**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].

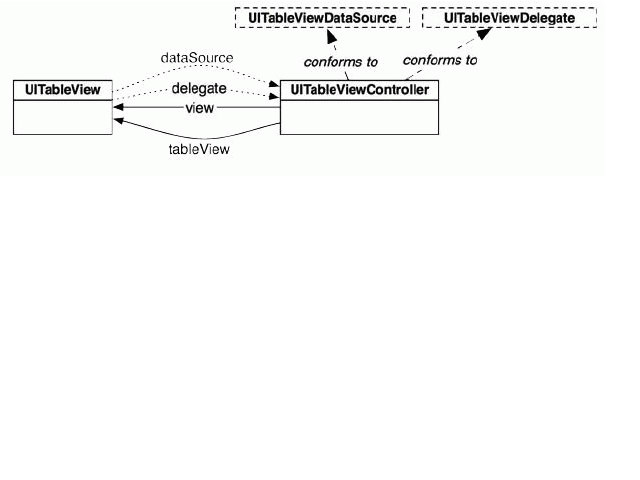
* Needs a **view controller** [appearance on the screen].
* Needs a **data source** [the number of rows to display, the data to be shown in rows]. Without a data source, a table view is just an empty container.
* Needs a **delegate** that can inform other objects of events involving the UITableView [any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

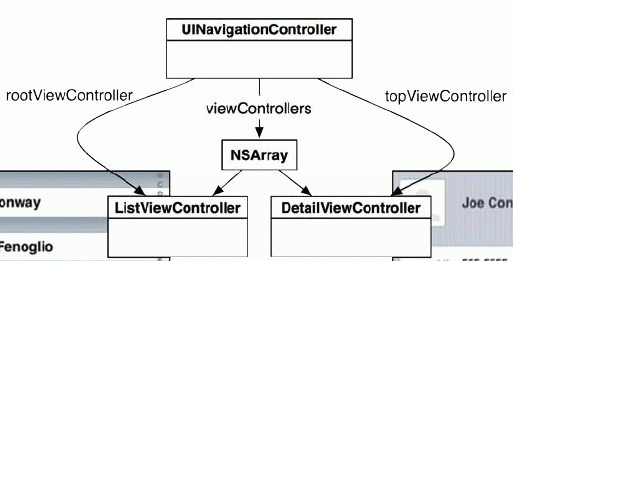
}

**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]];

}

…

}

12

Camera

In this chapter, you’re going to add photos to the Homepwner application. You will present a UIImagePickerController so that the user can take and save a picture of each item. The image will then be associated with a BNRItem instance, stored in an image store, and viewable in the item’s detail view. Then, when the insurance company demands proof, the user will have a visual record of owning that 70" HDTV.

Figure12.1 Homepwnerwithcameraaddition



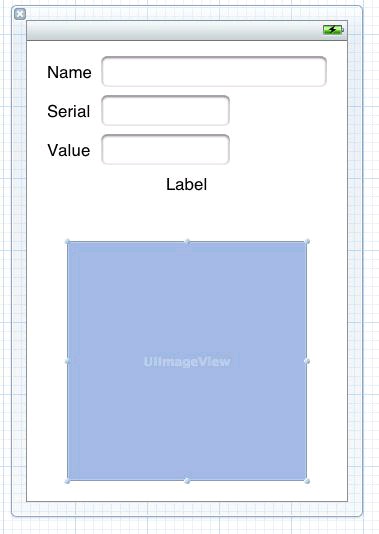


Displaying Images and UIImageView

Because we want the image to appear in the item’s detail view, our first step is to have the DetailViewController get and display an image. An easy way to display an image is to put an instance of UIImageView on the screen. Open Homepwner.xcodeproj and DetailViewController.xib. Then drag an instance of UIImageView onto the view, as shown in Figure 12.2.

Figure12.2 UIImageViewonDetailViewController’sview

A UIImageView displays an image according to its contentMode property. This property determines where to

position and how to resize the content within its frame.

position and how to resize the content within its frame. UIImageView’s default value for contentMode is UIViewContentModeScaleToFill, which will adjust the image to exactly match the bounds of the image view. If you keep the default, the image taken by the camera will be contorted to fit into the square UIImageView. You have to change the contentMode of the image view so that it resizes the image with the same aspect ratio.

Select the UIImageView and open the attributes inspector. Find the Mode attribute and change it to Aspect Fit, as shown in Figure 12.3. This will resize the image to fit within the bounds of the UIImageView.

