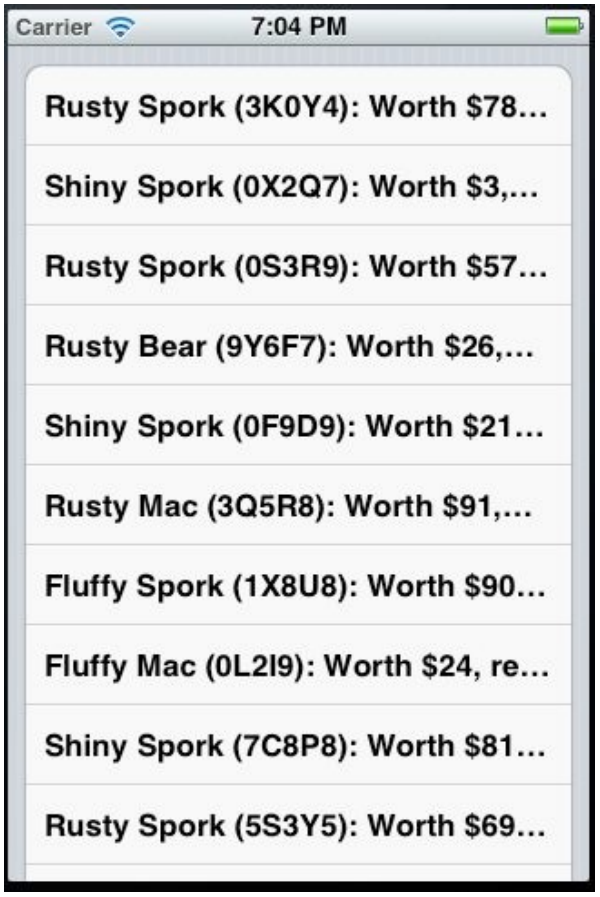
Chapter 8
UITableView and UITableViewController

- The homeowner application
- UITableViewController
- UITableView Data Source
- UITableViewCell

The homeowner application

- The first phase of the homeowner application is a table listing of the items that the homeowner owns
- The items are generated randomly as instances of INIItem.

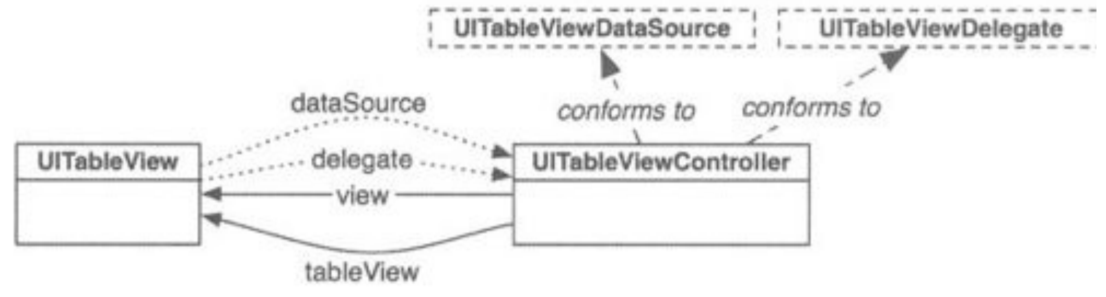


Rusty Spork (3K0Y4): Worth $78...
Shiny Spork (0X2Q7): Worth $3,...
Rusty Spork (0S3R9): Worth $57...
Rusty Bear (9Y6F7): Worth $26,...
Shiny Spork (0F9D9): Worth $21...
Rusty Mac (3Q5R8): Worth $91,...
Fluffy Spork (1X8U8): Worth $90...
Fluffy Mac (0L2I9): Worth $24, re...
Shiny Spork (7C8P8): Worth $81...
Rusty Spork (5S3Y5): Worth $69...

UITableView and UITableViewController

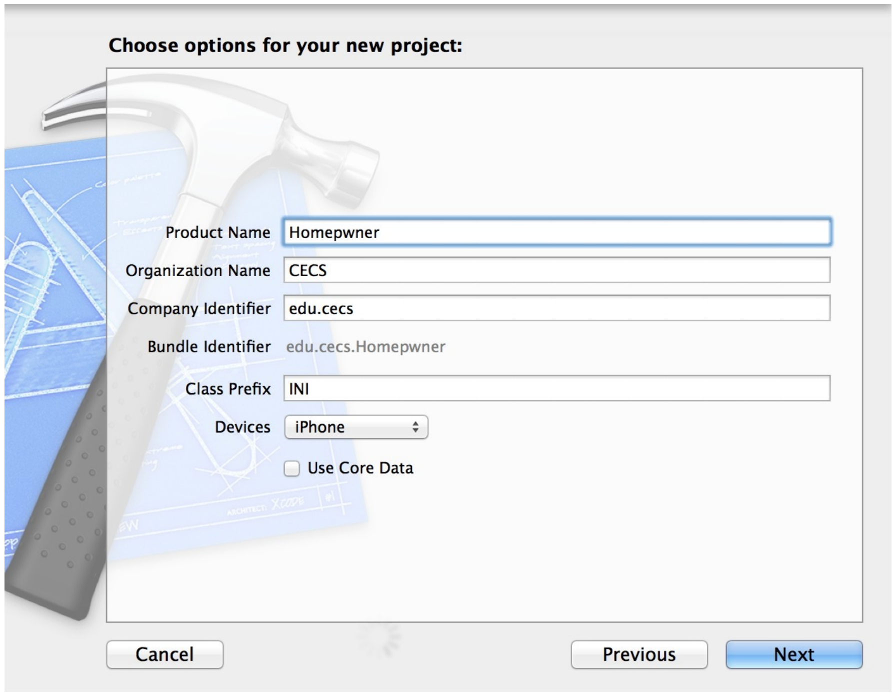
- UITableView is a view object that simply knows how to draw itself but nothing else, i.e. it does not handle logic nor data
- UItableView needs a "`UITableViewController`" to handle its appearance on the screen.
- UITableView needs a data source as follows:
  - The data source knows how many entries there are to display.
  - The data source will supply the data so that it can be displayed in the rows of the view.
  - The data source can be any Objective-C that conforms to the protocol "`UITableViewDataSource`"
- UITableView needs a delegate to react to various events involving UITableView and inform other objects when these events are generated. Thus, it needs a "`UITableViewDelegate`"
- It is possible for an instance of `UITableViewControlle` to fulfill these three roles

