**Chapter 9 [Phase 1]: UITableView and UITableViewController**

**Step 1:** Create a new iOS Empty Application project [Product Name/Class Prefix: Homepwner; Device: iPhone; Check “Use Automatic…”].

**Step 2**: Write a subclass ItemsViewController of UITableViewController: File 🡪 New 🡪 File.... 🡪 Cocoa Touch [iOS section] 🡪 Objective-C class 🡪 Next 🡪 Class: ItemsViewController, Subclass of: UITableViewController 🡪 Next 🡪 Create.

A **UITableView** displays a single column of data with rows. UITableView is a view object [draws itself but no logic or data].

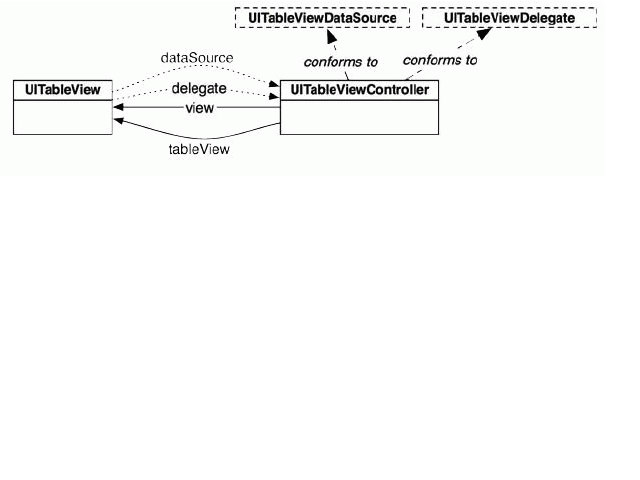
* Needs a **view controller** [appearance on the screen].
* Needs a **data source** [the number of rows to display, the data to be shown in rows]. Without a data source, a table view is just an empty container.
* Needs a **delegate** that can inform other objects of events involving the UITableView [any object that conforms to the UITableViewDelegate protocol].

🡪 An instance of the class **UITableViewController** fills all three roles: view controller, data source, and delegate.

**---------------------------------**

**PART I: View Controller**

**---------------------------------**



**UITableViewController** [a subclass of UIViewController] has a view [an instance of UITableView]. UITableViewController handles the UITableView.

**initWithStyle:** [the designated initializer of UITableViewController - determines the style of the table view]. Two options: **UITableViewStylePlain** [each row is a rectangle], **UITableViewStyleGrouped** [top/bottom rows have rounded corners].

|  |
| --- |
| **ItemsViewController.m**:  // All instances of ItemsViewController will use the UITableViewStyleGrouped style.  - (id)init  {  // Call the superclass's designated initializer.  self = [super initWithStyle:UITableViewStyleGrouped];  if (self)  { }  return self;  }  - (id)initWithStyle:(UITableViewStyle)style  {  return [self init];  } |
| **HomepwnerAppDelegate.m**: [import ItemsViewController.h]  - (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions  {  self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];    // Create an instance of ItemsViewController.  ItemsViewController \*itemsViewController = [[ItemsViewController alloc] init];    // Set itemsViewController as the rootViewController of the window.  [[self window] setRootViewController:itemsViewController];    self.window.backgroundColor = [UIColor whiteColor];  [self.window makeKeyAndVisible];  return YES;  } |
| **Result**: The default plain UITableView with no content [an empty table view] is on the screen. A UITableViewController inherits the **view method** from its superclass UIViewController which calls **loadView method** [creates and loads an empty view object if none exists]. |

**Step 3**: Import Item.h and Item.m from Chapter 3. [Drag the files onto the project window 🡪 Check: “Copy items…“ - it copies the files from their current directory to the project’s directory on the filesystem and adds them to the project.]