Figure 12.3 Image viewattributes

Now, Option-click DetailViewController.h in the project navigator to open it in the assistant editor. Control-drag from the UIImageView to the instance variable area in DetailViewController.h. Name the outlet imageView and choose Weak as the storage type. Click Connect.

DetailViewController’s instance variable area should now

look like this:

@interface DetailViewController : UIViewController {

\_\_weak IBOutlet UITextField \*nameField; \_\_weak IBOutlet UITextField \*serialNumberField; \_\_weak IBOutlet UITextField \*valueField; \_\_weak IBOutlet UILabel \*dateLabel; \_\_weak IBOutlet UIImageView \*imageView;

}

Taking pictures and UIImagePickerController

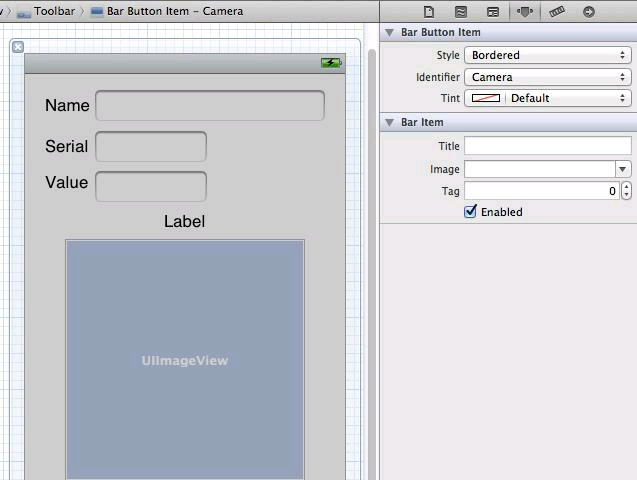
Now you need a button to initiate the photo-taking process. It

Now you need a button to initiate the photo-taking process. It would be nice to put this button on the navigation bar, but we will need the navigation bar for another button later. Instead, we will create an instance of UIToolbar and place it at the bottom of DetailViewController’s view. In DetailViewController.xib, drag a UIToolbar onto the bottom of the view.

A UIToolbar works a lot like a UINavigationBar – you can add UIBarButtonItems to it. However, where a navigation bar has two slots for bar button items, a toolbar has an array of bar button items. You can place as many UIBarButtonItems in a toolbar as can fit on the screen.

By default, a new instance of UIToolbar created in a XIB file comes with one UIBarButtonItem. Select this bar button item and open the attribute inspector. Change the Identifier to Camera, and the item will show a camera icon (Figure 12.4).

Figure12.4 UIToolbarwithbarbuttonitem

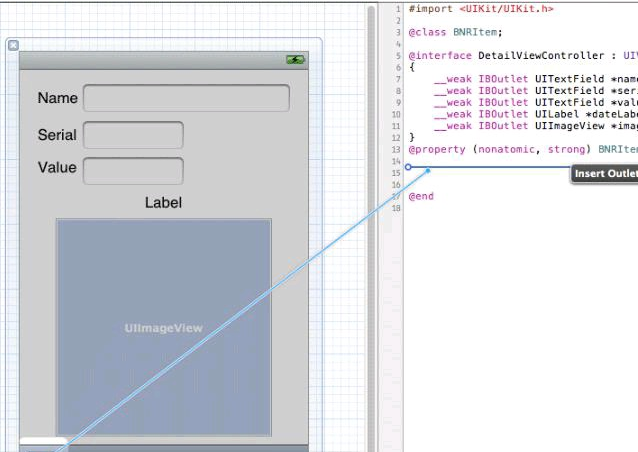
 



The camera button will need to send a message to the instance of DetailViewController when it is tapped. In previous exercises, you connected action methods in two steps: declaring them in the header file and then hooking them up in the XIB file. Just like you did with outlets, you can do both steps at once by opening a source file in the assistant editor and Control-dragging from a XIB file to the file. Option-click DetailViewController.h in the project navigator to open it in the assistant editor .

In DetailViewController.xib, select the camera button and Control-drag from the button to the method declaration area in DetailViewController.h (Figure 12.5).

Figure12.5 Creatingandconnectinganactionmethod fromaXIB



Let go of the mouse, and a window will appear that allows you to specify the type of connection you are creating. From the Connection pop-up menu, choose Action. Then, name this method takePicture: and click Connect (Figure 12.6).