UITableViewController is a subclass of UIViewController
UITableViewController has a view which is an instance of UITableView
When a UITableViewController creates its view it sets the data source and the delegate to point back to itself



Creating Homeowner

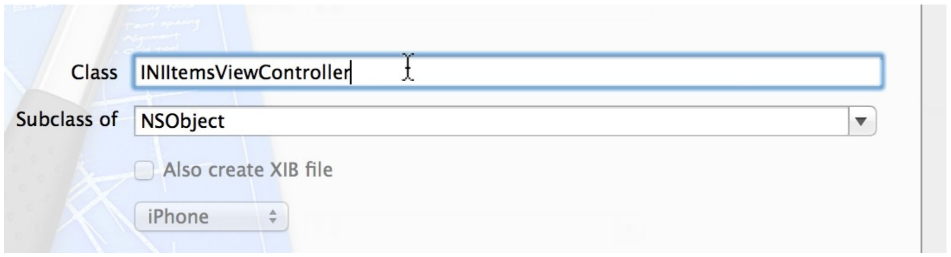
- Create an empty iOS application project and call it Homeowner or Homepwner, whichever you prefer.
- Create an instance of UITableViewController call it XXXItemsViewController (Start with NSObject)



Choose options for your new project:

Product Name	Homepwner
Organization Name	CECS
Company Identifier	edu.cecs
Bundle Identifier	edu.cecs.Homepwner
Class Prefix	INI
Devices	iPhone

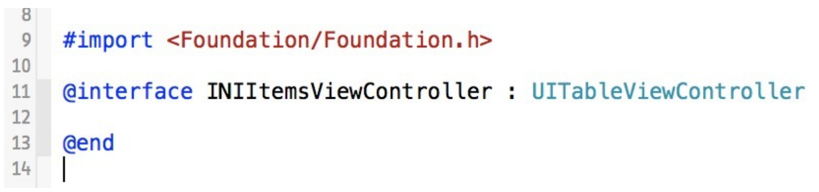
☐ Use Core Data



Class	INIItemsViewController
Subclass of	NSObject

☐ Also create XIB file

iPhone



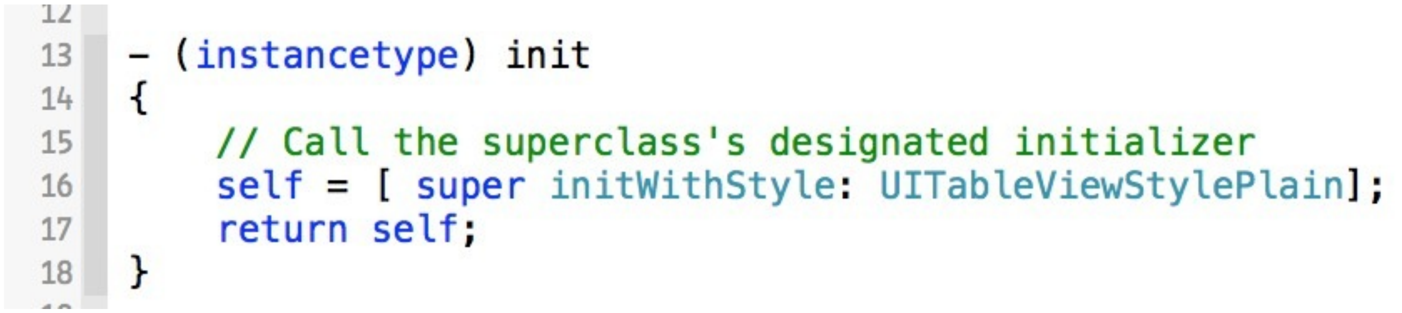
```
#import <Foundation/Foundation.h>

@interface INIItemsViewController : UITableViewController

@end
```

Initializing INIItemsView

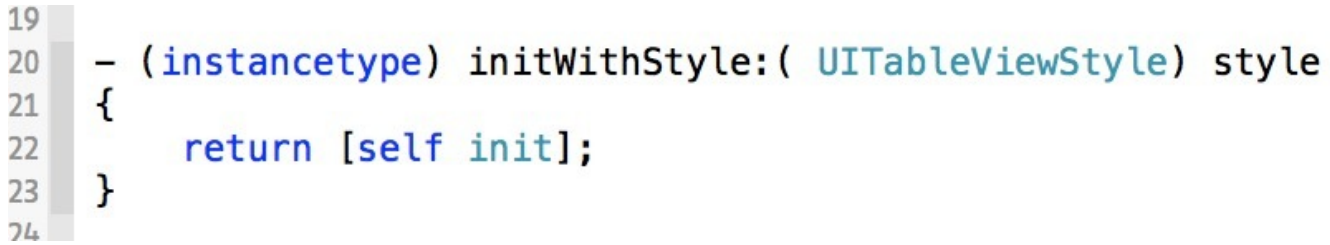
- The designated initializer of UITableViewController is initWithStyle:, which takes a constant that determines the style of the table view.
- There are two options:
    - UITableViewStylePlain and
    - UITableViewStyleGrouped.
- These looked quite different on iOS 6, but the differences are quite minor as of iOS 7.
- I want to ensure that all instances of INIItemsViewController use the UITableViewStylePlain style, no matter what initialization message is sent to them.
- You are changing the designated initializer to init. As such, you need to follow the two rules of initializers:
    - Call the superclass's designated initializer from yours
    - Override the superclass's designated initializer to call yours

Initializing INIItemsView (Cont.)