|  |  |
| --- | --- |
| **Item.h**:  @interface Item : NSObject  {}  + (id)randomItem;  - (id)initWithItemName:(NSString \*)name  valueInDollars:(int)value  serialNumber:(NSString \*)sNumber;  @property (nonatomic, copy) NSString \*itemName;  @property (nonatomic, copy) NSString \*serialNumber;  @property (nonatomic) int valueInDollars;  @property (nonatomic, readonly, strong) NSDate \*dateCreated;  @end | **Item.m**:  #import "Item.h"  @implementation Item  @synthesize itemName, serialNumber, dateCreated, valueInDollars;  + (id)randomItem  {  // Create an array of three adjectives  NSArray \*randomAdjectiveList = [NSArray arrayWithObjects:@"Fluffy",  @"Rusty",  @"Shiny", nil];    // Create an array of three nouns  NSArray \*randomNounList = [NSArray arrayWithObjects:@"Bear",  @"Spork",  @"Mac", nil];    // Get the index of a random adjective/noun from the lists  // The adjectiveIndex is a random number from 0 to 2 inclusive.    NSInteger adjectiveIndex = rand() % [randomAdjectiveList count];  NSInteger nounIndex = rand() % [randomNounList count];    // Create random name, value and a serial number  NSString \*randomName = [NSString stringWithFormat:@"%@ %@",  [randomAdjectiveList objectAtIndex:adjectiveIndex],  [randomNounList objectAtIndex:nounIndex]];  int randomValue = rand() % 100;  NSString \*randomSerialNumber = [NSString stringWithFormat:@"%c%c%c%c%c",  '0' + rand() % 10,  'A' + rand() % 26,  '0' + rand() % 10,  'A' + rand() % 26,  '0' + rand() % 10];    // Ignore the memory problems  Item \*newItem =  [[self alloc] initWithItemName:randomName  valueInDollars:randomValue  serialNumber:randomSerialNumber];  return newItem;  }  - (id)initWithItemName:(NSString \*)name  valueInDollars:(int)value  serialNumber:(NSString \*)sNumber  {  // Call the superclass's designated initializer  self = [super init];    // Did the superclass's designated initializer succeed?  if(self)  {  // Give the instance variables initial values  [self setItemName:name];  [self setSerialNumber:sNumber];  [self setValueInDollars:value];  dateCreated = [[NSDate alloc] init];  }    // Return the address of the newly initialized object  return self;  }  - (id)init  {  return [self initWithItemName:@"Possession"  valueInDollars:0  serialNumber:@""];  }  - (NSString \*)description  {  NSString \*descriptionString =  [[NSString alloc] initWithFormat:@"%@ (%@): Worth $%d, recorded on %@",  itemName,  serialNumber,  valueInDollars,  dateCreated];  return descriptionString;  }  @end |

**Step 4**: Write a class ItemStore: File 🡪 New 🡪 File.... 🡪 Cocoa Touch [iOS section] 🡪 Objective-C class 🡪 Next 🡪 Class: ItemStore, Subclass of: NSObject 🡪 Next 🡪 Create.

**------------------------------**

**PART II: Data Source**

**------------------------------**

In Cocoa Touch, the table view object asks another object [its dataSource] what it should display.

**Homepwner:** Data source [ItemsViewController] will store item data. The NSMutableArray [holds the Item instances] will be abstracted into another object [ItemStore]. If an object wants to see all of the items, it will ask the ItemStore for the array that contains them.

**ItemStore** is a singleton [only one instance in the app – many objects talk to it]. When another instance is created, the class will return the existing instance instead. Those objects can ask the singleton class for its one instance, instead of passing that instance as an argument to every method that will use it.

