

# Persistencia

## Índice

|  |    |
|--|----|
| 1 Persistencia en Android: ficheros y SQLite.....  | 3  |
| 1.1 Introducción.....  | 3  |
| 1.2 Manejo de ficheros tradicionales en Android.....                                       | 3  |
| 1.3 Base de datos SQLite.....  | 6  |
| 2 Persistencia en Android: ficheros y SQLite - Ejercicios.....                             | 15 |
| 2.1 Uso de ficheros.....   | 15 |
| 2.2 Persistencia con ficheros (*)......  | 16 |
| 2.3 Base de datos: SQLiteOpenHelper.....   | 16 |
| 2.4 Base de datos: inserción y borrado.....  | 17 |
| 2.5 Base de datos: probar nuestro adaptador.....   | 18 |
| 2.6 Base de datos: cambios en la base de datos (*)......                                   | 18 |
| 3 Persistencia en Android: proveedores de contenidos y SharedPreferences.....              | 19 |
| 3.1 Shared Preferences.....  | 19 |
| 3.2 Proveedores de contenidos.....   | 24 |
| 4 Persistencia en Android: proveedores de contenidos y SharedPreferences - Ejercicios..... | 33 |
| 4.1 Compartir datos entre actividades con Shared Preferences.....                          | 33 |
| 4.2 Actividad de preferencias.....   | 33 |
| 4.3 Proveedor de contenidos propio.....  | 34 |
| 4.4 ¿Por qué conviene crear proveedores de contenidos? (*)......                           | 35 |
| 5 Persistencia de datos en iOS: Ficheros y SQLite.....                                     | 37 |
| 5.1 Introducción.....  | 37 |
| 5.2 Ficheros plist (Property Lists).....   | 37 |
| 5.3 SQLite.....  | 43 |
| 6 Persistencia de datos en iOS: Ficheros y SQLite - Ejercicios.....                        | 53 |
| 6.1 Creando y leyendo un fichero plist.....  | 53 |
| 6.2 Implementar vista de detalle de la serie.....  | 54 |

|   |    |
|---|----|
| 6.3 (*) Uso básico de una base de datos SQLite.....                         | 54 |
| 7 Persistencia de datos en iOS: User Defaults y Core Data.....              | 56 |
| 7.1 Introducción.....   | 56 |
| 7.2 User Defaults.....  | 56 |
| 7.3 Core Data.....  | 60 |
| 8 Persistencia de datos en iOS: User Defaults y Core Data - Ejercicios..... | 76 |
| 8.1 Usando las variables de usuario (User Defaults).....                    | 76 |
| 8.2 Diseñando un modelo de datos sencillo con Core Data.....                | 76 |
| 8.3 (*) Migrando datos con Core Data.....                                   | 77 |

## 1. Persistencia en Android: ficheros y SQLite

### 1.1. Introducción

En este módulo trataremos diferentes mecanismos existentes en los sistemas Android e iOS para el almacenamiento de datos, tanto en la memoria del dispositivo como en memoria externa. Comenzaremos en primer lugar tratando el manejo de ficheros en Android para a continuación explicar cómo utilizar el motor de base de datos SQLite en nuestras aplicaciones en dicho sistema. A continuación presentaremos los proveedores de contenidos, los cuales nos proporcionan un mecanismo estándar para el intercambio de información entre aplicaciones Android. Veremos cómo usar los proporcionados por el sistema y crear los nuestros propios.

En las sesiones 3 y 4 comentaremos los métodos más usados para almacenar datos dentro de nuestras aplicaciones iOS. Empezaremos explicando de forma detallada el uso de los ficheros de propiedades (*plist*), un tipo de ficheros con formato XML que es muy usado en sistemas iOS. Continuaremos con el manejo de bases de datos SQLite dentro de las aplicaciones iOS, explicando la librería que disponemos dentro del SDK de iPhone. Por último, en la sesión 4 del módulo empezaremos explicando el método de almacenamiento mediante variables de tipo *User Defaults* para terminar con algo más complejo como es el uso de *Core Data*, forma de almacenamiento de datos muy extendida en estos últimos años dentro de los sistemas iOS.

### 1.2. Manejo de ficheros tradicionales en Android

El uso de ficheros tradicionales está permitido para los programas de Android, aunque como veremos en ésta y otras sesiones, hay otras tecnologías mucho más avanzadas, como el uso del motor de base de datos SQLite y, para el caso de preferencias de las aplicaciones, las *SharedPreferences*.

#### 1.2.1. Apertura de ficheros

En Android podemos abrir un fichero para lectura o para escritura de la siguiente forma:

```
// Abrir un fichero de salida privado a la aplicación
FileOutputStream fos = openFileOutput("fichero.txt",
Context.MODE_PRIVATE);
// Abrir un fichero de entrada
FileInputStream fis = openFileInput("fichero.txt");
```

Al abrir el fichero para salida, el modo `Context.MODE_PRIVATE` hace que éste sea privado a la aplicación. Si el fichero a abrir no existe, éste se crearía. Existe un parámetro para permitir compartir el fichero, pero lo adecuado en este caso sería hacer uso de proveedores de contenidos, por lo que no trataremos dicho parámetro.

Por otra parte, el parámetro `Context.MODE_APPEND` permite añadir contenido a un fichero que ya se hubiera creado previamente (o crear uno nuevo en el caso en el que éste no existiera).

Estos métodos tan sólo permiten abrir ficheros en la carpeta de la aplicación, por lo que si se intenta añadir algún separador de ruta se producirá una excepción. El fichero abierto utilizando `OpenFileOutput` (usando cualquiera de los dos parámetros anteriores) se guardaría, dentro del árbol de directorios del dispositivo, en la carpeta `/data/data/[paquete]/files/`, donde `[paquete]` sería el nombre del paquete de la aplicación. Se trata de ficheros de texto normales, por los que se puede acceder a ellos utilizando el shell de adb y utilizando el comando `cat`. Para acceder al shell podemos hacer lo siguiente:

```
adb shell
```

Por supuesto no debemos olvidarnos de cerrar el fichero una vez vayamos a dejar de utilizarlo mediante la correspondiente llamada al método `close()`:

```
fos.close();  
fis.close();
```

### 1.2.2. Ficheros como recursos

Es posible añadir ficheros como recursos de la aplicación. Para ello los almacenamos en la carpeta `/res/raw/` de nuestro proyecto. Estos ficheros serán de sólo lectura, y para acceder a ellos llamaremos al método `openRawResource` del objeto `Resource` de nuestra aplicación. Como resultado obtendremos un objeto de tipo `InputStream`. Un ejemplo sería el que se muestra a continuación:

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

Añadir ficheros a los recursos de la aplicación es un mecanismo para disponer de fuentes de datos estáticas de gran tamaño, como podría ser por ejemplo un diccionario. Esto será así en aquellos casos en los que no sea aconsejable (o deseable) almacenar esta información en una base de datos.

Como en el caso de cualquier otro recurso, sería posible guardar diferentes versiones de un mismo archivo para diferentes configuraciones del dispositivo. Podríamos tener por ejemplo un fichero de diccionario para diferentes idiomas.

### 1.2.3. Operar con ficheros

Una alternativa simple para escribir en un fichero o leer de él es utilizar las clases `DataInputStream` y `DataOutputStream`, que nos permiten leer o escribir a partir de diferentes tipos de datos, como enteros o cadenas. En el siguiente ejemplo vemos cómo es posible guardar una cadena en un fichero y a continuación leerla.

```

FileOutputStream fos = openFileOutput("fichero.txt",
Context.MODE_PRIVATE);
String cadenaOutput = new String("Contenido del fichero\n");
DataOutputStream dos = new DataOutputStream(fos);
dos.writeBytes(cadenaOutput);
fos.close();

FileInputStream fin = openFileInput("fichero.txt");
DataInputStream dis = new DataInputStream(fin);
String string = dis.readLine();
// Hacer algo con la cadena
fin.close();

```

#### 1.2.4. Almacenar datos en la tarjeta de memoria

Veamos ahora qué pasos debemos seguir para poder escribir datos en la tarjeta de memoria (SD card). Para ello se requiere el uso del método `Environment.getExternalStorageDirectory()` y una instancia de la clase `FileWriter`, tal como se muestra en el siguiente ejemplo:

```

try {
    File raiz = Environment.getExternalStorageDirectory();
    if (raiz.canWrite()){
        File file = new File(raiz, "fichero.txt");
        BufferedWriter out = new BufferedWriter(new
FileWriter(file));
        out.write("Mi texto escrito desde Android\n");
        out.close();
    }
} catch (IOException e) {
    Log.e("FILE I/O", "Error en la escritura de fichero: " +
e.getMessage());
}

```

Para probarlo en el emulador necesitamos tener creada una tarjeta SD, lo cual se puede conseguir mediante la herramienta del Android SDK `mksdcard`:

```
mksdcard 512M sdcard.iso
```

Podremos cargar esta tarjeta SD en nuestra terminal emulada seleccionando el archivo recién creado, tras seleccionar la opción adecuada, dentro del apartado SD card de la ventana de edición de las propiedades de una terminal emulada, la cual a su vez puede ser accedida mediante la opción Android SDK and AVD Manager bajo el menú Window. Para copiar fuera del emulador el archivo que hemos grabado en la tarjeta de memoria podemos utilizar el comando

```
adb pull /sdcard/fichero.txt fichero.txt
```

Por último, no debemos olvidar añadir el permiso correspondiente en el archivo Manifest de nuestra aplicación para poder escribir datos en un dispositivo de almacenamiento externo. Para ello añadimos lo siguiente al elemento manifest del Manifest.xml (recuerda que esto no va ni dentro del elemento application ni dentro del elemento activity):

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
```

```
/>
```

### 1.3. Base de datos SQLite

SQLite es un gestor de bases de datos relacional y es de código abierto, cumple con los estándares, y es extremadamente ligero. Además guarda toda la base de datos en un único fichero. Su utilidad es evidente en el caso de aplicaciones de pequeño tamaño, pues éstas no requerirán la instalación adicional de ningún gestor de bases de datos. Esto también será así en el caso de dispositivos empujados con recursos limitados. Android incluye soporte a SQLite, y en esta sección veremos cómo usarlo.

Por medio de SQLite podremos crear bases de datos relacionales independientes para nuestras aplicaciones. Con ellas podremos almacenar datos complejos y estructurados. Aunque el diseño de bases de datos quedará fuera del contenido de este curso, ya que necesitaríamos de más tiempo para poder tratarlo, es conveniente resaltar el hecho de que las mejores prácticas en este campo siguen siendo útiles en el caso de aplicaciones Android. En particular, la normalización de datos para evitar redundancia es un paso muy importante en el caso en el que estemos diseñando una base de datos para dispositivos con recursos limitados.

En el caso del sistema Android, SQLite se implementa como una librería C compacta, en lugar de ejecutarse en un proceso propio. Debido a ello, cualquier base de datos que creemos será integrada como parte de su correspondiente aplicación. Esto reduce dependencias externas, minimiza la latencia, y simplifica ciertos procesos como la sincronización.

**Nota:**

Las bases de datos de Android se almacenan en el directorio `/data/data/[PAQUETE]/databases/`, donde `[PAQUETE]` es el paquete correspondiente a la aplicación. Por defecto todas las bases de datos son privadas y tan sólo accesibles por la aplicación que las creó.

#### 1.3.1. Content Values

Para insertar nuevas filas en tablas haremos uso de objetos de la clase `ContentValues`. Cada objeto de este tipo representa una única fila en una tabla y se encarga de emparejar nombres de columnas con valores concretos. Hay que tener en cuenta que SQLite difiere de muchos otros motores de bases de datos en cuanto al tipado de las columnas. La idea básicamente es que los valores para las columnas no tienen por qué ser de un único tipo; es decir, los datos en SQLite son débilmente tipados. La consecuencia de esto es que no es necesario comprobar tipos cuando se asignan o extraen valores de cada columna de una fila, con lo que se mejora la eficiencia.

#### 1.3.2. Cursores

---

Los cursores son una herramienta fundamental en el tratamiento de información en Android, y también cuando se trabaja con bases de datos. Por lo tanto, en esta sección introducimos este concepto tan importante.

Cualquier consulta a una base de datos en Android devolverá un objeto de la clase `Cursor`. Los cursores no contienen una copia de todos los resultados de la consulta, sino que más bien se trata de punteros al conjunto de resultados. Estos objetos proporcionan un mecanismo para controlar nuestra posición (fila) en el conjunto de resultados obtenidos tras la consulta.

La clase `Cursor` incluye, entre otras, las siguientes funciones de navegación entre resultados:

- `moveToFirst`: desplaza el cursor a la primera fila de los resultados de la consulta.
- `moveToNext`: desplaza el cursor a la siguiente fila.
- `moveToPrevious`: desplaza el cursor a la fila anterior.
- `getCount`: devuelve el número de filas del conjunto de resultados de la consulta.
- `getColumnName`: devuelve el nombre de la columna especificada mediante un índice que se pasa como parámetro.
- `getColumnNames`: devuelve un array de cadenas conteniendo el nombre de todas las columnas en el cursor.
- `moveToPosition`: mueve el cursor a la fila especificada.
- `getPosition`: devuelve el índice de la posición apuntada actualmente por el cursor.

A lo largo de esta sesión aprenderemos cómo realizar consultas a una base de datos y como extraer valores o nombres de columnas concretos a partir de los cursores obtenidos por medio de dichas consultas.

### 1.3.3. Trabajar con bases de datos SQLite

---

Suele considerarse una buena práctica el crear una clase auxiliar que nos facilite la interacción con la base de datos. Se tratará de una clase de tipo adaptador, que creará una capa de abstracción que encapsulará dichas interacciones. De esta forma podremos añadir, eliminar o actualizar elementos en la base de datos sin necesidad de introducir ni una única instrucción en SQL en el resto del código, permitiendo además que se hagan las comprobaciones de tipos correspondientes.

Una clase de este tipo debería incluir métodos para crear, abrir o cerrar la base de datos. También puede ser un lugar conveniente para publicar constantes estáticas relacionadas con la base de datos, como por ejemplo constantes que almacenen el nombre de la tabla o las diferentes columnas.

A continuación se muestra el código de lo que podría ser un adaptador para una base de datos estándar. Incluye una extensión de la clase `SQLiteOpenHelper`, que será tratada en más detalle en la siguiente sección, y que se utiliza para simplificar la apertura, la creación y la actualización de la base de datos.

```

import android.content.Context;
import android.database.*;
import android.database.sqlite.*;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.util.Log;

public class MiAdaptadorBD {
    private static final String NOMBRE_BASE_DATOS =
"mibasededatos.db";
    private static final String TABLA_BASE_DATOS = "mainTable";
    private static final int VERSION_BASE_DATOS = 1;

    // Nombre de la columna de clave primaria
    public static final String CP_ID="_id";
    // El nombre e índice de cada columna en la base de datos
    public static final String COLUMNA_NOMBRE="nombre";
    public static final int COLUMNA_INDICE = 1;
    // PENDIENTE: crear campos adicionales para el resto de columnas
    // de la base de datos

    // Sentencia SQL para crear una nueva base de datos
    private static final String CREAR_BASE_DATOS = "create table " +
        TABLA_BASE_DATOS + " (" + CP_ID +
        " integer primary key autoincrement, " +
        COLUMNA_NOMBRE + " text not null);";

    // Variable para almacenar la instancia de la base de datos
    private SQLiteDatabase db;
    // Contexto de la aplicación que está usando la base de datos
    private final Context contexto;
    // Clase helper, usada para crear o actualizar la base de datos
    // (hablamos de ella en la siguiente sección)
    private miHelperBD dbHelper;

    // Crea la base de datos con ayuda del helper
    public MiAdaptadorBD(Context _contexto) {
        contexto = _contexto;
        dbHelper = new miHelperBD(contexto, NOMBRE_BASE_DATOS,
null,
        VERSION_BASE_DATOS);
    }

    // Abre una base de datos ya existente
    public MiAdaptadorBD open() throws SQLException {
        db = dbHelper.getWritableDatabase();
        return this;
    }

    // Cierra la base de datos, cuando ésta ya no va a ser utilizada
    public void close() {
        db.close();
    }

    // Método para insertar un elemento en la tabla de la base de
datos
    public int insertar(MiObjeto _miObjeto) {
        // PENDIENTE: crear nuevos elementos ContentValues para
representar
        // al objeto e insertarlos en la base de datos
        // Devolvemos el índice del nuevo elemento en la base de
datos
        return index;
    }
}

```



```

// Método para eliminar un elemento de la tabla de la base de
datos
// Devuelve un valor booleano indicando el éxito o no de la
// operación
public boolean eliminar(long _indiceFila) {
    return db.delete(TABLA_BASE_DATOS, CP_ID + "=" +
        _indiceFila, null) > 0;
}

// Método para obtener un Cursor que apunte a todas las filas
contenidas
// en la tabla de la base de datos
public Cursor obtenerTodos () {
    return db.query(TABLA_BASE_DATOS, new String[] {CP_ID,
        COLUMNA_NOMBRE},
        null, null, null, null,
        null);
}

// Método para obtener un elemento determinado de la base de datos
// a partir de su índice de fila
public MiObjeto getObjeto(long _indiceFila) {
    // PENDIENTE: obtener un cursor a una fila de la base
    // de datos y usar sus valores para inicializar una
    // instancia de MiObjeto

    return objectInstance;
}

// Método para actualizar un elemento de la tabla de la base de
datos
public boolean actualizar(long _indiceFila, MiObjeto _miObjeto) {
    // PENDIENTE: crear nuevos ContentValues a partir del
    // y usarlos para actualizar una fila en la tabla
    // correspondiente

    return true;
}

// Clase privada auxiliar para ayudar en la creación y
actualización
// de la base de datos
private static class miHelperBD extends SQLiteOpenHelper {

    public miHelperBD(Context contexto, String nombre,
        CursorFactory factory, int version) {
        super(contexto, nombre, factory, version);
    }

    // Método llamado cuando la base de datos no existe en el
    // y la clase Helper necesita crearla desde cero
    @Override
    public void onCreate(SQLiteDatabase _db) {
        _db.execSQL(CREAR_BASE_DATOS);
    }

    // Método llamado cuando la versión de la base de
    // datos en disco es diferente a la indicada; en
    // este caso la base de datos en disco necesita ser
    // actualizada
    @Override
    public void onUpgrade(SQLiteDatabase _db, int
        _versionAnterior,

```

```

        int _versionNueva) {
            // Mostramos en LogCat la operación
            Log.w("AdaptadorBD", "Actualizando desde la
versión " +
                                _versionAnterior + " a la versión " +
                                _versionNueva + ", lo cual borrará los
datos
                                previamente almacenados");
            // Actualizamos la base de datos para que se
adapte a la nueva versión.
            // La forma más sencilla de llevar a cabo dicha
actualización es
            // eliminar la tabla antigua y crear una nueva
            _db.execSQL("DROP TABLE IF EXISTS " +
TABLA_BASE_DATOS);
            onCreate(_db);
        }
    }
}

```

### 1.3.4. La clase SQLiteOpenHelper

La clase `SQLiteOpenHelper` es una clase abstracta utilizada como medio para implementar un patrón de creación, apertura y actualización de una determinada base de datos. Al implementar una subclase de `SQLiteOpenHelper` podemos abstraernos de la lógica subyacente a la decisión de crear o actualizar una base de datos de manera previa a que ésta deba ser abierta.

En el ejemplo de código anterior se vio cómo crear una subclase de `SQLiteOpenHelper` por medio de la sobrecarga de sus métodos `onCreate` y `onUpgrade`, los cuales permiten crear una nueva base de datos o actualizar a una nueva versión, respectivamente.

#### Nota:

En el ejemplo anterior, el método `onUpgrade` simplemente eliminaba la tabla existente y la reemplazaba con la nueva definición de la misma. En la práctica la mejor solución sería migrar los datos ya existentes a la nueva tabla.

Mediante los métodos `getReadableDatabase` y `getWritableDatabase` podemos abrir y obtener una instancia de la base de datos de sólo lectura o de lectura y escritura respectivamente. La llamada a `getWritableDatabase` podría fallar debido a falta de espacio en disco o cuestiones relativas a permisos, por lo que es una buena práctica tener en cuenta esta posibilidad por medio del mecanismo de manejo de excepciones de Java. En caso de fallo la solución podría pasar por al menos proporcionar una copia de sólo lectura, tal como se muestra en el siguiente ejemplo:

```

dbHelper = new miHelperBD(contexto, NOMBRE_BASE_DATOS, null,
VERSION_BASE_DATOS);

SQLiteDatabase db;
try {
    db = dbHelper.getWritableDatabase();
} catch (SQLiteException ex) {
    db = dbHelper.getReadableDatabase();
}

```