- You are changing the designated initializer to init. As such, you need to follow the two rules of initializers:
  - Call the superclass's designated initializer from yours
  - Override the superclass's designated initializer to call yours

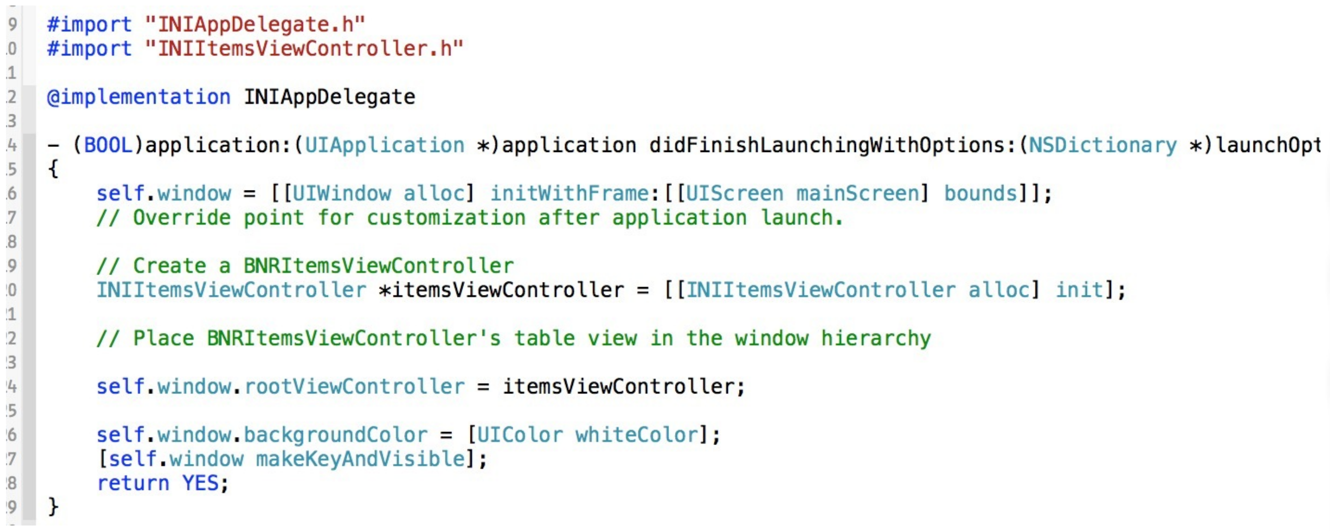


```
- (instancetype) init
{
    // Call the superclass's designated initializer
    self = [ super initWithStyle: UITableViewStylePlain];
    return self;
}
```

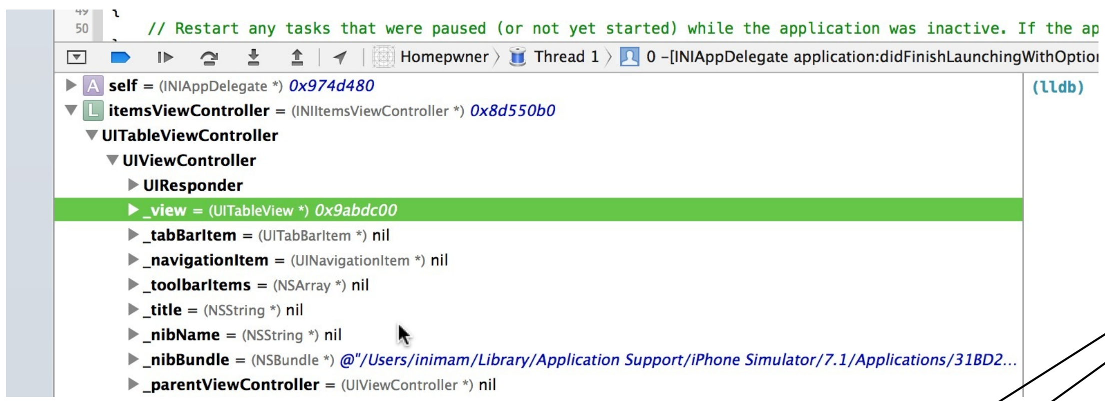


```
- (instancetype) initWithStyle:( UITableViewStyle) style
{
    return [self init];
}
```

Setting Root Controller

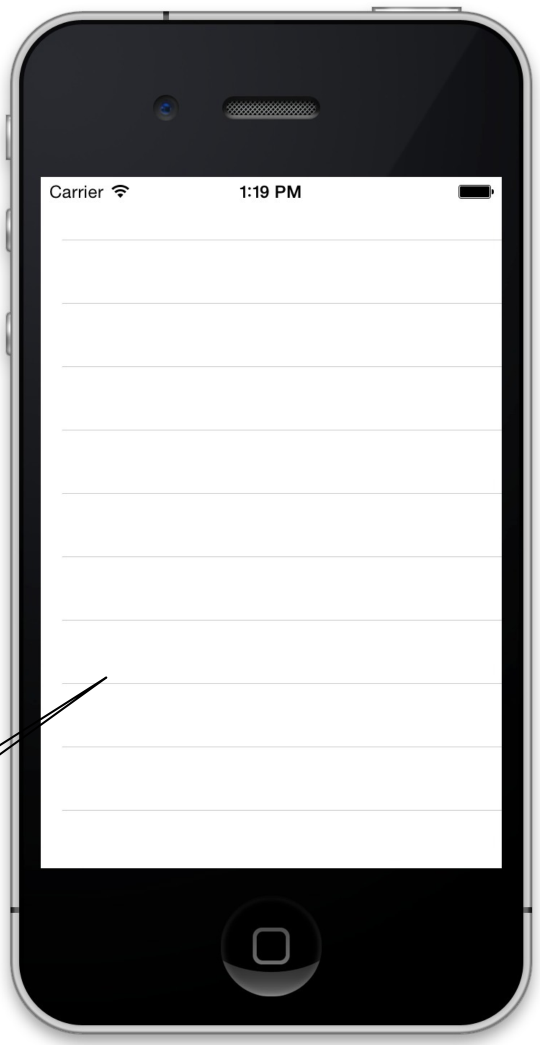


```
#import "INIAppDelegate.h"
#import "INIItemsViewController.h"

@implementation INIAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOpt
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.

    // Create a BNRItemsViewController
    INIItemsViewController *itemsViewController = [[INIItemsViewController alloc] init];

    // Place BNRItemsViewController's table view in the window hierarchy

    self.window.rootViewController = itemsViewController;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```