Figure 12.6 Creating the action

Now the action method is declared in the header file, and the UIBarButtonItem instance in the XIB is hooked up to send this message to the DetailViewController when tapped. Connecting an action method in this way also automatically adds a stub implementation in DetailViewController.m:

- (IBAction)takePicture:(id)sender { }

The interface for DetailViewController now looks like this:

@interface DetailViewController : UIViewController {

\_\_weak IBOutlet UITextField \*nameField; \_\_weak IBOutlet UITextField \*serialNumberField; \_\_weak IBOutlet UITextField \*valueField; \_\_weak IBOutlet UILabel \*dateLabel; \_\_weak IBOutlet UIImageView \*imageView;

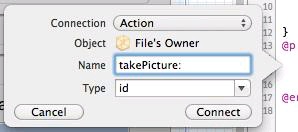
} @property (nonatomic, strong) BNRItem \*item;

- (IBAction)takePicture:(id)sender;

@end

If you made any mistakes while making this connection and later realize it (or realize it now), make sure you open DetailViewController.xib and disconnect any bad connections. (Look for yellow warning signs in the connections inspector.)

Also, notice that Xcode is smart enough to tell you when an action method is connected in the XIB file. See that little circle within a circle in the gutter area next to takePicture:’s

method name (Figure 12.7)? When this circle is filled in, this

method name (Figure 12.7)? When this circle is filled in, this action method is connected in a XIB file; an empty circle means it still needs connecting.

Figure 12.7 Source file connection status

In the takePicture: method, you will instantiate a UIImagePickerController and present it on the screen. When creating an instance of UIImagePickerController, you must set its sourceType property and assign it a delegate.

The sourceType is a constant that tells the image picker where to get images. It has three possible values:

UIImagePickerControllerSourceTypeCamera – The user will take a new picture.

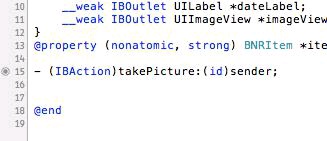
UIImagePickerControllerSourceTypePhotoLibrary

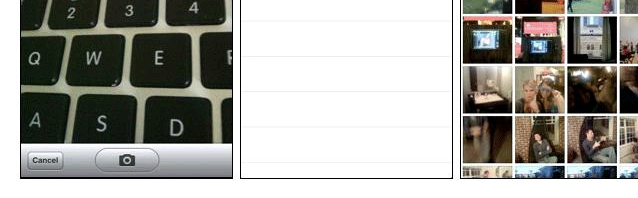
– The user will be prompted to select an album and then a photo from that album.

UIImagePickerControllerSourceTypeSavedPhotosAlbum

– The user picks from the most recently taken photos.

Figure 12.8 shows the results of using each constant. Figure 12.8 UIImagePickerControllerTypes



The first source type, UIImagePickerControllerSourceTypeCamera, won’t work on a device that doesn’t have a camera. So before using this type, you have to check for a camera by sending the message isSourceTypeAvailable: to the UIImagePickerController class. Sending this message to UIImagePickerController with one of the source type constants returns a Boolean value for whether the device supports that source type.

In addition to a source type, the UIImagePickerController instance needs a delegate to handle requests from its view. When the user taps the Use Photo button on the UIImagePickerController’s interface, the delegate is sent the message imagePickerController:didFinishPickingMediaWithInfo:. (The delegate receives another message – imagePickerControllerDidCancel: – if the process was cancelled.)

In DetailViewController.m, add the following code to takePicture:. (Remember – there’s already a stub for this method, so locate the stub in DetailViewController.m and add the following code there.)

- (IBAction)takePicture:(id)sender {

UIImagePickerController \*imagePicker = [[UIImagePickerController alloc] init];

// If our device has a camera, we want to take a picture, otherwise, we

// just pick from photo library if ([UIImagePickerController

isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {

[imagePicker setSourceType:UIImagePickerControllerSourceTypeCamera];

setSourceType:UIImagePickerControllerSourceTypeCamera];

} else { [imagePicker

setSourceType:UIImagePickerControllerSourceTypePhotoLibrary]; }

// This line of code will generate a warning right now, ignore it

[imagePicker setDelegate:self];

}

Once the UIImagePickerController has a source type and a delegate, it’s time to put its view on the screen. Unlike other UIViewController subclasses you’ve used, an instance of UIImagePickerController is presented modally. When a view controller is modal, it takes over the entire screen until it has finished its work.

To present a view modally, presentViewController:animated:completion: is sent to the UIViewController whose view is on the screen. The view controller to be presented is passed to it, and its view slides up from the bottom of the screen.

In DetailViewController.m, add code to the end of takePicture: to present the UIImagePickerController.