```
}
```

Al hacer una llamada a cualquiera de los dos métodos anteriores se invocarán los manejadores de evento adecuados. Así pues, si por ejemplo la base de datos no existiera en disco, se ejecutaría el código de `onCreate`. Si por otra parte la versión de la base de datos se hubiera modificado, se dispararía el manejador `onUpgrade`. Lo que sucede entonces es que al hacer una llamada a `getWritableDatabase` o `getReadableDatabase` se devolverá una base de datos nueva, actualizada o ya existente, según el caso.

### 1.3.5. Crear una base de datos sin SQLiteHelper

También es posible crear y abrir bases de datos sin necesidad de utilizar una subclase de `SQLiteHelper`. Para ello haremos uso del método `openOrCreateDatabase`. En este caso será necesario también hacer uso del método `execSQL` sobre la instancia de la base de datos devuelta por el método anterior para ejecutar los comandos SQL que permitirán crear las tablas de la base de datos y establecer las relaciones entre ellas. El siguiente código muestra un ejemplo:

```
private static final String NOMBRE_BASE_DATOS = "mibasededatos.db";
private static final String TABLA_BASE_DATOS = "tablaPrincipal";

private static final String CREAR_BASE_DATOS =
    "create table " + TABLA_BASE_DATOS + " /_id integer primare key
    autoincrement," +
    "columna_uno text not null);";

SQLiteDatabase db;

private void crearBaseDatos() {
    db = openOrCreateDatabase(NOMBRE_BASE_DATOS, Context.MODE_PRIVATE,
    null);
    db.execSQL(CREAR_BASE_DATOS);
}
```

#### Nota:

Aunque no es necesariamente obligatorio, es recomendable que cada tabla incluya un campo de tipo clave primaria con autoincremento a modo de índice para cada fila. En el caso en el que se desee compartir la base de datos mediante un proveedor de contenidos, un campo único de este tipo es obligatorio.

### 1.3.6. Realizar una consulta

El resultado de cualquier consulta a una base de datos será un objeto de la clase `Cursor`. Esto permite a Android manejar los recursos de manera más eficiente, de tal forma que en lugar de devolver todos los resultados, éstos se van proporcionando conforme se necesitan.

Para realizar una consulta a una base de datos haremos uso del método `query` de la instancia correspondiente de la clase `SQLiteDatabase`. Los parámetros que requiere este

método son los siguientes:

- Un booleano opcional que permite indicar si el resultado debería contener tan sólo valores únicos.
- El nombre de la tabla sobre la que realizar la consulta.
- Un array de cadenas que contenta un listado de las columnas que deben incluirse en el resultado.
- Una cláusula *where* que defina las filas que se deben obtener de la tabla. Se puede utilizar el comodín *?*, el cual será reemplazado por los valores que se pasen a través del siguiente parámetro.
- Un array de cadenas correspondientes a argumentos de selección que reemplazarán los símbolos *?* en la cláusula *where*.
- Una cláusula *group by* que define cómo se agruparán las filas obtenidas como resultado.
- Un filtro *having* que indique que grupos incluir en el resultado en el caso en el que se haya hecho uso del parámetro anterior.
- Una cadena que describa la ordenación que se llevará a cabo sobre las filas obtenidas como resultado.
- Una cadena opcional para definir un límite en el número de resultados a devolver.

**Nota:**

Como se puede observar, la mayoría de estos parámetros hacen referencia al lenguaje SQL. Tratar este tema queda fuera de los objetivos de este curso, por lo que en nuestros ejemplos le daremos en la mayoría de las ocasiones un valor *null* a estos parámetros.

El siguiente código muestra un ejemplo de consultas utilizando diversos parámetros:

```
// Devolver los valores de las columnas uno y tres para todas las
// filas, sin duplicados
String[] columnas = new String[] {CLAVE_ID, CLAVE_COL1, CLAVE_COL3};

Cursor todasFilas = db.query(true, TABLA_BASE_DATOS, columnas, null, null,
    null, null,
    null, null);

// Devolvemos los valores de todas las columnas para aquellas filas cuyo
// valor de la
// columna 3 sea igual al de una determinada variable. Ordenamos las filas
// según el
// valor de la columna 5
// Como queremos los valores de todas las columnas, le damos al parámetro
// correspondiente a las columnas a devolver el valor null
// En este caso no se ha hecho uso del primer parámetro booleano, que es
// optativo
String where = CLAVE_COL3 + "=" + valor;
String orden = CLAVE_COL5;
Cursor miResultado = db.query(TABLA_BASE_DATOS, null, where, null, null,
    null, orden);
```

### 1.3.7. Extraer resultados de un cursor

El primer paso para obtener resultados de un cursor será desplazarnos a la posición a partir de la cual queremos obtener los datos, usando cualquiera de los métodos especificados anteriormente (como por ejemplo `moveToFirst` o `moveToPosition`). Una vez hecho esto hacemos uso de métodos `get[TIPO]` pasando como parámetro el índice de la columna. Esto tendrá como resultado la devolución del valor para dicha columna de la fila actualmente apuntada por el cursor:

```
String valor = miResultado.getString(indiceColumna);
```

A continuación podemos ver un ejemplo más completo, en el que se va iterando a lo largo de los resultados apuntados por un cursor, extrayendo y sumando los valores de una columna de flotantes:

```
int COLUMNA_VALORES_FLOTANTES = 2;
Cursor miResultado = db.query("Tabla", null, null, null, null, null, null);
float total = 0;

// Nos aseguramos de que se haya devuelto al menos una fila
if (miResultado.moveToFirst()) {
    // Iteramos sobre el cursor
    do {
        float valor =
miResultado.getFloat(COLUMNA_VALORES_FLOTANTES);
        total += valor;
    } while (miResultado.moveToNext());
}

float media = total / miResultado.getCount();
```

#### Nota:

Debido a que las columnas de las bases de datos en SQLite son debilmente tipadas, podemos realizar castings cuando sea necesario. Por ejemplo, los valores guardados como flotantes en la base de datos podrían leerse más adelante como cadenas.

### 1.3.8. Añadir, actualizar y borrar filas

La clase `SQLiteDatabase` proporciona los métodos `insert`, `delete` y `update`, que encapsulan las instrucciones SQL requeridas para llevar a cabo estas acciones. Además disponemos de la posibilidad de utilizar el método `execSQL`, el cual nos permite ejecutar cualquier sentencia SQL válida en nuestras tablas de la base de datos en el caso en el que queramos llevar a cabo estas (u otras) operaciones manualmente.

#### Nota:

Cada vez que modifiques los valores de la base de datos puedes actualizar un cursor que se pueda ver afectado por medio del método `refreshQuery` de la clase `Cursor`.

Para **insertar** una nueva fila es necesario construir un objeto de la clase `ContentValues` y usar su método `put` para proporcionar valor a cada una de sus columnas. Para insertar la

nueva fila pasaremos como parámetro este objeto del tipo `ContentValues` al método `insert` de la instancia de la base de datos, junto por supuesto con el nombre de la tabla. A continuación se muestra un ejemplo:

```
// Crear la nueva fila
ContentValues nuevaFila = new ContentValues();

// Asignamos valores a cada columna
nuevaFila.put(NOMBRE_COLUMNNA, nuevoValor);
[... Repetir para el resto de columnas ...]

// Insertar la nueva fila en la tabla
db.insert(TABLA_BASE_DATOS, null, nuevaFila);
```

**Nota:**

Los ficheros, como imágenes o ficheros de audio, no se suelen almacenar dentro de tablas en una base de datos. En estos casos se suele optar por almacenar en la base de datos una cadena con la ruta al fichero o una URI.

La operación de **actualizar** una fila también puede ser llevada a cabo por medio del uso de la clase `ContentValues`. En este caso deberemos llamar al método `update` de la instancia de la base de datos, pasando como parámetro el nombre de la tabla, el objeto `ContentValues` y una cláusula *where* que especifique la fila o filas a actualizar, tal como se puede ver en el siguiente ejemplo:

```
// Creamos el objeto utilizado para definir el contenido a actualizar
ContentValues valoresActualizar = new ContentValues();

// Asignamos valores a las columnas correspondientes
valoresActualizar.put(NOMBRE_COLUMNNA, nuevoValor);
[... Repetir para el resto de columnas a actualizar ...]

String where = CLAVE_ID + "=" + idFila;

// Actualizamos la fila con el índice especificado en la instrucción
anterior
db.update(TABLA_BASE_DATOS, valoresActualizar, where, null);
```

Por último, para **borrar** una fila, simplemente llamaremos al método `delete` de la instancia de la base de datos, especificando en este caso el nombre de la tabla que se verá afectada por la operación y una cláusula *where* que utilizaremos para especificar las filas que queremos borrar. Un ejemplo podría ser el siguiente:

```
db.delete(TABLA_BASE_DATOS, CLAVE_ID + "=" + idFila, null);
```

## 2. Persistencia en Android: ficheros y SQLite - Ejercicios

### 2.1. Uso de ficheros

En este ejercicio vamos a crear una aplicación que muestre un listado de cadenas por pantalla. Estas cadenas se almacenarán en un fichero de texto privado para la aplicación. Podremos añadir nuevas cadenas a partir de un cuadro de edición presente en la interfaz. Partiremos del proyecto *Ficheros* proporcionado en las plantillas de la aplicación. Debes seguir los siguientes pasos:

- Añadir un manejador al botón de la actividad principal para que cada vez que sea pulsado se guarde en un fichero de texto. El fichero se llamará `fichero.txt`. Al abrir el fichero para escritura se utilizará el parámetro `Context.MODE_APPEND`, con lo que cada nueva cadena se añadirá al final del fichero en el caso en el que éste ya existiera. Para escribir en el fichero utilizaremos el método `writeBytes` del objeto `DataOutputStream` correspondiente.
- Al pulsar el botón también se deberá borrar el contenido del elemento `EditText` (le asignamos a la vista una cadena vacía con el método `setText`).
- Ejecutamos la aplicación e introducimos algunas líneas en el fichero. Vamos a comprobar ahora que todo ha funcionado correctamente. Para ello accedemos al sistema de ficheros de nuestro dispositivo ejecutando el comando `./adb shell` dentro de la carpeta *platform-tools* de nuestra carpeta de instalación del SDK de Android.
- Comprueba que se ha creado el fichero *fichero.txt* en la carpeta `/data/data/es.ua.jtech.android.ficheros/files/`. Examina su contenido ejecutando `cat fichero.txt`. Si algo no ha salido bien siempre podrás eliminar el fichero con `rm fichero.txt`.
- En la interfaz de la actividad se ha incluido una vista de tipo `TextView` debajo del botón. Añade un manejador para dicha vista, de tal forma que cada vez que se haga click sobre ella se muestre el contenido del fichero *fichero.txt*. Para ello leeremos el fichero línea a línea, añadiendo cada una al `TextView` por medio del método `append`.



Interfaz de la aplicación Ficheros

## 2.2. Persistencia con ficheros (\*)

Seguramente habrás observado que si abandonas la aplicación anterior (pulsando el botón *BACK* del dispositivo), al volver a ejecutarla el contenido del `TextView` ha desaparecido. Debes volver a pulsar sobre la vista para que se vuelva a mostrar el contenido del fichero. Haz las modificaciones pertinentes para que esto no sea así; es decir, para que cuando vuelva a mostrarse la actividad tras haber sido eliminada de la memoria se muestre el `TextView` tal cual se veía antes de abandonar la aplicación.

**Aviso:**

En este ejercicio no se está pidiendo que cargues el contenido del fichero en el `TextView` nada más arrancar la aplicación. Puede darse el caso de que lo que esté mostrando el `TextView` cuando la actividad sea eliminada de memoria no se corresponda con el contenido actualizado del fichero.

**Nota:**

Para poder completar el ejercicio deberás repasar los manejadores de evento de la primera sesión del módulo de Android relacionados con el ciclo de vida de ejecución de actividades.

## 2.3. Base de datos: SQLiteOpenHelper

En las plantillas de la aplicación se incluye la aplicación *BaseDatos*. En el proyecto de la aplicación se incluye el esqueleto de una clase `MiAdaptadorBD` que tendremos que completar. Se trata de un patrón que nos va a permitir acceder a una base de datos de



usuarios dentro de la aplicación sin necesidad de hacer uso de código SQL.

La clase `MiAdaptadorBD` incluye a su vez otro patrón, en este caso implementado como una subclase de `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.

En este primer ejercicio se pide hacer lo siguiente:

- 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 tabla que tenemos con la sentencia SQL `"DROP TABLE IF EXISTS " + NOMBRE_TABLA` y volveremos a crearla.
- En el constructor de `MiAdaptadorBD` debemos obtener en el campo `db` la base de datos, utilizando `MiOpenHelper`.

## 2.4. Base de datos: inserción y borrado

Continuamos trabajando con el proyecto anterior. En este ejercicio completaremos el código relacionado con las sentencias de inserción y borrado. Para realizar la inserción vamos a hacer uso de un mecanismo que no hemos visto en la sesión de teoría, y que consiste en utilizar una sentencia de inserción SQL precompilada que se completará con los valores concretos a insertar en la base de datos justo antes de realizar dicha inserción. La ventaja de las sentencias compiladas es que evitan que se produzcan ataques de inyección de código.

Como se puede observar, se ha incluido un atributo a la clase que representa la inserción SQL:

```
private static final String INSERT = "insert into " + NOMBRE_TABLA +
    "(" + COLUMNAS[1] + ")" values (?)";
```

En esta sentencia no se indica ningún valor concreto para la columna *nombre*. En el lugar en el que deberían aparecer los valores de dicho campo se ha escrito simplemente un símbolo de interrogación. También se ha añadido una instancia de la clase `SQLiteStatement` como parte de los atributos de la clase:

```
private SQLiteStatement insertStatement;
```

Realizamos los siguientes pasos:

- En el constructor de la clase `MiAdaptadorBD` llevamos a cabo la compilación de la sentencia:

```
this.insertStatement = this.db.compileStatement(INSERT);
```

- Una vez hecho esto, cada vez que deseemos insertar un nuevo usuario en la base de datos, deberemos dar valores concretos a la columna *nombre*, y ejecutar la sentencia. Para ello debemos añadir el siguiente código dentro del método `insert` de la clase `MiAdaptadorBD`:

```
this.insertStatement.bindString(1, nombreUsuario);
return this.insertStatement.executeInsert();
```

- Por último completamos el código del método `deleteAll`, cuyo contenido es eliminar a todos los usuarios de la base de datos. Para que además se devuelva el número de filas afectadas podemos insertar la siguiente línea en dicho método:

```
return db.delete(NOMBRE_TABLA, null, null);
```

Una vez hechos todos estos cambios podremos utilizar la clase `MiAdaptadorBD` para hacer operaciones con la base de datos de usuarios de manera transparente.

## 2.5. Base de datos: probar nuestro adaptador

Añade código en la actividad `Main` para eliminar todos los usuarios de la base de datos, añadir dos cualesquiera, y listarlos por medio de la vista de tipo `TextView` que se encuentra en el layout de dicha actividad.

Podemos comprobar mediante la línea de comandos (comando `./adb shell`) que la base de datos ha sido creada. Para ello puede ser útil hacer uso de los siguientes comandos:

```
#cd /data/data/es.ua.jtech.android.basedatos/databases
#sqlite3 misusuarios.db
sqlite> .schema
sqlite> .tables
sqlite> select * from usuarios;
```

## 2.6. Base de datos: cambios en la base de datos (\*)

Ahora vamos a cambiar en la clase `MiAdaptadorBD` el nombre de la segunda columna, que en lugar de *nombre* se va a llamar *nombres*. Ejecutamos la aplicación y comprobamos que sale con una excepción. Comprueba por medio de LogCat cuál ha sido el error. ¿Cómo lo podemos solucionar?

### Nota:

Pista: conforme hemos programado la clase `MiAdaptadorBD` y siguiendo el patrón de diseño de `SQLiteOpenHelper`, es posible arreglar el problema simplemente modificando el valor de un campo.

### 3. Persistencia en Android: proveedores de contenidos y SharedPreferences

#### 3.1. Shared Preferences

Comenzamos esta sesión hablando de una de las técnicas más simples de persistencia en Android junto al uso de ficheros: *Shared Preferences*. Se trata de un mecanismo ligero para almacenar datos basado en pares clave-valor, utilizado normalmente para guardar preferencias u opciones de la aplicación. También puede ser utilizado para almacenar el estado de la interfaz gráfica para contemplar el caso de que nuestra actividad deba finalizar abruptamente su ejecución. Otro posible uso es el de compartir información entre componentes de una misma aplicación.

**Nota:**

Los *Shared Preferences* nunca son compartidos entre diferentes aplicaciones.

Este método se basa en el uso de la clase `SharedPreferences`. Es posible utilizarla para guardar datos en muy diversos formatos: booleanos, cadenas, flotantes y enteros.

##### 3.1.1. Guardar Shared Preferences

Para crear o modificar un *Shared Preference* llamamos al método `getSharedPreferences` del contexto de la aplicación, pasando como parámetro el identificador del conjunto de valores de preferencias con el que queremos trabajar. Para realizar la modificación del valor de un *Shared Preference* hacemos uso de la clase `SharedPreferences.Editor`. Para obtener el objeto editor invocamos el método `edit` del objeto *Shared Preferences* que queremos modificar. Los cambios se guardan mediante el método `commit`. Veamos un ejemplo:

```
int modo = Activity.MODE_PRIVATE;
SharedPreferences misSharedPreferences = getSharedPreferences("Mis
preferencias", modo);

// Obtenemos un editor para modificar las preferencias
SharedPreferences.Editor editor = misSharedPreferences.edit();

// Guardamos nuevos valores en el objeto SharedPreferences
editor.putBoolean("isTrue",true);
editor.putFloat("unFloat",1.0);
editor.putInt("numeroEntero",12);
editor.putLong("otroNumero",2);
editor.putString("unaCadena","valor");

// Guardamos los cambios
editor.commit();
```

### 3.1.2. Leer Shared Preferences

Para leer *Shared Preferences* utilizamos el método `getSharedPreferences`, tal como se ha comentado anteriormente. Pasamos como parámetro el identificador del conjunto de *Shared Preferences* al que deseamos acceder. Una vez hecho esto ya podemos hacer uso de métodos de tipo `get` para acceder a preferencias individuales. Hay diferentes métodos `get` para diferentes tipos de datos. Cada uno de ellos pasamos como parámetro la clave que identifica la preferencia cuyo valor deseamos obtener, y un valor por defecto, el cual se usará en el caso concreto en el que no existiera una preferencia con la clave pasada como primer parámetro. Podemos ver un ejemplo a continuación:

```
public static String MIS_PREFS = "MIS_PREFS";

public void cargarPreferences() {
    // Obtener las preferencias almacenadas
    int modo = Activity.MODE_PRIVATE;
    SharedPreferences misSharedPreferences =
    getSharedPreferences(MIS_PREFS, modo);

    boolean isTrue = misSharedPreferences.getBoolean("isTrue", false);
    float unFloat = misSharedPreferences.getFloat("unFloat", 0);
    int numeroEntero = misSharedPreferences.getInt("numeroEntero", 0);
    long otroNumero = misSharedPreferences.getLong("otroNumero", 0);
    String unaCadena = misSharedPreferences.getString("unaCadena", "");
}
```

### 3.1.3. Interfaces para Shared Preferences

Android proporciona una plataforma basada en XML para la creación de interfaces gráficas para el manejo de preferencias. Estas interfaces tendrán un aspecto similar al de las del resto de aplicaciones del sistema. Al utilizarla, nos estaremos asegurando de que nuestras actividades para el manejo de preferencias serán consistentes con las del resto de aplicaciones del sistema. De esta forma los usuarios estarán familiarizados con el manejo de esta parte de nuestra aplicación.

La plataforma consiste en tres elementos:

- **Layout de la actividad de preferencias:** se trata de un archivo XML que definirá la interfaz gráfica de la actividad que muestre los controles para modificar los valores de las preferencias. Permite especificar las vistas a mostrar, los posibles valores que se pueden introducir, y a qué clave de *Shared Preferences* se corresponde cada vista.
- **Actividad de preferencias:** una subclase de `PreferenceActivity` que se corresponderá con la pantalla de preferencias de nuestra aplicación.
- **Shared Preference Change Listener:** una implementación de la clase `onSharedPreferencesChangeListener` que actuará en el caso en el que cambie el valor de alguna *Shared Preference*.

### 3.1.4. Definiendo una pantalla de preferencias con un layout en XML

Esta es sin duda la parte más importante de una actividad de preferencias. Se trata de un archivo XML utilizado para definir varios aspectos de dicha actividad. Al contrario que en el caso de los archivos de layout para interfaces gráficas que vimos en la sesión correspondiente del módulo de Android, estos archivos de layout se guardan en la carpeta `/res/xml/` de los recursos de la aplicación.

Aunque conceptualmente estos layouts son similares a los utilizados para definir interfaces gráficas, los layouts de actividades de preferencias utilizan un conjunto especializado de componentes específicos para este tipo de actividades. Están pensados para proporcionar interfaces con un aspecto similar al del resto de actividades de preferencias del sistema. Estos controles se describen en más detalle en la siguiente sección.

Los layout de preferencias contendrán un elemento `PreferenceScreen`:

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

Se pueden añadir más elementos de este tipo si se desea. Al hacerlo, éstos se representarán como un elemento seleccionable en un listado que mostrará una nueva ventana de preferencias en caso de que sea seleccionado.

Dentro de cada elemento `PreferenceScreen` podemos incluir tantos elementos de tipo `PreferenceCategory` y elementos específicos para añadir controles como se desee. Los elementos `PreferenceCategory` son utilizados para dividir cada pantalla de preferencias en subcategorías mediante una barra de título. Su sintaxis es la siguiente:

