MODULE Deep Dive into iOS App Development
SESSION [2]
SESSION TITLE: [Performing Data Persistence]

Sources:

1. iOS Database application [Chapter 2] [9781118391846]

MODULE Objectives

At the end of this module, you will be able to:

- Create a SQLite database
- Build a database and an iOS application to view master-detail relationships
- Connect your application to a database and display its data
- Run SQL statements against an SQLite database to insert and select data

Session Objectives

At the end of this session, you will be able to:

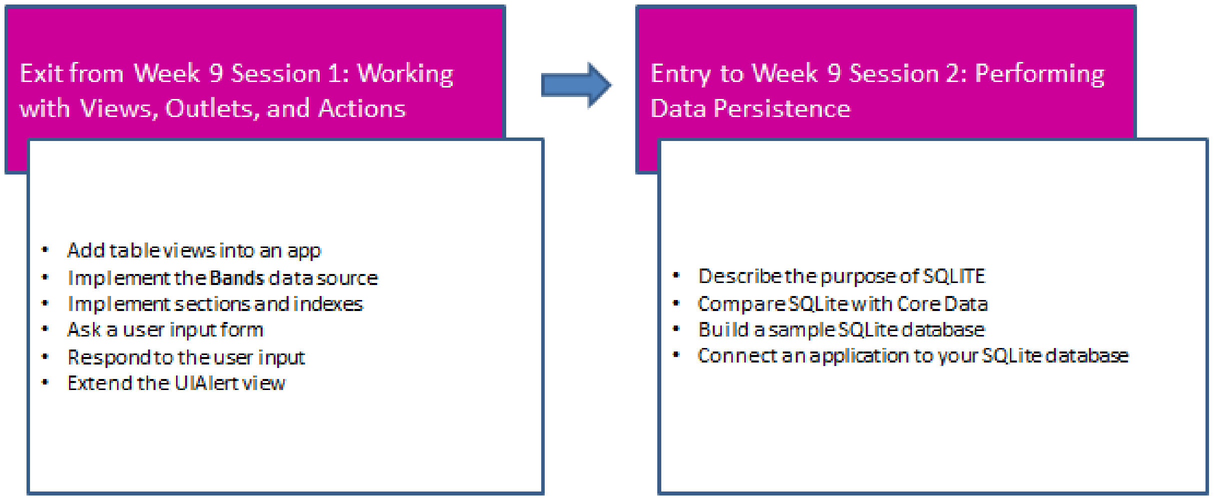
- Describe the purpose of SQLITE
- Compare SQLite with Core Data
- Build a sample SQLite database
- Connect an application to your SQLite database

<H1> Introduction

SQLite is the database engine that backs many iOS applications. **Xcode** uses the SQLite database that inherits from `NSObject`, which is declared in the `FMDatabase.h` header file in Objective-C.

In this session, you will learn how to build an application that uses **SQLite** as its backing data store. To use SQLite, you will first learn to create a database using the **command-line application** provided with Mac OS X. This tool will enable you to create the schema of your database, populate it with data, and perform queries.

Then, you will learn how to **deploy your database with an iOS application**, **connect to it in Xcode**, and **display the results or your SQL queries**. Finally, you will learn how to **build a fully functional database application** to view data that has a Master-Detail relationship, which is a **Product Catalog** in this case.



Exit from Week 9 Session 1: Working with Views, Outlets, and Actions

- Add table views into an app
- Implement the **Bands** data source
- Implement sections and indexes
- Ask a user input form
- Respond to the user input
- Extend the UIAlert view

Entry to Week 9 Session 2: Performing Data Persistence

- Describe the purpose of SQLITE
- Compare SQLite with Core Data
- Build a sample SQLite database
- Connect an application to your SQLite database

<H1> What is SQLite?

# iOS Database application, Chapter 2 and Page No. 22#

SQLite is an open source library, written in C, which implements a self-contained SQL relational database engine. You can use SQLite to store a large amount of relational data. The developers of SQLite have optimized it for use on embedded devices like the iPhone and iPad.

Although the **Core Data** application programming interface (API) is also designed to store data on iOS, its primary purpose is to persist objects created by your application. SQLite excels when preloading your application with a large amount of data, whereas Core Data excels at managing data created on the device.

<H2> The SQLite Library

# iOS Database application, Chapter 2 and Page No. 22#

SQLite is a library because it is a fully self-contained SQL database engine. All the data required to implement the database is stored in a single, cross-platform disk file. It requires a few external libraries and a little support from the operating system, and is therefore ideal for a mobile platform like iOS.

Apple has adopted SQLite for use on iOS for other reasons as well, including its small footprint.

- Weighing in at less than 300K, the library is small enough to use effectively on mobile devices with limited memory.
- It requires no configuration files, has no setup procedure, and needs no administration.
- You can just drop your database file on the device, include the SQLite library in your iOS project, and you are ready to roll.

SQLite implements most of the SQL92 standard. Therefore, working with a SQLite database is intuitive if you already know SQL. You should keep in mind that there are some features of SQL92 that SQLite does not currently support. These features include:

- `RIGHT` and `FULL OUTER JOIN`
- Complete support for `ALTER TABLE`
- `FOR EACH STATEMENT` triggers
- Writeable `VIEWs`
- `GRANT` and REVOKE permissions

Technical Stuff

Objective-C has the `FMDatabase.h` file for the database. You can use this header file in your **Swift** project by bridging it into your project from the `NSObject` class, which is importing this header file into your project.

Quick Tip

As you learn about SQLite, you can also read about the SQL92 standard. Since SQLite adopts most of the features of SQL92, its knowledge will help you understand their similarities and differences.

<H1> SQLite and Core Data

# iOS Database application, Chapter 2 and Page No. 23#

Core Data is not a relational database like **SQLite**. Core Data is an object persistence framework. Its primary purpose is to provide the developer with a framework to retain objects that the application creates.

Core Data allows you to model your data as objects using a convenient graphical interface built into Xcode. You can then manipulate those objects in code with an extensive set of APIs. Designing and defining your data objects using the graphical interface can simplify the creation of the Model portion of the Model-View-Controller (MVC) architecture.

Core Data can use SQLite, among other storage types, as a **backing store for its data**. This causes some confusion for developers. It is a common misconception that because Core Data can use SQLite to store data, Core Data is a relational database. This is not correct. As mentioned, Core Data is not an implementation of a relational database. Although Core Data uses SQLite in the background to store your data, it does not store the data in a way that is directly accessible to the developer. In fact, you should never attempt to manually modify the backing database structure or its data. Only the Core Data framework should manipulate the structure of the database and the data itself.

Quick Tip

Feel free to open the SQLite database and look at it if you are curious, but making any modifications to the data or database structure will likely invalidate it and cause problems when trying to access it using Core Data.

The Big Picture

Although Core Data is the preferred framework for dealing with data that you create on a device, SQLite remains a useful tool for iOS developers. If you need the functionality that a relational database provides, you should strongly consider using SQLite directly. However, if you only need to persist objects created during the use of your application, you should consider using Core Data.

Using Core data in your project is more appropriate as per metrics i.e. the code you write for application model layer is 50% to 70% less as measured by lines. Core Data also offers memory scalability, error-handling, and excellent security as it integrates well with the OS X tool chain. This enables you to further shorten your application design and debugging cycles.

You might choose to use a SQLite database if you need to preload a large amount of data on a device. Take, for example, a global positional system (GPS) navigation application. Navigation applications need a great deal of data, including points-of-interest (POIs) and maps. A good option for the architectural design is to create a SQLite database that contains all the POIs and map data. You can then deploy that database with your application and use SQLite APIs to access the database.

Quick Tip

Core Data is the recommended framework for creating data on a device, but you may want to forego Core Data and use the SQLite API directly.

<H1> Building a Sample Database

# iOS Database application, Chapter 2 and Page No. 24#

Before you start designing a database and an application, you need to understand what the application will do. In the real world, you will (or should) get a detailed specification defining what the application should do and how it should look. Of course, the implementation, or how it should work, is up to the designer and developer.

Here are some simple requirements for the catalog application:

- The purpose of the application is to display your company's catalog of widgets. Each widget will have:
    - A manufacturer
    - A product name
    - Some details about the product
    - The price of the product
    - The quantity on hand
    - The country of origin
    - A picture of the product

- The application should start up by showing a list of products. Tapping on a product should bring up a detail page with detailed information about the product.

The Big Picture

It is suspected that the database gurus out there are pulling their hair out right now because it is common wisdom that the user interface and the data should be completely decoupled and that you should normalize the data independently.

However, when developing applications that you will run on an embedded device like the iPhone or iPad, performance is an important concern. Data that you have fully normalized, with no duplicated data, can have a negative impact on performance.

Sometimes the cost to execute a complicated query is higher than the cost of maintaining the same data in two tables. This does not suggest that you should not normalize your data at all; just keep in mind how you will display the data while working through the database design process.

<H2> Designing the Database

# iOS Database application, Chapter 2 and Page No. 25#

Normalization is the process of breaking down your data in a way that makes it easy to query.

This process helps to avoid common problems in database storage, such as duplication of data. For example, when creating your database, a designer may want to store all the data in a single table, as in **Figure 1**.

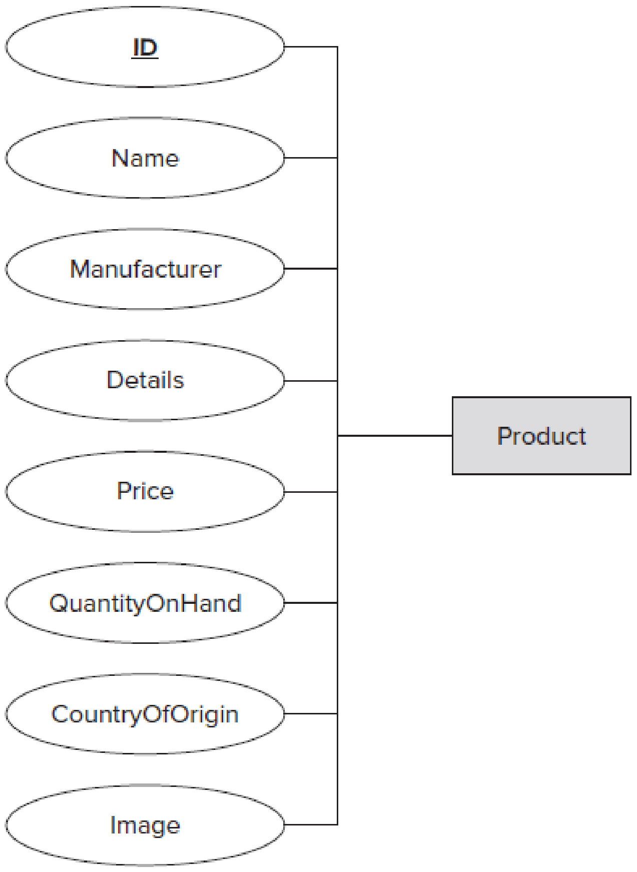
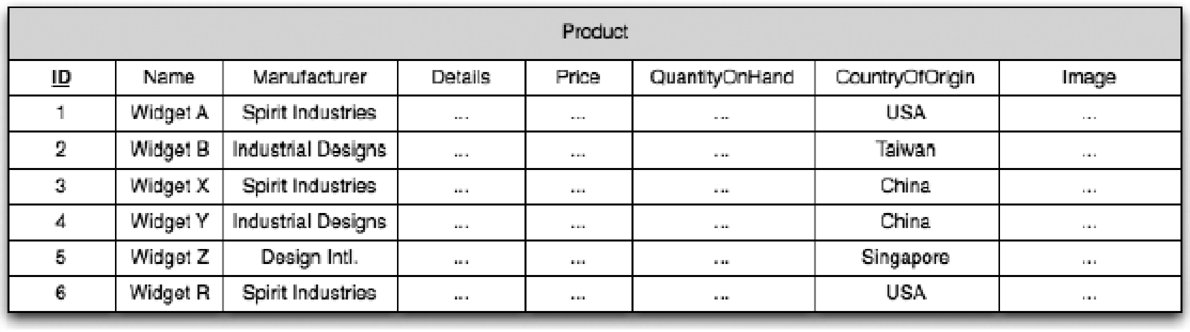


Figure 1: Storing the Data in a Product Single Table

If you are unfamiliar with **Entity-Relationship Diagrams** (ERDs), the box represents an entity or table in the database. The ovals that are connected to the entity are the attributes of that entity or the fields of the table. So, this diagram shows one table with each attribute as a field in the table.

The problem with this database design is that you have not normalized it. There will be duplication of data if more than one product in the catalog is manufactured by the same manufacturer or if more than one product is manufactured in a specific country. In that case, the data may look something like **Figure 2**.



Product							
ID	Name	Manufacturer	Details	Price	QuantityOnHand	CountryOfOrigin	Image
1	Widget A	Spirit Industries	...	...	...	USA	...
2	Widget B	Industrial Designs	...	...	...	Taiwan	...
3	Widget X	Spirit Industries	...	...	...	China	...
4	Widget Y	Industrial Designs	...	...	...	China	...
5	Widget Z	Design Intl.	...	...	...	Singapore	...
6	Widget R	Spirit Industries	...	...	...	USA	...

Figure 2: Data in a Single Table

You can see that the same manufacturer has more than one product in the database. In addition, there is more than one product made in a specific country. This design is a maintenance problem.

What happens if the data entry person populating the database types in **Spirit Industries** for item 1 writes **Spit Industries** for item 3?

It will appear in the application that two different companies make these products, when in reality **Spirit Industries** manufactures both products. This is a data integrity problem that you can avoid by normalizing the data.

You can remove the manufacturer and country of origin from the **Product** table and create new tables for these fields. Then, in the **Product** table, you can just reference the value in the related tables. Additionally, this new design could allow you to add more detail about the manufacturer, such as address, contact name, and so on. For the sake of simplicity, you will not be doing that in this example, but proper normalization makes this type of flexibility possible.

The new design should look something like **Figure 3**.