[imagePicker setDelegate:self];

// Place image picker on the screen

[self presentViewController:imagePicker animated:YES completion:nil];

}

(Don’t worry about the third argument, completion:. We’ll talk about it more in Chapter 13.)

You can build and run the application now. Select a BNRItem to see its details and then tap the camera button on the UIToolbar. UIImagePickerController’s interface will appear on the screen (Figure 12.9), and you can take a picture (or choose an existing image if you don’t have a camera). Tapping the Use Photo button dismisses the UIImagePickerController.

(If you are working on the simulator, there won’t be any images available. However, you can open Safari in the simulator, navigate to a page with an image, and click and hold on that image. When the action sheet appears, choose Save Image. This image will be saved in the simulator’s photo library. But, the simulator can be flaky, so you might have to try a few different images before one actually saves.)

images before one actually saves.)

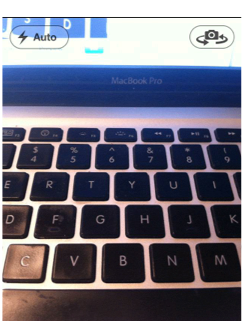


Figure12.9 UIImagePickerControllerpreviewinterface



But, oops – you dismissed the controller without keeping a reference to the image anywhere in the code. To hold on to the selected image, you need to implement the delegate method imagePickerController:didFinishPickingMediaWithInfo: in DetailViewController.

But before you implement this method, let’s take care of the two warnings that appeared during the last build telling you that DetailViewController does not conform to the UIImagePickerControllerDelegate or the UINavigationControllerDelegate protocol. In DetailViewController.h, add the protocols to the class declaration. (Why UINavigationControllerDelegate? UIImagePickerController is a subclass of UINavigationController.)

@interface DetailViewController : UIViewController

<UINavigationControllerDelegate, UIImagePickerControllerDelegate> {

That’s better. Notice that the two warnings have gone away.

The

imagePickerController:didFinishPickingMediaWithInfo:

message will be sent to the image picker’s delegate when a photo is selected. In DetailViewController.m, implement this method to put the image into the UIImageView that you created earlier.

- (void)imagePickerController:(UIImagePickerController \*)picker didFinishPickingMediaWithInfo:(NSDictionary \*)info {

// Get picked image from info dictionary

UIImage \*image = [info objectForKey:UIImagePickerControllerOriginalImage];

// Put that image onto the screen in our image view [imageView setImage:image];

// Take image picker off the screen - // you must call this dismiss method [self dismissViewControllerAnimated:YES completion:nil];

}

}

Build and run the application again. Take a photo, the image picker is dismissed, and you are returned to the DetailViewController’s view. Do you see your image? Oddly enough, you might or you might not. Let’s figure out what’s going on and fix the problem.

When a photo is taken, that image is loaded into memory. However, the image file is so large that it causes a low-memory warning. Recall that a low-memory warning gives the system the option of requiring view controllers to release their views if they are not currently visible. When a modal view controller is on the screen, its view is visible, and the view of the view controller that presented it is not. In our case, the low-memory warning destroys DetailViewController’s view, and the imageView is no longer available when we try to set it.

To get around this problem, we must create a separate store for images. Instead of putting the image directly into the imageView, we will put it into this store. Then when the DetailViewController’s view next appears on screen, we’ll have the DetailViewController grab the image from the image store and put it into its own imageView. In general, this is a best practice: a view controller should re-populate its view’s subviews with data whenever it is sent the message viewWillAppear: to eliminate the possibility that a low- memory warning could wipe out its content.

Creating BNRImageStore

The image store will hold all the pictures the user will take. In Chapter 14, you will have the BNRItem objects write out their instance variables to a file, which will then be read in when the application starts. However, as we’ve seen, images tend to be very large, so it’s a good idea to keep them separate from other data. The image store will fetch and cache the images as they are needed. It will also be able to flush the cache if the device runs low on memory. Create a new NSObject subclass called BNRImageStore. Open BNRImageStore.h and create its interface:

#import <UIKit/UIKit.h> @interface BNRImageStore : NSObject



{

{ NSMutableDictionary \*dictionary;

} + (BNRImageStore \*)sharedStore;

-  (void)setImage:(UIImage \*)i forKey:(NSString \*)s;

-  (UIImage \*)imageForKey:(NSString \*)s;

-  (void)deleteImageForKey:(NSString \*)s;  @end  Like the BNRItemStore, the BNRImageStore needs to be a singleton. In BNRImageStore.m, write the following code to ensure BNRImageStore’s singleton status. @implementation BNRImageStore

+  (id)allocWithZone:(NSZone \*)zone {  return [self sharedStore]; }

+  (BNRImageStore \*)sharedStore {  static BNRImageStore \*sharedStore = nil; if (!sharedStore) {  // Create the singleton  sharedStore = [[super allocWithZone:NULL] init]; }

return

- (id)init {

sharedStore;

}

self = if (self) {

[super init];

dictionary = [[NSMutableDictionary alloc] init]; }

return self; }

Then, implement the other three methods declared in the header

file.