```
// Restart any tasks that were paused (or not yet started) while the application was inactive. If the ap
```

```
▶ self = (INIAppDelegate *) 0x974d480
▼ itemsViewController = (INIItemsViewController *) 0x8d550b0
  ▼ UITableViewController
    ▼ UIViewController
      ▶ UIResponder
      ▶ _view = (UITableView *) 0x9abdc00
      ▶ _tabBarItem = (UITabBarItem *) nil
      ▶ _navigationItem = (UINavigationItem *) nil
      ▶ _toolbarItems = (NSArray *) nil
      ▶ _title = (NSString *) nil
      ▶ _nibName = (NSString *) nil
      ▶ _nibBundle = (NSBundle *) @"/Users/inimam/Library/Application Support/iPhone Simulator/7.1/Applications/31BD2...
      ▶ _parentViewController = (UIViewController *) nil
```

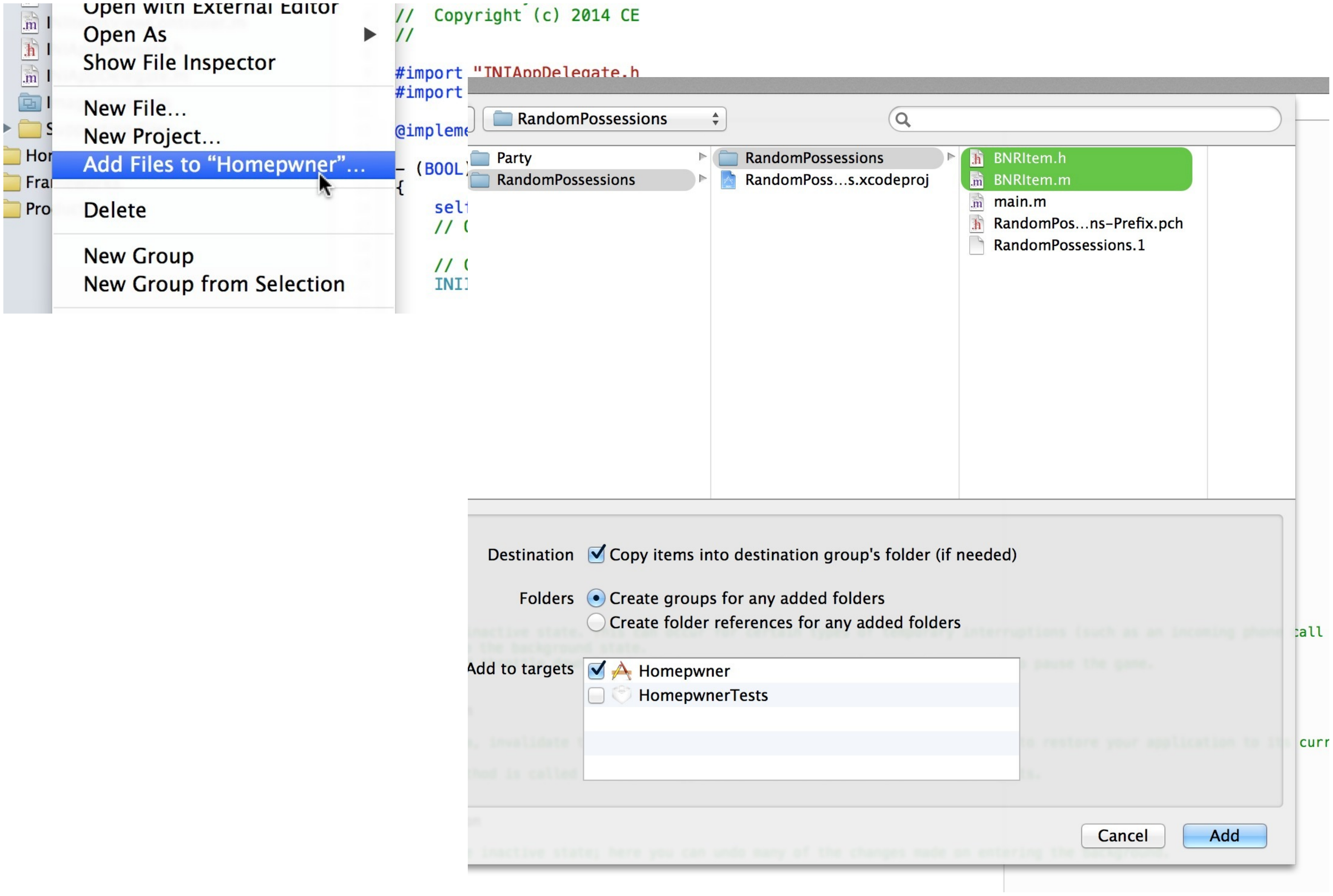


Empty Table, No datasource

UITableView’s Data Source

- The process of providing a UITableView with rows in Cocoa Touch is different from the typical procedural programming task.
- In a procedural design, you tell the table view what it should display.
- In Cocoa Touch, the table view asks another object – its dataSource – what it should display.
- In our case, the INIItemsViewController is the data source, so it needs a way to store item data.

Getting INIItems



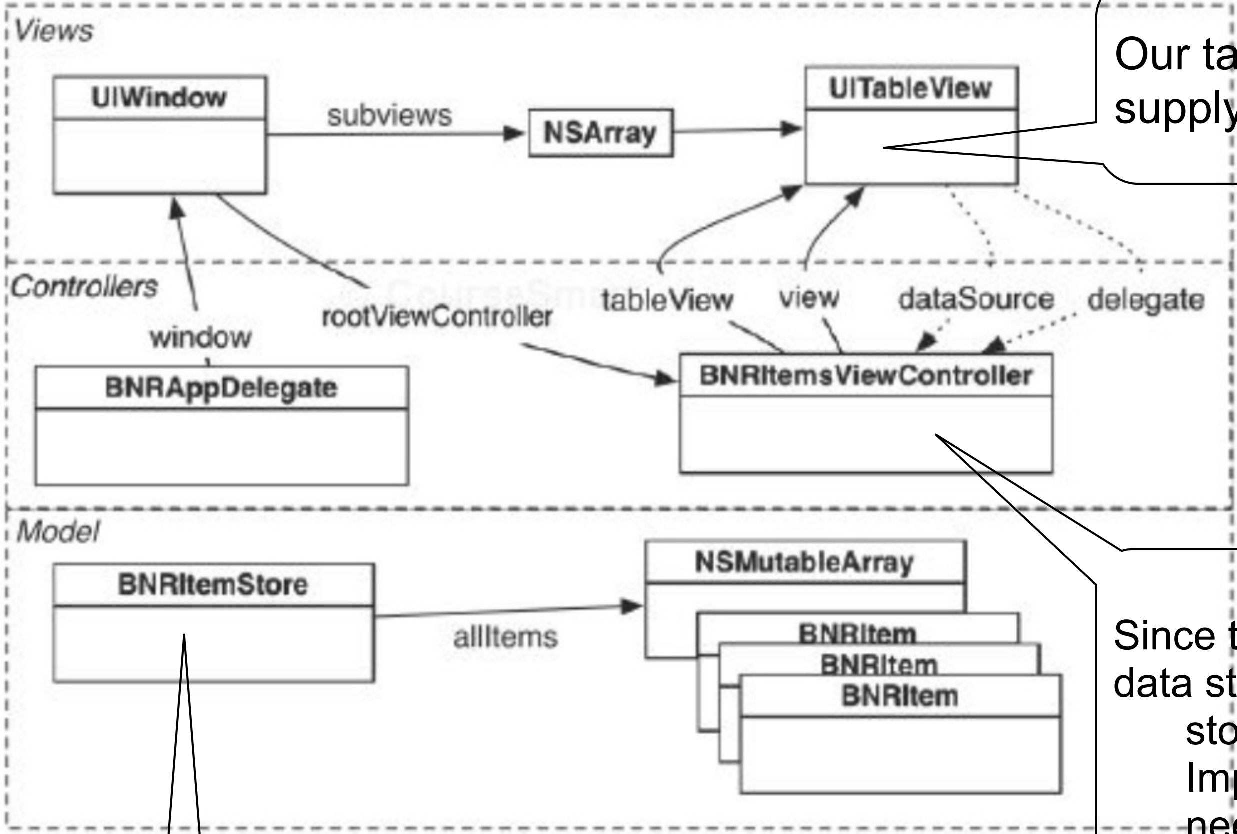
```
//  Copyright (c) 2014 CE
//