* **sharedStore method** – gets the ItemStore instance. If ItemStore instance created, returns the instance. If not, creates the instance and returns it.
* **sharedStore variable** is **static** - does not live on the stack, declared when the app is loaded into memory, and never destroyed. Access variable in the method in which it is declared. No other object or method can use the variable except via the sharedStore method. The initial value of sharedStore is nil. The first time this method runs, an instance of ItemStore will be created, and sharedStore will be set to point to it.
* **The singleton status** of ItemStore – only one instance of ItemStore can be allocated. **allocWithZone**: [overridden method] returns single ItemStore instance. Send allocWithZone: to super - skips trap and gets an instance of ItemStore when needed. Not sending allocWithZone to NSObject causes loop.
* **@Item directive** - tells the compiler that there is a Item class but it doesn’t need to know this class’s details in the current file. Using the class without importing it - speeds up compile times considerably because fewer files have to be recompiled when one file changes. The class must import the file when it sends messages to the Item class or instances of it.

|  |  |
| --- | --- |
| **ItemStore.h:**  #import <Foundation/Foundation.h>  @class Item;  @interface ItemStore : NSObject  {  NSMutableArray \*allItems;  }  + (ItemStore \*)sharedStore;  - (NSArray \*)allItems;  - (Item \*)createItem;  @end | **ItemStore.m:**  #import "ItemStore.h"  #import "Item.h"  @implementation ItemStore  + (ItemStore \*)sharedStore  {  static ItemStore \*sharedStore = nil;    if(!sharedStore)  sharedStore = [[super allocWithZone:nil] init];    return sharedStore;  }  + (id)allocWithZone:(NSZone \*)zone  {  return [self sharedStore];  }  - (id)init  {  self = [super init];  if(self)  allItems = [[NSMutableArray alloc] init];  return self;  }  - (NSArray \*)allItems  {  return allItems;  }  - (Item \*)createItem  {  Item \*p = [Item randomItem];  [allItems addObject:p];  return p;  }  @end  **ItemsViewController.m [continued]**:  #import "Item.h"  #import "ItemStore.h"  // All instances of ItemsViewController will use the UITableViewStyleGrouped style.-  - (id)init  {  …  if (self)  {  // Add 5 random items to the ItemStore  for(int i = 0; i < 5; i++)  [[ItemStore sharedStore] createItem];  }  …  } |

**Step 5**: Create a row for each entry in the store. Each cell will display the description of an Item as its textLabel.

**--------------------------**

**PART III: Delegate**

**--------------------------**

For ItemsViewController to conform to UITableViewDataSource, it must implement **tableView:numberOfRowsInSection:** [tells the table view how many rows to display] and **tableView:cellForRowAtIndexPath:** [tells the table view what content to display in each row].

**ItemsViewController.m:**

-(NSInteger)tableView:(UITableView \*)tableView

numberOfRowsInSection:(NSInteger)section

{

// Obtain the number of rows in the section.

return [[[ItemStore sharedStore] allItems] count];

}

Table views have sections, each section has its own set of rows. A table has a one column, a column has many cells, and a row has only one cell [Hierarchy: UIView 🡪 UITableView 🡪 UITableViewCell (subviews of contentView: textLabel, detailTextLabel, imageView; accessory indicator such as check mark)].

**Reusing UITableViewCells:** To preserve the lives of iOS devices, reuse table view cells. When the user scrolls the table, some cells move offscreen.

Offscreen cells are put into a pool of cells available for reuse. Then, instead of creating a brand new cell for every request, the data source first checks the pool. If there is an unused cell, the data source configures it with new data and returns it to the table view.

In **ItemsViewController.m**, implement **tableView:cellForRowAtIndexPath:** so that the nth row displays the nth entry in the allItems array.

- (UITableViewCell \*)tableView:(UITableView \*)tableView cellForRowAtIndexPath:(NSIndexPath \*)indexPath

{

// Create an instance of UITableView with default appearance. Check for a reusable cell first, use that if it exists.

UITableViewCell \*cell = [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"];

// If there is no reusable cell of this type, create a new one

if(!cell)

cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"UITableViewCell"];

// Set the text on the cell with the description of the item. That is the nth index of items, where n = row this cell will appear in on the tableview.

Item \*p = [[[ItemStore sharedStore] allItems] objectAtIndex:[indexPath row]];

[[cell textLabel] setText:[p description]];

return cell;

}

**Result**: a UITableView populated with a list of random Items.

**Chapter 10 [Phase 2]: Editing UITableView**

**Editing Mode**: UITableView has an editing property, and when this property is set to YES, the UITableView enters editing mode. Once the table view is in editing mode, the rows of the table can be manipulated by the user [re-order, add, or remove rows]. It does not allow editing the content of a row.