-  (void)setImage:(UIImage \*)i forKey:(NSString \*)s {  [dictionary setObject:i forKey:s]; }

-  (UIImage \*)imageForKey:(NSString \*)s {  return [dictionary objectForKey:s]; }

-  (void)deleteImageForKey:(NSString \*)s {  if (!s) return;  [dictionary removeObjectForKey:s]; }

NSDictionary

Notice that the dictionary is an instance of NSMutableDictionary. A dictionary is a lot like an array: it is a collection object and it has an immutable (NSDictionary) and mutable version (NSMutableDictionary). However, dictionaries and arrays differ in how they store their objects. An array is an ordered list of pointers to objects that is accessed by an index. When you have an array, you can ask it for the object at the nth index:

// Put some object at the beginning of an array [someArray insertObject:someObject atIndex:0];

// Get that same object out someObject = [someArray objectAtIndex:0];

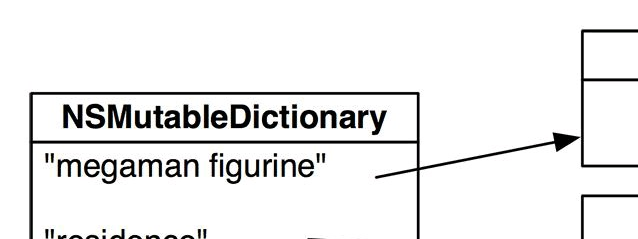
A dictionary’s objects are not ordered within the collection. So instead of accessing entries with an index, you use a key. The key is usually an instance of NSString.

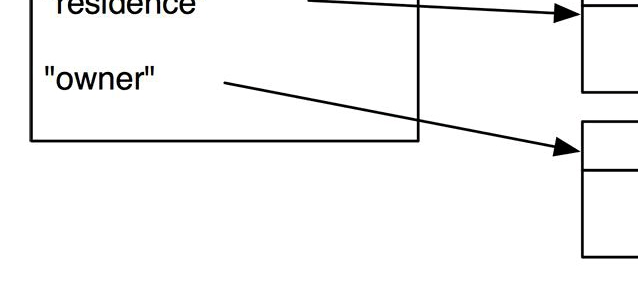
// Add some object to a dictionary for the key "MyKey" [someDictionary setObject:someObject forKey:@"MyKey"];

// Get that same object out someObject = [someDictionary objectForKey:@"MyKey"];

We call each entry in a dictionary a key-value pair. The value is the object being stored in the collection, and the key is a unique value (usually a string) that you use to retrieve the value later. (In other development environments, a dictionary is called a hash map or hash table, but we still use the term key-value pair to talk about the information they store.)

Figure 12.10 NSDictionary diagram





The purpose of a dictionary is difficult to understand if you are new to programming, but the general idea is that a dictionary maps one value to another (Figure 12.10). Consider a robot that responded to the facial expressions of a human. For example, if a human smiled, the robot would wink. Thus, the human-to- robot response would be stored in a dictionary. The “wink” response would be the value stored for the “smile” key. When a human made another facial expression at the robot, it would look up that expression in its dictionary and find the appropriate response.

Consider solving the same human-to-robot response problem with an array as the collection object. One approach would be to store every response in an ordered array. With this approach, you would have to remember the index of every response by assigning a numeric value to each facial expression. If you added a new expression-response pair, you’d have to recompute the indices of each response. At some point early in your programming career, you’ve probably done something like that:

int smileIndex = 0; int scowlIndex = 1; RobotResponses responses[2] = { "wink", "scream"};

if ([human isSmiling]) response = responses[smileIndex];

else if ([human isScowling]) response = responses[scowlIndex];

Another approach would be to store an object that held both the facial expression and the response in an array. When the human made a facial expression, you would search the list for the



appropriate response.

appropriate response.

for (ExpressionResponsePair \*p in allResponses) { if ([p expression] == [human facialExpression])

response = [p response];

}

Both of these approaches are inefficient and clumsy. A dictionary

makes this lookup process a lot faster and easier to understand:

NSMutableDictionary \*dictionary = [NSMutableDictionary dictionary]; [dictionary setValue:@"wink" forKey:@"smile"];

response = [dictionary objectForKey:[human facialExpression]];

When using a dictionary, there can only be one object for each key. If you add an object to a dictionary with a key that matches the key of an object already present in the dictionary, the earlier object is removed. If you need to store multiple objects under one key, you can put them in an array and add the array to the dictionary as the value.