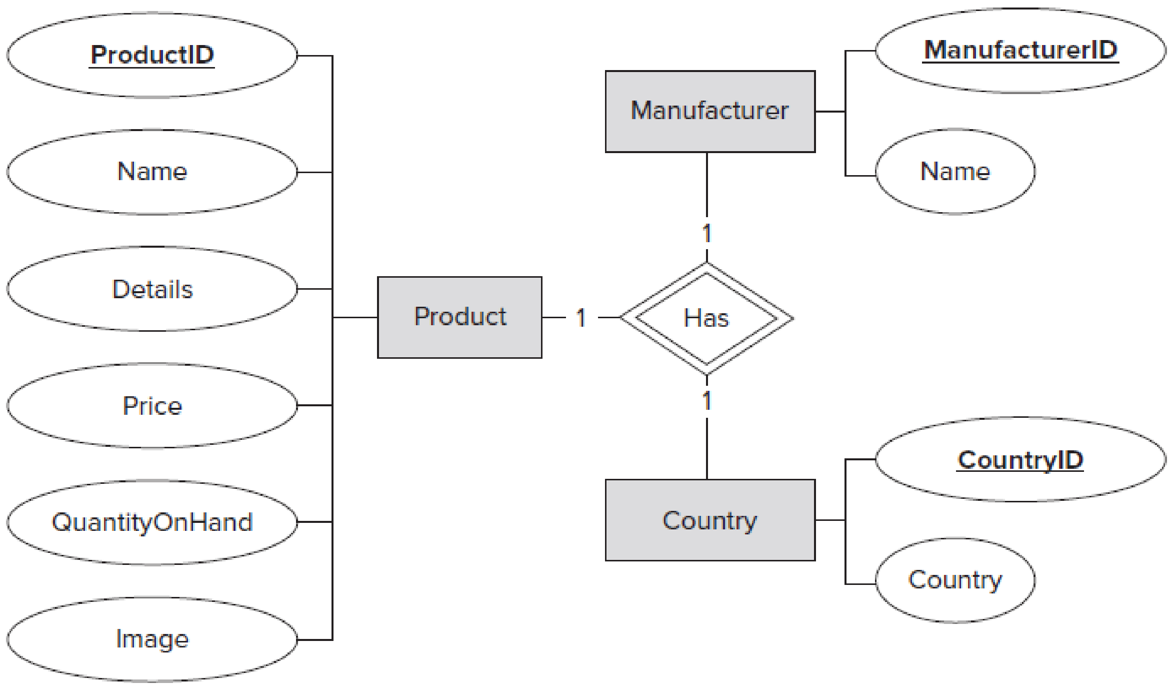


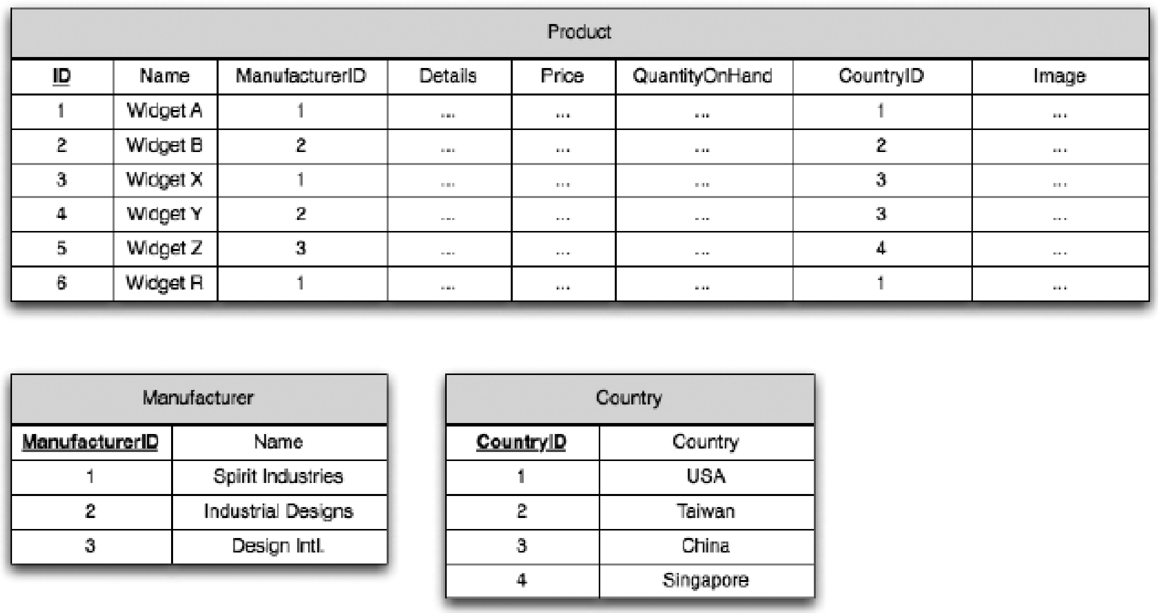
Figure 3: Normalized Database Tables

You can see that, instead of specifying the manufacturer and origin explicitly in the main **Product** table, you just reference the ID in the related tables. The fact that you can relate data in one table to data in another gives a relational database both its name and its power.

QUICK TIP

Although normalization is important, you can take it too far and overnormalize your data. For instance, you could create a separate table for price. In that case, all products that cost $1 would reference a row in a **Price** table that contains the value $1. Although this would eliminate the duplication of data for all products that cost $1, it would be painful to write the code to maintain the relationship between the Product and Price tables. This is overnormalized, and you should avoid it.

Figure 4 shows what the new normalized database will look like.



Product							
ID	Name	ManufacturerID	Details	Price	QuantityOnHand	CountryID	Image
1	Widget A	1	...	...	...	1	...
2	Widget B	2	...	...	...	2	...
3	Widget X	1	...	...	...	3	...
4	Widget Y	2	...	...	...	3	...
5	Widget Z	3	...	...	...	4	...
6	Widget R	1	...	...	...	1	...

Manufacturer	
ManufacturerID	Name
1	Spirit Industries
2	Industrial Designs
3	Design Intl.

Country	
CountryID	Country
1	USA
2	Taiwan
3	China
4	Singapore

Figure 4: Data in Normalized Tables

The Big Picture

It is good to use the normalization process because it helps to avoid the problem of duplication of data. Normalization, therefore, helps to remove the data integrity problem. However, remember that you are writing applications for a mobile platform with limited central processing unit (CPU) capability. You will pay a penalty for using overly complex SQL to access your data. You are better off in some instances repeating data instead of using relationships.

<H2> *Creating a Database*

# iOS Database application, Chapter 2 and Page No. 27#

You can use a couple of different methods to create, modify, and populate your SQLite database. Using a command-line interface may not seem optimal in these days of graphical interfaces, but the command line does have its advantages.

One feature that stands out is that you can **create and populate a database** using the command-line interface and scripts. For example, you can write a PERL script that gets data out of an enterprise database such as Oracle or MySQL and then creates a SQLite database with a subset of the data.

Although scripting is beyond the scope of this course, you will learn how to create and populate a database using the command-line tool.

You can also use the command-line interface to import data from a file into a table, read in and execute a file that contains SQL, and output data from a database in a variety of formats, including:

- Comma-separated values

- Left-aligned columns
- HTML <table> code
- SQL INSERT statements for TABLE
- One value per line
- Values delimited by .separator string
- Tab-separated values
- Tool Command Language (TCL) list elements

<H3> Steps to Create a Database

- To start the command-line tool, you need to bring up a terminal window.
- Next, change to the directory where you want to store your database file. Start the command-line tool and create your new database by typing `sqlite3 catalog.db` at the command prompt. This command starts the command-line tool and attaches the database `catalog.db`.
- The ATTACH DATABASE command either attaches an existing database to the SQLite tool or creates a new database if the specified file does not already exist. You can attach multiple databases to a single instance of the command-line tool and reference data in each database by using dot notation in the form `database-name.table-name`. You can use this powerful feature to migrate data from one database to another.

Technical Stuff

Aside from being able to execute SQL at the command line, the command-line interface tool has various meta commands that you can use to control the tool itself. You can display these by typing **.help** from the command line. You can see what databases you have attached to the current instance of the tool by typing **.databases** at the command line. You can quit the command-line tool by typing **.exit** or **.quit**.

To create your main **Product** table, type the CREATE TABLE statement at the SQLite command prompt as follows:

```
CREATE TABLE "main"."Product"
("ID" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL ,"Name" TEXT, "ManufacturerID" INTEGER, "Details" TEXT,"Price" DOUBLE, "QuantityOnHand" INTEGER,"CountryOfOriginID" INTEGER, "Image" TEXT );
```

- Now that you have created the **Product** table, let’s move on to creating the **Manufacturer** and **Country** tables.
- Type the following SQL commands at the command prompt:

```
CREATE TABLE "main"."Manufacturer"
("ManufacturerID" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL ,"Name" TEXT NOT NULL );
CREATE TABLE "main"."Country"("CountryID" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL ,"Country" TEXT NOT NULL );
```

The foreign key constraints allow the database to validate that the values used as a foreign key in a child table map to the values stored in the parent table. In our model, the `ManufacturerID` field in the **Product** table is a foreign key that references values in the **Manufacturer** table. In the Manufacturer table, the `ManufacturerID` field is its primary key. If you wanted the database to enforce the rule that a manufacturer must exist in the **Manufacturer** table to use a `ManufacturerID` in the Product table, you would define the `ManufacturerID` in the **Product** table as a foreign key that references the `ManufacturerID` in the Manufacturer table.

You have just successfully created your database. You should have a database file that contains three tables: **Product**, **Manufacturer**, and **Country Of Origin**. Now you can put some data into the tables.

<H2> Populating the Database

# iOS Database application, Chapter 2 and Page No. 29#

Having a database is great, but the data is what really counts. You can populate your database one item at a time from the command line by using `INSERT  SQL` statements.

<H3> Creating Records with the `INSERT` Command

# iOS Database application, Chapter 2 and Page No. 29#

Figure 5 shows the syntax for the `INSERT` statement.

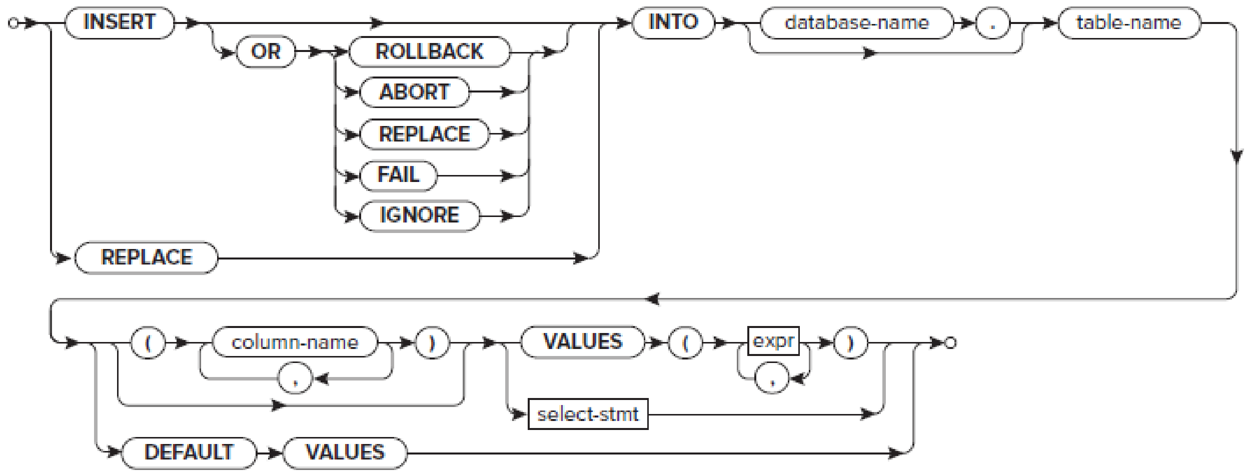


Figure 5: The `INSERT` Statement Syntax

- The open circles at the beginning and the end are **terminators**. They show where the SQL statement starts and ends. The arrow that comes out of the terminator indicates the main branch of the statement.

- Keywords are in all caps. Keywords on the main branch are required. So, for the INSERT statement, INSERT, INTO, and VALUES are required for an INSERT statement to be valid SQL.

- Anything that is not on the main branch is optional. Choices for optional keywords are left aligned. For example, the OR after INSERT is optional. If you do use OR, you must pick one and only one of the options ROLLBACK, ABORT, REPLACE, FAIL, or IGNORE.

- Text that is not in all caps is the data that the user provides. Therefore, the INSERT SQL in **Figure 5** indicates that the user needs to specify the database name and table name into which the data will be inserted. Additionally, the user must specify the columns into which the data will be inserted, and finally the values to be inserted.

You can insert a row into the **Product** table by using the following INSERT statement:

```
INSERT INTO "main"."Product"
("Name","ManufacturerID","Details","Price","QuantityOnHand",
"CountryOfOriginID","Image")
VALUES ('Widget A','1','Details of Widget A','1.29','5','1', 'Canvas_1');
```

<H3>Reading Your Rows with the SELECT Command

# iOS Database application, Chapter 2 and Page No. 30#

To verify that your data was successfully imported, you can display it using the SQL SELECT statement. The syntax for the SELECT statement appears in **Figure 6**.

