia la query para que acceda a ContactsContract.Contacts.CONTENT_URI, así como la uri de notificación del cursor.
- En el `adapter` mapea los mismos campos de texto del anterior ejercicio (Id y Nombre) a las columnas `new String[]{ContactsContract.Contacts._ID, ContactsContract.Contacts.DISPLAY_NAME}`.
- Para comprobar que funciona correctamente deberás introducir algunos números de teléfono desde la agenda de Android.
- Opcional: Una vez que tengas el ID de un contacto, puedes acceder a sus números de teléfono que se encuentran en otra tabla, `ContactsContract.Data` y necesitarías otro cursor diferente para recorrerlos.

Nota:

Una forma mucho más directa de acceder desde una aplicación a los contactos es lanzando la actividad de los contactos, nativa de Android.