**Homepwner:** The **header view** appears at the top of the item list. It will have **2 subviews** [instances of UIButton]: one for editing mode and the second adds a new Item to the table. View and its subviews are created in **XIB file**. **ItemsViewController** unarchives the XIB file when it needs to display the header view.

**-------------------------------**

**PART I: Enable Editing**

**-------------------------------**

**Step 1:** In **ItemsViewController.h**, declare an instance variable of type UIView for your header view and three new methods. headerView is a strong reference because it is a top-level object in the XIB file.

#import <UIKit/UIKit.h>

@interface ItemsViewController : UITableViewController

{

IBOutlet UIView \*headerView;

}

- (UIView \*)headerView;

- (IBAction)addNewItem:(id)sender;

- (IBAction)toggleEdittingMode:(id)sender;

@end

**Step 2:** Create the new XIB file for laying out view objects, archiving them, and having them loaded at runtime. File 🡪 New 🡪 File... 🡪 User Interface [iOS section] 🡪 Empty template 🡪 Next 🡪 iPhone 🡪 Save file as HeaderView 🡪 select File's Owner object and change its Class to ItemsViewController in the identity inspector .

**Step 3:** Drag a Custom UIView onto the canvas [background - completely transparent, Opacity slider 0]. Then drag two instances of UIButton onto that view. **Make the connections**: File’s Owner [right-click]: drag from headerView to UIView; Edit UIButton [right-click]: drag to File’s Owner [toggleEditingMode:]; New UIButton [right-click]: drag to File’s Owner [addNewItem:]

**Step 4:** HeaderView.xib will have ItemsViewController load XIB file manually using **NSBundle**. NSBundle is interface between an app and the appbundle it lives in. An instance of NSBundle is created when app launches, get a pointer to the instance by sending the message mainBundle to NSBundle. Once there is a pointer to the main bundle object, ask it to load a XIB file. In **ItemsViewController.m**, implement headerView. Passing self as the owner of the XIB file places the instance of ItemsViewController in the File's Owner hole of the XIB file. The first time the headerView message is sent to the ItemsViewController, it will load HeaderView.xib and keep a pointer to the view object in the instance variable headerView. The buttons in this view will send messages to the ItemsViewController when tapped.

- (UIView \*)headerView

{

// if the headerView is not loaded yet

if(!headerView)

[[NSBundle mainBundle] loadNibNamed:@"HeaderView" owner:self options:nil];

return headerView;

}

**Step 5:** Set headerView as the header view of the table. UITableView sends messages to its delegate, ItemsViewController, when it needs to show the header view. The first time tableView:heightForHeaderInSection: is sent to ItemsViewController, it will send itself the message headerView. At this time, headerView will be nil, which causes headerView to be loaded from XIB file. Implement methods from UITableViewDelegate protocol in **ItemsViewController.m**:

- (UIView \*)tableView:(UITableView \*)tv viewForHeaderInSection:(NSInteger)section

{

return [self headerView];

}

- (CGFloat)tableView:(UITableView \*)tableView heightForHeaderInSection:(NSInteger)section

{

// The height of the header view should be determined from the height of the view in the XIB file.

return [[self headerView] bounds].size.height;

}

**Step 6:** To set the editing property for a view controller using **setEditing:animated:** method. In **ItemsViewController.m**, implement **toggleEditingMode**:

**Result**: Tap the Edit button, and the UITableView will enter editing mode

**---------------------------**

**PART II: Add a row**

**---------------------------**

**Step 7:** When the New button is tapped, a new row will be added to the UITableView. DataSource of the UITableView determines # of rows the table view should display. ItemsViewController consults the store and returns the #of rows. The UITableView and its dataSource must agree on the # of rows. Add a new Item to the ItemStore before inserting the new row. In **ItemsViewController.m** implement **addNewItem**:

- (IBAction)addNewItem:(id)sender

{

// Create a new Item and add it to the store

Item \*newItem = [[ItemStore sharedStore] createItem];

// Figure out where that item is in the array

int lastRow = [[[ItemStore sharedStore] allItems] indexOfObject:newItem];

NSIndexPath \*ip = [NSIndexPath indexPathForRow:lastRow inSection:0];

// Insert this new row into the table.