Finally, note that a dictionary’s memory management is like that of an array. Whenever you add an object to a dictionary, the dictionary owns it, and whenever you remove an object from a dictionary, the dictionary releases its ownership.

Creating and using keys

When an image is added to the store, it will be put into a dictionary under a unique key, and the associated BNRItem object will be given that key. When the DetailViewController wants an image from the store, it will ask its item for the key and search the dictionary for the image. Add a property to BNRItem.h to store the key.

@property (nonatomic, readonly, strong) NSDate \*dateCreated;

@property (nonatomic, copy) NSString \*imageKey;

Synthesize this new property in the implementation file.

@implementation BNRItem

@synthesize imageKey;

The image keys need to be unique in order for your dictionary to work. While there are many ways to hack together a unique string, we’re going to use the Cocoa Touch mechanism for creating universally unique identifiers (UUIDs), also known as globally unique identifiers (GUIDs). Objects of type CFUUIDRef

globally unique identifiers (GUIDs). Objects of type

CFUUIDRef

represent a UUID and are generated using the time, a counter, and a hardware identifier, which is usually the MAC address of the ethernet card.

Import BNRImageStore.h at the top of DetailViewController.m.

#import "DetailViewController.h" #import "BNRItem.h" #import "BNRImageStore.h"

In DetailViewController.m, update

imagePickerController:didFinishPickingMediaWithInfo:

to generate a UUID when a new picture is taken.

- (void)imagePickerController:(UIImagePickerController \*)picker didFinishPickingMediaWithInfo:(NSDictionary \*)info

{ UIImage \*image = [info

objectForKey:UIImagePickerControllerOriginalImage];

// Create a CFUUID object - it knows how to create unique identifier strings

CFUUIDRef newUniqueID = CFUUIDCreate(kCFAllocatorDefault);

The prefix CF means CFUUIDRef comes from the Core Foundation framework (and remember that the Ref suffix means that it is a pointer). Core Foundation is a collection of C “classes” and functions. Core Foundation objects are created by calling a function that begins with the type of object being created and contains the word Create (CFUUIDCreate). When creating a Core Foundation object, the first argument specifies how memory is allocated. In practice, you pass kCFAllocatorDefault and let the system make that choice.

Once created, a CFUUIDRef is just an array of bytes and, if represented as a string, will look something like this: 28153B74-4D6A-12F6-9D61-155EA4C32167

This UUID will be used in two ways: it will be the key in the BNRImageStore’s dictionary and in a later chapter, it will be the name of the image file on the filesystem. Because keys in a dictionary and paths on the filesystem are typically strings, we want to represent the UUID as a string instead of an array of bytes.

You can create a string object from a CFUUIDRef by calling the C function CFUUIDCreateString. In DetailViewController.m, add the following line of code in

imagePickerController:didFinishPickingMediaWithInfo:.

imagePickerController:didFinishPickingMediaWithInfo:. - (void)imagePickerController:(UIImagePickerController \*)picker

didFinishPickingMediaWithInfo:(NSDictionary \*)info

{ UIImage \*image = [info

objectForKey:UIImagePickerControllerOriginalImage];

CFUUIDRef newUniqueID = CFUUIDCreate (kCFAllocatorDefault);

// Create a string from unique identifier CFStringRef newUniqueIDString =

CFUUIDCreateString (kCFAllocatorDefault, newUniqueID);

Notice that newUniqueIDString’s type is CFStringRef. The imageKey property of BNRItem is an NSString. Clearly, you need some way to move between CFStringRef and NSString to set the imageKey property.

Fortunately, many classes in Core Foundation are toll-free bridged with their Objective-C counterpart. For example, CFStringRef is toll-free bridged with NSString; CFArrayRef with NSArray. Instances of classes that are toll-free bridged look exactly the same as their counterpart in memory. Therefore, you can use a simple C-style typecast to treat a toll-free bridged Core Foundation object as an Objective-C object.

Typecast newUniqueIDString and set it as the imageKey of the selected BNRItem in imagePickerController:didFinishPickingMediaWithInfo:. Also, place this image in the BNRImageStore.