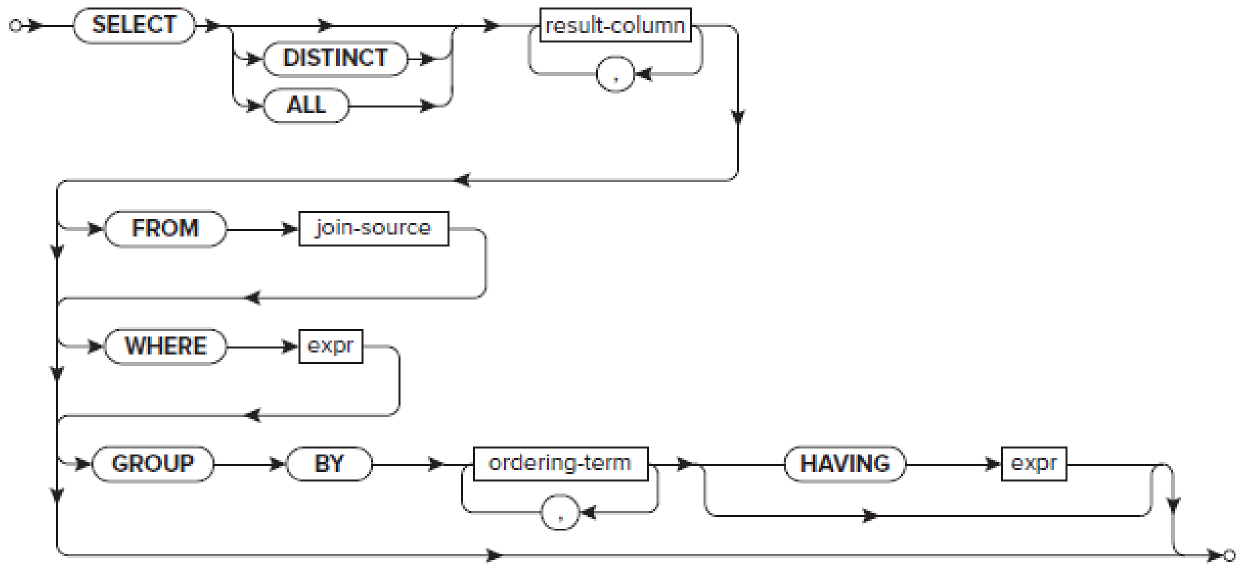


Figure 6: The Select Statement Syntax

To see all the rows in your **Product** table, type the following:

```
select * from Product
```

The output should look as follows:

```
1 Widget  A  1  Details of Widget A       1.29  5     1     Canvas_1
2 Widget  B  1  Details of Widget B       4.29  15    2     Canvas_2
3 Widget  X  1  Details of Widget X       0.29  25    3     Canvas_3
4 Widget  Y  1  Details of Widget Y       1.79  5     3     Canvas_4
5 Widget  Z  1  Details of Widget Z       6.26  15    4     Canvas_5
6 Widget  R  1  Details of Widget R       2.29  45    1     Canvas_6
7 Widget  S  1  Details of Widget S       3.29  55    1     Canvas_7
8 Widget  T  1  Details of Widget T       4.29  15    2     Canvas_8
9 Widget  L  1  Details of Widget L       5.29  50    3     Canvas_9
10 Widget N  1  Details of Widget N       6.29  50    3     Canvas_10
11 Widget E  1  Details of Widget E       17.29 25    4     Canvas_11
12 Part Alpha 2 Details of Part Alpha     1.49  25    1     Canvas_12
13 Part Beta  2 Details of Part Beta      1.89  35    1     Canvas_13
14 Part Gamma 2 Details of Part Gamma     3.46  45    2     Canvas_14
15 Device N  3  Details of Device N       9.29  15    3     Canvas_15
16 Device O  3  Details of Device O       21.29 15    3     Canvas_16
17 Device P  3  Details of Device P       51.29 15    4     Canvas_17
18 Tool   A  4  Details of Tool A         14.99 5     1     Canvas_18
19 Tool   B  4  Details of Tool B         44.79 5     1     Canvas_19
20 Tool   C  4  Details of Tool C         6.59  5     1     Canvas_20
21 Tool   D  4    Details of Tool D       8.29  5     1     Canvas_21
```

This is identical to the input data file, so you are ready to proceed.

<H2> Tools to Visualize the SQLite Database

# iOS Database application, Chapter 2 and Page No. 33#

As powerful as the command-line interface to SQLite is, sometimes it is easier to use a graphical user interface (GUI) to examine the database. Many applications provide this functionality.

For developing simple iOS applications that do not require intense database development, you can use the **SQLite Manager plug-in** for the Firefox web browser. This free plug-in, available at the Google code website (*http://code.google.com/p/sqlite-manager/*), provides the following features:

- Dialog interface for creation and deletion of tables, indexes, views, and triggers
- Ability to modify tables by adding and dropping columns
- Ability to create or open any existing SQLite databases
- Ability to execute arbitrary SQL or simply view all the data in your tables
- Visual interface for database settings, eliminating the need to write pragma statements to view and change the SQLite library settings
- Ability to export tables/views, such as CSV, SQL, or XML files
- A tree view that shows all tables, indexes, views, and triggers
- Ability to import tables from CSV, SQL, or XML files
- Interface to browse data from any table/view
- Ability to edit and delete records while browsing data

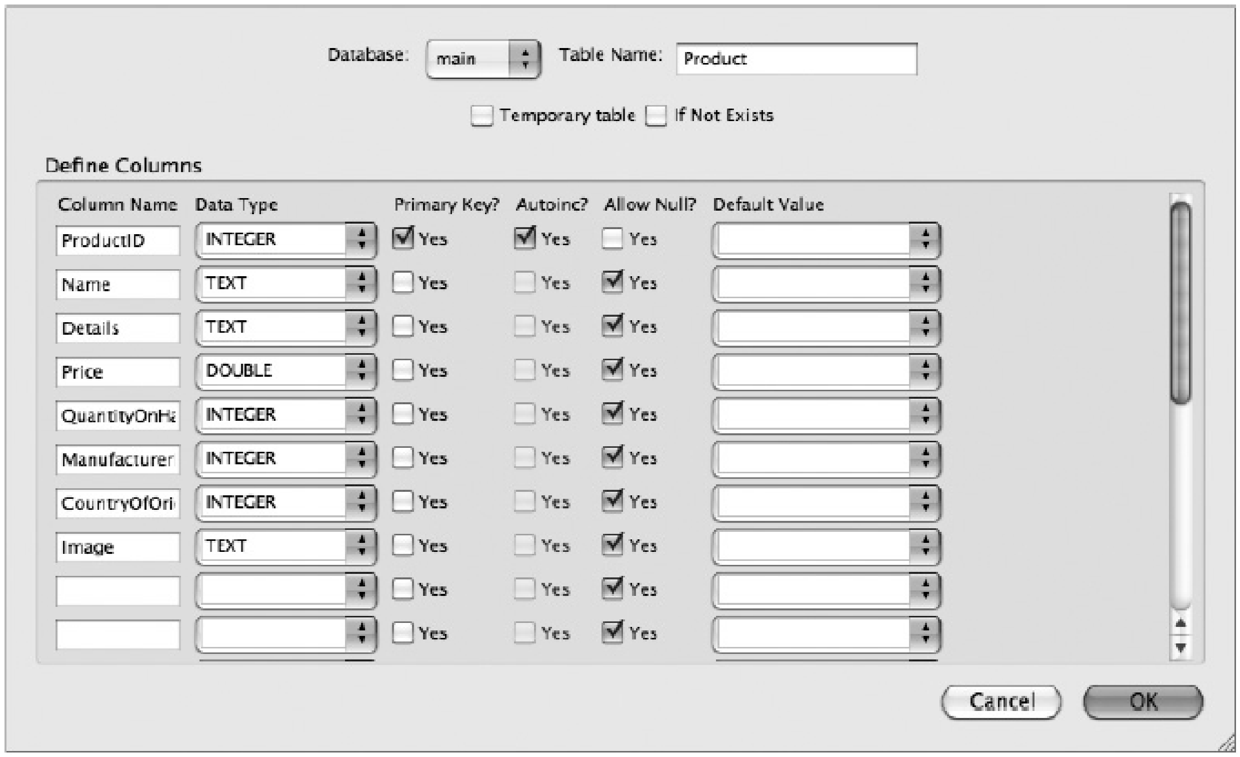
Technical Stuff

A **pragma statement** is the SQL extension specific to SQLite that is used to modify the operation of the SQLite library or query the SQLite library for internal (non-table) data.

<H3> Creating Tables with SQLite Manager Plug-in

The SQLite Manager plug-in is easy to install and use. You can perform the following steps to use the plug-in to create new tables:

1. Click the **Create Table** icon.
2. A dialog box opens, as shown in **Figure 7**. It contains all the data that you need to create a new table. In this dialog box, the fields are populated from the **Entity-Relationship Diagrams** (ERD).



Database: main Table Name: Product

☐ Temporary table ☐ If Not Exists

Define Columns

Column Name	Data Type	Primary Key?	Autoinc?	Allow Null?	Default Value
ProductID	INTEGER	✓ Yes	✓ Yes	☐ Yes	
Name	TEXT	☐ Yes	☐ Yes	✓ Yes	
Details	TEXT	☐ Yes	☐ Yes	✓ Yes	
Price	DOUBLE	☐ Yes	☐ Yes	✓ Yes	
QuantityOnHa	INTEGER	☐ Yes	☐ Yes	✓ Yes	
Manufacturer	INTEGER	☐ Yes	☐ Yes	✓ Yes	
CountryOfOri	INTEGER	☐ Yes	☐ Yes	✓ Yes	
Image	TEXT	☐ Yes	☐ Yes	✓ Yes	
		☐ Yes	☐ Yes	✓ Yes	
		☐ Yes	☐ Yes	✓ Yes	

Figure 7: Dialog Box Containing Data to Create a Table

3. Expand **Tables** in the left pane of the interface to see a list of all the tables in the database.
4. Select a table, like **Product** in **Figure 8**. This will reveal the details of the table. You can see the SQL that you originally used to create the table, the number of fields in the table, the number of records, and detailed information on all the columns in the table. You can also add, alter, and drop columns from this view.

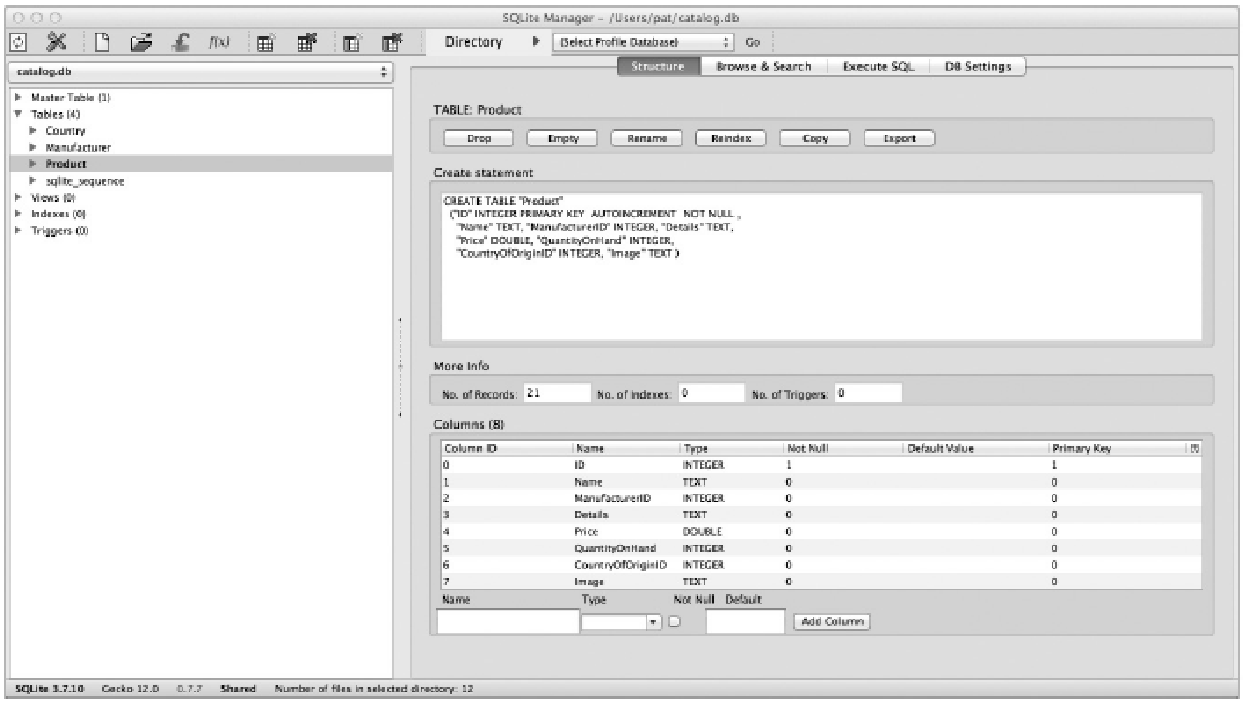


TABLE: Product

Create statement

```
CREATE TABLE "Product"
 ("ID" INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL ,
  "Name" TEXT, "ManufacturerID" INTEGER, "Details" TEXT,
  "Price" DOUBLE, "QuantityOnHand" INTEGER,
  "CountryOfOriginID" INTEGER, "Image" TEXT )
```

More Info

No. of Records: 21   No. of Indexes: 0   No. of Triggers: 0

Columns (8)

Column ID	Name	Type	Not Null	Default Value	Primary Key
0	ID	INTEGER	1		1
1	Name	TEXT	0		0
2	ManufacturerID	INTEGER	0		0
3	Details	TEXT	0		0
4	Price	DOUBLE	0		0
5	QuantityOnHand	INTEGER	0		0
6	CountryOfOriginID	INTEGER	0		0
7	Image	TEXT	0		0

Name   Type   Not Null   Default

Figure 8: View Table Definition Data with SQLite Manager

5. You can select the **Browse & Search** tab at the top of the right pane to view and edit the data in the selected table, as shown in **Figure 9**.
6. Select the **Execute SQL** tab to execute arbitrary SQL statements against the database.
7. Finally, the **DB Settings** tab enables you to view and edit various database settings that are normally only available via pragma statements at the command prompt.