[[self tableView] insertRowsAtIndexPaths:[NSArray arrayWithObject:ip] withRowAnimation:UITableViewRowAnimationTop];

}

**Result:** Tap the New button and watch the new row slide into the bottom position of the table.

**---------------------------**

**PART III: Delete row**

**---------------------------**

**Step 8:** Remove the Item from the ItemStore. The ItemStore must know how to remove objects from itself. In **ItemStore.h**, declare a **removeItem** method:

@interface ItemStore : NSObject

…

- (void)removeItem:(Item \*)p;

@end

NSMutableArray’s **removeObject:** sends a message isEqual: to objects. **removeObjectIdenticalTo:** removes an object if it is the same object as object passed to the message, it does not override isEqual:. Use removeObjectIdenticalTo: when a particular instance is specified. In **ItemStore.m**, implement removeItem:

- (void)removeItem:(Item \*)p

{

[allItems removeObjectIdenticalTo:p];

}

**Step 9:** Remove the row from the UITableView. Implement **tableView:commitEditingStyle:forRowAtIndexPath:** from the **UITableViewDataSource protocol**. It passes two arguments [**UITableViewCellEditingStyle** (Delete) and **NSIndexPath** is the row in the table]. In **ItemsViewController.m**, implement this method and confirm the row deletion by sending the message **deleteRowsAtIndexPaths:withRowAnimation:** back to the table view.

- (void)tableView:(UITableView \*)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath \*)indexPath

{

// If the table view is asking to commit a delete command...

if(editingStyle == UITableViewCellEditingStyleDelete)

{

ItemStore \*ps = [ItemStore sharedStore];

NSArray \*items = [ps allItems];

Item \*p = [items objectAtIndex:[indexPath row]];

[ps removeItem:p];

// Remove the row from the table view with an animation

[tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] withRowAnimation:UITableViewRowAnimationFade];

}

}

**Result:** create some rows, and then delete a row. It will disappear.

**------------------------------**

**PART IV: Moving rows**

**------------------------------**

**Step 10:** Write an ItemStore a method to change the order of Items in its allItems array. In **ItemStore.h**, declare **moveItemAtIndex**

@interface ItemStore : NSObject

…

- (void)moveItemAtIndex:(int)from toIndex:(int)to;

@end

Implement this method in **ItemStore.m**:

- (void)moveItemAtIndex:(int)from toIndex:(int)to

{

if(from == to)

return;

// Get pointer to object being moved so we can re-insert it

Item \*p = [allItems objectAtIndex:from];

// Remove p from array

[allItems removeObjectAtIndex:from];

// Insert p in array at new location

[allItems insertObject:p atIndex:to];

}

**Step 11:** Moving a row doesn’t require confirmation, the table view moves the row and reports the move to its data source by sending the message **tableView:moveRowAtIndexPath:toIndexPath**:. Catch this message to update your data source to match the new order.

- (void)tableView:(UITableView \*)tableView

moveRowAtIndexPath:(NSIndexPath \*)fromIndexPath

toIndexPath:(NSIndexPath \*)toIndexPath

{

[[ItemStore sharedStore] moveItemAtIndex:[fromIndexPath row] toIndex:[toIndexPath row]];