- (void)imagePickerController:(UIImagePickerController \*)picker didFinishPickingMediaWithInfo:(NSDictionary \*)info

{ UIImage \*image = [info

objectForKey:UIImagePickerControllerOriginalImage];

CFUUIDRef newUniqueID = CFUUIDCreate (kCFAllocatorDefault);

CFStringRef newUniqueIDString = CFUUIDCreateString (kCFAllocatorDefault,

newUniqueID);

// Use that unique ID to set our item's imageKey NSString \*key = (\_\_bridge NSString \*)newUniqueIDString; [item setImageKey:key];

// Store image in the BNRImageStore with this key [[BNRImageStore sharedStore] setImage:image

forKey:[item imageKey]];

Notice the use of \_\_bridge in this typecast. To understand what this keyword does, you must understand how memory is managed for Core Foundation objects.

Core Foundation and toll-free bridging

When a variable that points to an Objective-C object is destroyed, ARC knows that object has lost an owner. ARC doesn’t do this with Core Foundation objects. Thus, when a Core Foundation object loses a pointer, you must call a function that tells the object to lose an owner before you lose the pointer. This function is CFRelease.

If you do not call CFRelease before losing a pointer, the pointed-to object still thinks it has an owner. Losing a pointer to an object before telling it to lose an owner results in a memory leak: you can no longer access that object, and it still has an owner. Add code to imagePickerController:didFinishPickingMediaWithInfo: to tell the objects pointed to by newUniqueIDString and newUniqueID to lose an owner since these are both local variables that will be destroyed when this method ends.

[[BNRImageStore sharedStore] setImage:image forKey:[item imageKey]];

CFRelease(newUniqueIDString); CFRelease(newUniqueID);

[imageView setImage:image];

[self dismissViewControllerAnimated:YES completion:nil]; }

Here are the memory management rules when it comes to Core Foundation objects.

A variable only owns the object it points to if the function that created the object has the word Create or Copy in it.

If a pointer owns a Core Foundation object, you must call CFRelease before you lose that pointer. Remember that a pointer can be lost if it is set to point at something else (including nil) or if the pointer itself is being destroyed.

Once you call CFRelease on a pointer, you cannot access that pointer again.

As you can see, the rules of memory management are a bit more complicated when dealing with Core Foundation because you don’t have the luxury of ARC. However, you typically won’t use Core Foundation objects as much as Objective-C objects. As long as you stick to these rules, you will be okay.

Now, back to the \_\_bridge keyword. ARC doesn’t know how to manage memory with Core Foundation objects very well, so it gets confused if you typecast a Core Foundation pointer into its Objective-C counterpart. Placing \_\_bridge in front of the cast tells ARC, “Hey, don’t even worry about it.” Thus, when ARC sees this line of code, it doesn’t give ownership to the key variable as it normally would:

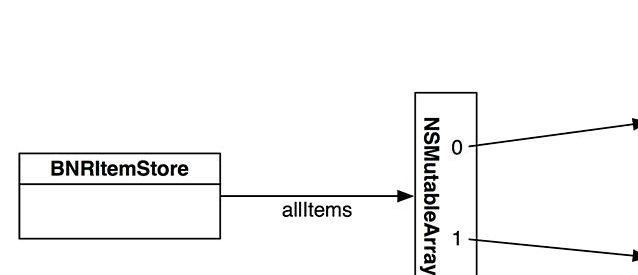
NSString \*key = (\_\_bridge NSString \*)newUniqueIDString;

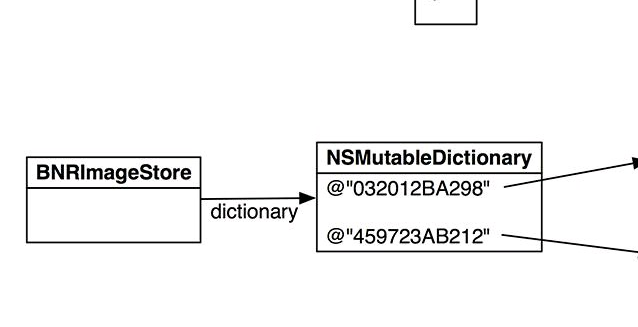
Once key is an Objective-C pointer, ARC can do its work as normal. When this object is passed to setImageKey:, BNRItem’s imageKey instance variable takes ownership of that object.

Wrapping up BNRImageStore

Now that the BNRImageStore can store images and BNRItems have a key to get that image (Figure 12.11), we need to teach DetailViewController how to grab the image for the selected BNRItem and place it in its imageView.