```
<PreferenceCategory
    android:title="My Preference Category"/>
</PreferenceCategory>
```

Una vez dividida la pantalla en subcategorías tan sólo quedaría añadir los elementos correspondientes a los controles específicos, que veremos en la siguiente sección. Los atributos de los elementos XML para estos controles pueden variar, aunque existe un conjunto de atributos comunes:

- `android:key`: la clave de Shared Preference que se usará para guardar el valor del control.
- `android:title`: texto a mostrar para describir la preferencia.
- `android:summary`: una descripción más larga a mostrar debajo del texto definido con el campo anterior.
- `android:defaultValue`: el valor por defecto que se mostrará inicialmente y también el que finalmente se guardará si ningún otro valor fue asignado a este campo.

El siguiente listado muestra una pantalla de preferencias muy simple con una única categoría y un único control (un check box):

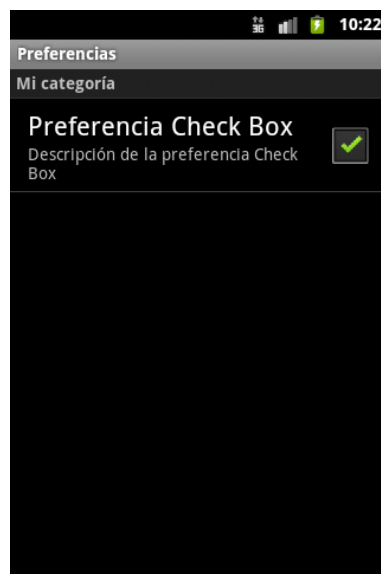
```
<?xml version="1.0" encoding="utf-8"?>
```

```

<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:title="Mi categoría">
        <CheckBoxPreference
            android:key="MI_CHECK_BOX"
            android:title="Preferencia Check Box"
            android:summary="Descripción de la preferencia
Check Box"
            android:defaultValue="true"
        />
    </PreferenceCategory>
</PreferenceScreen>

```

Esta pantalla de preferencias se mostraría de la siguiente forma:



Ejemplo sencillo de ventana de preferencias

### 3.1.5. Controles nativos para preferencias

Android incluye diferentes controles que podemos añadir a nuestras ventanas de preferencias por medio de los elementos XML que se indican a continuación:

- **CheckBoxPreference:** un check box estándar que puede ser utilizado para preferencias de tipo booleano.
- **EditTextPreference:** permite al usuario introducir una cadena como valor para una preferencia. Al seleccionar el texto se mostrará un diálogo con el que introducir el nuevo valor.
- **ListPreference:** el equivalente a un Spinner. Seleccionar este elemento de la interfaz mostrará un diálogo con las diferentes opciones que es posible seleccionar. Se utilizan arrays para asociar valores y textos a las opciones de la lista.
- **RingtonePreference:** un tipo específico de preferencia de tipo listado que permite seleccionar entre diferentes tonos de teléfono. Esta opción es particularmente útil en

el caso en el que se esté creando una pantalla de preferencias para configurar las opciones de notificación de una aplicación o componente de la misma.

También es posible crear nuestros propios controles personalizados definiendo una subclase de `Preference` (o cualquiera de sus subclases). Se puede encontrar más información en la documentación de Android: <http://developer.android.com/reference/android/preference/Preference.html>.

### 3.1.6. Actividades de preferencias

Para mostrar una pantalla de preferencias debemos definir una subclase de `PreferenceActivity`:

```
public class MisPreferencias extends PreferenceActivity {
```

La interfaz gráfica de la interfaz se puede crear a partir del layout en el método `onCreate`, mediante una llamada a `addPreferencesFromResource`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.milayout);
}
```

Como en el caso de cualquier otra actividad, la actividad de preferencias que hayamos definido debe ser incluida en el *Manifest* de la aplicación:

```
<activity android:name=".MisPreferencias"
    android:label="Mis preferencias">
</activity>
```

Eso es todo lo que necesitamos para crear este tipo de actividades y mostrar una ventana con las preferencias definidas en el layout. Esta actividad puede ser iniciada como cualquier otra mediante un `Intent`, ya sea a través de una llamada a `startActivity` o bien a `startActivityForResult`:

```
Intent i = new Intent(this, MisPreferencias.class);
startActivityForResult(i, MOSTRAR_PREFERENCIAS);
```

Los valores para las diferentes preferencias de la actividad son almacenadas en el contexto de la aplicación. Esto permite a cualquier componente de dicha aplicación, incluyendo a cualquier actividad y servicio de la misma, acceder a los valores tal como se muestra en el siguiente código de ejemplo:

```
Context contexto = getApplicationContext();
SharedPreferences prefs =
    PreferenceManager.getDefaultSharedPreferences(contexto);
// PENDIENTE: hacer uso de métodos get para obtener los valores
```

### 3.1.7. Shared Preference Change Listeners

El último elemento que nos queda por examinar en esta plataforma de *Shared Preferences* es la interfaz `onSharedPreferenceChangeListener`, que es utilizada para invocar un evento cada vez que una preferencia es añadida, eliminada o modificada. Para registrar *Listeners* usamos un código como el que se muestra a continuación, en el que la parte más importante es sin duda el método `onSharedPreferenceChanged`, dentro del cual deberemos determinar qué preferencia ha cambiado de estado a partir de sus parámetros para llevar a cabo las acciones oportunas:

```
public class MiActividad extends Activity implements
OnSharedPreferenceChangeListener {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        // Registramos este objeto
        OnSharedPreferenceChangeListener
        Context contexto = getApplicationContext();
        SharedPreferences prefs =
        PreferenceManager.getDefaultSharedPreferences(contexto);
        prefs.registerOnSharedPreferenceChangeListener(this);
    }

    public void onSharedPreferenceChanged(SharedPreferences prefs,
String clave) {
        // PENDIENTE: comprobar qué clave concreta ha cambiado y
su nuevo valor
        // para modificar la interfaz de una actividad o el
comportamiento de
        // algún componente
    }
}
```

## 3.2. Proveedores de contenidos

Los proveedores de contenidos o `ContentProvider` proporcionan una interfaz para publicar y consumir datos, identificando la fuente de datos con una dirección URI que empieza por `content://`. Son una forma más estándar que los adaptadores (como el que vimos en la sesión anterior en la sección dedicada a `SQLite`) de desacoplar la capa de aplicación de la capa de datos.

Podemos encontrar dos tipos principales de proveedores de contenidos en Android. Los proveedores nativos son proveedores que el propio sistema nos proporciona para poder acceder a datos del dispositivo, incluyendo audio, vídeo, imágenes, información personal, etc. Por otra parte, también es posible que creamos nuestros propios proveedores con tal de permitir a nuestra aplicación compartir datos con el resto del sistema. En esta sesión hablaremos de ambos tipos.

### 3.2.1. Proveedores nativos

Android incluye una serie de proveedores de contenidos que nos pueden proporcionar información de diferentes tipos: información sobre nuestros contactos, información del calendario e incluso ficheros multimedia. Ejemplos de estos proveedores de contenidos



nativos son el Browser, CallLog, ContactsContract, MediaStore, Settings y el UserDictionary. Para poder hacer uso de estos proveedores de contenidos y acceder a los datos que nos proporcionan debemos añadir los permisos adecuados al fichero AndroidManifest.xml. Por ejemplo, para acceder al listín telefónico podemos añadir el permiso correspondiente de la siguiente forma:

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

Para obtener información de un ContentProvider debemos hacer uso del método query de la clase ContentResolver. El primero de los parámetros de dicho método será la URI del proveedor de contenidos al que deseamos acceder. Este método devuelve un cursor que nos permitirá iterar entre los resultados, tal como se vio en la sesión anterior en el apartado de cursores:

```
ContentResolver.query(
    Uri uri,
    String[] projection,
    String selection,
    String[] selectionArgs,
    String sortOrder)
```

Por ejemplo, la siguiente llamada nos proporcionará un cursor que nos permitirá acceder a todos los contactos de nuestro teléfono móvil. Para ello hemos hecho uso de la constante ContactsContract.Contacts.CONTENT\_URI, que almacena la URI del proveedor de contenidos correspondiente. Al resto de parámetros se le ha asignado el valor null:

```
ContentResolver cr = getContentResolver();
Cursor cursor = cr.query(ContactsContract.Contacts.CONTENT_URI,
    null, null, null, null);
```

**Nota:**

En versiones anteriores a Android 2.0 las estructuras de datos de los contactos son diferentes y no se accede a esta URI.

Una vez obtenido el cursor es posible mapear sus campos con algún componente de la interfaz gráfica de la actividad. De esta forma cualquier cambio que se produzca en el cursor se reflejará automáticamente en el componente gráfico sin que sea necesario que programemos el código que lo refresque. Esto se consigue con el siguiente método:

```
cursor.setNotificationUri(cr,
    ContactsContract.Contacts.CONTENT_URI);
```

Para asignar un adaptador a una lista de la interfaz gráfica de la actividad, más

concretamente una `ListView`, utilizamos el método `setAdapter(Adapter)`:

```
ListView lv = (ListView)findViewById(R.id.ListView01);
SimpleCursorAdapter adapter = new SimpleCursorAdapter(
    getApplicationContext(),
    R.layout.textviewlayout,
    cursor,
    new String[]{
        ContactsContract.Contacts._ID,
        ContactsContract.Contacts.DISPLAY_NAME},
    new int[]{
        R.id.TextView1,
        R.id.TextView2});
lv.setAdapter(adapter);
```

En este ejemplo los identificadores `R.id.TextView1` y `R.id.TextView2` se corresponden con views del layout que define cada fila de la `ListView`, como se puede ver en el archivo `textviewlayout.xml` que tendríamos que crear:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <TextView android:id="@+id/TextView1"
        android:textStyle="bold"
        android:ems="2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
    </TextView>
    <TextView android:id="@+id/TextView2"
        android:textStyle="bold"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
    </TextView>
</LinearLayout>
```

La lista en sí (identificada por `R.id.ListView01` en el presente ejemplo) estaría en otro XML layout, por ejemplo en `main.xml`.

Se debe tener en cuenta que en el caso de este ejemplo, si quisiéramos obtener los números de teléfono de cada uno de nuestros contactos, tendríamos que recorrer el cursor obtenido a partir del proveedor de contenidos, y por cada persona en nuestra agenda, crear un nuevo cursor que recorriera los teléfonos, ya que una persona puede tener asignados varios teléfonos.

Puedes acceder a un listado completo de los proveedores de contenidos nativos existentes en Android consultando su manual de desarrollo. Concretamente, leyendo la sección dedicada al paquete `android.provider` en <http://developer.android.com/reference/android/provider/package-summary.html>.

### 3.2.2. Proveedores propios: crear un nuevo proveedor de contenidos

Para acceder a nuestras fuentes de datos propias de manera estándar nos interesa implementar nuestros propios `ContentProvider`. Gracias a que proporcionamos nuestros datos a partir de una URI, quien use nuestro proveedor de contenidos no deberá preocuparse de dónde provienen los datos indicados por dicha URI; podrían provenir de ficheros locales en la tarjeta de memoria, de un servidor en Internet, o de una base de datos SQLite. Dada una URI nuestro proveedor de contenidos deberá proporcionar en su interfaz los métodos necesarios para realizar las operaciones básicas en una base de datos: inserción, lectura, actualización y borrado.

Para crear un nuevo proveedor de contenidos definimos una subclase de `ContentProvider`. Utilizaremos una sobrecarga del método `onCreate` para crear e inicializar la fuente de datos que queramos hacer pública a través de dicho proveedor. Un esqueleto de subclase de `ContentProvide` podría ser el siguiente:

```
public class MiProveedor extends ContentProvider {  
  
    @Override  
    public boolean onCreate() {  
        // Inicializar la base de datos  
        return true;  
    }  
}
```

Dentro de la clase deberemos crear un atributo público y estático de nombre `CONTENT_URI`, en el cual almacenaremos el URI completo del proveedor que estamos creando. Este URI debe ser único, por lo que una buena idea puede ser tomar como base el nombre de paquete de nuestra aplicación. La forma general de un URI es la siguiente:

```
content://[NOMBRE_PAQUETE].provider.[NOMBRE_APLICACION]/[RUTA_DATOS]
```

Por ejemplo:

```
content://es.ua.jtech.android.provider.miaplicacion/elementos
```

Estos URIs se pueden presentar en dos formas. La URI anterior representa una consulta dirigida a obtener todos los valores de ese tipo (en este caso todos los elementos). Si a la URI anterior añadimos un sufijo `/[NUMERO_FILA]`, estamos representando una consulta destinada a obtener un único elemento (por ejemplo, el quinto elemento en el siguiente ejemplo):

```
content://es.ua.jtech.android.provider.miaplicacion/elementos/5
```

**Nota:**

Se considera una buena práctica de programación el proporcionar acceso a nuestro proveedor de contenidos utilizando cualquiera de estas dos formas.

La forma más simple de determinar cuál de las dos formas anteriores de URI se utilizó para hacer una petición a nuestro proveedor de contenidos es mediante un *Uri Matcher*. Crearemos y configuramos un elemento *Uri Matcher* para analizar una URI y determinar a cuál de las dos formas se corresponde. En el siguiente código se muestra el código

básico que deberemos emplear para implementar este patrón:

```
public class MiProveedor extends ContentProvider {

    private static final String miURI =
"content://es.ua.jtech.android.provider.miaplicacion/elementos";
    public static final Uri CONTENT_URI = Uri.parse(miUri);

    @Override
    public boolean onCreate() {
        // PENDIENTE: inicializar la base de datos
        return true;
    }

    // Creamos las constantes que nos van a permitir diferenciar
    // entre los dos tipos distintos de peticiones a partir
    // de la URI
    private static final int TODAS_FILAS = 1;
    private static final int UNA_FILA = 2;

    private static final UriMatcher uriMatcher;

    // Inicializamos el objeto UriMatcher. Una URI que termine en
    // 'elementos' se corresponderá con una consulta para todas las
    // filas, mientras que una URI con el sufijo 'elementos/[FILAS]'
    // representará una única fila
    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("es.ua.jtech.android.provider.miaplicacion",
            "elementos", TODAS_FILAS);
        uriMatcher.addURI("es.ua.jtech.android.provider.miaplicacion",
            "elementos/#", UNA_FILA);
    }
}
```

Esta misma técnica se puede emplear para proporcionar diferentes alternativas de URI para diferentes subconjuntos de datos, como por ejemplo diferentes tablas en una base de datos, a través del mismo proveedor de contenidos.

### 3.2.3. Proveedores propios: crear la interfaz de consultas

Para permitir realizar operaciones con los datos publicados a través de nuestro proveedor de contenidos deberemos implementar los métodos `delete`, `insert`, `update` y `query` para el borrado, inserción, actualización y realización de consultas, respectivamente. Estos métodos son la interfaz estándar utilizada con proveedores de contenidos, de tal forma que para compartir datos entre aplicaciones no se tendrán interfaces distintos para diferentes fuentes de datos.

Lo más habitual suele ser implementar un proveedor de contenidos como un mecanismo para permitir el acceso a una base de datos SQLite privada a una aplicación, aunque a través de estos métodos indicados se podría en principio acceder a cualquier fuente de datos.

El siguiente código muestra el esqueleto de estos métodos dentro de una subclase de `ContentProvider`. Obsérvese que se hace uso del objeto `UriMatcher` para determinar que tipo de consulta se está realizando:

```

@Override
public Cursor query(Uri uri,
                    String[] projection,
                    String selection,
                    String[] selectionArgs,
                    String sort) {

    // Si se trata de una consulta para obtener una única fila,
    limitamos
    // los resultados a obtener de la fuente de datos por medio de una
    // cláusula where
    switch(UriMatcher.match(uri)) {
        case UNA_FILA:
            // PENDIENTE: modifica la sentencia SELECT
            mediante una cláusula
            // where, obteniendo el identificador de fila
            como:
            // numeroFila = uri.getPathSegments().get(1));
            }
        return null;
    }

@Override
public Uri insert(Uri _uri, ContentValues _valores) {
    long idFila = [ ... Añadir un nuevo elemento ... ]

    // Devolvemos la URI del elemento recién añadido
    if (idFila > 0)
        return ContentUris.withAppendendId(CONTENT_URI, idFila);

    throw new SQLException("No se pudo añadir un nuevo elemento a " +
        _uri);
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
    switch(uriMatcher.match(uri)) {
        case TODAS_FILAS:
        case UNA_FILA:
        default:
            throw new IllegalArgumentException("URI no
    soportada: " + uri);
    }
}

@Override
public int update(Uri uir, ContentValues valores, String where, String[]
whereArgs) {
    switch(uriMatcher.match(uri)) {
        case TODAS_FILAS:
        case UNA_FILA:
        default:
            throw new IllegalArgumentException("URI no
    soportada: " + uri);
    }
}

```

### 3.2.4. Proveedores propios: tipo MIME

El último paso para crear un proveedor de contenidos es definir el tipo MIME que identifica el tipo de datos que devuelve el proveedor. Debemos sobrecargar el método `getType` para que devuelva una cadena que identifique nuestro tipo de datos de manera

única. Se deberían definir dos tipos posibles, uno para el caso de una única fila y otro para el caso de devolver todos los resultados. Se debe utilizar una sintaxis similar a la del siguiente ejemplo:

- Un único elemento:  
vnd.es.ua.jtech.android.cursor.item/mproveedorcontent
- Todos los elementos:  
vnd.es.ua.jtech.android.cursor.dir/mproveedorcontent

Como se puede observar, la cadena comienza siempre por vnd. A continuación tendríamos el nombre del paquete de nuestra aplicación. Lo siguiente es cursor, ya que nuestro proveedor de contenidos está devolviendo datos de tipo Cursor. Justo antes de la barra pondremos item en el caso del tipo MIME para un único elemento, y dir para el caso del tipo MIME para todos los elementos. Finalmente, tras la barra, pondremos el nombre de nuestra clase (en minúsculas) seguido de content. En el siguiente código de ejemplo se muestra cómo sobrecargar getType según la URI:

```
@Override
public String getType(Uri _uri) {
    switch (uriMatcher.match(_uri)) {
        case TODAS_FILAS:
            return
            "vnd.es.ua.jtech.android.cursor.dir/mproveedorcontent";
        case UNA_FILA:
            return
            "vnd.es.ua.jtech.android.cursor.item/mproveedorcontent";
        default:
            throw new IllegalArgumentException("URI no
soportada: " + _uri);
    }
}
```

### 3.2.5. Proveedores propios: registrar el proveedor

No debemos olvidar incorporar nuestro proveedor de contenidos al *Manifest* de la aplicación una vez que lo hemos completado. Esto se hará por medio del elemento provider, cuyo atributo authorities podremos utilizar para especificar la URI base, tal como se puede ver en el siguiente ejemplo:

```
<provider android:name="MiProveedor"
    android:authorities="es.ua.jtech.android.miaplicacion"/>
```

### 3.2.6. Content Resolvers

Aunque en la sección de proveedores nativos ya se ha dado algún ejemplo de cómo hacer uso de un *Content Resolver* para realizar una consulta, ahora que hemos podido observar en detalle la estructura de un proveedor de contenidos propio es un buen momento para ver en detalle cómo realizar cada una de las operaciones que éstos permiten.

El contexto de una aplicación incluye siempre una instancia de la clase ContentResolver, a la cual se puede acceder mediante el método getContentResolver.

Esta clase incorpora diversos métodos para realizar consultas a proveedores de contenidos. Cada uno de estos métodos acepta como parámetro una URI que indica con qué proveedor de contenidos se desea interactuar.

Las consultas realizadas sobre un proveedor de contenidos recuerdan a las consultas realizadas sobre bases de datos. Los resultados de las consultas se devuelven como objetos de la clase `Cursor`. Se pueden extraer valores del cursor de la misma forma en la que se explicó en el apartado de bases de datos de la sesión anterior. El método `query` de la clase `ContentResolver` acepta los siguientes parámetros:

- La URI del proveedor del contenidos al que se desea realizar la consulta.
- Una proyección que enumera las columnas que se quieren incluir en los resultados obtenidos.
- Una cláusula *where* que define las filas a obtener. Se pueden utilizar comodines `?` que se reemplazarán por valores incluidos en el siguiente parámetro.
- Una matriz de cadenas que se pueden utilizar para reemplazar los comodines `?` en la cláusula *where* anterior.
- Una cadena que describa la ordenación de los elementos devueltos.

El siguiente código muestra un ejemplo:

```
ContentResolver cr = getContentResolver();
// Devolver todas las filas
Cursor todasFilas = cr.query(MiProveedor.CONTENT_URI, null, null, null,
null);
// Devolver todas columnas de las filas para las que la columna 3
// tiene un valor determinado, ordenadas según el valor de la columna 5
String where = COL3 + "=" + valor;
String orden = COL5;
Cursor algunasFilas = cr.query(MiProveedor.CONTENT_URI, null, where, null,
orden);
```

### 3.2.7. Otras operaciones con Content Resolvers

Para realizar otro tipo de operaciones con los datos de un proveedor de contenidos haremos uso de los métodos `delete`, `update` e `insert` de un objeto `ContentResolver`.

Con respecto a la **inserción**, la clase `ContentResolver` proporciona en su interfaz dos métodos diferentes: `insert` y `bulkInsert`. Ambos aceptan como parámetro la URI del proveedor de contenidos, pero mientras que el primer método tan solo toma como parámetro un objeto de la clase `ContentValues`, el segundo toma como parámetro una matriz de elementos de este tipo. El método `insert` devolverá la URI del elemento recién añadido, mientras que `bulkInsert` devolverá el número de filas correctamente insertadas:

```
// Obtener el Content Resolver
ContentResolver cr = getContentResolver();

// Crear una nueva fila de valores a insertar
ContentValues nuevosValores = new ContentValues();

// Asignar valores a cada columna
```

```
nuevosValores.put(NOMBRE_COLUMNA, nuevoValor);
[ ... Repetir para cada columna ... ]

Uri miFilaUri = cr.insert(MiProveedor.CONTENT_URI, nuevosValores);

// Insertamos ahora una matriz de nuevas filas
ContentValues[] arrayValores = new ContentValues[5];
// PENDIENTE: rellenar el array con los valores correspondientes
int count = cr.bulkInsert(MiProveedor.CONTENT_URI, arrayValores);
```

Para el caso del **borrado** hacemos uso del método `delete` del *Content Resolver* pasando como parámetro la URI de la fila que deseamos eliminar de la fuente de datos. Otra posibilidad es incluir una cláusula *where* para eliminar múltiples filas. Ambas técnicas se muestran en el siguiente ejemplo:

```
ContentResolver cr = getContentResolver();

// Eliminar una fila específica
cr.delete(miFilaUri, null, null);
// Eliminar las primeras cinco filas
String where = "_id < 5";
cr.delete(MiProveedor.CONTENT_URI, where, null);
```

Una **actualización** se lleva a cabo mediante el método `update` del objeto *Content Resolver*. Este método toma como parámetro la URI del proveedor de contenidos objetivo, un objeto de la clase *ContentValues* que empareja nombres de columna con valores actualizados, y una cláusula *where* que indica qué filas deben ser actualizadas. El resultado de la actualización será que en cada fila en la que se cumpla la condición de la cláusula *where* se cambiarán los valores de las columnas especificados por el objeto de la clase *ContentValues*; también se devolverá el número de filas correctamente actualizadas. A continuación se muestra un ejemplo:

```
ContentValues nuevosValores = new ContentValues();

// Utilizamos el objeto ContentValues para especificar en qué columnas
// queremos que se produzca un cambio, y cuál debe ser el nuevo valor
// de dichas columnas
nuevosValores.put(NOMBRE_COLUMNA, nuevoValor);

// Aplicamos los cambios a las primeras cinco filas
String where = "_id < 5";

getContentResolver().update(MiProveedor.CONTENT_URI, nuevosValores, where,
null);
```



## 4. Persistencia en Android: proveedores de contenidos y SharedPreferences - Ejercicios

### 4.1. Compartir datos entre actividades con Shared Preferences

Descarga de las plantillas el proyecto *Dni*. Dicho proyecto está compuesto de dos actividades, *Formulario* y *Resumen*. El objetivo del ejercicio es conseguir que al pulsar el botón *Siguiente* en la actividad *Formulario* se muestre en la actividad *Resumen* cuáles fueron los datos introducidos.

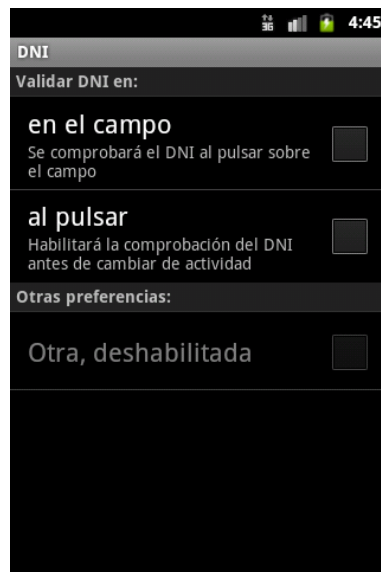
En este ejercicio vamos a hacer uso de `SharedPreferences` para pasar los datos de una actividad a la siguiente. Al pulsar en *Siguiente* en *Formulario* deberemos guardar el valor de todos los campos mediante esta plataforma. Al mostrarse la actividad *Resumen* deberemos leer estos datos, también a partir de `SharedPreferences`, para mostrarlos por pantalla.

Por último, crea un método que valide el DNI (debe tratarse de una secuencia de ocho dígitos entre 0 y 9 y una letra al final). Si se pulsa *Siguiente* y el DNI no tiene el formato correcto, no se deberá pasar a la actividad *Resumen*. En lugar de esto se mostrará un `Toast` en pantalla con un mensaje de error.

### 4.2. Actividad de preferencias

En este ejercicio seguimos trabajando con el proyecto del ejercicio anterior. Vamos a añadir un menú de opciones que permita escoger en qué momento se comprobará la validez del DNI. Para ello añade en primer lugar un menú a la actividad *Formulario* con una única opción, cuyo nombre será *Opciones*.

Al seleccionar esta opción se deberá mostrar una actividad de preferencias cuyo aspecto será el siguiente:



Preferencias de la aplicación Dni

El significado de las opciones será el siguiente:

- *En el campo*: con esta opción seleccionada, al pulsar sobre el campo de DNI en el formulario deberá mostrarse un `Toast` indicando si la sintaxis del DNI es correcta o no.
- *Al pulsar*: si se desactiva esta opción (que deberá estar activada por defecto), no se hará la comprobación del DNI al pulsar el botón *Siguiente*, por lo que siempre será posible pasar de la actividad *Formulario* a la actividad *Resumen* sea cual sea el formato del DNI introducido.
- El último check box se encontrará siempre deshabilitado. ¿Qué atributo debemos utilizar para conseguir esto?

Crea dos variables booleanas llamadas `enElCampo` y `alPulsar`. Su valor dependerá del estado de los checkboxes anteriores y se actualizará por medio del evento `onSharedPreferenceChange`. Utiliza el valor de estas variables en el código de tu actividad para que esta tenga el comportamiento indicado.

### 4.3. Proveedor de contenidos propio

Vamos a implementar otra forma de acceder a la base de datos de usuarios de la aplicación *BaseDatos* desarrollada durante los ejercicios de la sesión anterior, siguiendo esta vez el patrón de diseño `ContentProvider` de Android. Seguiremos trabajando pues con dicho proyecto.

- Creamos una nueva clase llamada `UsuariosProvider` que herede de `ContentProvider`. Esto 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.android.basedatos/usuarios");
private static final int TODAS_FILAS = 1;
private static final int UNA_FILA = 2;

private static final UriMatcher uriMatcher;

static{
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI("es.ua.jtech.android.basedatos", "usuarios",
TODAS_FILAS);
    uriMatcher.addURI("es.ua.jtech.android.basedatos", "usuarios/#",
UNA_FILA);
}

```

- Vamos a acceder a la misma base de datos de usuarios utilizada en los ejercicios de la sesión anterior, pero no vamos a hacerlo a través del adaptador que tuvimos que implementar, sino que vamos a copiar de él el código que nos haga falta. Copia 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 necesarios en el constructor. Para inicializar la referencia a la base de datos vamos a utilizar, una vez más, MiOpenHelper. Podemos copiarlo del adaptador que ya implementamos en los ejercicios de la sesión anterior.
- 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 consulta 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 registrarlo en el *AndroidManifest.xml*:

```

...
        <provider android:name=".UsuariosProvider"
android:authorities="es.ua.jtech.android.basedatos"/>
    </application>
</manifest>

```

- En la clase `Main` se añadió código para insertar una serie de valores en la base de datos y mostrarlos en el campo de texto. Manteniendo este código, vamos a añadir al campo de texto el resultado obtenido con la consulta del `UsuariosProvider` para comprobar que tanto el adaptador de la base de datos como el proveedor nos devuelven el mismo resultado.

#### 4.4. ¿Por qué conviene crear proveedores de contenidos? (\*)

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

- Utiliza el proyecto *ProveedorContenidos* de las plantillas. Implementa la inserción en el proveedor de contenidos. Pruébala insertando algunos usuarios de ejemplo en la clase *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 siguiente línea y al estar usando un proveedor de contenidos, la lista se actualiza automáticamente cuando ocurre algún cambio, sin necesidad de pedir explícitamente la actualización al pulsar el botón.

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

**Nota:**

Esto no va a funcionar si se notifica que se ha producido un cambio al insertar o borrar un elemento. Para ello usamos

```
getContext().getContentResolver().notifyChange(
    UsuariosProvider.CONTENT_URI, null);
```

- Implementa el método *delete* del proveedor de contenidos. Pruébalo en la clase *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).

## 5. Persistencia de datos en iOS: Ficheros y SQLite

### 5.1. Introducción

La persistencia de datos en iOS se produce gracias a una pequeña memoria flash que tiene nuestro dispositivo, el cual es equivalente a un disco duro de tamaño limitado. Los datos que ahí se almacenen se conservarán aunque el dispositivo se apague. Por motivos de seguridad, el SO de iOS no permite que las aplicaciones accedan directamente a la memoria interna, pero a cambio existe una API completa de acceso a la porción de memoria que le corresponde a la app que desarrollemos.

En una aplicación iOS encontraremos básicamente 4 modos de almacenar nuestros datos de forma persistente, estos son:

- Ficheros de propiedades (*plist*)
- Bases de datos embebidas *SQLite*
- Datos de usuario (*User Defaults*)
- Core Data

En esta primera sesión veremos cómo usar los ficheros de texto (*Property Lists*) y el almacenamiento en base de datos de tipo *SQLite*.

### 5.2. Ficheros plist (Property Lists)

El almacenamiento de datos en ficheros de propiedades es uno de los más usados en el desarrollo de las aplicaciones para **iOS**. Se usa principalmente para guardar datos de configuración de la aplicación y acceder a ellos de forma sencilla. Si estamos desarrollando un juego podemos usarlos para la definición de los niveles, almacenar las puntuaciones del jugador, etc.

#### 5.2.1. Leyendo datos desde ficheros plist

Los ficheros plist (*Property Lists*) tienen formato **XML** y son "*amigables*" con xCode. Para entender el funcionamiento de este tipo de ficheros vamos a realizar un ejemplo en el que primero crearemos un fichero plist, lo completaremos con datos de ejemplo y después lo leeremos para mostrarlo por pantalla.

Un ejemplo de un fichero plist podría ser el siguiente:

| Key                  | Type       | Value      |
|----------------------|------------|------------|
| ▼ DatosPartida       | Diction... | (19 items) |
| ▼ 1                  | Diction... | (4 items)  |
| bloqueado            | Boolean    | NO         |
| estrellasConseguidas | Number     | 0          |
| puntos               | Number     | 0          |
| tiempo               | Number     | 0          |
| ▼ 10                 | Diction... | (4 items)  |
| bloqueado            | Boolean    | YES        |
| estrellasConseguidas | Number     | 0          |
| puntos               | Number     | 0          |
| tiempo               | Number     | 0          |
| ▼ 11                 | Diction... | (4 items)  |
| bloqueado            | Boolean    | YES        |
| estrellasConseguidas | Number     | 0          |
| puntos               | Number     | 0          |
| tiempo               | Number     | 0          |
| ▶ 12                 | Diction... | (4 items)  |
| ▶ 13                 | Diction... | (4 items)  |

### Fichero plist

Antes de nada creamos un proyecto nuevo en xCode de tipo Window-based Application al que llamaremos sesion03-ejemplo1. Para crear un fichero plist debemos hacer click derecho sobre el directorio Supporting Files de nuestro proyecto y seleccionamos iOS > Resource > Property List. Lo guardaremos con el nombre configUsuario. Seguidamente ya podemos acceder al fichero recién creado haciendo click sobre el, en este momento xCode nos mostrará un pequeño editor totalmente vacío en el que iremos rellenando con datos. Podemos verlo también en formato XML si hacemos click derecho sobre el y seleccionamos Open As > Preview.

Para empezar a rellenar el fichero que acabamos de crear, hacemos click derecho dentro de su contenido (ahora vacío) y seleccionamos "Add row". Entonces nos aparecerá una fila vacía. Dentro del campo "Key" escribimos la clave, un nombre descriptivo que deberemos utilizar más adelante para acceder a los datos. En nuestro caso vamos a escribir "config" como clave. Dentro de la columna "Type" seleccionamos "Dictionary". Veremos que aparece una flecha en el lado izquierdo que indica que pueden haber subniveles dentro de nuestro diccionario.

Ahora añadimos un nuevo ítem que cuelgue del diccionario que acabamos de crear, para ello hacemos click sobre la flecha de la izquierda, esta se girará hacia abajo, entonces hacemos click ahora sobre el símbolo + (más). Nos aparecerá una nueva línea que cuelga del diccionario.

En esta nueva línea, dentro del campo "key" escribimos "nombre", el campo "type" lo dejamos como de tipo "String" ya que va a ser una cadena de texto y en la columna de "Value" escribimos nuestro nombre, por ejemplo "Javi".

Ahora vamos a añadir otro campo nuevo en nuestra configuración, para ello hacemos click sobre el símbolo + y nos aparecerá una nueva línea justo al mismo nivel que la anterior y colgando del diccionario. Dentro del campo de "Key" escribimos "ciudad" y en "Value" escribimos "Alicante". El campo de "Type" lo dejamos como "String",

Ahora dentro de nuestro fichero *plist* vamos a indicar los dispositivos que dispone el usuario, para ello creamos una nueva línea dentro del diccionario y le pondremos como clave "dispositivos". Seleccionamos el tipo "Array" y seguimos los mismos pasos que al crear el diccionario, para ello hacemos click sobre la flecha de la izquierda y después sobre el símbolo + (más). Ahora nos aparecerá una nueva fila con clave Item 0. Seleccionamos el tipo "String" y en el campo "Value": "iPhone".

Ahora repetimos el paso 3 veces, indicando como values: "iPad", "Android" y "Blackberry".

Una vez seguidos todos estos pasos tendremos nuestro fichero de propiedades listo. Debe de quedar como se muestra a continuación:

| Key            | Type       | Value      |
|----------------|------------|------------|
| ▼ config       | Diction... | (3 items)  |
| ▼ dispositivos | Array      | (4 items)  |
| Item 0         | String     | Blackberry |
| Item 1         | String     | Android    |
| Item 2         | String     | iPhone     |
| Item 3         | String     | iPad       |
| ciudad         | String     | Alicante   |
| nombre         | String     | Javi       |

### Fichero plist terminado

Este fichero se almacenará dentro del directorio de *bundle* cuando compilemos la aplicación. En el caso de que queramos escribir en el, deberemos almacenar el fichero dentro del directorio de "Documents" del binario.

Ahora desde nuestro código vamos a acceder a él y a mostrarlo por pantalla, para ello escribimos el siguiente código dentro del método `didFinishLaunchingWithOptions` de la clase delegada:

```
// Cargamos el fichero PLIST
NSString *mainBundlePath = [[NSBundle mainBundle] bundlePath];
NSString *plistPath = [mainBundlePath
    stringByAppendingPathComponent:@"configUsuario.plist"];
NSDictionary *diccionario = [[NSDictionary alloc]
    initWithContentsOfFile:plistPath];

// Cargamos el Diccionario inicial que esta en el raiz del
fichero
NSDictionary *dicConfig = [diccionario
    objectForKey:@"config"];

// Cargamos los campos que estan dentro del diccionario del
raiz
NSString *nombre = [dicConfig objectForKey:@"nombre"];
```

```

        NSString *ciudad = [dicConfig objectForKey:@"ciudad"];
        NSArray *dispositivos = [dicConfig
objectForKey:@"dispositivos"];

        // Mostramos por consola los datos obtenidos
        NSLog(@"Nombre: %@", nombre);
        NSLog(@"Ciudad: %@", ciudad);

        for (NSString *dispositivo in dispositivos){
            NSLog(@"Dispositivo: %@", dispositivo);
        }

        [self.window makeKeyAndVisible];
        return YES;

```

Si todo ha ido correctamente, deberemos obtener la siguiente salida por la consola:

```

2011-09-01 20:10:25.981 sesion03-ejemplo0[2311:207] Nombre: Javi
2011-09-01 20:10:25.983 sesion03-ejemplo0[2311:207] Ciudad: Alicante
2011-09-01 20:10:25.984 sesion03-ejemplo0[2311:207] Dispositivo: Blackberry
2011-09-01 20:10:25.985 sesion03-ejemplo0[2311:207] Dispositivo: Android
2011-09-01 20:10:25.986 sesion03-ejemplo0[2311:207] Dispositivo: iPhone
2011-09-01 20:10:25.986 sesion03-ejemplo0[2311:207] Dispositivo: iPad

```

Lista de dispositivos

### 5.2.2. Escribiendo datos en ficheros plist

Partiendo del ejemplo anterior vamos a escribir ahora en el fichero, para ello primero debemos comprobar que el fichero plist se encuentre dentro del directorio de Documents del dispositivo, en el caso de que no lo esté (por ejemplo, si es la primera vez que arranca la aplicación), debemos copiarlo a ese directorio. Una vez que tenemos el fichero dentro del directorio de documentos ya podemos escribir en el.

Para realizar todo el proceso de comprobación de la existencia del fichero dentro del directorio de documentos, copiarlo en ese directorio y leer los datos debemos de sustituir el código que hay dentro del método `didFinishLaunchingWithOptions` de la clase delegada por este otro:

#### Atención

Para que funcione la escritura en un fichero plist, este debe de estar dentro del directorio de Documents de nuestro dispositivo iOS, no en el bundle.

```

        NSDictionary *diccionarioRaiz;

        // Ruta del directorio de documentos de nuestro dispositivo
        NSArray *paths =
        NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask, YES);

        if ([paths count] > 0)
        {

```



```

        NSString *documentsDirectory = [paths objectAtIndex:0];
        NSString *documentsFilename = [documentsDirectory
        stringByAppendingPathComponent:@"configUsuario.plist"];

        // Primero comprobamos si existe el fichero en el
directorio de documentos
        BOOL fileExists = [[NSFileManager defaultManager]
        fileExistsAtPath:documentsFilename];
        if (fileExists)
        {
            NSLog(@"Fichero encontrado en directorio de documentos
OK: %@!",
                documentsFilename);

            // Si existe ya, cargamos los datos desde aqui
directamente
            diccionarioRaiz = [[NSDictionary alloc]
            initWithContentsOfFile:documentsFilename];
        }
        else
        {
            NSLog(@"No se encuentra el fichero configUsuario.plist
en
                el directorio de documentos->Lo creamos.");

            // Si no existe, primero cargamos el fichero PLIST
desde el Bundle
            NSString *mainBundlePath = [[NSBundle mainBundle]
            bundlePath];
            NSString *plistPath = [mainBundlePath
            stringByAppendingPathComponent:@"configUsuario.plist"];
            diccionarioRaiz = [[NSDictionary alloc]
            initWithContentsOfFile:plistPath];

            // Y despues escribimos los datos cargados (el
            // a un fichero nuevo de la carpeta de documentos
            [diccionarioRaiz writeToFile:documentsFilename
            atomically:YES];

            NSLog(@"plist de configUsuario creado!");
        }

        // Cargamos el Diccionario inicial que esta en el raiz del
fichero
        NSDictionary *dicConfig = [diccionarioRaiz
        objectForKey:@"config"];

        // Cargamos los campos que estan dentro del diccionario
del raiz
        NSString *nombre = [dicConfig objectForKey:@"nombre"];
        NSString *ciudad = [dicConfig objectForKey:@"ciudad"];
        NSArray *dispositivos = [dicConfig
        objectForKey:@"dispositivos"];

        // Mostramos por consola los datos obtenidos
        NSLog(@"Nombre: %@", nombre);
        NSLog(@"Ciudad: %@", ciudad);

        for (NSString *dispositivo in dispositivos){
            NSLog(@"Dispositivo: %@", dispositivo);
        }
    }
}

```

```

    }

    // Por ultimo liberamos la memoria del diccionario raiz
    [diccionarioRaiz release];
}

[self.window makeKeyAndVisible];
return YES;

```

Para comprobar el funcionamiento arrancamos la aplicación y veremos que la primera vez se creará el fichero dentro de la carpeta de Documents de nuestro dispositivo y en las siguientes ya no se creará y simplemente lo leerá desde la carpeta de Documents.