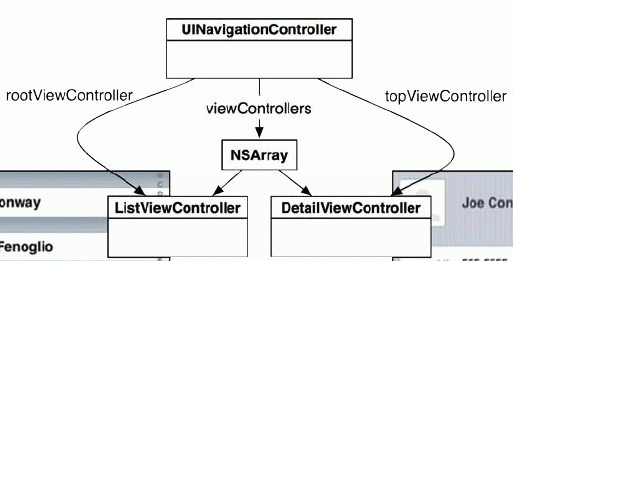
}

**Result**: check out the new reordering controls on the side of each row. Touch and hold a reordering control and move the row to a new position.

**Chapter 11 [Phase 3]: UINavigationController**

**Homepwner:** add a UINavigationController to the app. The ItemsViewController is the UINavigationController’s rootViewController. Then, create another subclass of UIViewController that can be pushed onto the UINavigationController’s stack. When a user selects one of the rows, the new UIViewController’s view will slide onto the screen. This view controller will allow the user to view and edit the properties of the selected Item.

**UINavigationController** has a stack [array] of screens [views of a UIViewController]. When a UINavigationController instance is initialized, it gets 1 UIViewController. The navigation controller’s root view controller is on the bottom of the stack [first object in the array]. More view controllers are pushed on top of the stack [end of the array] while app is running [only the root view controller is guaranteed to always be in the stack.]. When a UIViewController is pushed onto the stack, its view slides onto the screen from the right. When the stack is popped, the top view controller is removed from the stack, and the view of the one below it slides onto the screen from the left. When a UIViewController is on top of the stack [last view controller], its view is visible. **UITabBarController** gets all of its view controllers when it is initialized.



**UINavigationController** [subclass of UIViewController has a view] has two subviews: a UINavigationBar and the view of topViewController.

**Step 1:** Create UINavigationController, give it a root view controller of its own, and set the INavigationController as the root view controller of the window:

In **HomepwnerAppDelegate.m**:

- (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions

{

…

// Create an instance of ItemsViewController.

ItemsViewController \*itemsViewController = [[ItemsViewController alloc] init];

// Create an instance of UINavigationController, its stack contains only ItemsViewController

UINavigationController \*navController = [[UINavigationController alloc] initWithRootViewController:itemsViewController];

// Place navigation controller's view in the window hierarchy

[[self window] setRootViewController:navController];

…

}

**Result**: a UINavigationBar appears at the top of the screen.

**Step 2:** Write a subclass DetailViewController [The screen will have editable text fields for each property of the Item] of UIViewController: File 🡪 New 🡪 File.... 🡪 Cocoa Touch [iOS section] 🡪 Objective-C class 🡪 Next 🡪 Class: DetailViewController, Subclass of: UIViewController [check “With XIB for user interface”] 🡪 Next 🡪 Create.

**Step 3:** Open DetailViewController.xib [project navigator] 🡪 Option [alt]-click on DetailViewController.h [project navigator] 🡪 Drag subviews onto the view [4 UILabels, 3 UITextFields] 🡪 In DetailViewController.h, add the instance variable curly brackets 🡪 In XIB file, control-drag from the UITextField next to the Name label to the instance variable area [inside the curly brackets] in DetailViewController.h 🡪 Let go inside the instance variable area, and a pop-up window will appear [Enter nameField into the field, select Weak (the object it will point to is not a top-level object in the XIB file), Connect]. Create the other three outlets the same way. **DetailViewController.h**:

@interface DetailViewController : UIViewController

{

\_\_weak IBOutlet UITextField \*nameField;

\_\_weak IBOutlet UITextField \*serialNumberField;

\_\_weak IBOutlet UITextField \*valueField;

\_\_weak IBOutlet UILabel \*dateLabel;

}

@end

**Step 4:** For each of the UITextFields in the XIB file. [Control-drag from the UITextFields to the File's Owner and select delegate from the list.]