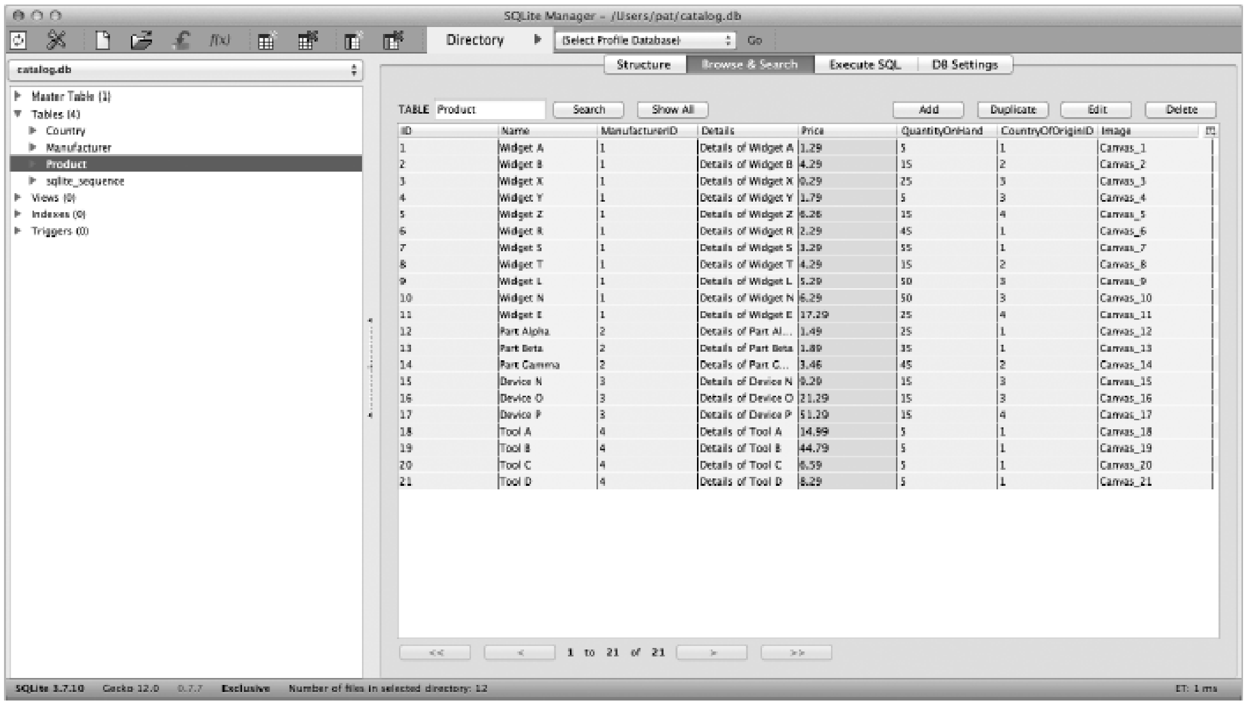


TABLE Product

ID	Name	ManufacturerID	Details	Price	QuantityOnHand	CountryOfOriginID	Image
1	Widget A	1	Details of Widget A	1.29	5	1	Canvas_1
2	Widget B	1	Details of Widget B	4.29	15	2	Canvas_2
3	Widget X	1	Details of Widget X	0.29	25	3	Canvas_3
4	Widget Y	1	Details of Widget Y	1.79	5	3	Canvas_4
5	Widget Z	1	Details of Widget Z	6.28	15	4	Canvas_5
6	Widget R	1	Details of Widget R	2.29	45	1	Canvas_6
7	Widget S	1	Details of Widget S	3.29	55	1	Canvas_7
8	Widget T	1	Details of Widget T	4.29	15	2	Canvas_8
9	Widget L	1	Details of Widget L	5.29	50	3	Canvas_9
10	Widget N	1	Details of Widget N	6.29	50	3	Canvas_10
11	Widget E	1	Details of Widget E	17.29	25	4	Canvas_11
12	Part Alpha	2	Details of Part Al...	1.49	25	1	Canvas_12
13	Part Beta	2	Details of Part Beta	1.89	35	1	Canvas_13
14	Part Gamma	2	Details of Part G...	3.46	45	2	Canvas_14
15	Device N	3	Details of Device N	9.29	15	3	Canvas_15
16	Device O	3	Details of Device O	21.29	15	3	Canvas_16
17	Device P	3	Details of Device P	51.29	15	4	Canvas_17
18	Tool A	4	Details of Tool A	14.99	5	1	Canvas_18
19	Tool B	4	Details of Tool B	44.79	5	1	Canvas_19
20	Tool C	4	Details of Tool C	6.59	5	1	Canvas_20
21	Tool D	4	Details of Tool D	8.29	5	1	Canvas_21

Figure 9: Browse Table Data with SQLite Manager

Big Picture

Before creating a sample SQLite database, you need to understand what the application will do. This can help you understand the advantages of creating a normalized database. You can use various methods to create, modify, and populate your SQLite database.

.

<H1> Connecting to Your Database

# iOS Database application, Chapter 2 and Page No. 36 #

Now that you have a catalog database, you can write the iOS application that you will use to view the catalog. To do this, you will need to create an application with a table view to display the catalog. Clicking a cell in the table view should navigate to a detail page that shows detailed information about the selected catalog entry. To build this interface, you need to be able to connect to your database and run SQL statements against it. You will also use a **Navigation Controller** to implement a Master-Detail interface.

It is a good idea to **mock up your application interface** before you get started.

It helps to get buy-in from customers that the interface you have designed meets their needs. It is far easier to move around interface items or redesign the look and feel of a mockup than it is to rework your actual application code. You want to find any problems with the design as early as possible to avoid costly and time-consuming changes to the software. A picture can go a long way in explaining to customers what the application will look like.

Figure 10 shows a mocked-up interface in **OmniGraffle**. The interface might not look pretty, but it will get the job done. You will spruce it up a bit in the next session. However, for now, it will demonstrate how to get data out of your SQLite database.

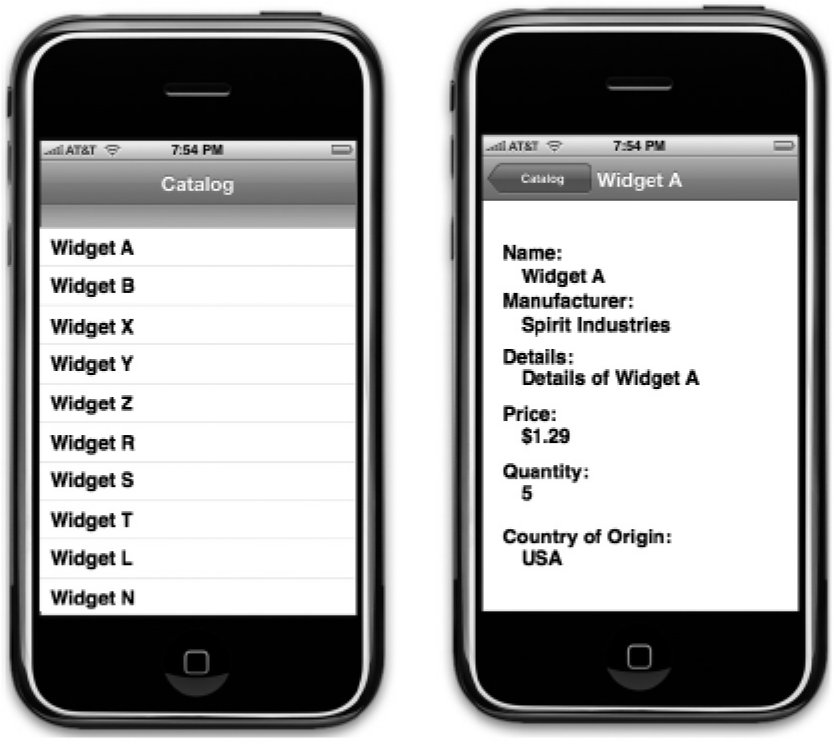


Figure 10: Application Interface Mockup

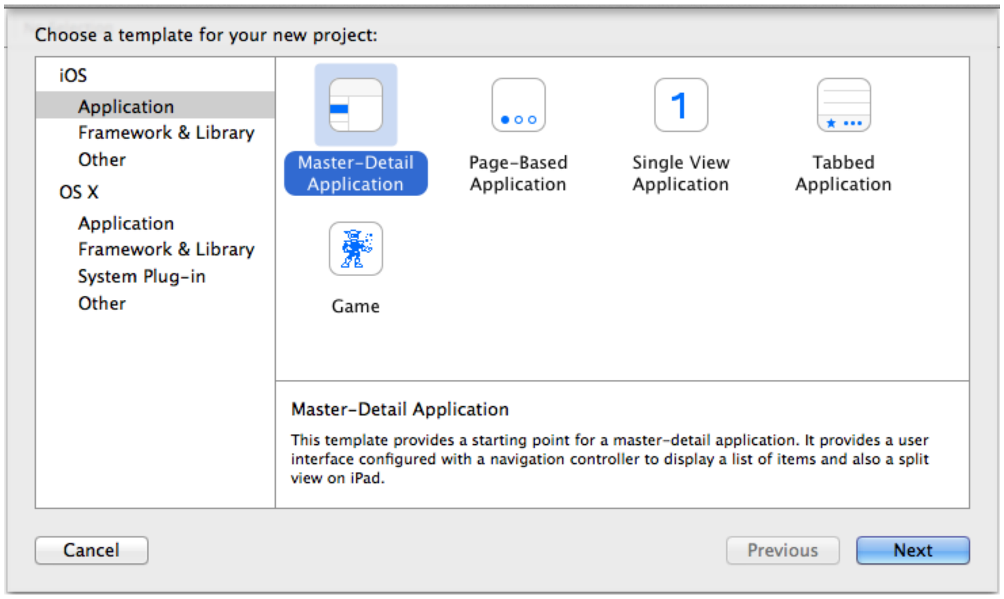
<H2> Starting the Project

# iOS Database application, Chapter 2 and Page No. 36 #

For this project, you are going to implement a Master-Detail interface. As seen in the mockup in **Figure 10**, the main screen shows the entire product catalog, and tapping an item should display a screen with the details for that item. The `UINavigationController` is perfect for building this kind of interface.

To get started, you can use the following steps:

1. Open Xcode and create a new project using the **Master-Detail Application** template (see **Figure 11**).

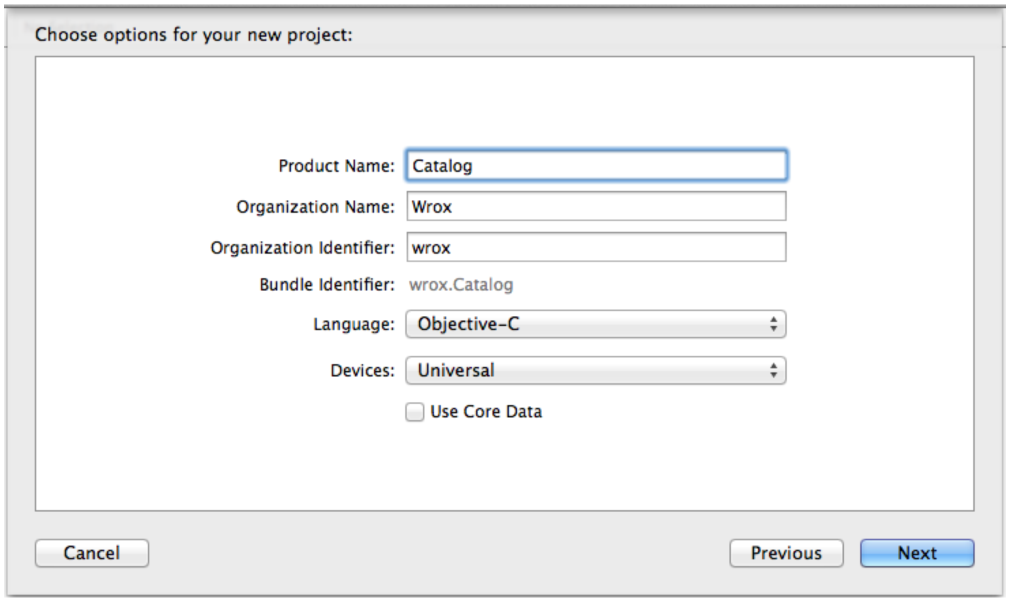


Master-Detail Application

This template provides a starting point for a master-detail application. It provides a user interface configured with a navigation controller to display a list of items and also a split view on iPad.

Figure 11: Create a Master-Detail Application Project

2. When you create your project, set the **Devices** option to **iPhone** and check the **Use Automatic Reference Counting** check box in the **Choose options for your new project** dialog in iOS 7 with Xcode 5. This option is absent in case of iOS 8 with Xcode 6.1.1, as shown in **Figure 12**. Here, you have only option available, which is **Use Core Data** check box.


Choose options for your new project:

Product Name: Catalog

Organization Name: Wrox

Organization Identifier: wrox

Bundle Identifier: wrox.Catalog

Language: Objective-C

Devices: Universal

☐ Use Core Data

Figure 12: Choose Options for the New Project

The Master-Detail Application template creates a project that contains two Interface Builder `xib` files:

- `MasterViewController.xib File`: This file contains the table view that will hold your master data.
- `DetailViewController.xib File`: This is a placeholder that you will fill with your detail data.

When you choose the **Swift** language, you will get two Interface Builder files: `MasterViewController.`**`Swift`** and `DetailViewController.`**`Swift`**.

<H3>The `UINavigationController`

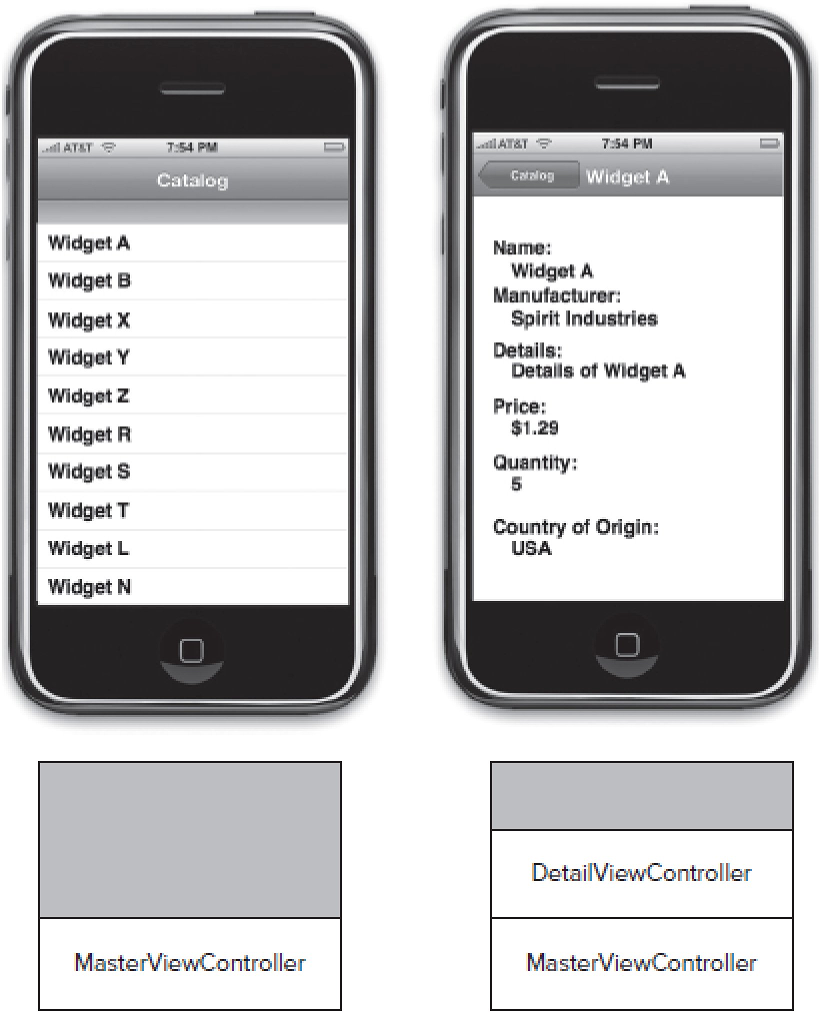
You use the Navigation Controller to display and manage a hierarchy of View Controllers. Anytime that you need to display hierarchical data, consider using the `UINavigationController`.