#import "INIAppDelegate.h
#import
```

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

Folders [● Create groups for any added folders] ○ Create folder references for any added folders

Add to targets [✓ Homepwner] ☐ HomepwnerTests

UITableView Data Source

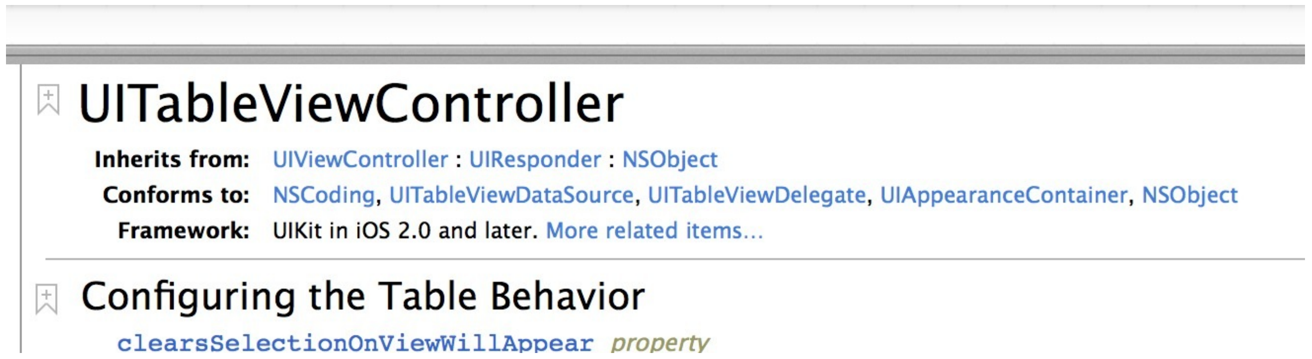


Our table view will ask the dataStore to supply it with the rows to be displayed

Since this controller performs the role of the data store, it needs to
- store the items to be displayed
- Implement the two required methods needed to conform to "UITableViewDataSource" protocol namel "tableView:numberOfRowsInSection:" anc "tableView:cellForRowAtIndexPath"

INIItemStore is an abstraction of the array that holds the individual items. It will perform additions tasks such as storing and retrieving on and from disk

UITableView Data Source Required Methods

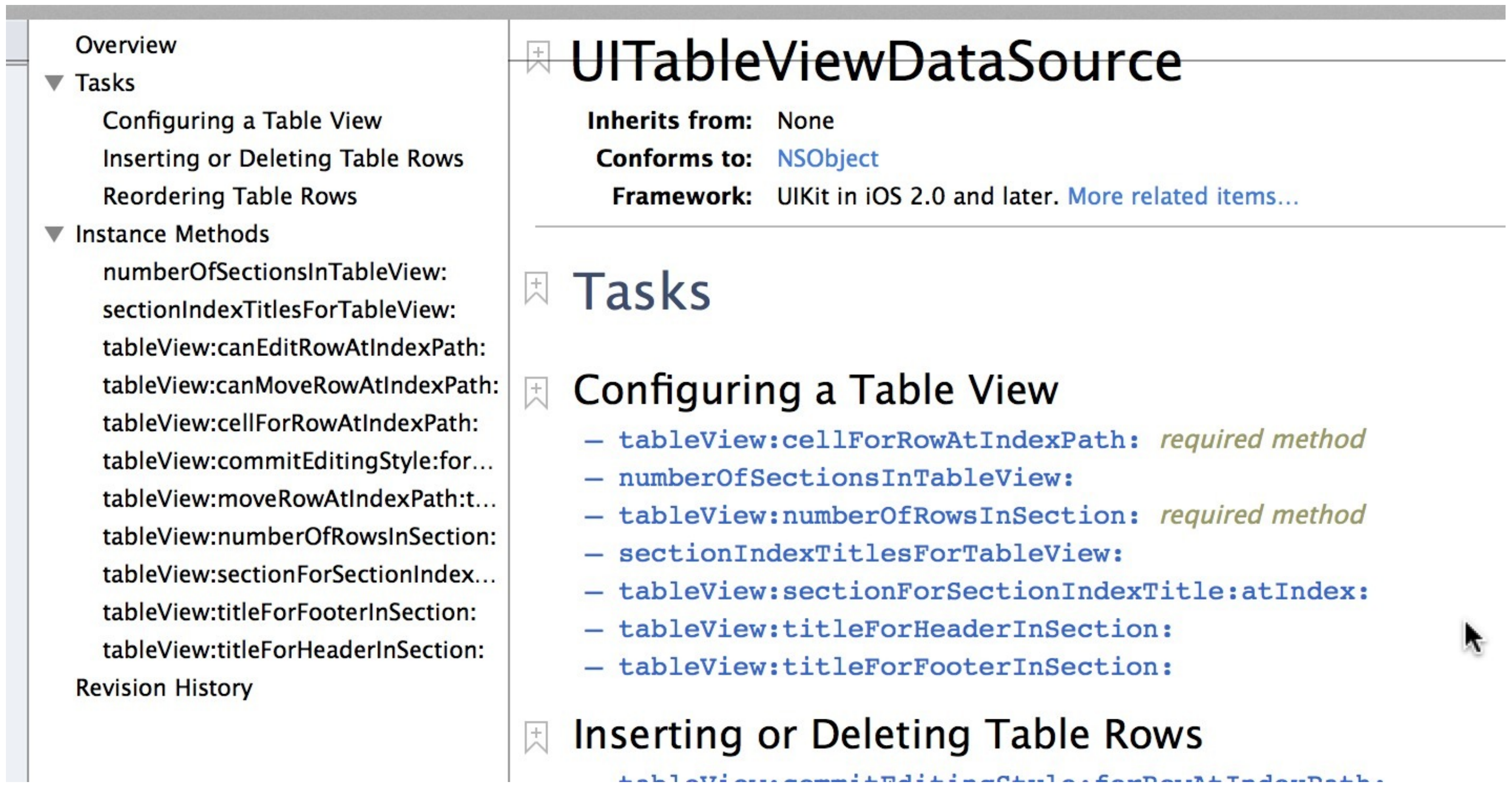


UITableViewController

Inherits from: UIViewController : UIResponder : NSObject
Conforms to: NSCoding, UITableViewDataSource, UITableViewDelegate, UIAppearanceContainer, NSObject
Framework: UIKit in iOS 2.0 and later. More related items…

Configuring the Table Behavior

`clearsSelectionOnViewWillAppear` *property*



Overview
Tasks
Configuring a Table View
Inserting or Deleting Table Rows
Reordering Table Rows
Instance Methods
numberOfSectionsInTableView:
sectionIndexTitlesForTableView:
tableView:canEditRowAtIndexPath:
tableView:canMoveRowAtIndexPath:
tableView:cellForRowAtIndexPath:
tableView:commitEditingStyle:for…