**Step 5:** Change DetailViewController’s view background to the same background as the UITableView. Recall that a view controller’s view is not created until the view controller loads it the first time, so the code to perform extra setup on the view should be in **viewDidLoad**. **DetailViewController.m:**

- (void)viewDidLoad

{

[super viewDidLoad];

[[self view] setBackgroundColor:[UIColor groupTableViewBackgroundColor]];

}

**Step 6:** To fill the UITextFields, pass the selected Item from the ItemsViewController to the DetailViewController. Give DetailViewController a property to hold an Item. When a row is tapped, ItemsViewController will give the corresponding Item to the instance of DetailViewController that is being pushed onto the stack. The DetailViewController will populate its text fields with the properties of that Item. Editing the text in the UITextFields on DetailViewController’s view will change the properties of that Item.

|  |  |
| --- | --- |
| In **DetailViewController.h**:  @class Item;  @interface DetailViewController : UIViewController  {…}  @property (nonatomic, strong) Item \*item;  @end | In **DetailViewController.m**, synthesize the accessors for item and import Item.h:  #import "Item.h"  @implementation DetailViewController  @synthesize item;  …  @end |

**Step 7**: **Appearing** and **disappearing views:** whenever a UINavigationController is about to swap views, it sends out two messages: **viewWillDisappear:** [UIViewController that is about to be popped off the stack] and **viewWillAppear:** [UIViewController that will be on top of the stack]. When implementing these methods, call the superclass’s implementation.

When the DetailViewController’s view appears on the screen, it needs to setup its subviews to show the properties of the item. In DetailViewController.m, override **viewWillAppear**: to transfer the item’s properties to the various UITextFields. In **DetailViewController.m**:

- (void)viewWillAppear:(BOOL)animated

{

[super viewWillAppear:animated];

// Set three data fields.

[nameField setText:[item itemName]];

[serialNumberField setText:[item serialNumber]];

[valueField setText:[NSString stringWithFormat:@"%d", [item valueInDollars]]];

// Set the data, turn the date into a date string.

NSDateFormatter \*dateFormatter = [[NSDateFormatter alloc] init];

[dateFormatter setDateStyle:NSDateFormatterMediumStyle];

[dateFormatter setTimeStyle:NSDateFormatterNoStyle];

// Use filtered NSDate object to set dateLabel contents

[dateLabel setText:[dateFormatter stringFromDate:[item dateCreated]]];

// Change the navigation item to display name of item

[[self navigationItem] setTitle:[item itemName]];

}

When a DetailViewController is popped off the stack, set the properties of its item to the contents of the UITextFields. In **DetailViewController.m**:

- (void)viewWillDisappear:(BOOL)animated

{

[super viewWillDisappear:animated];

// Clear first responder

[[self view] endEditing:YES];

// Save changes to item

[item setItemName:[nameField text]];

[item setSerialNumber:[serialNumberField text]];

[item setValueInDollars:[[valueField text] intValue]];

}

When the message **endEditing:** is sent to a view, if it or any of its subviews is currently the first responder, it will resign its first responder status, and the keyboard will be dismissed. The values of the Item will be updated when the user taps the Back button on the UINavigationBar. When ItemsViewController appears back on the screen, it is sent the message **viewWillAppear:.** Reload the UITableView so to see the changes. In **ItemsViewController.m:**

- (void)viewWillAppear:(BOOL)animated

{

[super viewWillAppear:animated];

[[self tableView] reloadData];

}

**Result:** move back and forth between the UIViewControllers you created and change the data with ease.

**Step 8:** The user should be able to tap a row in ItemsViewController’s table view and have the DetailViewController’s view slide onto the screen and display the properties of the selected Item instance. ItemsViewController will create the instance of DetailViewController and add it to the stack.

**1.** It knows when to push DetailViewController onto the stack. As the table view’s delegate, it receives message **tableView:didSelectRowAtIndexPath:**.

**2.** It needs a pointer to the navigation controller to send the navigation controller message **pushViewController:animated:**.