The Navigation Controller manages the state of display using a "navigation stack." You push View Controllers that you want to display onto the navigation stack when you are ready to display them. Pressing the **Back** button causes the current View Controller to be popped off the navigation stack. At the bottom of the stack is the Root View Controller — in this example, the `MasterViewController`.

You will implement navigation in the catalog application using the `UINavigationController`.

Figure 13 shows the application mockup along with the navigation stack. The left side shows the product catalog displayed in the `UITableView`, which is included in the `MasterViewController`.

Selecting a row in the table view causes the `DetailViewController` to be pushed onto the navigation stack. You see this in the image on the right. The status of the navigation stack appears at the bottom of **Figure 13**.



Widget A
Widget B
Widget X
Widget Y
Widget Z
Widget R
Widget S
Widget T
Widget L
Widget N

Name:
Widget A
Manufacturer:
Spirit Industries
Details:
Details of Widget A
Price:
$1.29
Quantity:
5
Country of Origin:
USA

Figure 13: Application Screens and Navigation Stack State

Tapping the **Catalog** button in the navigation bar at the top of the detail screen causes the `DetailViewController` to be popped from the navigation stack, thus displaying the `MasterViewController` again. The most important thing to remember is that the `UINavigationController` always displays the View Controller, which is at the top of the navigation stack.

<H3>The `UITableViewController`

If you look at the code header for the MasterViewController, you will notice that the MasterViewController is not a subclass of the UIViewController.

Instead, it is a subclass of the UITableViewController. When implementing a View Controller that will control a table view, you can subclass the UITableViewController class instead of UIViewController. UITableViewController is a great shortcut to use. When you use a UITableViewController, you do not have to declare that you will be implementing the UITableViewDataSource and UITableViewDelegate protocols.

The UITableViewController already has a table view associated with it. You can get a reference to the table view by using the tableView property.

Additional Knowhow

The UITableViewController class immediately creates a table view and assigns itself as a delegate and a data source when created. The table view data source creates the number of sections, rows, and cells of the table view. You can populate the table cells and manage application using the database.

Quick Tip

When implementing a View Controller that will control a table view, you can subclass the UITableViewController class instead of UIViewController. UITableViewController is a great shortcut to use. When you use a UITableViewController, you do not have to declare that you will be implementing the UITableViewDataSource and UITableViewDelegate protocols.

<H2> The Model Class

By simply creating a project based on the **Master-Detail Application** template, you get a lot of free functionality. In fact, if you build and run the application, you should get something that looks like **Figure 14**.

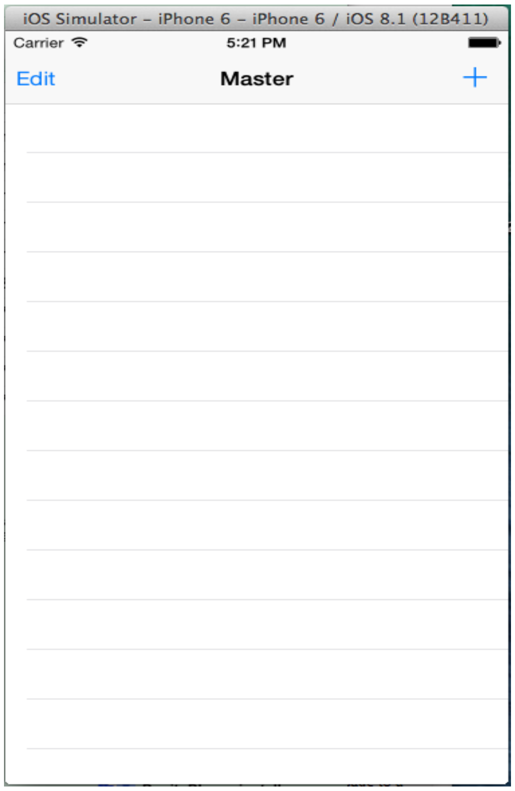


Figure 14: Master-Detail Application

In **Swift** language, you will get the same output for the Master-Detail application. You have added no code, yet you already have a navigation bar (the blue-gray area at the top) and a table view (the lines). You also can add new rows with the plus button and edit the set of rows using the **Edit** button. Now you need to fill the table view with data.

In keeping with the preferred application architecture on iOS, you will design this application by following the **Model-View-Controller** (MVC) design pattern.

You already have your views (in the `xib` files) and controllers (**Master View Controller** and **Detail View Controller**); you just need a model. You need to design a model class that represents your data. The model class should also have a method that returns the number of rows in the database and provides access to the data for a specific row.

For this application, you will base your model on the `Product` class. The `Product` class will mirror the fields in the Product table in the database. Your model will be a collection of Product objects.

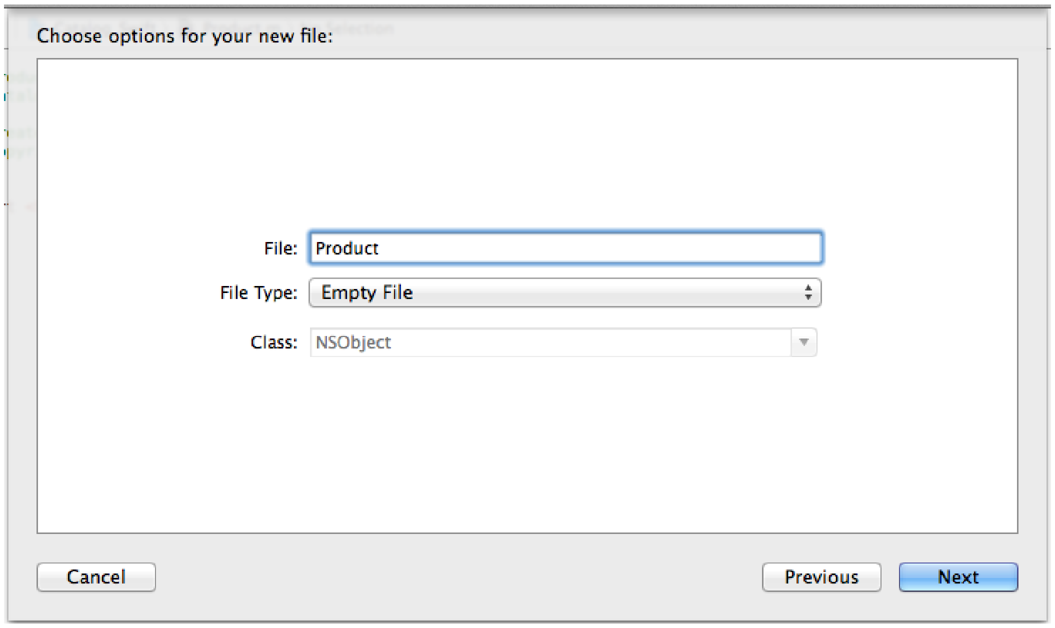
To implement this model, create a new Objective-C class called `Product`. In the header, you will add a property for each database field. The following is the code for the header:

```
#import <Foundation/Foundation.h>
```

```
@interface Product : NSObject {
      int ID;
      NSString* name;
      NSString* manufacturer;
      NSString* details;
      float price;

       int quantity;
       NSString* countryOfOrigin;
       NSString* image;
}

@property (nonatomic) int ID;
@property (strong, nonatomic) NSString *name;
@property (strong, nonatomic) NSString *manufacturer;
@property (strong, nonatomic) NSString *details;
@property (nonatomic) float price;
@property (nonatomic) int quantity;
@property (strong, nonatomic) NSString *countryOfOrigin;
@property (strong, nonatomic) NSString *image;

@end
```

You can see that you simply declare a member variable for each database field and then create a property to access each field.

If you are working with **Swift**, then you select **Objective-C file -> Next**. In the **New File Type**, select **Empty File** and click **Next**, as shown in **Figure 15**.


Choose options for your new file:

File: Product

File Type: Empty File

Class: NSObject

Figure 15: New Objective-C File Product

The above code will be given in **Swift** as follows:

```
      class Product : NSObject {
          var name : String = String()
          var manufacturer : String = String()
          var details : String = String()
          let price : Float = 0.0
          let quantity : Int
          var countryoforigin : String = String()
          var image : String = String()
      }
```

In **Swift**, you add a bridging header to access the `NSObject` file and define the `var` member with the `var` keyword.

The implementation for this class is even easier:

```
#import "Product.h"

@implementation Product
@synthesize ID;
@synthesize name;
@synthesize manufacturer;
@synthesize details;
@synthesize price;
@synthesize quantity;
```

```
@synthesize countryOfOrigin;
@synthesize image;

@end
```

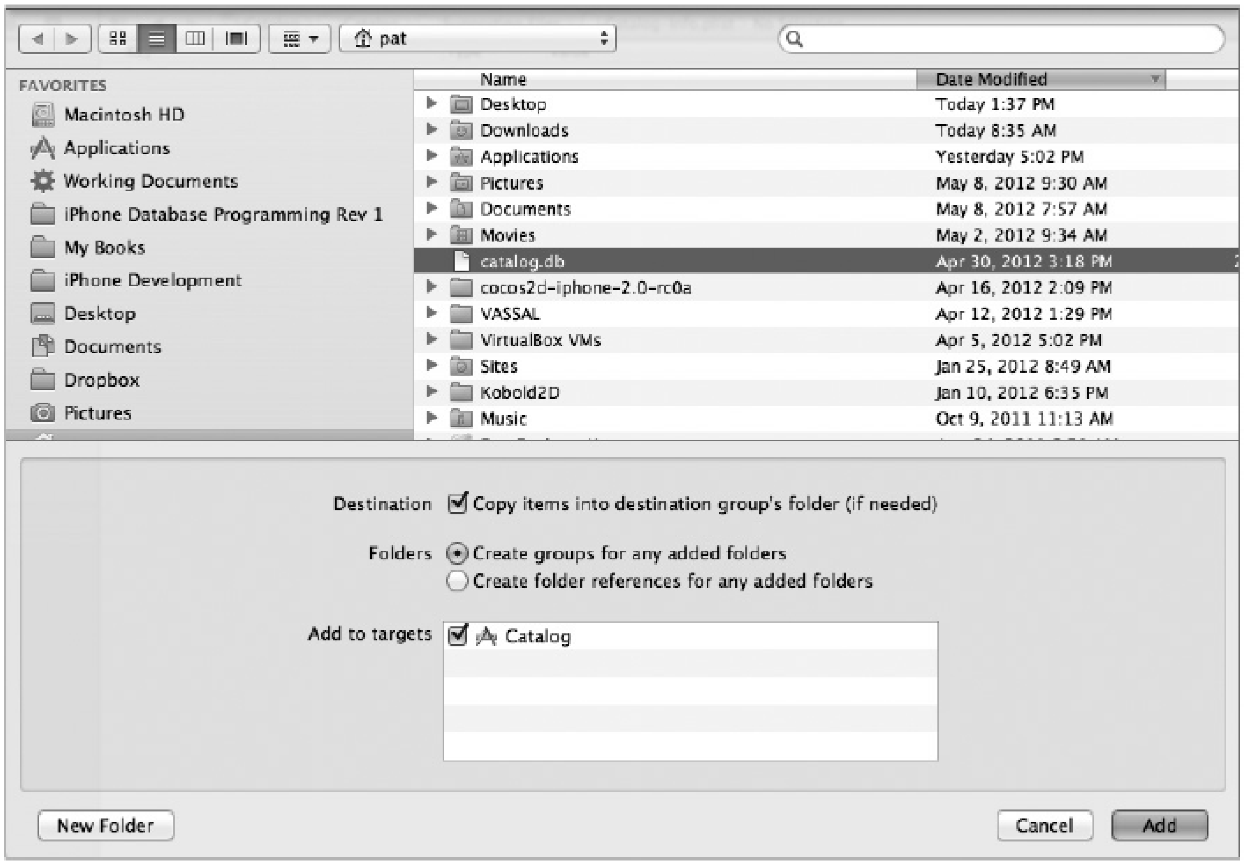
Here, you just synthesize all the properties declared in the header. At this point, it is a good idea to build and verify that there are no errors in your application.

<H3>The `DBAccess` Class

Now that you have your model class completed, you need to write the code to get the data out of the database and into your model. In general, it is a good idea to abstract away access to the database. This means writing a general class to perform common database functions. This gives you the flexibility if you want to move to a different database engine later. To do this, you will create a database access class that talks to the database. This class will have methods to initialize the database, close the database, and most importantly, build and return an array of Product objects.

Before you get started on coding the database access class, you need to add your SQLite database to the Xcode project. The steps to do this are as follows:

1. Add the SQLite database to the project's **Supporting Files** folder. To do this, right-click on the **Supporting Files** folder and select **Add Files to "Catalog"** from the right-click menu.
2. In the dialog box that opens, navigate to your home directory or wherever you stored the catalog database, select it, and click **Add**. Make sure that you have selected the **Copy items into destination group's folder (if needed)** check box, as shown in **Figure 16**.



Name	Date Modified
Desktop	Today 1:37 PM
Downloads	Today 8:35 AM
Applications	Yesterday 5:02 PM
Pictures	May 8, 2012 9:30 AM
Documents	May 8, 2012 7:57 AM
Movies	May 2, 2012 9:34 AM
catalog.db	Apr 30, 2012 3:18 PM
cocos2d-iphone-2.0-rc0a	Apr 16, 2012 2:09 PM
VASSAL	Apr 12, 2012 1:29 PM
VirtualBox VMs	Apr 5, 2012 5:02 PM
Sites	Jan 25, 2012 8:49 AM
Kobold2D	Jan 10, 2012 6:35 PM
Music	Oct 9, 2011 11:13 AM

Destination [✓ Copy items into destination group's folder (if needed)]

Folders [✓ Create groups for any added folders] Create folder references for any added folders

Add to targets [✓ Catalog]

Figure 16: Adding an Existing File to a Project

To create the database access class, create a new Objective-C class called `DBAccess`. In the header file, `DBAccess.h`, you need to add an import statement for `sqlite3.h` because you intend to use functions from the sqlite3 library in the data access class.

You also need to add the following three method signatures for the methods that you plan to implement:

- `getAllProducts`: This method returns an array of all the Product objects in the catalog. Because you will be referencing the Product object in this class, you need to add an import statement for `Product.h`.
- `closeDatabase:` This method closes the database.
- `initializeDatabase`: This method initializes the database.