Ahora que ya tenemos la gestión de ficheros básica completada vamos a escribir sobre el algún dato. Para ello simplemente escribimos el siguiente fragmento de código justo antes de la línea `[self.window makeKeyAndVisible];`:

```

//*****
// Cambiamos el nombre y lo guardamos en el fichero
// Ruta del directorio de documentos de nuestro dispositivo

if ([paths count] > 0)
{
    // Cargamos el fichero desde el directorio de
documentos del dispositivo
    NSString *documentsDirectory = [paths objectAtIndex:0];
    NSString *documentsFilename = [documentsDirectory
        stringByAppendingPathComponent:@"configUsuario.plist"];

    diccionarioRaiz = [[NSDictionary alloc]
        initWithContentsOfFile:documentsFilename];

    NSDictionary *dicConfig = [diccionarioRaiz
objectForKey:@"config"];

    // Cambiamos el valor del nombre en el diccionario
    [dicConfig setValue:@"Otro nombre" forKey:@"nombre"];

    // Escribimos todo el diccionario de nuevo al fichero del
// directorio de documentos
    [diccionarioRaiz writeToFile:documentsFilename
atomically:YES];

    // Por ultimo liberamos el diccionario de la
memoria
    [diccionarioRaiz release];
}

```

Lo que hemos hecho en el código anterior es cargar todo el diccionario de nuevo desde el directorio de documentos, modificar los datos que queremos cambiar o incluso añadir y después guardar el diccionario en el fichero.

Volvemos a arrancar la aplicación y veremos que ha cambiado el campo de nombre.

#### Nota

La lectura y escritura de datos en un fichero plist se debe de hacer de una sólo vez, nunca por partes. Siempre se debe de cargar o escribir un diccionario `NSDictionary` como elemento principal del fichero.

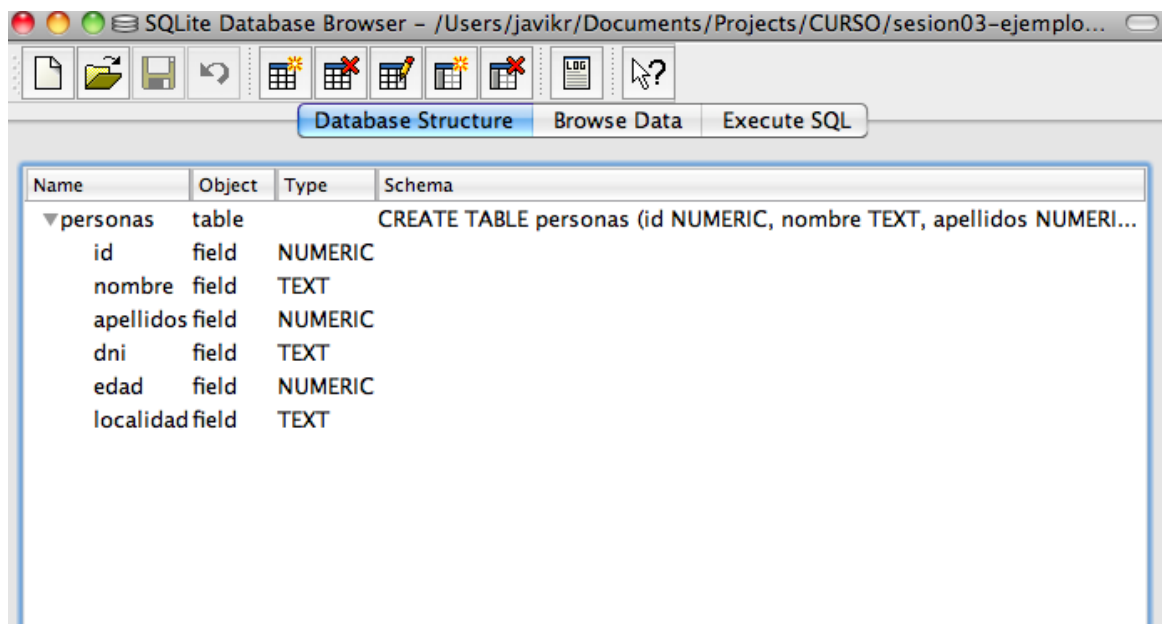
### 5.3. SQLite

**SQLite** es una base de datos relacional ligera con la que se puede trabajar de forma sencilla mediante sentencias **SQL**. La base de datos se almacenará en un fichero dentro de nuestra aplicación. Este método de persistencia puede ser el adecuado cuando los datos a almacenar estén en un formato de tablas, filas y columnas y se necesite que los datos sean accesibles de forma rápida.

#### 5.3.1. Herramientas disponibles

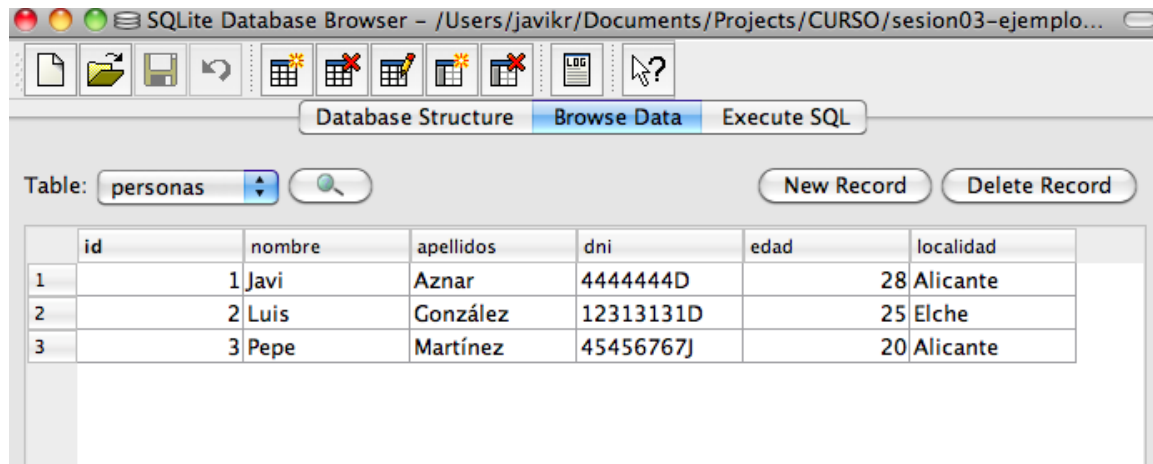
Para explicar el funcionamiento de las librerías de *SQLite* de *Cocoa Touch* vamos a seguir un ejemplo completo. Comenzamos creando la base de datos, para ello podemos utilizar uno de los frontends que facilitan la gestión del *SQLite*: uno bastante sencillo y recomendado es el [SQLite Database Browser](#) y otro es [fmdb](#).

Nosotros utilizaremos el primero (*SQLite Database Browser*). Abrimos la aplicación y hacemos click sobre el botón de crear nueva base de datos que llamaremos: `personasDB.sqlite`. Seguidamente creamos las tablas, para simplificar en nuestro ejemplo crearemos una sólo tabla llamada "personas". Esta tabla tendrá las siguientes columnas: `id` (NUMERIC), `nombre` (TEXT), `apellidos` (TEXT), `dni` (TEXT), `edad` (NUMERIC) y `localidad` (TEXT).



### Estructura de la Base de Datos

Una vez creada la tabla pasamos a insertar datos dentro de ella, para ello abrimos la pestaña de `Browse Data` y hacemos click en `New Record`. Cuando tengamos al menos 3 registros metidos dentro de la tabla ya podemos guardar la base de datos. Hacemos click sobre *guardar*, le ponemos el nombre que queramos y elegimos cualquier directorio del MAC.

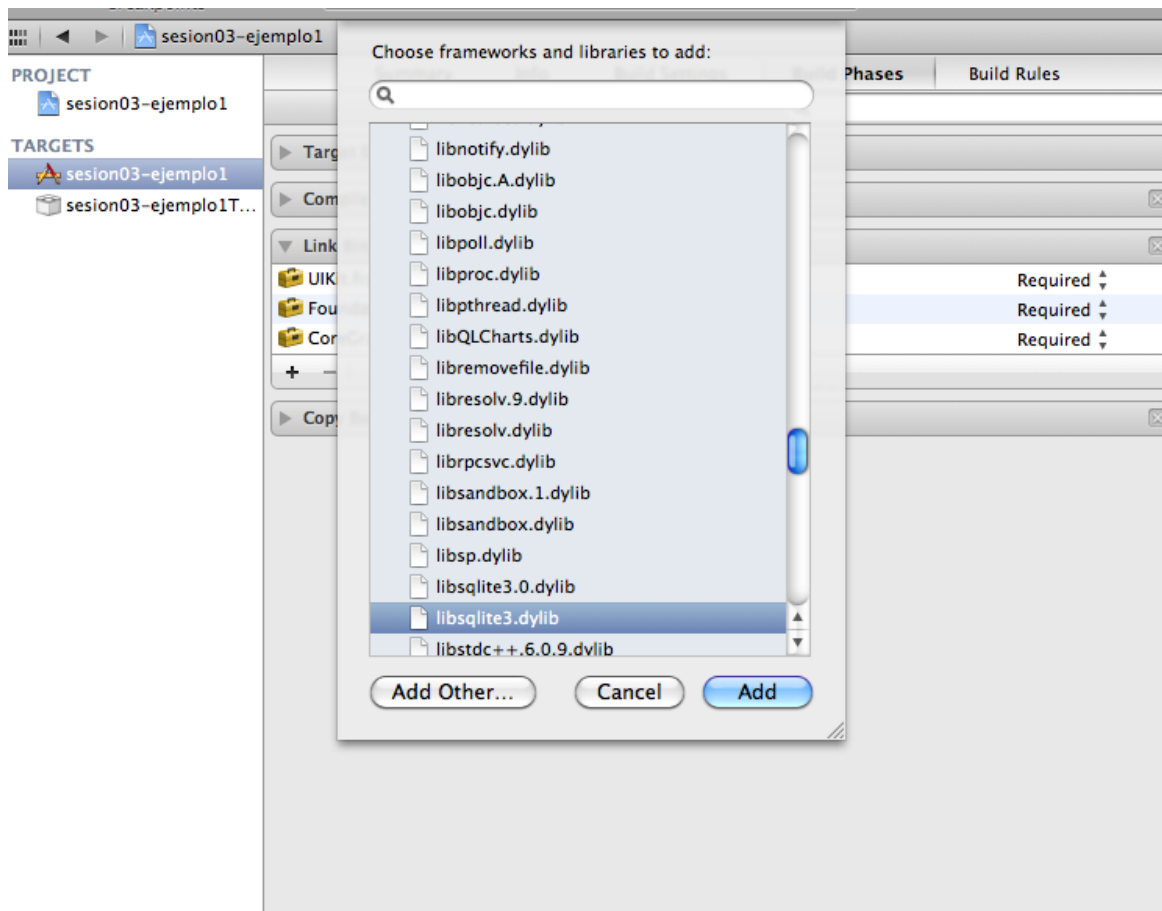


Datos de la Base de Datos

### 5.3.2. Lectura de datos

Una vez que tenemos la base de datos *SQLite* creada ya podemos empezar a escribir el código para leer y mostrar los datos dentro de nuestra aplicación:

Primero empezamos creando un proyecto nuevo en xCode usando la plantilla basada en vistas `View-based application` y lo llamaremos `sesion03-ejemplo2`. Ahora añadimos el framework de `sqlite` a nuestro proyecto, para ello hacemos click sobre el raíz del proyecto y seleccionamos la pestaña `Build Phases`. Desplegamos el listado de `Link Binary With Libraries` y hacemos click sobre el botón (+). En el listado seleccionamos el framework `libsqlite3.dylib` y hacemos click en `Add`.



Añadiendo el framework libsqlite3.dylib

Ahora añadimos a nuestro proyecto la base de datos que hemos creado anteriormente, para ello arrastramos el fichero `personasDB.sqlite` a nuestro directorio Supporting Files. Asegurate de seleccionar la opción de "Copy items to destination group's folder (if needed)" para que se copie el fichero dentro de la carpeta del proyecto.

Para acelerar el proceso de lectura de la base de datos y ahorrar memoria vamos a cargar primero todos los datos en objetos, para ello vamos a crear una clase que se encargue de almacenar los datos. Hacemos click derecho sobre el raíz del proyecto "New file > Objective-C class", seleccionamos "NSObject" como clase principal y le damos a "Next". Escribimos como nombre de la clase: "Persona" y hacemos click en "save".

Persona.h

```
//imports iniciales

@interface Persona : NSObject {
    int _uId;
    NSString *_nombre;
    NSString *_apellidos;
```

```

        NSInteger _edad;
        NSString *_localidad;

    }

    @property (nonatomic, assign) int uId;
    @property (nonatomic, copy) NSString *nombre;
    @property (nonatomic, copy) NSString *apellidos;
    @property (nonatomic) NSInteger edad;
    @property (nonatomic, copy) NSString *localidad;

    - (id)initWithUid:(int)uId nombre:(NSString *)nombre
    apellidos:(NSString *)apellidos edad:(NSInteger)edad
    localidad:(NSString *)localidad;

@end

```

Persona.m

```

#import "Persona.h"

@implementation Persona

@synthesize uId = _uId;
@synthesize nombre = _nombre;
@synthesize apellidos = _apellidos;
@synthesize edad = _edad;
@synthesize localidad = _localidad;

- (id)initWithUid:(int)uId nombre:(NSString *)nombre
apellidos:(NSString *)apellidos
edad:(NSInteger)edad localidad:(NSString *)localidad {

    if ((self = [super init])) {
        self.uId = uId;
        self.nombre = nombre;
        self.apellidos = apellidos;
        self.edad = edad;
        self.localidad = localidad;
    }
    return self;
}

- (void) dealloc {
    self.nombre = nil;
    self.apellidos = nil;
    self.localidad = nil;
    [super dealloc];
}

@end

```

Una vez creada la clase "Persona" vamos a crear la clase encargada de manejar la base de datos *sqlite3*. Otra opción sería cargar todos los datos directamente desde la clase delegada al arrancar la aplicación pero no es recomendable ya que si en algún momento tenemos que cambiar de motor de base de datos lo tendremos más complicado.

Creamos entonces una clase dentro de nuestro proyecto llamada "BDPersistencia" que herede de NSObject y escribimos el siguiente código:

BDPersistencia.h

```
//imports iniciales

@interface BDPersistencia : NSObject {
    sqlite3 *_database;
}

+ (BDPersistencia*)database;
- (NSArray *)arrayPersonas;

@end
```

BDPersistencia.m

```
#import "BDPersistencia.h"
#import "Persona.h"

@implementation BDPersistencia

static BDPersistencia *_database;

+ (BDPersistencia*)database {
    if (_database == nil) {
        _database = [[BDPersistencia alloc] init];
    }
    return _database;
}

- (id)init {
    if ((self = [super init])) {
        NSString *sqliteDb = [[NSBundle mainBundle]
            pathForResource:@"personasDB"
            ofType:@"sqlite"];

        if (sqlite3_open([sqliteDb UTF8String],
            &_database) != SQLITE_OK)
        {
            NSLog(@"Fallo al abrir la BD!");
        }
        return self;
    }
}

- (void)dealloc {
    sqlite3_close(_database);
    [super dealloc];
}

- (NSArray *)arrayPersonas {
    NSMutableArray *arraySalida = [[NSMutableArray alloc]
init]
    autorelease];
}
```

```

localidad, edad      NSString *query = @"SELECT id, nombre, apellidos,
                      FROM personas ORDER BY apellidos DESC";

                      sqlite3_stmt *statement;
                      if (sqlite3_prepare_v2(_database, [query UTF8String],
-1,
                      &statement, nil)
                          == SQLITE_OK) {
                          while (sqlite3_step(statement) == SQLITE_ROW) {
                              int uniqueId = sqlite3_column_int(statement,
0);
                              char *nombreChar = (char *)
sqlite3_column_text(statement, 1);
                              char *apellidosChar = (char *)
sqlite3_column_text(statement, 2);
                              char *localidadChar = (char *)
sqlite3_column_text(statement, 3);
                              int edad = sqlite3_column_int(statement, 4);

                              NSString *nombre = [[NSString alloc]
initWithUTF8String:nombreChar];
                              NSString *apellidos = [[NSString alloc]
initWithUTF8String:apellidosChar];
                              NSString *localidad = [[NSString alloc]
initWithUTF8String:localidadChar];
                              Persona *persona = [[Persona alloc]
initWithUid:uniqueId nombre:nombre
                              apellidos:apellidos edad:edad
                              localidad:localidad];

                              [arraySalida addObject:persona];
                              [nombre release];
                              [apellidos release];
                              [localidad release];
                              [persona release];
                          }
                      }
                      sqlite3_finalize(statement);
                  }
                  return arraySalida;
            }
        }
    @end

```

Con esto ya podemos leer de nuestra base de datos las personas. Para ver que funciona todo a la perfección escribimos el siguiente código dentro de la clase delegada en el método `applicationDidFinishLaunching`. Antes debemos incluir en la parte superior las clases creadas:

```

#import "BDPersistencia.h"
#import "Persona.h"

```

```

        NSArray *personas = [BDPersistencia
database].arrayPersonas;
        for (Persona *persona in personas) {
            NSLog(@"%d: %@, %@, %@, %d", persona.uId,
persona.nombre,
            persona.apellidos, persona.localidad, persona.edad);
        }
    }

```



```
}
```

Si ejecutamos el proyecto, obtendremos la siguiente salida por consola:

```
2011-09-02 17:57:11.520 sesion03-ejemplo1[4567:207] Carga OK BD
2011-09-02 17:57:11.523 sesion03-ejemplo1[4567:207] DATOS:
2011-09-02 17:57:11.524 sesion03-ejemplo1[4567:207] 3: Pepe, Martínez, Alicante, 20
2011-09-02 17:57:11.524 sesion03-ejemplo1[4567:207] 2: Luis, González, Elche, 25
2011-09-02 17:57:11.525 sesion03-ejemplo1[4567:207] 1: Javi, Aznar, Alicante, 28
2011-09-02 17:57:11.531 sesion03-ejemplo1[4567:207] total: 3
```

Listado de personas

### 5.3.3. Mostrando los datos en una tabla

Una vez que tenemos el código para obtener todas las filas de la tabla de personas de la base de datos queda lo más fácil, mostrarlas dentro de una vista de la aplicación. En este caso utilizaremos la vista de tabla "UITableView".

Comenzamos creando una nueva vista en nuestro proyecto, para ello hacemos click derecho sobre el raíz y seleccionamos: "New File > UIViewController Subclass". Asegurate de marcar "Subclass of UITableViewController" y la opción de "With XIB for user interface". Hacemos click en "Next" y guardamos el fichero como "PersonasViewController", por ejemplo.