When a row is tapped, its delegate is sent **tableView:didSelectRowAtIndexPath:** [contains the index path of the selected row]. Implement this method to create a DetailViewController and then push it on top of the navigation controller’s stack.

The UINavigationController’s stack is an array, it will take ownership of any view controller added to it. Thus, the DetailViewController is owned only by the UINavigationController after tableView:didSelectRowAtIndexPath: finishes. When the stack is popped, the DetailViewController is destroyed. The next time a row is tapped, a new instance of DetailViewController is created.

In **ItemsViewController.h** [import DetailViewController.h]. In **ItemsViewController.m:**

- (void)tableView:(UITableView \*)aTableView didSelectRowAtIndexPath:(NSIndexPath \*)indexPath

{

// Set DetailViewController to have an item before viewWillAppear gets called

DetailViewController \*detailViewController = [[DetailViewController alloc] init];

NSArray \*items = [[ItemStore sharedStore] allItems];

Item \*selectedItem = [items objectAtIndex:[indexPath row]];

// Give detail view controller a pointer to the view object in row

[detailViewController setItem:selectedItem];

// Push it into the top of the navigation controller's stack

[[self navigationController] pushViewController:detailViewController animated:YES];

}

**Step 9**: A **UINavigationBar** should display a descriptive title for the UIViewController that is currently on top of the UINavigationController’s stack. It is not a subclass of UIView, so it cannot appear on the screen. The navigation item supplies the navigation bar with the content it needs to draw. When a UIViewController comes to the top of a UINavigationController’s stack, the UINavigationBar uses the UIViewController’s navigationItem to configure itself.

By default, a UINavigationItem is empty. At the most basic level, a UINavigationItem has a simple title string. In **ItemsViewController.m**, modify **init** to set the navigationItem’s title to read Homepwner:

- (id)init

{

if (self)

{

…

// Set the tile to Homepwner

UINavigationItem \*n = [self navigationItem];

[n setTitle:@"Homepwner"];

}

…

}

**Result:** The string Homepwner appears on the navigation bar.

Set the DetailViewController’s navigation item title be the name of the Item it is displaying. In DetailViewController.m, implement setItem:

- (void)setItem:(Item \*)i

{

item = i;

[[self navigationItem] setTitle:[item itemName]];

}

**Result:** Create and tap a row, the title of the navigation bar is the name of the Item that was selected.

**Step 9**: There are 3 areas for each UINavigationItem: a leftBarButtonItem, a rightBarButtonItem, and a titleView. The left and right bar button items are pointers to instances of UIBarButtonItem, which contains the information for a button that can only be displayed on a UINavigationBar or a UIToolbar.

Add a **UIBarButtonItem** to the UINavigationBar on the right side of the navigation bar when the ItemsViewController is on top of the stack. When tapped, it should add a new Item to the list. A bar button item has a target-action pair that works like UIControl’s target-action mechanism. When you set a target-action pair in a XIB file, you Control-drag from a button to its target and then select a method from the list of IBActions.

Add another UIBarButtonItem to replace the Edit button in the table view header. UIViewController has an editButtonItem property, and when sent editButtonItem, the view controller creates a UIBarButtonItem with the title Edit. This button comes with a target-action pair: it sends the message **setEditing:animated:** to its UIViewController when tapped. In **ItemsViewController.m**, create a UIBarButtonItem instance and give it its target and action:

- (id)init

{

…

if (self)

{

…

[n setTitle:@"Homepwner"];

// Create a new bar button item that will send addNewItem: to ItemsViewController

UIBarButtonItem \*bbi = [[UIBarButtonItem alloc]

initWithBarButtonSystemItem:UIBarButtonSystemItemAdd

target:self

action:@selector(addNewItem:)];

// Set this bar button item as the right item in the navigation item

[[self navigationItem] setRightBarButtonItem:bbi];

// Set this bar button item as the left item in the navigation item

[[self navigationItem] setLeftBarButtonItem:[self editButtonItem]];

}

…

}