tableView:moveRowAtIndexPath:t…
tableView:numberOfRowsInSection:
tableView:sectionForSectionIndex…
tableView:titleForFooterInSection:
tableView:titleForHeaderInSection:
Revision History

UITableViewDataSource

Inherits from: None
Conforms to: NSObject
Framework: UIKit in iOS 2.0 and later. More related items…

Tasks

Configuring a Table View

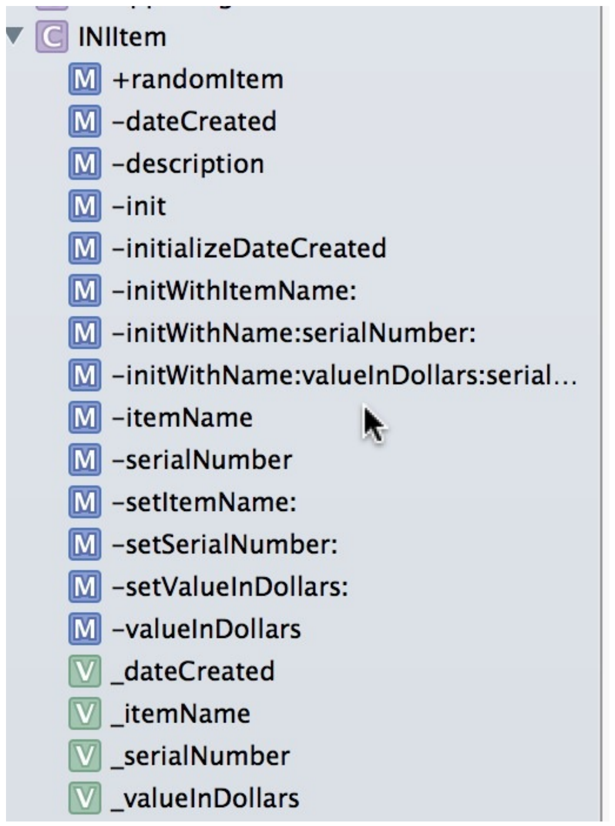
– `tableView:cellForRowAtIndexPath:` *required method*
– `numberOfSectionsInTableView:`
– `tableView:numberOfRowsInSection:` *required method*
– `sectionIndexTitlesForTableView:`
– `tableView:sectionForSectionIndexTitle:atIndex:`
– `tableView:titleForHeaderInSection:`
– `tableView:titleForFooterInSection:`

Inserting or Deleting Table Rows

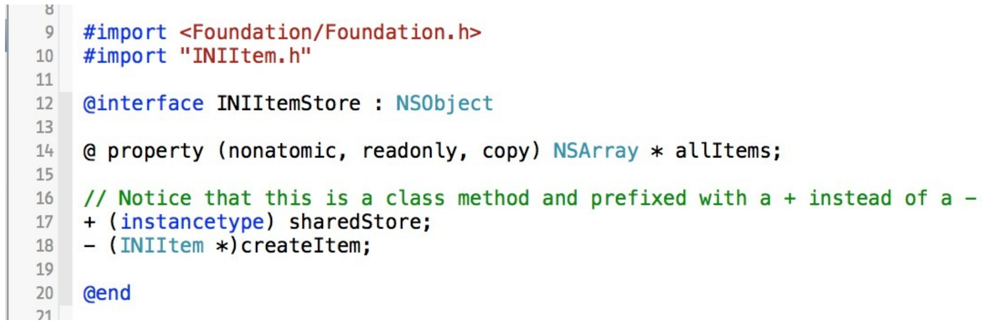
Creating the INIItemStore

- Recall: A singleton is a design pattern that
  - restricts the number of instances that can be created in a single application to one instance.
  - is useful when you have one object needed by more than one object but you still want the object to be encapsulated and protected.
- You must read the excellent discussion at:
  - http://sourcemaking.com/design_patterns/singleton
- INIItemStore will be a singleton:
  - It will have one static variable called “shareStore”.
  - Since it is static it will not be on the stack.
  - Will never get destroyed
  - It is private in the sense that it cannot be accessed or altered by any other object.

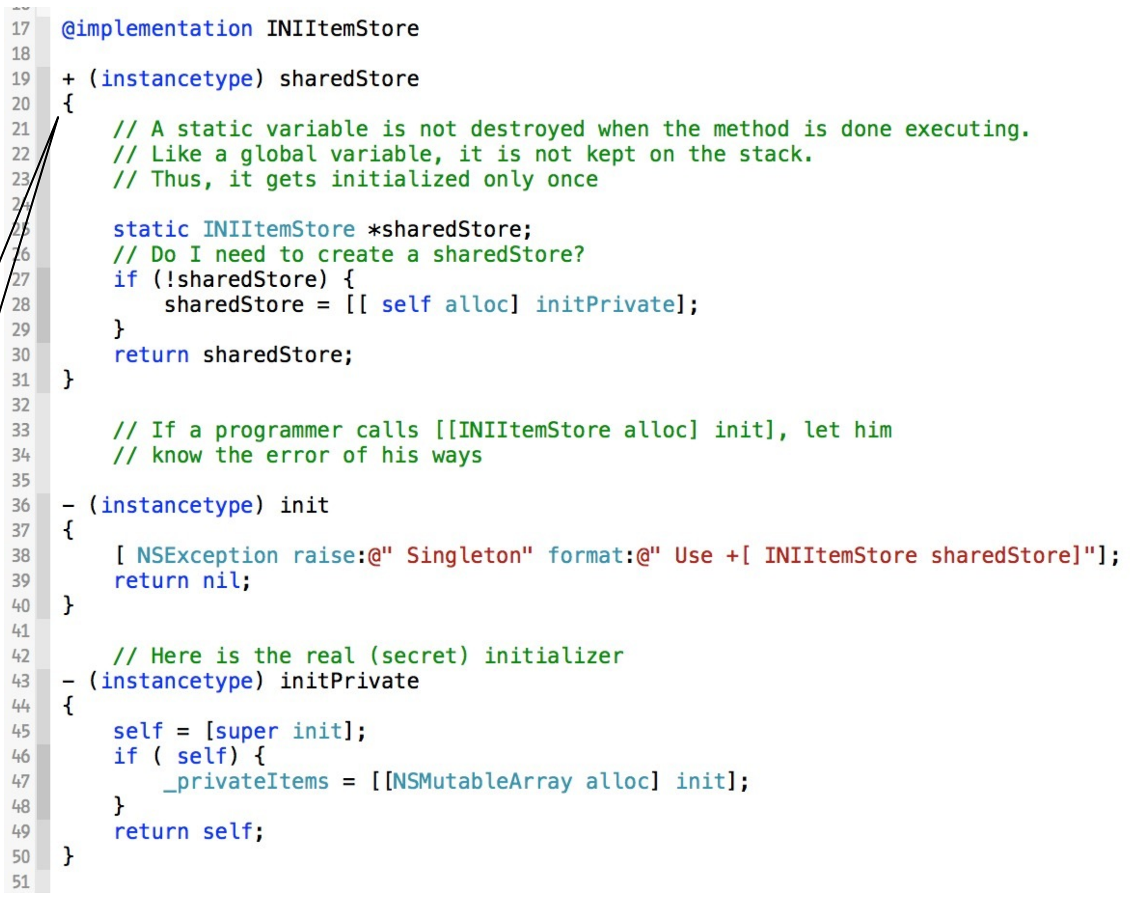
INIItem and INIItemStore