Ahora abrimos el fichero "PersonasViewController.h" y añadimos un NSArray que será el que almacene las personas de la base de datos:

```
//imports iniciales

@interface PersonasViewController : UITableViewController
{
    NSArray *_listadoPersonas;
}

@property (nonatomic, retain) NSArray *listadoPersonas;

@end
```

Ahora abrimos el fichero "PersonasViewController.m", añadimos el @synthesize, importamos los ficheros de gestión de la base de datos, la clase Persona y todo el código básico para la gestión de la vista de la tabla:

```
#import "PersonasViewController.h"
#import "Persona.h"
#import "BDPersistencia.h"

@implementation PersonasViewController

@synthesize listadoPersonas = _listadoPersonas;
```

```

- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)dealloc
{
    [super dealloc];

    self.listadoPersonas = nil;
}

- (void)didReceiveMemoryWarning
{
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc that aren't in
use.
}

#pragma mark - View lifecycle

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = @"Listado de personas";

    self.listadoPersonas = [BDPersistencia
database].arrayPersonas;
}

- (void)viewDidUnload
{
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
}

- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
}

- (void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:

```

```

        (UIInterfaceOrientation)interfaceOrientation
        {
            // Return YES for supported orientations
            return (interfaceOrientation ==
UIInterfaceOrientationPortrait);
        }

#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView
*)tableView
{
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:
(NSInteger)section
{
    // Return the number of rows in the section.
    return [self.listadoPersonas count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleSubtitle
reuseIdentifier:CellIdentifier] autorelease];
    }

    // Configure the cell...
    Persona *persona = (Persona *)[self.listadoPersonas
objectAtIndex:indexPath.row];
    cell.textLabel.text = [NSString stringWithFormat:@"%@@
%@ (%d)", persona.nombre,
persona.apellidos, persona.edad];
    cell.detailTextLabel.text = persona.localidad;

    return cell;
}

#pragma mark - Table view delegate

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:
(NSIndexPath *)indexPath
{
    // Navigation logic may go here. Create and push
another view controller.
}

@end

```

Ahora sólo nos queda incorporar la vista que acabamos de crear dentro de nuestra aplicación. Para ello vamos a llamarla desde el Delegate y a meterla dentro de un "Navigation Controller". Primero debemos de añadir un "Outlet" en la clase delegada hacia un "Navigation Controller":

```
//imports iniciales

@interface sesion03_ejemplo1AppDelegate : NSObject
<UIApplicationDelegate> {
    UIWindow *_window;
    UINavigationController *_navController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet
UINavigationController *navController;

@end
```

Ahora hacemos click sobre la vista "MainWindow.xib" y arrastramos un "Navigation Controller" desde el listado de la columna de la derecha hacia la columna de la izquierda donde pone "Objects". Nos aparecerá el "Navigation Controller" dibujado en pantalla. Ahora desplegamos el "Navigation Controller" y hacemos click sobre el "View Controller" que hay dentro. Nos vamos a la pestaña de "Identity Inspector" de la columna de la derecha y escribimos dentro de "Class" el nombre de la vista del listado de personas: "PersonasViewController".

Finalmente hacemos click sobre el objeto "*Delegate*", vamos a la pestaña de "Show Connections Inspector" y arrastramos desde "navController" hasta el objeto "Navigation Controller" de la columna de la izquierda. Ya tenemos los controladores y las vistas conectadas.

Una vez hecho esto nos queda indicar dentro de nuestro código que la aplicación arranque con el Navigation Controller, para ello escribimos lo siguiente al final del método "didFinishLaunchingWithOptions" de la implementación de la clase delegada:

```
self.window.rootViewController = self.navController;
[self.window makeKeyAndVisible];
return YES;
```

Ya está listo todo y preparado para compilar. Si ejecutamos el proyecto veremos que nos aparece el listado de personas en la vista de tabla.

## 6. Persistencia de datos en iOS: Ficheros y SQLite - Ejercicios

### 6.1. Creando y leyendo un fichero plist

En este primer ejercicio vamos a crear nuestra primera aplicación usando el método de persistencia de ficheros. La aplicación mostrará en una vista de tabla `TableView` un listado de series en la que si seleccionamos una veremos su información más importante en otra vista. Comenzaremos creando un fichero de propiedades (*plist*) siguiendo una estructura determinada y posteriormente mostraremos en una vista de tabla todos sus datos. ¡Comenzamos!

1) Crear un proyecto usando la plantilla "Master-Detail application". Lo llamaremos *ejercicio-plist-ios*, como identificador de empresa escribiremos *es.ua.jtech* y como prefijo de clase *UA*. Por último seleccionaremos *iPhone* como dispositivo y marcaremos *Use Automatic Reference Counting* para "olvidarnos" de la gestión de memoria. El resto de opciones las dejaremos sin marcar.

2) Crear un archivo de propiedades (plist) con el nombre de "series".

3) Editar el archivo plist creando la siguiente estructura de datos (5 series mínimo):

| Key         | Type       | Value     |
|-------------|------------|-----------|
| ▼ series-db | Diction... | (1 item)  |
| ▼ series    | Array      | (5 items) |
| ▼ Item 0    | Diction... | (5 items) |
| titulo      | String     |           |
| anyo        | Number     | 0         |
| director    | String     |           |
| sinopsis    | String     |           |
| duracion    | String     |           |
| ▶ Item 1    | Diction... | (5 items) |
| ▶ Item 2    | Diction... | (5 items) |
| ▶ Item 3    | Diction... | (5 items) |
| ▶ Item 4    | Diction... | (5 items) |

Plist series

4) Rellenar el archivo plist con datos de series reales.

5) Escribir el código necesario en `UAMasterViewController.m` que recorra el archivo plist que acabamos de crear y muestre por consola los títulos de las series.

6) Mostrar en la tabla que hay implementada los títulos de la serie. Para ello deberemos de completar todos los métodos del protocolo del `TableView`.

7) Ejecutar la aplicación y comprobar que funciona correctamente.

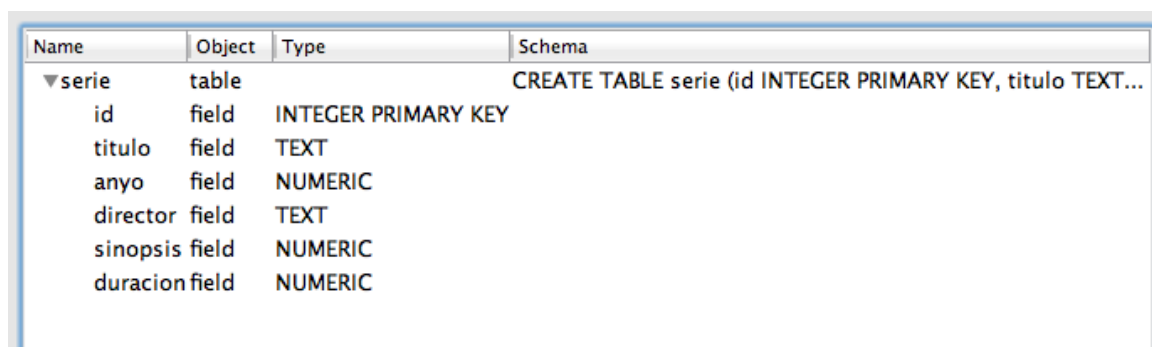
## 6.2. Implementar vista de detalle de la serie

Dentro del proyecto anterior (no hace falta crear uno nuevo) vamos a mostrar toda la información adicional de cada una de las series en una vista de detalle. Cuando seleccionemos una fila de la tabla nos deberá aparecer una nueva vista con el resto de información (año, director principal, sinopsis y duración).

## 6.3. (\*) Uso básico de una base de datos SQLite

En este ejercicio vamos a crear la misma aplicación que en el ejercicio anterior pero usando el método de persistencia de SQLite. Para completar el ejercicio deberemos seguir los siguientes pasos:

- 1) Crear un proyecto usando la plantilla "Master-Detail application", lo llamaremos *ejercicio-sqlite-ios*, como identificador de empresa escribiremos *es.ua.jtech* y como prefijo de clase *UA*. Por último seleccionaremos *iPhone* como dispositivo y marcaremos *Use Automatic Reference Counting* para "olvidarnos" de la gestión de memoria. El resto de opciones las dejaremos sin marcar.
- 2) Nos descargamos e instalamos el programa SQLite Database Browser desde [esta](#) dirección.
- 3) Diseñar la estructura de la base de datos usando SQLite Database Browser tal y como se muestra en la imagen siguiente:



| Name     | Object | Type                | Schema   |
|----------|--------|---------------------|--|
| ▼ serie  | table  |                     | CREATE TABLE serie (id INTEGER PRIMARY KEY, titulo TEXT... |
| id       | field  | INTEGER PRIMARY KEY |  |
| titulo   | field  | TEXT                |  |
| anyo     | field  | NUMERIC             |  |
| director | field  | TEXT                |  |
| sinopsis | field  | NUMERIC             |  |
| duracion | field  | NUMERIC             |  |

Sqlite series

- 4) Insertar al menos 5 series en la base de datos usando el programa SQLite Database Browser.
- 5) Cargar todos los datos de la base de datos desde el método `didFinishLaunchingWithOptions` de la clase *UAppDelegate*. Mostrar por consola los datos cargados.
- 6) Mostrar los datos cargados en el paso anterior en la vista de tabla `UITableView` que hay creada en la clase `UAMasterViewController`.

7) Arrancar la aplicación y comprobar que se muestran las series en la tabla de forma correcta.

## 7. Persistencia de datos en iOS: User Defaults y Core Data

### 7.1. Introducción

En esta sesión veremos como utilizar dos nuevos métodos de persistencia en iOS, ambos muy interesantes y usados muy frecuentemente:

- **User Defaults:** Muy útil para almacenar datos simples y de uso repetido dentro de nuestras aplicaciones iOS.
- **Core Data:** Método avanzado de almacenamiento de objetos enteros dentro de la memoria del dispositivo iOS. Muy útil para almacenar datos complejos, con relaciones, etc.

### 7.2. User Defaults

Este método es muy útil cuando queremos almacenar pequeñas cantidades de datos como puntuaciones, información de usuario, el estado de nuestra aplicación, configuraciones, etc.

Este método es el más rápido de usar ya que no requiere de una base de datos, ni ficheros, ni ninguna otra capa intermedia. Los datos se almacenan directamente en la memoria del dispositivo dentro de un objeto de la clase `NSUserDefaults`.

Para aprender a usar este método de almacenamiento de datos vamos a crear una aplicación muy simple que lea un nombre y una edad de pantalla y las guarde en un objeto `NSUserDefaults`. Para ello seguimos los siguientes pasos:

- 1) Comenzamos creando un nuevo proyecto llamado "sesion04-ejemplo1" dentro de xCode.
- 2) Abrimos la vista `sesion04_ejemplo1ViewController` y arrastramos dos *Text Field*, un botón y dos labels quedando la vista de la siguiente manera:





Vista de la aplicación

- 3) Abrimos el fichero `sesion04_ejemplo1ViewController.h` y definimos el Outlet del evento onclick del botón. El fichero quedará de la siguiente manera:

```
//imports iniciales

@interface sesion04_ejemplo1ViewController : UIViewController {
    UITextField *tfNombre;
    UITextField *tfEdad;
}

@property(n nonatomic, retain) IBOutlet UITextField *tfNombre;
@property(n nonatomic, retain) IBOutlet UITextField *tfEdad;

-(IBAction) guardaDatos;

@end
```

- 4) Ahora implementamos el código para el método del botón dentro de `sesion04_ejemplo1ViewController.m`:

```
@synthesize tfEdad;
@synthesize tfNombre;
```

```
-(IBAction) guardaDatos {
    NSLog(@"Guardamos los datos...");
}
```

```

// creamos el objeto UserDefaults
NSUserDefaults *datos = [NSUserDefaults standardUserDefaults];

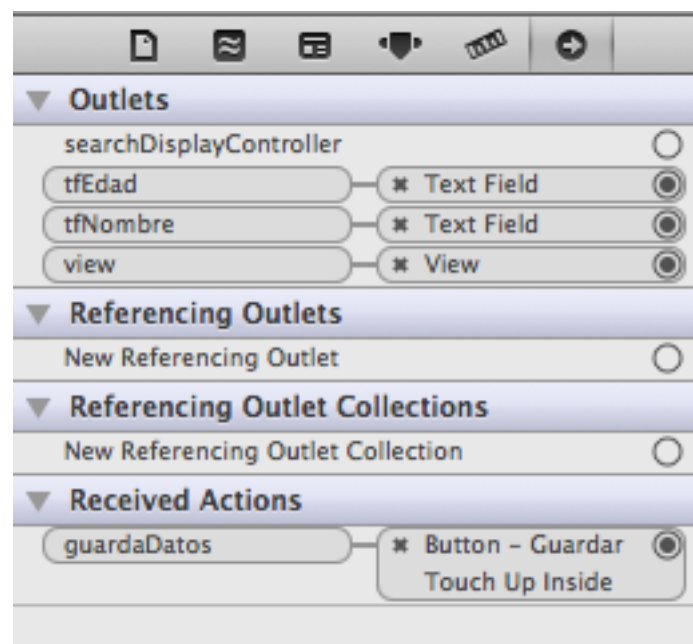
// guardamos el nombre (NSString)
[datos setObject:tfNombre.text forKey:@"nombre"];

// guardamos la edad (Integer)
[datos setInteger:[tfEdad.text integerValue] forKey:@"edad"];

// almacenamos los datos en la memoria
[datos synchronize];
}

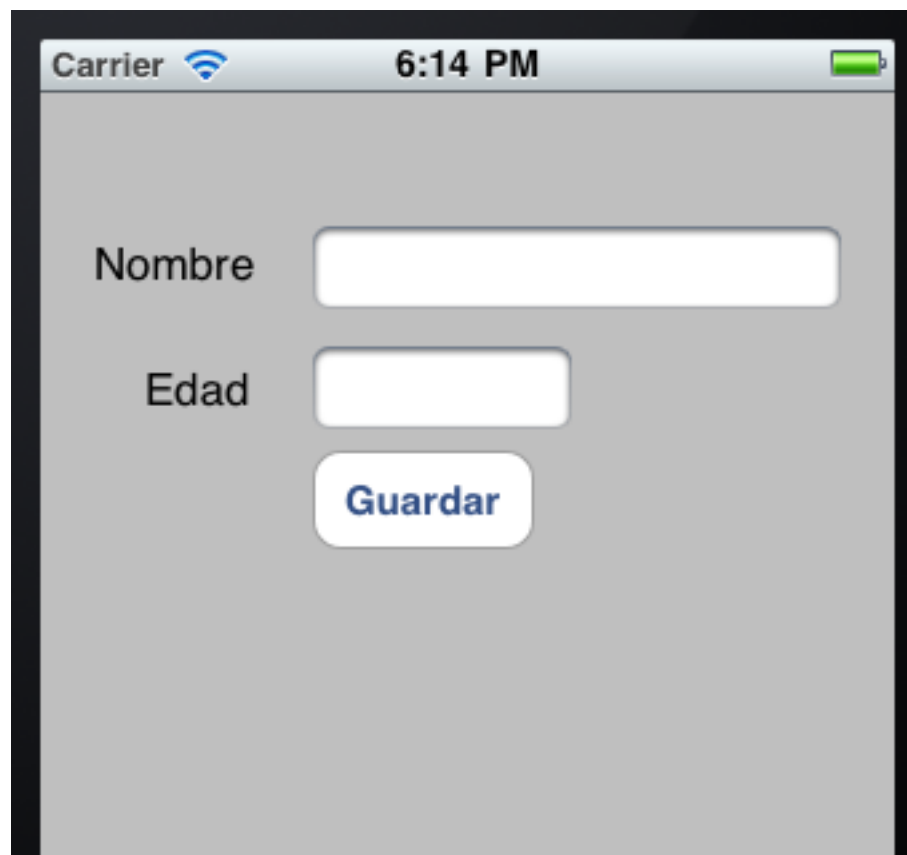
```

- 5) Tenemos que enlazar el botón con el evento que hemos programado en el paso anterior, para ello abrimos la vista `sesion04_ejemplo1ViewController.xib` y con la tecla *ctrl* apretada, arrastramos desde el botón hasta `File's Owner` (columna de la izquierda, parte superior) y seleccionamos el método que hemos creado (`guardaDatos`). De esta forma ya tenemos "conectado" el botón.
- 6) Ahora hacemos lo mismo con los *Text Field*. Tenemos que "conectarlos" al controlador, para ello seleccionamos el objeto "File's Owner" y abrimos la pestaña de conexiones (la que está más a la derecha en la columna de la derecha). Hacemos click en `tfEdad` y arrastramos hasta el *Text Field* de edad. Hacemos lo mismo para el *text field* de nombre. Ya tenemos todos los elementos de la vista conectados.



Conexión de Outlets

- 7) Arrancamos la aplicación y la probamos:



Aplicación

- 8) Ahora vamos a hacer que la aplicación cargue los datos que se han guardado al arrancar la vista, para ello simplemente tenemos que escribir el siguiente código en el método `(void)viewDidLoad` de la clase `sesion04_ejemplo1ViewController.m`:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    UserDefaults *datos = [NSUserDefaults standardUserDefaults];

    // obtenemos un NSString
    NSString *nombre = [datos objectForKey:@"nombre"];

    // obtenemos un NSInteger
    NSInteger edad = [datos integerForKey:@"edad"];

    if (nombre!=nil && edad!=0){
        NSLog(@"Nombre cargado: %@, edad: %d",nombre,edad);
    }
    else {
        NSLog(@"Sin datos!");
    }
}
```

- 9) Ya podemos arrancar de nuevo la aplicación y veremos que si escribimos cualquier

texto en los *Text Fields*, estos se almacenarán dentro del objeto `NSUserDefaults`. Al apagar la aplicación y volver a arrancarla, estos datos se cargarán (aparecerán en consola).

```
2011-09-02 18:18:38.451 sesion04-ejemplo1[4730:207] Nombre cargado: Javier, edad: 28
```

Consola

#### Nota

Los datos permanecerán en nuestro dispositivo hasta que se elimine la aplicación, en ese caso todos los datos guardados en `NSUserDefaults` se borrarán.

- 10) Probar eliminar la aplicación del dispositivo/simulador y volver arrancar. Comprobamos que los datos se han borrado:

```
2011-09-02 18:16:28.020 sesion04-ejemplo1[4701:207] Sin datos!
```

Consola

## 7.3. Core Data

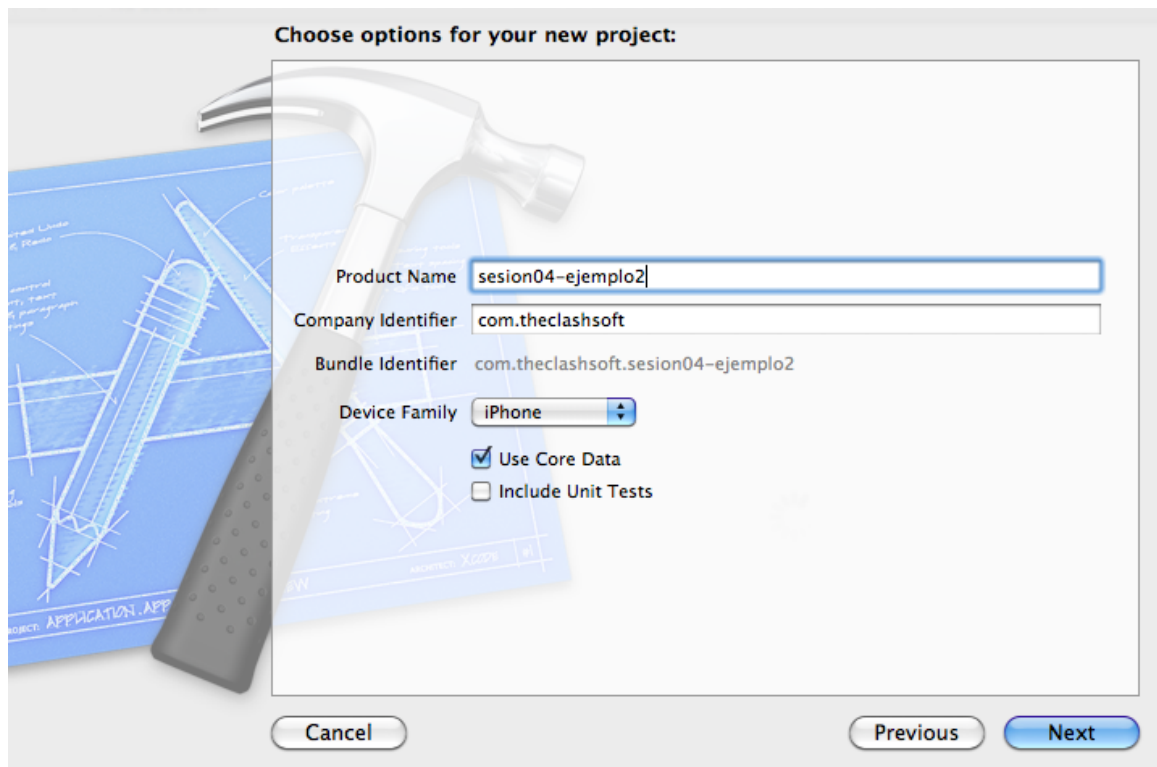
De todas las formas de persistencia de datos que existen en iOS, **Core Data** es la más apropiada para usar en caso de manejar datos no triviales, con relaciones algo más complejas entre ellos. El uso de *Core Data* dentro de nuestra aplicación optimiza al máximo el consumo de memoria, mejora el tiempo de respuesta y reduce la cantidad de código redundante a escribir.

*Core Data* utiliza de fondo el método de almacenamiento *SQLite* visto en la sesión 3 de este módulo, por tanto para explicar el funcionamiento de este método de persistencia vamos a utilizar el mismo modelo de datos que usamos en esa sesión: *Persona* -> *DetallePersona*.

En el siguiente ejemplo haremos la misma aplicación que hicimos con *SQLite*, pero usando este nuevo método de persistencia.

### 7.3.1. Creando el proyecto

Para empezar tenemos que crearnos un nuevo proyecto en xCode. Para ello abrimos xCode y seleccionamos crear una nueva aplicación basada en ventanas (window-based application). Hacemos click en "Next", escribimos como nombre del proyecto "sesion04-ejemplo2" y seleccionamos "Use Core Data".



Creando un proyecto nuevo

Con esto ya tenemos la estructura básica para empezar a programar. Antes de nada hechamos un vistazo rápido a toda la estructura de directorios creada y vemos que dentro del directorio principal tenemos un archivo que se llama "session04\_ejemplo2.xcdatamodeld". Al hacer click sobre el archivo se nos abre un editor especial que en principio está vacío y que más adelante utilizaremos para diseñar nuestro modelo de datos.

Ahora hechamos un vistazo a la clase delegada "session04\_ejemplo2AppDelegate.m", vemos que hay unos cuantos métodos nuevos que servirán para configurar el *Core Data*. Vamos a explicar los más importantes:

- - (NSManagedObjectModel \*)managedObjectModel. NSManagedObjectModel es la clase que contiene la definición de cada uno de los objetos o entidades que almacenamos en la base de datos. Normalmente este método no lo utilizaremos ya que nosotros vamos a utilizar el editor que hemos visto antes, con el que podremos crear nuevas entidades, crear sus atributos y relaciones.
- - (NSPersistentStoreCoordinator \*)persistentStoreCoordinator. Aquí es donde configuramos los nombres y las ubicaciones de las bases de datos que se usarán para almacenar los objetos. Cuando un "managed object" necesite guardar algo pasará por este método.
- - (NSManagedObjectContext \*)managedObjectContext. Esta clase NSManagedObjectContext será la más usada con diferencia de las tres, y por tanto, la

más importante. La utilizaremos básicamente para obtener objetos, insertarlos o borrarlos.

### 7.3.2. Definiendo el modelo de datos

En el ejemplo que hicimos en la sesión anterior de *SQLite* creamos dos tablas, una para los datos básicos de la persona y otra para el detalle. Ahora haremos lo mismo pero usando Core Data, la diferencia principal es que mientras que con *SQLite* obteníamos sólo los datos que necesitábamos de la base de datos, aquí obtenemos el objeto completo.

Para definir nuestro modelo de datos basado en dos objetos (Persona y DetallePersona) abrimos el editor visual haciendo click sobre el fichero `sesion04_ejemplo2.xcdatamodeld`. Empezaremos creando una nueva entidad, para ello hacemos click sobre el botón "Add Entity (+)" que se encuentra en la parte inferior de la pantalla. A la nueva entidad le pondremos como nombre "Persona".

Dentro de la sección de "Attributes" hacemos click sobre (+) y creamos un nuevo atributo para nuestra entidad, le llamamos "nombre" y le asignamos como tipo "String". Realizamos este mismo paso 2 veces más para añadir "apellidos" y "edad", este último de tipo "Integer 16".

| ▼ Attributes |              |  |
|--------------|--------------|--|
| Attribute ▲  | Type         |  |
| S apellidos  | String ↕     |  |
| N edad       | Integer 16 ↕ |  |
| S nombre     | String ↕     |  |
| + -          |              |  |

Atributos Persona

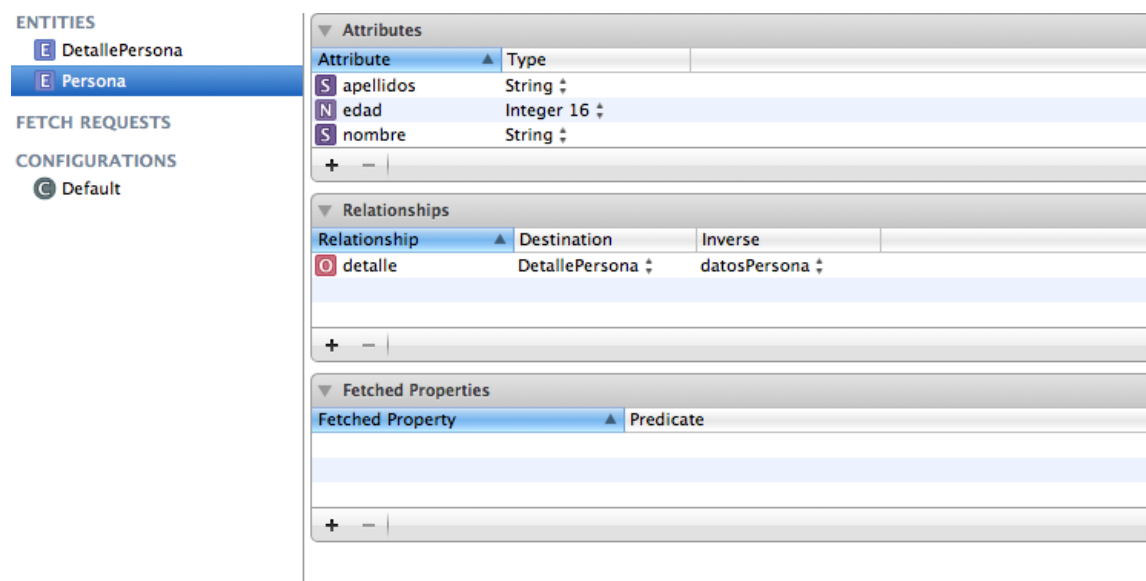
Ahora creamos la entidad de "DetallePersona" de la misma manera que la anterior pero creando los atributos: "localidad", "pais", "direccion", "cp" y "telefono". Todas de tipo "String".

| ▼ Attributes |          |  |
|--------------|----------|--|
| Attribute ▲  | Type     |  |
| S direccion  | String ↕ |  |
| S localidad  | String ↕ |  |
| S pais       | String ↕ |  |
| S telefono   | String ↕ |  |
| + -          |          |  |

Atributos DetallePersona

Finalmente nos queda relacionar de algún modo las dos entidades, para ello añadimos una

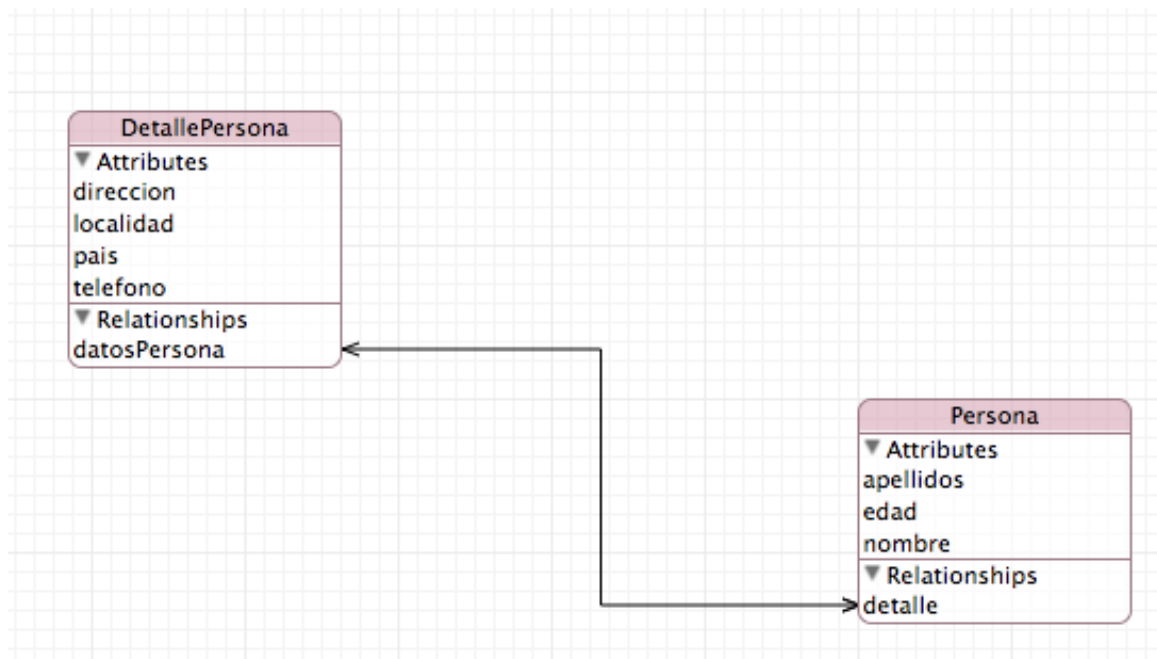
relación dentro de la entidad de "Persona" hacia "DetallePersona" simplemente haciendo click sobre el botón (+) de la sección de "Relationships". A la relación le ponemos como nombre "detalle", seleccionamos como destino la entidad "DetallePersona" y será "No inverse". Este tipo de relación es uno a uno. Por dentro, *Core Data* realizará crear una clave ajena en la tabla de persona que apunte a DetallePersona, pero eso a nosotros no nos interesa.



Vista general modelo de datos (1)

*Apple* recomienda que siempre que hagamos una relación de una entidad a otra hagamos también la relación inversa, por lo tanto hacemos justo lo anterior pero al revés (de "DetallePersona" a "Persona") con la diferencia que en la columna de "Inverse" seleccionamos "datosPersona".

Ya tenemos las entidades y las relaciones creadas. Si hacemos click en el botón de "Editor Style", en la parte de abajo a la derecha del editor, podemos ver de una forma más clara la estructura que hemos diseñado:



Vista general modelo de datos (2)

### 7.3.3. Probando el modelo

Una vez que tenemos el modelo creado pasamos a probarlo para comprobar que está todo correcto, para ello abrimos la clase delegada (`sesion04_ejemplo2AppDelegate.m`) y añadimos el siguiente fragmento de código arriba del método `applicationDidFinishLaunching`:

```

NSManagedObjectContext *context = [self managedObjectContext];
NSManagedObject *persona = [NSEntityDescription
insertNewObjectForEntityForName:@"Persona"
inManagedObjectContext:context];

[persona setValue:@"Juan" forKey:@"nombre"];
[persona setValue:@"Martinez" forKey:@"apellidos"];
[persona setValue:[NSNumber numberWithInt:28] forKey:@"edad"];

NSManagedObject *detallePersona = [NSEntityDescription
insertNewObjectForEntityForName:
@"DetallePersona"
inManagedObjectContext:context];

[detallePersona setValue:@"Calle Falsa, 123" forKey:@"direccion"];
[detallePersona setValue:@"Alicante" forKey:@"localidad"];
[detallePersona setValue:@"España" forKey:@"pais"];
[detallePersona setValue:@"965656565" forKey:@"telefono"];

[detallePersona setValue:persona forKey:@"datosPersona"];
[persona setValue:detallePersona forKey:@"detalle"];

NSError *error;
if (![context save:&error]) {
    NSLog(@"Uhh... Algo ha fallado: %@", [error
localizedDescription]);
}

```



```
}
```

Con este código hemos creado un objeto "Persona" enlazado con otro objeto "DetallePersona". ¡Como puedes ver no hemos necesitado nada de código *SQL*!. Primero creamos un objeto del tipo `NSManagedObjectContext` que almacena el contexto de *Core Data*. Después creamos el objeto "Persona" en sí con `NSManagedObject`. A este objeto le pondremos los valores para cada una de las propiedades definidas en el modelo. Hacemos lo mismo con "DetallePersona". Finalmente relacionamos el objeto "Persona" con el objeto "DetallePersona".

Compilamos el proyecto, comprobamos que todo funciona bien y no salen errores.

Para comprobar que los datos se han almacenado correctamente en nuestro dispositivo los mostramos en la consola de la siguiente manera:

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription
                               entityForName:@"Persona"
                               inManagedObjectContext:context];
[fetchRequest setEntity:entity];
NSArray *fetchedObjects = [context executeFetchRequest:fetchRequest
error:&error];
for (NSManagedObject *info in fetchedObjects) {
    NSLog(@"Nombre: %@", [info valueForKey:@"nombre"]);
    NSLog(@"Apellidos: %@", [info valueForKey:@"apellidos"]);
    NSLog(@"Edad: %d", (NSInteger)[info valueForKey:@"edad"]
intValue);

    NSManagedObject *details = [info valueForKey:@"detalle"];
    NSLog(@"Direccion: %@", [details valueForKey:@"direccion"]);
    NSLog(@"Localidad: %@", [details valueForKey:@"localidad"]);
    NSLog(@"Pais: %@", [details valueForKey:@"pais"]);
    NSLog(@"Telefono: %@", [details valueForKey:@"telefono"]);

    NSLog(@"-----");
}
[fetchRequest release];
```

El código de arriba lo escribiremos dentro del método `didFinishLaunchingWithOptions` de la clase delegada. Lo que hace este código es recorrer todos los objetos almacenados en memoria y mostrarlos por consola. Si arrancamos la aplicación veremos que se muestran todos los datos de los objetos almacenados. Cada vez que arranquemos la aplicación se mostrarán más objetos ya que también los creamos.

Para listar los objetos usamos la clase `NSFetchRequest`. Esta clase es equivalente a ejecutar una sentencia "*Select*" de *SQL*. Al objeto `NSFetchRequest` le asignamos la entidad que queremos listar ("*SELECT \* FROM ??*"), en nuestro caso será "Persona". El listado de los objetos se obtiene mediante el método `executeFetchRequest` de la clase `NSManagedObjectContext` que hemos definido en el bloque anterior de código. Este método devolverá un `NSArray` con todos los objetos `NSManagedObject`.

```

2011-09-02 18:38:58.404 sesion04-ejemplo2[4893:207] Nombre: Juan
2011-09-02 18:38:58.405 sesion04-ejemplo2[4893:207] Apellidos: Martinez
2011-09-02 18:38:58.406 sesion04-ejemplo2[4893:207] Edad: 28
2011-09-02 18:38:58.406 sesion04-ejemplo2[4893:207] Direccion: Calle Falsa, 123
2011-09-02 18:38:58.407 sesion04-ejemplo2[4893:207] Localidad: Alicante
2011-09-02 18:38:58.408 sesion04-ejemplo2[4893:207] Pais: España
2011-09-02 18:38:58.408 sesion04-ejemplo2[4893:207] Telefono: 965656565
2011-09-02 18:38:58.409 sesion04-ejemplo2[4893:207] -----

```

Salida de consola

#### Nota

Si queremos ver qué sentencias SQL está ejecutando *Core Data* por debajo tenemos que activar el "flag" de debug, para ello seleccionamos la opción: "Product" > "Edit Scheme". En la columna de la izquierda seleccionamos el esquema que estamos utilizando para Debug y en la parte de la derecha, en el cuadro de "Arguments Passed on Launch" añadimos 2 parámetros: "-com.apple.CoreData.SQLDebug" y "1".

Ahora arrancamos de nuevo la aplicación y veremos que en la consola aparecerán las sentencias SQL utilizadas por *Core Data*:

```

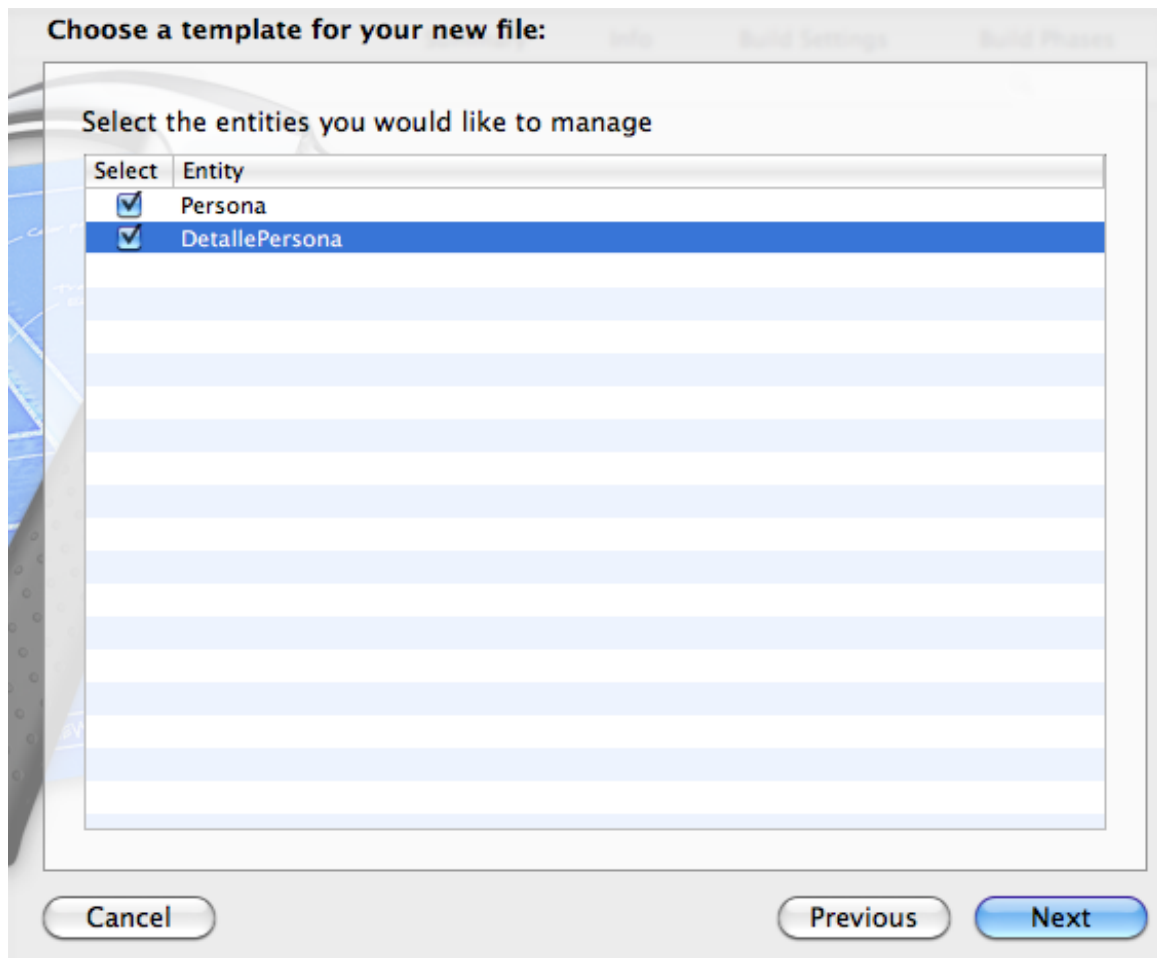
(...)
INSERT INTO ZDETALLEPERSONA(Z_PK, Z_ENT, Z_OPT, ZDATOSPERSONA, ZTELEFONO,
ZLOCALIDAD, ZDIRECCION, ZPAIS)
VALUES(?, ?, ?, ?, ?, ?, ?, ?)
 CoreData: sql: SELECT 0, t0.Z_PK, t0.Z_OPT, t0.ZTELEFONO, t0.ZLOCALIDAD,
t0.ZDIRECCION, t0.ZPAIS, t0.ZDATOSPERSONA
FROM ZDETALLEPERSONA t0 WHERE t0.Z_PK = ?
(...)

```

### 7.3.4. Generando los modelos automáticamente

Generar los modelos a mano directamente desde el editor no es lo más recomendado ya que podemos cometer errores de escritura, errores de tipo, etc. La mejor manera de hacerlo es creando un archivo para cada entidad. Esto se puede hacer desde 0, pero *xCode* tiene un generador de clases que es muy fácil de usar y ahorremos bastante tiempo. ¡Vamos a usarlo!

Hacemos click sobre el raíz de nuestro proyecto y seleccionamos "New File". En la columna de la izquierda seleccionamos "iOS" > "Core Data" y en el cuadro de la derecha "NSObject subclass". Click en "next". En la siguiente pantalla seleccionamos nuestro modelo de datos "sesion04\_ejemplo2" y click en "next". Ahora nos aseguramos de seleccionar las dos entidades que hemos creado anteriormente: "Persona" y "DetallePersona", click en "next". Seleccionamos el raíz de nuestro proyecto para guardar los ficheros que se generen y aceptamos.



### Ventana de selección de entidad

Ahora veremos que `xsource` ha creado automáticamente 4 ficheros nuevos dentro de nuestro proyecto: `Persona.m/h` y `DetallePersona.m/h` con todos los atributos especificados. Estas nuevas clases que heredan de `NSObject` serán las que utilicemos a partir de ahora.