Figure 12.11 Cache



The DetailViewController’s view will appear at two times: when the user taps a row in ItemsViewController and when the UIImagePickerController is dismissed. In both of these situations, the imageView should be populated with the image of the BNRItem being displayed. In DetailViewController.m, add code to viewWillAppear: to do this.

- (void)viewWillAppear:(BOOL)animated {

[super viewWillAppear:animated];

[nameField setText:[item itemName]]; [serialNumberField setText:[item serialNumber]]; [valueField setText:[NSString stringWithFormat:@"%d",

[item valueInDollars]]];

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

autorelease]; [dateFormatter setDateStyle:NSDateFormatterMediumStyle];

[dateFormatter setTimeStyle:NSDateFormatterNoStyle];

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

NSString \*imageKey = [item imageKey];

if (imageKey) { // Get image for image key from image store UIImage \*imageToDisplay =

[[BNRImageStore sharedStore] imageForKey:imageKey];

// Use that image to put on the screen in imageView

[imageView setImage:imageToDisplay]; } else {

// Clear the imageView [imageView setImage:nil];

[imageView setImage:nil];

}

}

Notice that if no image exists in the image store for that key (or there is no key for that item), the pointer to the image will be nil. When the image is nil, the UIImageView just won’t display an image.

Build and run the application. Create a BNRItem and select it from the UITableView. Then, tap the camera button and take a picture. The image will appear as it should.

There is another detail to take care of: if you select a new image for a BNRItem, the old one will still be in the BNRImageStore.

At the top of

imagePickerController:didFinishPickingMediaWithInfo:

in DetailViewController.m, add some code to tell the

BNRImageStore to remove the old image.

- (void)imagePickerController:(UIImagePickerController \*)picker didFinishPickingMediaWithInfo:(NSDictionary \*)info

{

NSString \*oldKey = [item imageKey];

// Did the item already have an image? if (oldKey) {

// Delete the old image

[[BNRImageStore sharedStore] deleteImageForKey:oldKey]; }

UIImage \*image = [info objectForKey:UIImagePickerControllerOriginalImage];

Build and run the application again. The behavior should remain the same, but the memory benefits are significant.

Dismissing the keyboard

When the keyboard appears on the screen in the item detail view, it obscures DetailViewController’s imageView. This is annoying when you’re trying to see an image, so you’re going to implement the delegate method textFieldShouldReturn: to have the text field resign its first responder status to dismiss the keyboard when the return key is tapped. (This is why you hooked up the delegate outlets earlier.) But first, in DetailViewController.h, have DetailViewController conform to the UITextFieldDelegate protocol.

@interface DetailViewController : UIViewController <UINavigationControllerDelegate,

UIImagePickerControllerDelegate,

UIImagePickerControllerDelegate, UITextFieldDelegate>

In DetailViewController.m, implement textFieldShouldReturn:.

- (BOOL)textFieldShouldReturn:(UITextField \*)textField {

[textField resignFirstResponder];

return YES; }

It would be stylish to also dismiss the keyboard if the user taps anywhere else on DetailViewController’s view. We can dismiss the keyboard by sending the view the message endEditing:, which will cause the text field (as a subview of the view) to resign as first responder. Now let’s figure out how to get the view to send a message when tapped.

We have seen how classes like UIButton can send an action message to a target when tapped. Buttons inherit this target- action behavior from their superclass, UIControl. You’re going to change the view of DetailViewController from an instance of UIView to an instance of UIControl so that it can handle touch events.

In DetailViewController.xib, select the main View object. Open the identity inspector and change the view’s class to UIControl (Figure 12.12).

Figure 12.12 Changing the class of DetailViewController’s view

Then, open DetailViewController.h in the assistant editor. Control-drag from the view (now a UIControl) to the method declaration area of DetailViewController. When the pop- up window appears, select Action from the Connection pop-up menu. Notice that the interface of this pop-up window is slightly different than one you saw when creating and connecting the UIBarButtonItem. A UIBarButtonItem is a simplified version of UIControl – it only sends its target an action message when it is tapped. A UIControl, on the other hand, can send action messages in response to a variety of events.

Therefore, you must choose the appropriate event type to trigger

the action message being sent. In this case, you want the action

the action message being sent. In this case, you want the action message to be sent when the user taps on the view. Configure this pop-up window to appear as it does in Figure 12.13 and click Connect.

Figure 12.13 Configuring a UIControl action

This will create a stub method in DetailViewController.m.

Enter the following code into that method.

- (IBAction)backgroundTapped:(id)sender {

[[self view] endEditing:YES];

}

Build and run your application and test both ways of dismissing the keyboard.