Overview

Demo three programs to your TA showing the use of:

- Part 1: Introduction to NSUserDefaults
- Part 2: Introduction to using Resource Files
- Part 3: Introduction to SQL database
- Part 4: (Optional) UITableView

You may combine the programs together as long as you prove to the TA that you have learned all three methods of persistent data storage.

Submission

Lab evaluation form
Online Lab feedback form

Note: If you are unable to complete the lab, please be ready to demonstrate it at the beginning of the next lab. iPods are available for checkout from CSG. Labs are considered late if you are not ready to demo at the beginning of the next lab.

Part 1: UserDefaults

NSUserDefaults allows you to store and recall a small amount of data from persistent storage, and is usually used for storing program settings. The backend of UserDefaults is a .plist file (a property list), a file that maintains value-key pairs. NSUserDefaults gives you a simple means to store objects (NSString, double, int, NSArray) and their associated key.

Here's a simple example that stores an int and retrieves that same int:

```
// Get the singleton instance of NSUserDefaults
NSUserDefaults *storage = [NSUserDefaults standardUserDefaults];

// Store an integer in the standard NSUserDefaults instance
[storage setInteger:15 forKey:@"myInt"];

// Synchronize the in-memory instance of NSUserDefaults with the file on the disk
[storage synchronize];

// Exit the program, come back and retrieve the stored data
[storage integerForKey:@"myInt"];
```

Step 1) Start a new project called **UserDefault** using the Single View application template.
Step 2) Using a UITextField, allow the user to type in text.
Step 3) Create a button to store the text of the UITextField into the standard NSUserDefaults.
Step 4) Create a button to restore the text in the standard NSUserDefaults to the UITextField.
Step 5) Demo your app to the TA. You will need to double tap the home button, close the program, and reopen the program to ensure your persistent storage works.
*Review the optional part and the end of Lab 3 if you're unfamiliar with using UITextField.

Part 2: Introduction to Using Resource Files

You've already used resource files in prior labs (Audio files, images). Now you will parse a text file to display a series of jokes for Billy's iPhone app.

Step 1) Create a new project called **Billy** using the Single View application template.
Step 2) Download jokes.txt from the course website.
Step 3) Add jokes.txt as a resource file to your project. To do this, right-click (ctrl-click) in the Navigator and select *Add Files to <Your Project>…*

Step 4) jokes.txt contains one joke per line ("\n" separates each joke). Add a UILabel and UIButton to your app. When the user presses the button, a joke from jokes.txt should appear on the Label. Pressing the button multiple times should display a different joke each time.

Hints: Do you recall Question #2 from Part 2 on the Lab 5 Evaluatsion Form? File paths must be related to your projects "bundle". You should be able to use the code on Question #2 of Lab 5 in order to store the contents of the entire file into a NSString. Then you should be able to find a method of that you can call on your NSString object that will split the file contents by "\n" and store each component into an NSArray.

Step 5) Show the TA your completed application.

Part 3: Introduction to SQL

Billy has been very busy, and has collected a massive amount of jokes. Being a savy programmer, he has collected his jokes into an SQLite database file called jokes.db. You will now remake your jokes application so that you can display jokes from a database. The database consists of one table called **Billy** that has two columns, **category** and **joke**:

```
\-------------------------------------------------------------\
| category (TEXT)    | joke (TEXT)                            |
|-------------------------------------------------------------|
| Knock-knock        | Knock-knock. Who's there. Berry. B...  |
| One Liners         | A grasshopper walks into a bar.        |
| ...                | ...                                    |
\-------------------------------------------------------------\
```

There are many rows in the table. Each row is a joke. Currently, there are only three different categories labeling the jokes in the database: **Knock-knock**, **One Liners**, and **Geek Jokes**. You can insert a row into the table to add a new joke; if you do, label the joke with a category of **User**.

Step 1) Create a new project called **BillyDB** using the Utility application template.
Step 2) Download jokes.db from the course website.
Step 3) Add jokes.db as a resource file to your project. To do this, right-click (ctrl-click) in the Navigator and select *Add Files to <Your Project>…*
Step 4) Add the libsqlite3.0 framework to your project.
Step 5) Import sqlite3.h for you view controllers.

```
#import <sqlite3.h>
```

Step 6) Unless you're experienced at using sqlite in C, read the following information:

No matter what programming language you're using, working with an SQL database usually follows this pattern:

1. Open or connect to a database
2. CRUD – Create, Read, Update, or Delete rows in the tables of the database
3. Repeat step 2 lots of times.
4. Close the connection to the database

The following code **opens** a SQLite database from a file at a path stored in NSString *path:

```
sqlite3 *database = NULL;
const char *filename = [path cStringUsingEncoding:NSUTF8StringEncoding];
sqlite3_open(filename, &database); // returns SQLITE_OK if the database is opened properly
```

The following code synchronizes the in memory copy of the database with the persistent storage and **closes** the database:

```
sqlite3_exec(database, "Commit", NULL, NULL, NULL);
sqlite3_close(database);
```

CRUD is accomplished by writing statements in SQL (Structured Query Language) and executing those statements on the database. The next page gives an example of a query.