Cambiamos el código de la clase delegada por el siguiente:

```

NSManagedObjectContext *context = [self managedObjectContext];
    Persona *datosPersona = [NSEntityDescription
                                insertNewObjectForEntityForName:@"Persona"
                                inManagedObjectContext:context];
    datosPersona.nombre = @"Juan";
    datosPersona.apellidos = @"Martinez";
    datosPersona.edad = [NSNumber numberWithInt:28];

    DetallePersona *detallePersona = [NSEntityDescription
insertNewObjectForEntityForName:@"DetallePersona"
                                inManagedObjectContext:context];
    detallePersona.direccion = @"Calle Falsa, 123";

```

```

detallePersona.localidad = @"Alicante";
detallePersona.pais = @"España";
detallePersona.telefono = @"965656565";

detallePersona.datosPersona = datosPersona;
datosPersona.detalle = detallePersona;

NSError *error;
if (![context save:&error]) {
    NSLog(@"Uhh... Algo ha fallado: %@", [error
localizedDescription]);
}

//Listamos los datos
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription
    entityForName:@"Persona"
    inManagedObjectContext:context];
[fetchRequest setEntity:entity];
NSArray *fetchedObjects = [context executeFetchRequest:fetchRequest
error:&error];
for (NSManagedObject *info in fetchedObjects) {
    NSLog(@"Nombre: %@", [info valueForKey:@"nombre"]);
    NSLog(@"Apellidos: %@", [info valueForKey:@"apellidos"]);
    NSLog(@"Edad: %d", (NSInteger)[[info valueForKey:@"edad"]
integerValue]);

    NSManagedObject *details = [info valueForKey:@"detalle"];
    NSLog(@"Direccion: %@", [details valueForKey:@"direccion"]);
    NSLog(@"Localidad: %@", [details valueForKey:@"localidad"]);
    NSLog(@"Pais: %@", [details valueForKey:@"pais"]);
    NSLog(@"Telefono: %@", [details valueForKey:@"telefono"]);

    NSLog(@"-----");
}
[fetchRequest release];

```

Ejecutamos el proyecto y comprobamos que en la consola aparece lo mismo que antes, pero ahora tenemos el código bastante más claro.

### 7.3.5. Creando la vista de tabla

Ahora vamos a mostrar los objetos que antes hemos listado por consola en una vista de tabla `UITableView`. Para ello creamos un `UITableViewController`:

Click en "New file..." > "UIViewController subclass". En la siguiente pantalla nos aseguramos de seleccionar "Subclass of UITableViewController", de esta forma nos aparecerán ya todos los métodos de la tabla creados por defecto. Guardamos el archivo con el nombre que queramos, por ejemplo: "PersonasViewController".

Abrimos ahora "PersonasViewController.h" y añadimos 2 atributos a la clase:

- Un `NSArray` que se será el listado de personas (objetos de la clase `Persona`).
- Una referencia a `NSManagedObjectContext`.

El fichero "PersonasViewController.h" quedará de la siguiente manera:

```
//imports iniciales

@interface PersonasViewController : UITableViewController {
    NSArray *_listaPersonas;
    NSManagedObjectContext *_context;
}

@property (nonatomic, retain) NSArray *listaPersonas;
@property (nonatomic, retain) NSManagedObjectContext *context;

@end
```

Ahora en el .m importamos en el encabezado las clases "Persona.h" y "DetallePersona.h" y en el método "viewDidLoad" creamos el array listaPersonas con todos los objetos Persona de forma similar a cuando lo hicimos en el apartado anterior:

```
#import "PersonasViewController.h"

#import "Persona.h"
#import "DetallePersona.h"

@implementation PersonasViewController

@synthesize listaPersonas = _listaPersonas;
@synthesize context = _context;
```

En viewDidLoad:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription
                                   entityForName:@"Persona"
                                   inManagedObjectContext:_context];

    [fetchRequest setEntity:entity];
    NSError *error;
    self.listaPersonas = [_context executeFetchRequest:fetchRequest
error:&error];
    self.title = @"Listado Personas";
    [fetchRequest release];
}
```

Ahora completamos el resto de métodos heredados de UITableViewController de la misma forma que hicimos en sesiones anteriores:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
}
```

```

    // Return the number of rows in the section.
    return [_listaPersonas count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

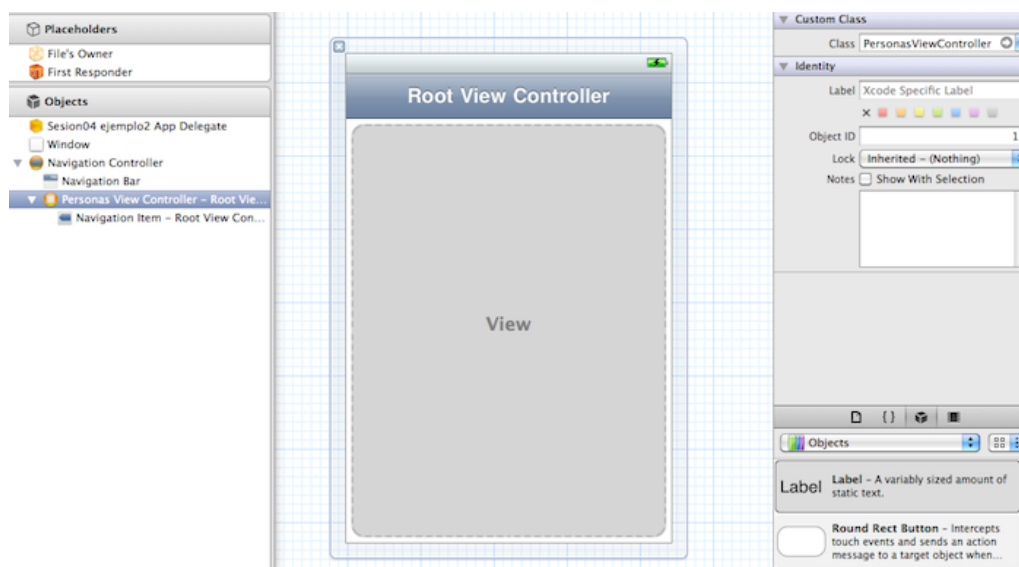
    UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleSubtitle
        reuseIdentifier:CellIdentifier] autorelease];
    }

    // Configure the cell...
    Persona *p = (Persona *)[_listaPersonas objectAtIndex:indexPath.row];
    cell.textLabel.text = p.nombre;
    cell.detailTextLabel.text = [NSString stringWithFormat:@"%d años",
    [p.edad intValue]];

    return cell;
}

```

Ahora para poder mostrar la tabla dentro de nuestra aplicación debemos asignarla dentro de la clase delegada, para ello abrimos la vista "MainWindow.xib" y arrastramos una controladora de navegación "Navigation Controller" a la lista de objetos de la izquierda. La desplegamos y dentro de "View Controller" en la pestaña de la derecha de "Identity Inspector" escribimos el nombre de nuestra clase que tiene la vista de la tabla: "PersonasViewController".



Pantalla del diseñador

Ahora abrimos el fichero "sesion04\_ejemplo2AppDelegate.h" y creamos un

"UINavigationController" dentro de la declaración de variables dejando el fichero como sigue:

```
//imports iniciales

@interface sesion04_ejemplo2AppDelegate : NSObject <UIApplicationDelegate>
{
    UINavigationController *_navController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController
*navController;

@property (nonatomic, retain, readonly) NSManagedObjectContext
*managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectContext
*managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectContext
*managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectContext
*managedObjectContext;

- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;

@end
```

Dentro del .m importamos la clase "PersonasViewController.h" y completamos el @synthesize:

```
@synthesize navController = _navController;
```

Y por último, justo antes de la llamada al método "makeKeyAndVisible" escribimos el siguiente código para asignar el contexto de la clase delegada al "PersonasViewController":

```
PersonasViewController *root = (PersonasViewController *) [_navController
topViewController];
root.context = [self managedObjectContext];
[self.window addSubview:_navController.view];
```

Ahora compilamos y ejecutamos nuestro proyecto y, si todo ha ido bien, debería de salir la tabla con los datos de las personas:



Listado de personas

### 7.3.6. Migraciones de bases de datos con Core Data

Una vez que tenemos diseñada nuestra base de datos inicial, ¿qué pasaría si queremos modificarla para añadirle un campo más a una tabla? ¿Qué hará Core Data para realizar la migración de la versión anterior a la actual?

Siguiendo con el ejemplo anterior, si después de ejecutar la aplicación en nuestro dispositivo o simulador modificamos la estructura de la base de datos añadiendo un atributo nuevo en la entidad "DetallePersona", por ejemplo: "codigoPostal", con el tipo "Integer 16", veremos que al volver a arrancar la aplicación nos aparecerá un error en tiempo de ejecución similar al siguiente:

```
Unresolved error Error Domain=NSCocoaErrorDomain Code=134100 "The
operation
couldn't be completed.
(Cocoa error 134100.)" UserInfo=0x6b6c680 {metadata=<CFBasicHash 0x6b68380
[0x1337b38]>{type = immutable dict, count = 7,entries =>
  2 : <CFString 0x6b70320 [0x1337b38]>{contents =
"NSStoreModelVersionIdentifiers"}
=
  <CFArray 0x6b706e0 [0x1337b38]>{type = immutable, count = 1,
values = (
  0 : <CFString 0x1332cd8 [0x1337b38]>{contents = ""}
)}}
  4 : <CFString 0x6b70350 [0x1337b38]>{contents =
"NSPersistenceFrameworkVersion"}
=
```



```

        <CFNumber 0x6b6f2c0 [0x1337b38]>{value = +386, type =
kCFNumberSInt64Type}
        6 : <CFString 0x6b70380 [0x1337b38]>{contents =
"NSStoreModelVersionHashes"} =
        <CFBasicHash 0x6b707e0 [0x1337b38]>{type = immutable dict, count =
2,
entries =>
        1 : <CFString 0x6b70700 [0x1337b38]>{contents = "Persona"} =
<CFData 0x6b70740
[0x1337b38]>{length = 32, capacity = 32, bytes =
0x1c50e5a379ff0ddc3cda7b82a3934c5e ... 75649ac3c919beea}
        2 : <CFString 0x6b70720 [0x1337b38]>{contents = "DetallePersona"}
=
        <CFData 0x6b70790
[0x1337b38]>{length = 32, capacity = 32, bytes =
0xb7ba2eb498f9alacdd6630d56f6cd12c ... e8963dd3e488ba95}
}

        7 : <CFString 0x10e3aa8 [0x1337b38]>{contents = "NSStoreUUID"} =
<CFString 0x6b70510
[0x1337b38]>{contents = "4F80A909-DA41-48ED-B24D-DBA73EC2BB9E"}
        8 : <CFString 0x10e3948 [0x1337b38]>{contents = "NSStoreType"} =
<CFString 0x10e3958
[0x1337b38]>{contents = "SQLite"}
        9 : <CFString 0x6b703b0 [0x1337b38]>{contents =
"_NSAutoVacuumLevel"} = <CFString
0x6b70830 [0x1337b38]>{contents = "2"}
        10 : <FString 0x6b703d0 [0x1337b38]>{contents =
"NSStoreModelVersionHashesVersion"}
=
        <CFNumber 0x6b61950 [0x1337b38]>{value = +3, type =
kCFNumberSInt32Type}
}
, reason=The model used to open the store is incompatible with the one
used to
create the store}, {
    metadata = {
        NSPersistenceFrameworkVersion = 386;
        NSStoreModelVersionHashes = {
            DetallePersona = <b7ba2eb4 98f9alac dd6630d5 6f6cd12c f8900077
46b16f00
            e8963dd3 e488ba95>;
            Persona = <1c50e5a3 79ff0ddc 3cda7b82 a3934c5e 5cb4ee4b
4ac98838 75649ac3
            c919beea>;
        };
        NSStoreModelVersionHashesVersion = 3;
        NSStoreModelVersionIdentifiers = (
            ""
        );
        NSStoreType = SQLite;
        NSStoreUUID = "4F80A909-DA41-48ED-B24D-DBA73EC2BB9E";
        "_NSAutoVacuumLevel" = 2;
    };
    reason = "The model used to open the store is incompatible with the
one used
to create the store";
}

```

Este error es muy común cuando estamos desarrollando aplicaciones con Core Data. Básicamente nos dice que el modelo de base de datos que está intentando cargar de memoria es distinto al que se indica en la aplicación, esto es porque no hemos indicado ningún método de migración. Realmente existen dos formas de solucionar este error:

- Por un lado, y **sólo cuando estemos desarrollando y no tengamos ninguna versión de nuestra aplicación lanzada en la App Store**, podremos eliminar la base de datos del dispositivo / simulador simplemente borrando la aplicación de él. También podemos ejecutar el siguiente código una sola vez: `[[NSFileManager defaultManager] removeItemAtURL:storeURL error:nil]`
- En el caso de que estemos realizando una migración sobre una versión ya publicada en la App Store el método anterior no nos serviría, ya que obligaríamos al usuario a borrar su aplicación del dispositivo con lo que todos los datos que tuviera se perderían. Esto no es viable, por lo que necesitaremos usar otro método para realizar la migración y solucionar el error de una forma más "limpia", este método es el que vamos a comentar a continuación.

XCode soporta Versioning para Core Data, esto es que podemos crear versiones de la base de datos de una forma sencilla si seguimos una serie de pasos:

1) Añadimos las siguientes opciones en forma de `NSDictionary` a nuestro código del `persistentStoreCoordinator`

```
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:YES],
    NSMigratePersistentStoresAutomaticallyOption,
    [NSNumber numberWithInt:YES],
    NSInferMappingModelAutomaticallyOption,
    nil];
```

Y modificamos el método `addPersistentStoreWithType` pasándole como parámetro el diccionario `options`:

```
if (![__persistentStoreCoordinator
    addPersistentStoreWithType:NSSQLiteStoreType
    configuration:nil URL:storeURL options:options error:&error])
{
    ...
}
```

Con el código anterior indicamos al `persistentStoreCoordinator` que la migración, en el caso de que se deba de realizar, sea de forma automática.

#### Atención

Deberemos de tener en cuenta que este tipo de migración que estamos estudiando en este punto sólo es válida para cambios simples en Core Data, lo que Apple llama como *lightweight migration*. Este tipo de migración admite una serie de cambios que se indican dentro de la documentación de Apple, para cambios más complejos deberemos realizar migraciones personalizadas y mucho más complejas. El documento en donde Apple explica el tipo de migraciones que existen para Core Data y su forma de implementarlas lo podemos consultar desde [esta web](#).

2) Ahora deberemos de crear la versión del esquema de Core Data dentro de XCode, para

ello seleccionamos el fichero del esquema `sesion04_ejemplo2.xcdatamodeld` y seleccionamos en el menú principal: *Editor > Add Model Version*. A la nueva versión la llamaremos: `sesion04_ejemplo_2` (versión 2). Una vez que hayamos pulsado sobre *Finish* se nos habrá creado la nueva versión del esquema dentro de la principal, y estará marcada con un "tick" la versión anterior.

3) Una vez que tenemos creada la nueva versión, deberemos modificarla añadiéndole el nuevo atributo. Seleccionamos la versión recién creada `sesion04_ejemplo_2.xcdatamodel` y añadimos a la entidad `DetallePersona` el atributo `codigoPostal`. **Atención:** Si este atributo lo hemos añadido anteriormente deberemos de borrarlo de la versión anterior y añadirlo en la segunda versión.

4) Ahora nos queda indicar a Core Data que queremos usar la versión 2 de la base de datos para nuestra aplicación. Para ello seleccionamos el esquema principal (el raíz) `sesion04_ejemplo2.xcdatamodeld` y en la ventana de la derecha, en la pestaña de *File Inspector* seleccionamos como Versión actual la versión 2, esto lo hacemos dentro del apartado de *Versioned Core Data Model*. Automáticamente veremos que el símbolo de *tick* cambia a la versión 2.

5) Ahora ya podemos volver a compilar y veremos que no nos aparece el error y funciona todo según lo esperado.

## 8. Persistencia de datos en iOS: User Defaults y Core Data - Ejercicios

### 8.1. Usando las variables de usuario (User Defaults)

En este ejercicio vamos a usar las variables de usuario para almacenar y mostrar los datos del dispositivo que estamos usando y las veces que hemos arrancado la aplicación.

Crear un proyecto nuevo usando la plantilla "Single View Application" con el nombre "ejercicio-userdefaults-ios". Se debe de ejecutar en iPhone e iPad (*Universal*). Como identificador de empresa escribiremos *es.ua.jtech* y como prefijo de clase *UA*. Debe de tener el ARC activado (desmarcar el resto de opciones).

#### Ayuda: métodos de UIDevice

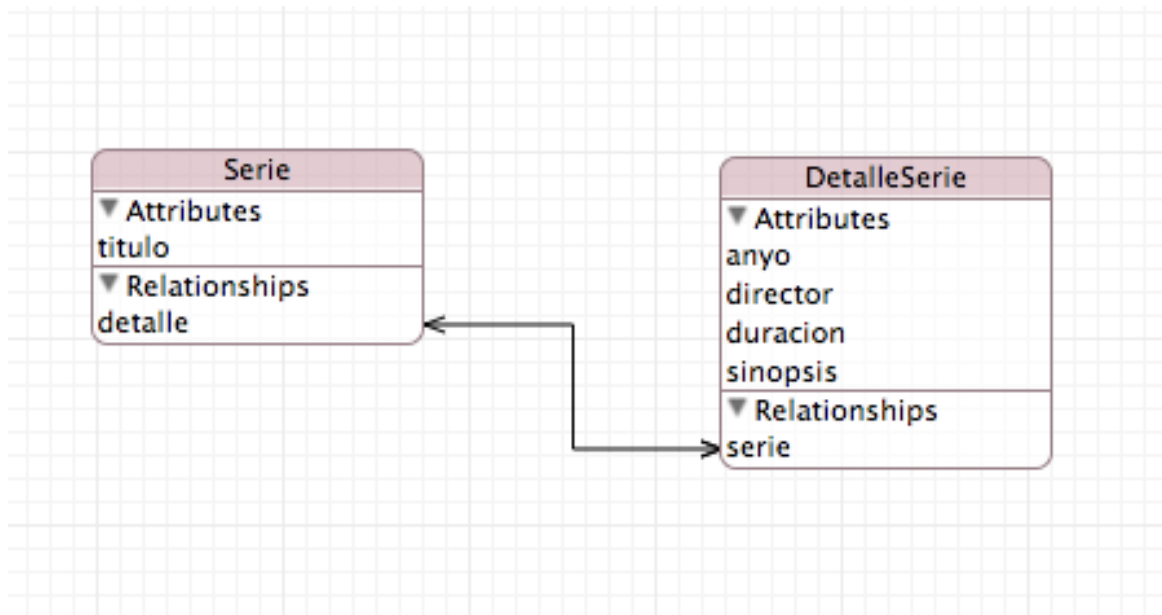
Para obtener los datos del dispositivo usaremos el *singleton* `UIDevice`, estos son algunos de los métodos que podemos usar: `[[UIDevice currentDevice] name];` `[[UIDevice currentDevice] systemName];` `[[UIDevice currentDevice] systemVersion];` `[[UIDevice currentDevice] model];` `[[UIDevice currentDevice] localizedModel];`

### 8.2. Diseñando un modelo de datos sencillo con Core Data

En este ejercicio diseñaremos un modelo de datos de dos tablas relacionadas entre ellas, añadiremos datos por código y los mostraremos en una vista de tabla. La base de datos que vamos a implementar constará de dos tablas, por un lado una que contendrá los títulos de series y otra que tendrá los detalles. A la primera la llamaremos "*Serie*" y la segunda "*DetalleSerie*". Para realizar correctamente el ejercicio deberemos seguir los siguientes pasos:

- 1) Crear un proyecto nuevo usando la plantilla "Master-Detail Application" con el nombre "ejercicio-coredata-ios". Se debe de ejecutar en iPhone, como identificador de empresa escribiremos *es.ua.jtech* y como prefijo de clase *UA*. Debe de tener el ARC activado y la opción de "Use Core Data" (desmarcar el resto de opciones).

- 2) Diseñar el siguiente modelo de datos:



Esquema modelo de datos

- 3) Revisar el fichero `UAppDelegate.m` y comprobar que tenemos todos los métodos necesarios para que funcione Core Data en nuestra aplicación.
- 4) Modificar el fichero `UAMasterViewController.m` para que se muestren los títulos de las series en la tabla.
- 5) Probar la aplicación añadiendo unos títulos de prueba.

### 8.3. (\*) Migrando datos con Core Data

Una vez que hemos diseñado la Base de Datos con Core Data vamos a modificarla añadiendo un atributo más a la tabla de *DetalleSerie*: "valoracion".

- a) ¿Qué pasa si cambiamos el modelo de datos y arrancamos la aplicación? ¿Qué error nos da y por qué?
- b) ¿Cómo podemos evitar este tipo de errores cuando estemos desarrollando? ¿Y si estamos en producción, con la aplicación ya a la venta?
- c) Crea una nueva versión del modelo.
- d) Modifica el código necesario dentro de la clase `UAppDelegate` para evitar este tipo de errores cuando estemos en un entorno de producción.
- e) Vuelve a arrancar la aplicación y comprueba que funciona bien.