```
▼ C INIItem
    M +randomItem
    M -dateCreated
    M -description
    M -init
    M -initializeDateCreated
    M -initWithItemName:
    M -initWithName:serialNumber:
    M -initWithName:valueInDollars:serial...
    M -itemName
    M -serialNumber
    M -setItemName:
    M -setSerialNumber:
    M -setValueInDollars:
    M -valueInDollars
    V _dateCreated
    V _itemName
    V _serialNumber
    V _valueInDollars
```



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

@interface INIItemStore : NSObject

@ property (nonatomic, readonly, copy) NSArray * allItems;

// Notice that this is a class method and prefixed with a + instead of a -
+ (instancetype) sharedStore;
- (INIItem *)createItem;

@end
```

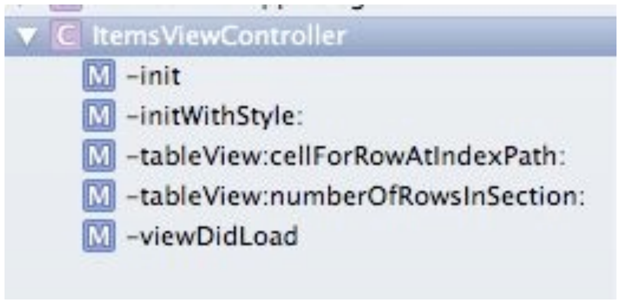


```
@implementation INIItemStore

+ (instancetype) sharedStore
{
    // A static variable is not destroyed when the method is done executing.
    // Like a global variable, it is not kept on the stack.
    // Thus, it gets initialized only once

    static INIItemStore *sharedStore;
    // Do I need to create a sharedStore?
    if (!sharedStore) {
        sharedStore = [[ self alloc] initPrivate];
    }
    return sharedStore;
}

    // If a programmer calls [[INIItemStore alloc] init], let him
    // know the error of his ways

- (instancetype) init
{
    [ NSException raise:@" Singleton" format:@" Use +[ INIItemStore sharedStore]"];
    return nil;
}

    // Here is the real (secret) initializer
- (instancetype) initPrivate
{
    self = [super init];
    if ( self) {
        _privateItems = [[NSMutableArray alloc] init];
    }
    return self;
}
```