The `DBAccess.h` header file should look like this:

```
#import <Foundation/Foundation.h>
// This includes the header for the SQLite library.
#import <sqlite3.h>
```

```
#import "Product.h"
@interface DBAccess : NSObject {
}
- (NSMutableArray*) getAllProducts;
- (void) closeDatabase;
- (void)initializeDatabase;
@end
```

In **Swift**, you can make class to access database of `NSObject` (such as FMDB and Core Data) as follows:

```
import “Product.m”
class DBAccess : NSObject
```

To add `NSMutableArray` in **Swift**, you can assign `getAllProducts` as follows:

```
      let getAllProduct = NSMutableArray()
// In Swift make function as :
   -  func closeDatabase()
   -  func initializeDatabase
```

In the implementation of the `DBAccess` class, add a class-level variable to hold a reference to the database, as follows:

```
// Reference to the SQLite database.
sqlite3* database;
```

You will populate this variable in the `initializeDatabase` function. Then, every other function in your class will have access to the database.

Now, you will create the `init` function that callers use to initialize instances of this class. In `init`, you will make an internal call to initialize the database. The `init` function should look like this:

```
-(id) init
{
// Call super init to invoke superclass initiation code
if ((self = [super init]))
{
// Set the reference to the database
[self initializeDatabase];
}
return self;
}
```

Your `initializeDatabase` function will do just that. It will go out, get the path to the database, and attempt to open it. Here is the code for `initializeDatabase`:

```
// Open the database connection
- (void)initializeDatabase {
// Get the database from the application bundle.
NSString *path = [[NSBundle mainBundle]
pathForResource:@"catalog"
ofType:@"db"];
}
```

In **Swift**, you get the database from the application bundle using the following command:

```
let path = NSBundle.mainBundle().pathForResource("catalog", ofType: "db")
```

```
// Open the database.
if (sqlite3_open([path UTF8String], &database) == SQLITE_OK)
{
NSLog(@"Opening Database");
}
else
{
// Call close to properly clean up
sqlite3_close(database);
NSAssert1(0, @"Failed to open database: '%s'.",
sqlite3_errmsg(database));
}
```

In **Swift**, you can print values using the following two primary functions of the **Swift** standard library:

- `print()`: This function prints the value as output.
- `println(_:)`: This function overloads to accept a value or no value for printing. It prints a new line character.

In the implementation of the `DBAccess` class, add a class-level variable to hold a reference to the database as follows:

```
    // Reference to the SQLite database.
    sqlite3* database;
```

You will populate this variable in the `initializeDatabase` function. Then, every other function in your class will have access to the database.

Now, you will create the `init` function that callers use to initialize instances of this class. In `init`, you will make an internal call to initialize the database. The `init` function should look like this:

```
    -(id) init
    {
          // Call super init to invoke superclass initiation code
          if ((self = [super init]))
          {
          // Set the reference to the database
          [self initializeDatabase];
          }
          return self;
    }
```

Your initializeDatabase function will do just that. It will go out, get the path to the database, and attempt to open it. Here is the code for initializeDatabase:

```
      // Open the database connection
      - (void)initializeDatabase {
      // Get the database from the application bundle.
      NSString *path = [[NSBundle mainBundle]
      pathForResource:@"catalog"
      ofType:@"db"];
      }
      // Open the database.
      if (sqlite3_open([path UTF8String], &database) == SQLITE_OK)
      {
      NSLog(@"Opening Database");
      }
      else
      {
      // Call close to properly clean up
      sqlite3_close(database);
      NSAssert1(0, @"Failed to open database: '%s'.",
      sqlite3_errmsg(database));
      }
```

You can see that you need the path to the database file. Because you put the catalog.db file in the **Supporting Files** folder, the database will be deployed to the device in the main bundle of the application. There is a handy class, NSBundle, for getting information about an application bundle. The mainBundle method returns a reference to the main application bundle, and the pathForResource:ofType: method returns the path to the specified file in the bundle. Because you specified the resource as catalog and the type as db, the method returns the path to catalog.db.

Next, you use the C function sqlite3_open to open a connection to the database. You pass in the path to the database and the address to a variable of type sqlite3*. The second parameter is populated with the handle to the database. The sqlite3_open function returns an int. It returns the constant SQLITE_OK if everything goes well. If not, it returns an error code. The most common errors that you will encounter are as follows:

- SQLITE_ERROR: This error indicates that the database cannot be found.
- SQLITE_CANTOPEN: This error indicates that there is some other reason that the database file cannot be opened.

You can find the full list of error codes in the sqlite3.h include file.

You make sure that you got back SQLITE_OK and then log that you are opening the database. If you get an error, you close the database and then log the error message.

Next, you will add a method to cleanly close the connection to the database. This is as simple as calling sqlite3_close and passing in a handle to the database as follows:

```
-(void) closeDatabase
{
      // Close the database.
      if (sqlite3_close(database) != SQLITE_OK) {
      NSAssert1(0, @"Error: failed to close database: '%s'.",
      sqlite3_errmsg(database));
      }
}
```

The sqlite3_close function returns a value just like sqlite3_open. If the call to sqlite3_close fails, you use the sqlite3_errmsg function to get a textual error message and print it to the console.

At this point, you should try to build the application. The build fails because the SQLite functions are not found. Although you included the proper header files, the compiler does not know where to find the binaries for the library. You need to add the libsqlite framework to your Xcode project.

1. Click on the **Catalog** project icon at the top of the Project Navigator.
2. Select the **Catalog Target** option on the left side of the Editor pane.
3. At the top of the Editor pane, select the **Build Phases** tab.
4. Expand the **Link Binary with Libraries** entry in the Editor pane and press the plus sign below the list of frameworks that are already included with your project.
5. Select libsqlite3.0.dylib from the list of frameworks.

Now you should be able to build your project successfully. You will still receive a warning that tells you that the getAllProducts method is not implemented, but you can fix that by implementing the function.

Now comes the heart of the database access class: the getAllProducts method. You will implement the getAllProducts method to return an array of Product objects that represent the records in the product catalog database. The method allocates an `NSMutableArray` to hold the list of products, construct your SQL statement, execute the statement, and loop through the results, constructing a Product object for each row returned from the query.

You will start the method by declaring and initializing the array that will hold the products. You will use an `NSMutableArray` because you want to be able to add Product objects to the array one by one as you retrieve rows from the database. The regular `NSArray` class is immutable, so you cannot add items to it on-the-fly.

Here is the beginning of your method:

```
- (NSMutableArray*) getAllProducts
{
      // The array of products that we will create
      NSMutableArray *products = [[NSMutableArray alloc] init];
```

The next step in implementing `getAllProducts` is to declare a `char*` variable and populate it with your SQL statement:

```
// The SQL statement that we plan on executing against the database
const char *sql = "SELECT product.ID,product.Name, \
Manufacturer.name,product.details,product.price,\
product.quantityonhand, country.country, \
product.image FROM Product,Manufacturer, \
Country where manufacturer.manufacturerid=product.manufacturerid \
and product.countryoforiginid=country.countryid";
```

The following is the SQL statement in a slightly more readable form:

```
SELECT product.ID,
      product.Name,
      Manufacturer.name,
      product.details,
      product.price,
      product.quantityonhand,
      country.country,
      product.image
FROM Product,Manufacturer, Country
WHERE manufacturer.manufacturerid=product.manufacturerid
      AND product.countryoforiginid=country.countryid
```

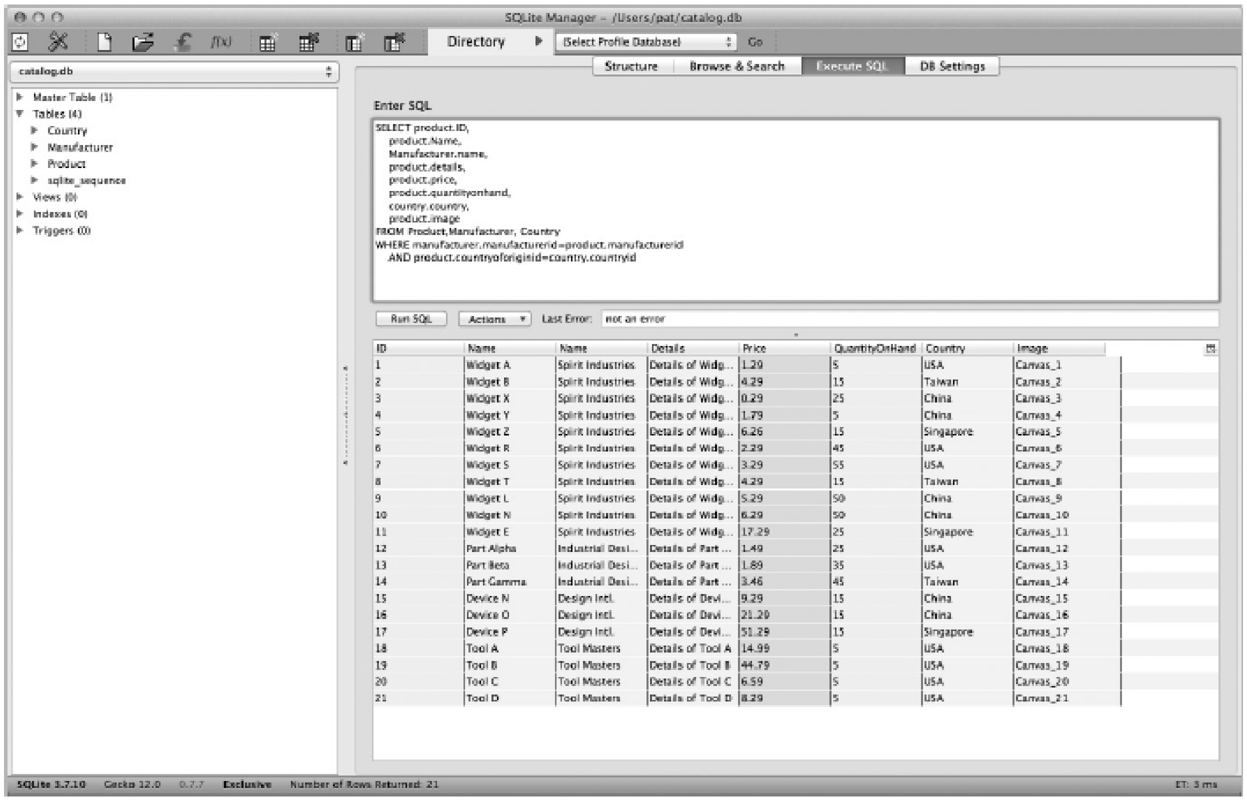
The above SQL statement gets data out of the three tables noted in the FROM clause: **Product**, **Manufacturer**, and **Country**.

You can see which fields are selected from each table in the SELECT portion of the query. The form for specifying fields to select is `table.field`. Therefore, you are first selecting the **ID** field and then the **Name** field from the Product table.

Finally, you set up the joins in the WHERE clause. You only want data from the Manufacturer table where the `manufacturerID` is the same as the `manufacturerID` in the **Product** table. Likewise, you only want data from the **Country** table where the `countryID` is the same as the `countryoforiginID` in the **Product** table. This allows you to display the actual manufacturer name and country name in the query and the application, instead of just displaying a meaningless ID number.

It is much easier to develop your query when you can run it and instantly see the results, especially as you get into queries that are more complex.

Figure 17 shows the query running in SQLite Manager. You get the desired results when you execute this query.



```
SELECT product.ID,
    product.Name,
    Manufacturer.name,
    product.details,
    product.price,
    product.quantityonhand,
    country.country,
    product.image
FROM Product,Manufacturer, Country
WHERE manufacturer.manufacturerid=product.manufacturerid
    AND product.countryoforiginid=country.countryid
```

ID	Name	Name	Details	Price	QuantityOnHand	Country	Image
1	Widget A	Spirit Industries	Details of Widg...	1.29	5	USA	Canvas_1
2	Widget B	Spirit Industries	Details of Widg...	4.29	15	Taiwan	Canvas_2
3	Widget X	Spirit Industries	Details of Widg...	0.29	25	China	Canvas_3
4	Widget Y	Spirit Industries	Details of Widg...	1.79	5	China	Canvas_4
5	Widget Z	Spirit Industries	Details of Widg...	6.26	15	Singapore	Canvas_5
6	Widget R	Spirit Industries	Details of Widg...	2.29	45	USA	Canvas_6
7	Widget S	Spirit Industries	Details of Widg...	3.29	55	USA	Canvas_7
8	Widget T	Spirit Industries	Details of Widg...	4.29	15	Taiwan	Canvas_8
9	Widget L	Spirit Industries	Details of Widg...	5.29	50	China	Canvas_9
10	Widget N	Spirit Industries	Details of Widg...	6.29	50	China	Canvas_10
11	Widget E	Spirit Industries	Details of Widg...	17.29	25	Singapore	Canvas_11
12	Part Alpha	Industrial Desi...	Details of Part ...	1.49	25	USA	Canvas_12
13	Part Beta	Industrial Desi...	Details of Part ...	1.89	35	USA	Canvas_13
14	Part Gamma	Industrial Desi...	Details of Part ...	3.46	45	Taiwan	Canvas_14
15	Device N	Design Intl.	Details of Devi...	9.29	15	China	Canvas_15
16	Device O	Design Intl.	Details of Devi...	21.29	15	China	Canvas_16
17	Device P	Design Intl.	Details of Devi...	51.29	15	Singapore	Canvas_17
18	Tool A	Tool Masters	Details of Tool A	14.99	5	USA	Canvas_18
19	Tool B	Tool Masters	Details of Tool B	44.79	5	USA	Canvas_19
20	Tool C	Tool Masters	Details of Tool C	6.59	5	USA	Canvas_20
21	Tool D	Tool Masters	Details of Tool D	8.29	5	USA	Canvas_21

Figure 17: SQL Dictionary

To run this SQL in your code, you need to create an SQLite statement object. This object will execute your SQL against the database. You can then prepare the SQL statement as follows:

```
// The SQLite statement object that will hold the result set
sqlite3_stmt *statement;
// Prepare the statement to compile the SQL query into byte-code
```

```
    int  sqlResult  =  sqlite3_prepare_v2(database,  sql,  -1,  &statement,
    NULL);
```

The parameters for the `sqlite3_prepare_v2` function are a connection to the database, your SQL statement, the maximum length of your SQL or –1 to read up to the first null terminator, the statement handle that will be used to iterate over the results, and a pointer to the first byte after the SQL statement or NULL, which you use here.