The following code **executes a statement** on the database (the statement is the 2nd argument):

```
NSMutableArray *rows = [[NSMutableArray alloc] init];
sqlite3_exec(database, "select joke from Billy", callback, rows, NULL);
```

Notes: *callback* is a C function, not an Objective-C method. You only need to implement it if you're querying the database (and need to run through the results of the query). The fourth argument to sqlite3_exec is the first argument to the callback. Thus, the following is an example of an implementation of the callback function:

```
int callback(void *context, int count, char **values, char **columns)
{
    NSMutableArray *rows = (NSMutableArray *)context;
    for (int i=0; i < count; i++) {
        const char *nameCString = values[i];
        [rows addObject:[NSString stringWithUTF8String:nameCString]];
    }
    return SQLITE_OK;
}
```

In this callback, all of the rows are stored as NSString objects into an array. Notice how we passed the array to the callback as the 4th parameter to the sqlite3_exec function.

Step 7) Search Google to brush up on SQL statements. You'll need to create statements to query for all jokes in a certain category and a statement to insert a row in the Billy table.

Here are a few good resources:
http://www.w3schools.com/sql/sql_select.asp
http://www.w3schools.com/sql/sql_where.asp
http://www.w3schools.com/sql/sql_insert.asp

Step 8) With your utility app, use the main view to allow the user to select all jokes that have the same category. Allow the user to tap through random jokes from that category. The design is up to you.

For the flipside view, allow the user to enter a joke in a text field and store that joke into the database under the **User** category.

Step 9) Show the TA your completed application. You should be able to query the database for different types of jokes.

Part 4: (Optional) UITable View

UITableView is a great way to organize lists of content. A typical UITableView consists of one or more sections, which consist of UITableViewCell cells (a cell is a row in the table).

Step 1) Create a new project called **Table** from the Single View application template.

Step 2) Open the view controller's .xib file or storyboard.

In Interface Builder:

1) Add a UITableView onto the view.
2) Ctrl-click (right click) on the UITableView instance and drag a connection from **delegate** to File's Owner (.xib) or View Controller (storyboard).
3) Ctrl-click (right click) on the UITableView instance and drag a connection from **dataSource** to File's Owner (.xib) or View Controller (storyboard).

You have now made your view controller the delegate and dataSource for the table view.

Step 3) In Xcode, declare in the header file that your view controller implements the delegate protocol for UITableViewDelegate and UITableViewDataSource:

```
@interface TableViewController : UIViewController <UITableViewDelegate, UITableViewDataSource>
```

TableViewControllers are a common design pattern for iOS devices. Alternatively, you could subclass the UITableViewController class to accomplish the same thing.

```
@interface TableViewController : UITableViewController
```

Right click on UITableViewController, UITableViewDelegate, and UITableViewDataSource, then select Jump to Definition to browse the header files for more information about the protocols.

Step 4) Implement methods from the UITableViewDelegate and UITableViewDataSource protocols. Most methods of the protocols have a default implementation. Some interesting methods for you to consider are:

numberOfSectionsInTableView:
tableView:numberOfRowsInSection:
tableView:titleForHeaderInSection:
tableView:didSelecRowAtIndexPath:

Most of these methods are very simple to implement, requiring you to return a number or an NSString *. The most complicated to implement is tableView:cellForRowAtIndexPath:, but a sample implementation has been provided for you below.

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"MainCell"];
    If (cell == nil) {
        cell = [[UITableViewCell alloc] initWithFrame:CGRectZero reuseIdentifier:@"MainCell"];
    }
    cell.textLabel.text = @"My Cell";
    //set other properties of cell

    return cell;
}
```

Notice how memory is saved by reusing the same instance of UITableViewCell. In cases where most cells in your table are similar, you'll want to follow the technique of the example. The caller will call the tableView:cellForRowAtIndexPath: method for every cell when it needs to draw the cells. For the first call, we allocate memory for a new UITableViewCell and set its text. After we pass back the cell to the caller, it will render the cell. For subsequent calls, we reuse the same cell (and reuse the memory) and simply change the text and other properties of the cell.