Like the other commands that you have run against SQLite in this session, `sqlite3_prepare_v2` returns an `int`, which will be either `SQLITE_OK` or an error code.

You should note that preparing a SQL statement does not actually execute the statement. The statement is not executed until you call the `sqlite3_step` function to begin retrieving rows. If the result is `SQLITE_OK`, you step through the results one row at a time using the `sqlite3_step` function as follows:

```
    if ( sqlResult== SQLITE_OK) {
          // Step through the results - once for each row.
          while (sqlite3_step(statement) == SQLITE_ROW) {
```

For each row, allocate a Product object as follows:

```
    // Allocate a Product object to add to products array
    Product *product = [[Product alloc] init];
```

In **Swift**, you can create a database table and then insert, update, delete, and display database items using **Swift** commands as follows:

```
// to insert into database
sharedInstance.database!.open()
let    insert    =    sharedInstance.dtabase!.executeUpdate("INSERT    INTO
Product(product.ID,  product.name)  VALUES  (?,   ?)",  withArgumentsInArray:
[Product.ID, Product.Name])
      sharedInstance.database!.close()
      return insert
}
// to update database
sharedInstance.database!.open()
let  update  =  sharedInstance.dtabase!.executeUpdate("UPDATE  Product  SET
product.ID=?  WHERE    product.name=?  ,  withArgumentInArray:  [ProductID,
Product.name])   sharedInstance.database!.close()
      return update
}
```

Now you have to **retrieve the data from the row**.

A group of functions called the “result set” interface is used to get the required field. The function that you use is based on the data type contained in the column that you are trying to retrieve.

The following is a list of the available functions:

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
sqlite3_int64 sqlite3_column_int64(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
sqlite3_value *sqlite3_column_value(sqlite3_stmt*, int iCol);
```

The first parameter to each function is the prepared statement. The second parameter is the index in the SQL statement of the field that you are retrieving. The index is 0-based. Therefore, to get the first field in your SQL statement, the product ID, which is an `int`, you would use this:

```
sqlite3_column_int(statement, 0);
```

To retrieve text or strings from the database, you use the `sqlite3_column_text` function.

You can create `char*` variables to hold the strings retrieved from the database. Then, you can use the ternary operator (?:) to either use the string if it is not null, or use an empty string if it is null. Here is the code to get all the data from the database and populate your product object:

```
// The second parameter is the column index (0 based) in
// the result set.
char *name = (char *)sqlite3_column_text(statement, 1);
char *manufacturer = (char *)sqlite3_column_text(statement, 2);
char *details = (char *)sqlite3_column_text(statement, 3);
char *countryOfOrigin = (char *)sqlite3_column_text(statement, 6);
char *image = (char *)sqlite3_column_text(statement, 7);
// Set all the attributes of the product
product.ID = sqlite3_column_int(statement, 0);
product.name = (name) ? [NSString stringWithUTF8String:name] : @"";
product.manufacturer = (manufacturer) ? [NSString
stringWithUTF8String:manufacturer] : @"";
product.details = (details) ? [NSString stringWithUTF8String:details] :
@"";
product.price = sqlite3_column_double(statement, 4);
product.quantity = sqlite3_column_int(statement, 5);
product.countryOfOrigin = (countryOfOrigin) ? [NSString
stringWithUTF8String:countryOfOrigin] : @"";
product.image = (image) ? [NSString stringWithUTF8String:image] : @"";
Finally, you add the product to the products array and move on to the
next row:
// Add the product to the products array
[products addObject:product];
}
```

When you work with **Swift**, you do not need to pick a specific size of integer to use in your code. **Swift** provides additional integer type as `Int`, which has the same size as the current platform's native word size:

- On 32 bit platform, `Int` is Int32.
- On 64 bit platform, `Int` is Int64.

The Unsigned integer type in **Swift** also has the same integer size as of platform, that is:

- On 32 bit platform, `UInt` is UInt32.
- On 64 bit platform, `UInt` is UInt64.

After you are finished looping through the result set, you call `sqlite3_finalize` to release the resources associated with the prepared statement. Then you log any errors and return your products array as follows:

```
// Finalize the statement to release its resources
sqlite3_finalize(statement);
}
else {
NSLog(@"Problem with the database:");
NSLog(@"%d",sqlResult);
}
return products;
}
```

The whole database access class should look like

```
#import "DBAccess.h"

@implementation DBAccess

// Reference to the SQLite database.
sqlite3* database;

-(id) init
```

```
{
      // Call super init to invoke superclass initiation code
      if ((self = [super init]))
      {
            // Set the reference to the database
            [self initializeDatabase];
      }
      return self;
}

// Open the database connection
- (void)initializeDatabase {

      // Get the database from the application bundle.
      NSString *path = [[NSBundle mainBundle]
                        pathForResource:@"catalog"
                        ofType:@"db"];

// Open the database.
if (sqlite3_open([path UTF8String], &database) == SQLITE_OK)
{
      NSLog(@"Opening Database");
}
else
{

      // Call close to properly clean up
      sqlite3_close(database);
      NSAssert1(0, @"Failed to open database: '%s'.",
            sqlite3_errmsg(database));
      }
}

-(void) closeDatabase
{
      // Close the database.
      if (sqlite3_close(database) != SQLITE_OK) {
            NSAssert1(0, @"Error: failed to close database: '%s'.",
                        sqlite3_errmsg(database));
      }
}
```

```
- (NSMutableArray*) getAllProducts
{
      // The array of products that we will create
      NSMutableArray *products = [[NSMutableArray alloc] init];

      // The SQL statement that we plan on executing against the database
      const char *sql = "SELECT product.ID,product.Name, \
      Manufacturer.name,product.details,product.price,\
      product.quantityonhand, country.country, \
      product.image FROM Product,Manufacturer, \
      Country where manufacturer.manufacturerid=product.manufacturerid \
      and product.countryoforiginid=country.countryid";

      // The SQLite statement object that will hold our result set
      sqlite3_stmt *statement;

      // Prepare the statement to compile the SQL query into byte-code
      int  sqlResult  =  sqlite3_prepare_v2(database,  sql,  -1,  &statement,
      NULL);
if ( sqlResult== SQLITE_OK) {
      // Step through the results - once for each row.
      while (sqlite3_step(statement) == SQLITE_ROW) {
                  // allocate a Product object to add to products array
                  Product *product = [[Product alloc] init];
                  // The second parameter is the column index (0 based) in
                  // the result set.
                  char *name = (char *)sqlite3_column_text(statement, 1);
                  char *manufacturer = (char *)sqlite3_column_text(statement,
                  2);
                  char *details = (char *)sqlite3_column_text(statement, 3);
                  char           *countryOfOrigin           =           (char
                  *)sqlite3_column_text(statement, 6);
                  char *image = (char *)sqlite3_column_text(statement, 7);

                  // Set all the attributes of the product
                  product.ID = sqlite3_column_int(statement, 0);
                  product.name       =       (name)       ?       [NSString
                  stringWithUTF8String:name] : @"";
                  product.manufacturer = (manufacturer) ? [NSString
                                    stringWithUTF8String:manufacturer] : @"";
                  product.details = (details) ? [NSString
                                    stringWithUTF8String:details] : @"";
                  product.price = sqlite3_column_double(statement, 4);
```

```
                product.quantity = sqlite3_column_int(statement, 5);
                product.countryOfOrigin = (countryOfOrigin) ? [NSString
                            stringWithUTF8String:countryOfOrigin] : @"";
                product.image       =       (image)       ?       [NSString
                stringWithUTF8String:image] : @"";

                // Add the product to the products array
                [products addObject:product];
          }
          // Finalize the statement to release its resources
          sqlite3_finalize(statement);
    }
    else {
          NSLog(@"Problem with the database:");
          NSLog(@"%d",sqlResult);
    }

    return products;

}
@end
```

Figure 18: DBAccess.m

<H2> Parameterized Queries

It is possible and quite common to use parameterized queries. For example, if you want to create a query that returns only products made in the USA, you can use the following SQL:

```
    SELECT Product.name, country.country
    FROM country,product
    WHERE countryoforiginid=countryid and country='USA'
```

This query is perfectly fine if you always want a list of the products made in the USA. If you want to decide at runtime which countries' products to display, you will need to use a parameterized query.

The first step in parameterizing this query is to replace the data that you want to parameterize with question marks (?). So your SQL will become this:

```
    SELECT Product.name, country.country
    FROM country,product
    WHERE countryoforiginid=countryid and country=?
```

After you have prepared your statement with `sqlite3_prepare_v2` but before you start stepping through your results with `sqlite3_step`, you need to bind your parameters. Remember that preparing the statement does not actually execute it. The statement is not executed until you begin iterating over the result set using `sqlite3_step`.

You use a series of functions to bind parameters just as you retrieve data fields. Here are the `bind` functions:

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_int64(sqlite3_stmt*, int, sqlite3_int64);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n, void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int, void(*)(void*));
int sqlite3_bind_value(sqlite3_stmt*, int, const sqlite3_value*);
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
```

You need to use the `bind` function that corresponds with the data type that you are binding. The first parameter in each function is the prepared statement. The second is the index (1-based) of the parameter in your SQL statement. The rest of the parameters vary based on the type that you are trying to bind.

To bind text at runtime, you use the `sqlite3_bind_text` function as follows:

```
sqlite3_bind_text (statement,1,value,-1, SQLITE_TRANSIENT);
```

In the above `bind` statement, `value` is the text that you would determine at runtime. In the example, that text would be “USA,” but you can set it to anything that you want dynamically at runtime. That is the advantage of using parameterized queries. Of course, you could just dynamically generate your SQL each time, but parameterized queries offer the performance advantage of preparing and compiling the statement once, and then caching and reusing the statement.

Quick Tip

The complete documentation of the `bind` functions is available on the SQLite web site.

<H2> Writing to the Database

If you modify a sample application or create your own SQLite application that attempts to write to the database, you will have a problem. The version of the database that you are using in the sample code is located in the application bundle, but the application bundle is read-only, so attempting to write to this database will result in an error.

To be able to write to the database, you need to make an editable copy. On the device, place this editable copy in the documents directory. Each application on the device is “sandboxed” and only has access to its own documents directory.

The following code snippet shows how to check whether a writable database already exists, and if not, create an editable copy.

```
// Create a writable copy of the default database from the bundle
// in the application Documents directory.
- (void) createEditableDatabase {
      // Check to see if editable database already exists
      BOOL success;
      NSFileManager *fileManager = [NSFileManager defaultManager];
      NSError *error;
      NSArray *paths = NSSearchPathForDirectoriesInDomains
            (NSDocumentDirectory, NSUserDomainMask, YES);
      NSString *documentsDir = [paths objectAtIndex:0];
      NSString *writableDB = [documentsDir
      stringByAppendingPathComponent:@"catalog.db"];
success = [fileManager fileExistsAtPath:writableDB];
// The editable database already exists
if (success) return;
}
// The editable database does not exist
// Copy the default DB to the application Documents directory.
NSString *defaultPath = [[[NSBundle mainBundle] resourcePath]
      stringByAppendingPathComponent:@"catalog.db"];
success = [fileManager copyItemAtPath:defaultPath
      toPath:writableDB error:&error];
if (!success) {
      NSAssert1(0, @"Failed to create writable database file:'%@'.",
      [error localizedDescription]);
}
}
```

You will need to change your database access code to call this function and then refer to the editable copy of the database instead of the bundled copy.

<H2> *Displaying the Catalog*

Now that you have the DBAccess class done, you can get on with displaying the catalog. In the MasterViewController, you will implement the code to retrieve the products array from the DBAccess class and display it in the table view. In the header, MasterViewController.h, you add import statements for the Product and DBAccess classes:

```
      #import "Product.h"
      #import "DBAccess.h"
```

You add a property to hold your products array:

```
      @property (strong, nonatomic) NSMutableArray* products;
```

In iOS 6, you can optionally synthesize properties by adding an @synthesize directive to the implementation file. Synthesizing a property causes the compiler to generate the getter and setter methods for the property. In iOS 6, the compiler synthesizes properties for you automatically. However, you can alternatively define the getter or setter, or both methods yourself. The compiler fills in the blanks by defining either of these methods if you don’t.

Continuing in the MasterViewController.m implementation class, you can add the following code to the viewDidLoad method to get your products array from the DBAccess class:

```
      - (void)viewDidLoad
      {
            [super viewDidLoad];
            // Do any additional setup after loading the view, typically from
            a nib.
            // Get the DBAccess object;
            DBAccess *dbAccess = [[DBAccess alloc] init];
            // Get the products array from the database
            self.products = [dbAccess getAllProducts];
            // Close the database because we are finished with it
            [dbAccess closeDatabase];
      }
```

To keep this example simple, you can remove the default code to provide the **Edit** and **Add** buttons at the top of the screen.

You should also modify the tableView:canEditRowAtIndexPath: method to disallow editing row data:

```
-(BOOL)tableView:(UITableView  *)tableView  canEditRowAtIndexPath:(NSIndexPath
*)indexPath
{
// Return NO if you do not want the specified item to be editable.
return NO;
}
```

Next, you have to implement your table view methods. The first thing you have to do is tell the table view the existing number of rows by implementing the `numberOfRowsInSection` method. You will get the count of items to display in the table view from your products array, as follows:

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
return [self.products count];
}
```

Finally, you have to implement `cellForRowAtIndexPath` to provide the table view with the table view cell that corresponds with the row, which is required by the table view. You look up the Product object using the row that the table view is asking for as the index into the array. The code for the `cellForRowAtIndexPath` method is as follows:

```
      // Customize the appearance of table view cells.
      - (UITableViewCell *)tableView:(UITableView *)tableView
            cellForRowAtIndexPath:(NSIndexPath *)indexPath {
      static NSString *CellIdentifier = @"Cell";
      UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:CellIdentifier];
      if (cell == nil) {
            cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
      }
      // Configure the cell.
      // Get the Product object
      Product* product = [self.products objectAtIndex:[indexPath row]];
      cell.textLabel.text = product.name;
      return cell;
      }
```

In **Swift**, you can view a table using an override function as follows:

```
override   func   tableView(tableView:   UITableView,   cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
    let     cell     =     tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)as UITableViewCell
    cell.textLabel?.text = Product!.options[indexPath.row].ID
    if(Product!.selectedoptionIndex == indexPath.row)
        cell.name = 123
    } else
        cell.name = 234
    }
```

```
        return cell
}
```

To set the Catalog text at the top in the Navigation Controller, navigate to the `initWithNibName:bundle:` method in the **MasterViewController.m** implementation file. Change the line that reads:

```
        self.title = NSLocalizedString(@"Master", @"Master");
```

to:

```
        self.title = NSLocalizedString(@"Catalog", @"Catalog");
```

You should now be able to build and run the application. When you run it, you should see the product catalog as in **Figure 19**. When you touch items in the catalog, nothing happens. However, the application should display the details of the product that is touched. To accomplish this, you need to implement the table view function `didSelectRowAtIndexPath`. However, before you do that, you need to build the view that you will display when a user taps a product in the table.

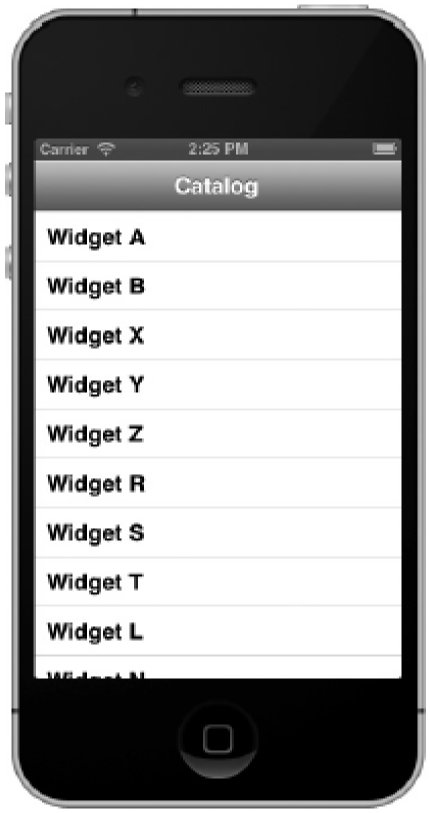


Figure 19: Product Catalog

<H2> Viewing Product Details

When a user taps a product in the table, the application should navigate to the product detail screen. Because this screen will be used with the `NavigationController`, it needs to be implemented using a `UIViewController`. A nice feature of the Master-Detail Application template is that the template automatically creates the detail view controller for you. Appropriately, the controller is called `DetailViewController`.

In the `DetailViewController.h` header, add Interface Builder outlets for the data that you want to display. You need to add an outlet property for each label control as follows:

```
@property (strong, nonatomic) IBOutlet UILabel* nameLabel;
@property (strong, nonatomic) IBOutlet UILabel* manufacturerLabel;
@property (strong, nonatomic) IBOutlet UILabel* detailsLabel;
@property (strong, nonatomic) IBOutlet UILabel* priceLabel;
@property (strong, nonatomic) IBOutlet UILabel* quantityLabel;
@property (strong, nonatomic) IBOutlet UILabel* countryLabel;
```

Because you will not be using it, you can delete the default Interface Builder outlet property that the project template created:

```
@property (strong, nonatomic) IBOutlet UILabel *detailDescriptionLabel;
```

You will use the outlets in the code to link to the user interface (UI) widgets.

Now, select the `DetailViewController.xib` file. In this file, you will add a series of `UILabels` to the interface as you designed in the mockup. Your interface should look something like **Figure 20**.

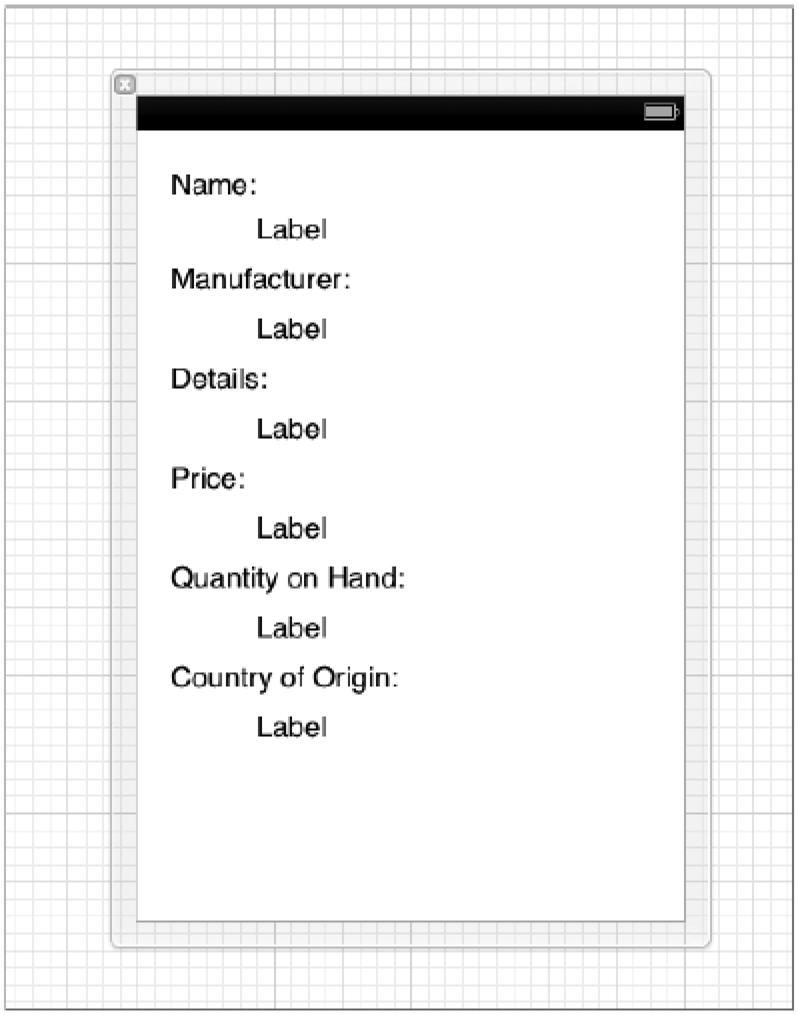


Figure 20: Interface after `UILabels`

Next, you need to hook up your labels to the outlets that you created in the `DetailViewController.h` header file. Make sure that you save the header file before you try to hook up the outlets, or the outlets that you created in the header will not be available.

To hook up the outlets, you can use the following steps:

1. Make sure that you have the `DetailViewController.xib` file open.
2. On the left side of the Interface Builder window, you should see two groups of objects: **Placeholders** and **Objects**. In the **Placeholders** group, select **File's Owner**.
3. Make sure that you have the Utility panel open, and click the last icon at the top of the pane to show the Connections inspector. In this pane, you can see all the class's outlets and their connections.

4. Click and drag from the open circle on the right side of the outlet name to the control to which that outlet should be connected in the user interface. When you are finished, the interface should look like **Figure 21**.

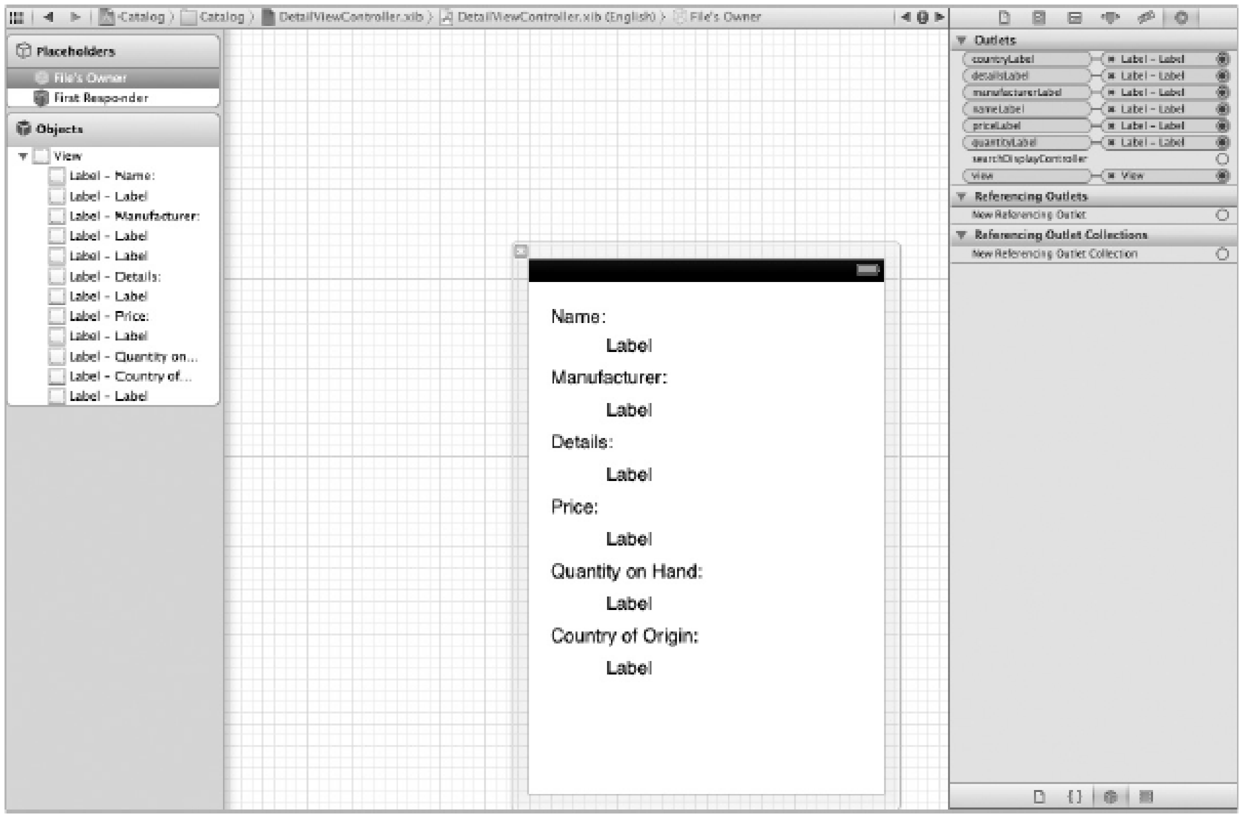


Figure 21: Interface after Hooking Up the Outlets

In `DetailViewController.m`, implement the `configureView` method to set the text in the labels, as follows:

```
- (void)configureView
{
// Update the user interface for the detail item.
if (self.detailItem) {
Product* theProduct = (Product*) self.detailItem;
// Set the text of the labels to the values passed in the Product object
[self.nameLabel setText:theProduct.name];
[self.manufacturerLabel setText:theProduct.manufacturer];
[self.detailsLabel setText:theProduct.details];
[self.priceLabel                                             setText:[NSString
stringWithFormat:@"%.2f",theProduct.price]];
[self.quantityLabel setText:[NSString stringWithFormat:@"%d",
theProduct.quantity]];
[self.countryLabel setText:theProduct.countryOfOrigin];
}
}
```

The Master-Detail Application template creates the `configureView` method to give you a place to configure the view. You add your code to set the text in the labels inside of the `if (self.detailItem)` block to ensure that you have an object that you want to display. In this application, the `detailItem` will be a Product object, so you cast `self.detailItem` to a `(Product*)`. In general, because `detailItem` is of type `id`, you can use `detailItem` in your own applications to display data from any type of object.

The code in the template calls the `configureView` method in the setter for the `detailItem` property, as you can see in the following code:

```
- (void)setDetailItem:(id)newDetailItem
{
if (_detailItem != newDetailItem) {

_detailItem = newDetailItem;
}
}
// Update the view.
[self configureView];
```

Therefore, when you set the `detailItem` property from the `MasterViewController`, the detail view controller will use the object in that property to configure the view.

To be able to navigate to your new screen, you need to modify the `tableView:didSelectRowAtIndexPath:` method in the `MasterViewController` to display the detail screen when a user selects a product.

In the `MasterViewController` implementation, modify the code in the `tableView:didSelectRowAtIndexPath:` method to set the `detailViewController.detailItem` to the object in the products array that the user selected:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
if (!self.detailViewController) {
self.detailViewController = [[DetailViewController alloc]
initWithNibName:@"DetailViewController" bundle:nil];
}
self.detailViewController.detailItem            =            [self.products
objectAtIndex:[indexPath row]];
[self.navigationController pushViewController:
self.detailViewController animated:YES];
}
```

That’s all there is to it. Now you should be able to build and run your application. Try tapping on an item in the catalog. The application should take you to the detail page for that item. Tapping the **Catalog** button in the navigation bar should take you back to the catalog. Tapping another row in the table view should take you to the data for that item.

You now have a fully functioning catalog application!

QUICK TIP

You can make an application more effective by learning about the various methods related to database access, and the use and management of database files. By creating effective queries, you can get quick desired results. You can easily maintain a large and useful database with the help of an effective database design and database manager.

THE BIG PICTURE

SQLite is the heart of an iOS application. It plays an important role for developing an application with immense features. Like our mind stores all the day-to-day information related to our life, SQLite stores all the features and history of an app.

Lab Connect

During the lab hour of this session, you will create a database, connect it with a lab application, and insert data into the database.

Cheat Sheet

- SQLite is an open source library, written in C that implements a self-contained SQL relational database engine. You can use SQLite to store a large amount of relational data.
- SQLite is a library because it is a fully self-contained SQL database engine. It is ideal for a mobile platform like iOS.
- Core Data is an object persistence framework. Its primary purpose is to provide the developer with a framework to retain objects that the application creates.

- SQLite excels when preloading your application with a large amount of data, whereas Core Data excels at managing data created on the device.
- Before you start designing a database and an application, you need to understand what the application will do.
- Normalization is the process of breaking down your data in a way that makes it easy to query. It helps to avoid common problems in database storage.
- You can create and populate a database using the command-line interface and scripts. You can also use the command-line interface to import data from a file into a table, read in and execute a file that contains SQL, and output data from a database in a variety of formats.
- To verify that your data was successfully imported, you can display it using the `SQL SELECT` statement.
- For developing simple iOS applications that do not require intense database development, you can use the SQLite Manager plug-in for the Firefox web browser.
- It is a good idea to mock up your application interface before you get started. It helps to get buy-in from customers that the interface you have designed meets their needs.
- You can design an application by following the Model-View-Controller (MVC) design pattern.
- It is a good idea to abstract away access to the database. This gives you the flexibility if you want to move to a different database engine later.
- You can use parameterized queries to prepare and compile a statement once, and then caching and reusing the statement.