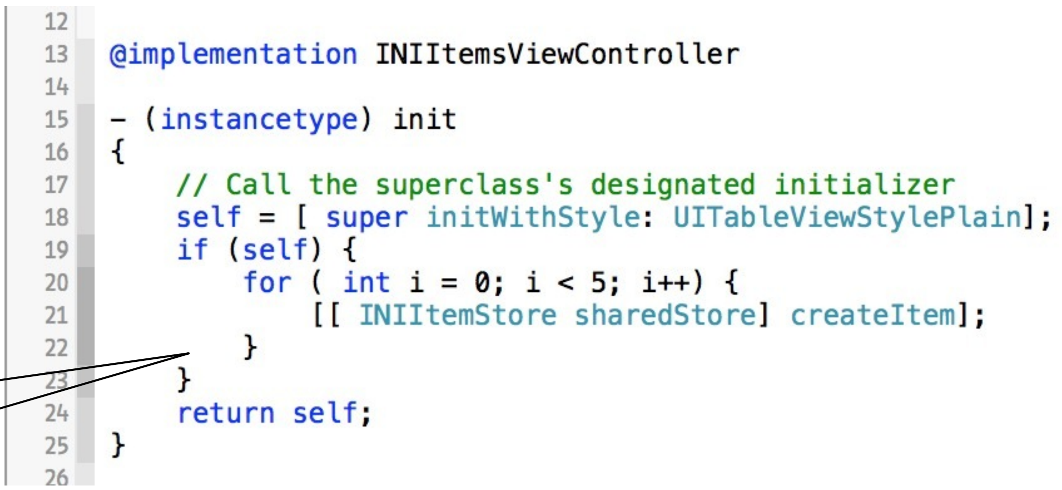
This method will create a new instance of sharedStore only if one does not exist.
Notice that it is declared as static.
This method will always point to the instance of shareStore that gets created the first time it is run.

Implementing data source methods

- ItemsViewController needs to do the following:
  - Create and store the items to be displayed.
  - Implement the two required methods needed to conform to "UITableViewDataSource" protocol namely:
    - tableView:numberOfRowsInSection:
    - tableView:cellForRowAtIndexPath



ItemsViewController
- -init
- -initWithStyle:
- -tableView:cellForRowAtIndexPath:
- -tableView:numberOfRowsInSection:
- -viewDidLoad

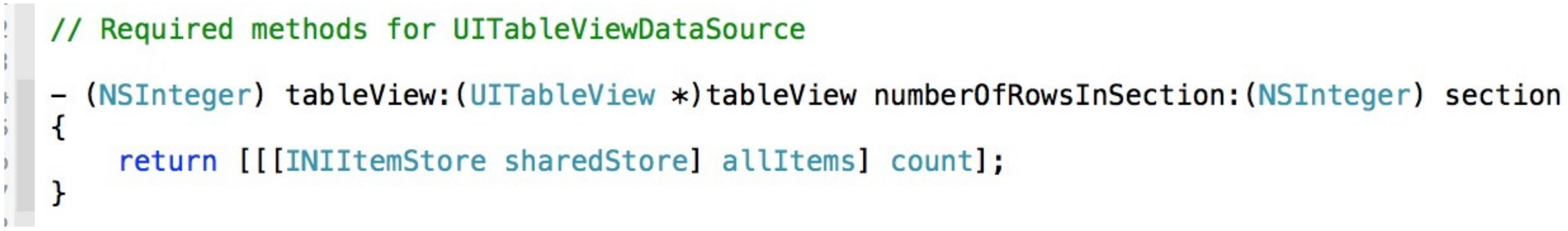


```
@implementation INIItemsViewController

- (instancetype) init
{
    // Call the superclass's designated initializer
    self = [ super initWithStyle: UITableViewStylePlain];
    if (self) {
        for ( int i = 0; i < 5; i++) {
            [[ INIItemStore sharedStore] createItem];
        }
    }
    return self;
}
```

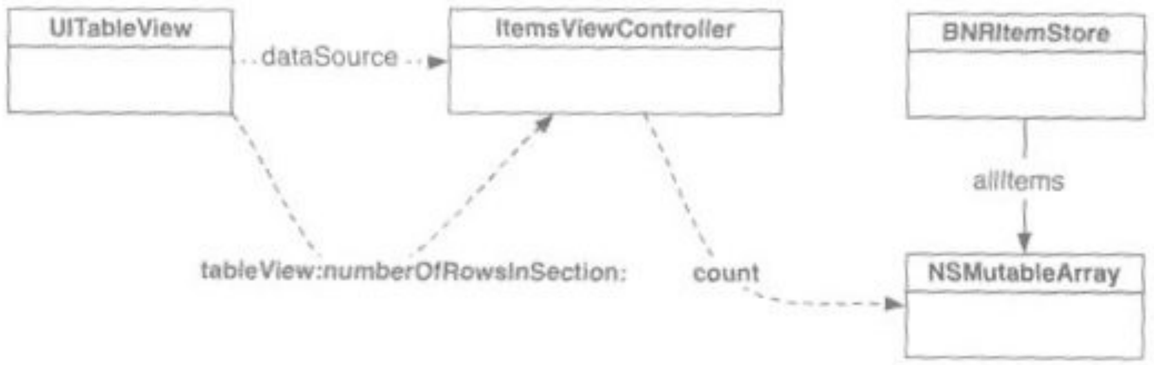
Use init to generate 5 random INIItems and store them in sharedStore

Implementing data source methods (Cont.)

- Implementing “tableView:numberOfRowsInSection:” involves asking sharedStore for it’s allItems, then asking the array allItems for the count of it’s items.



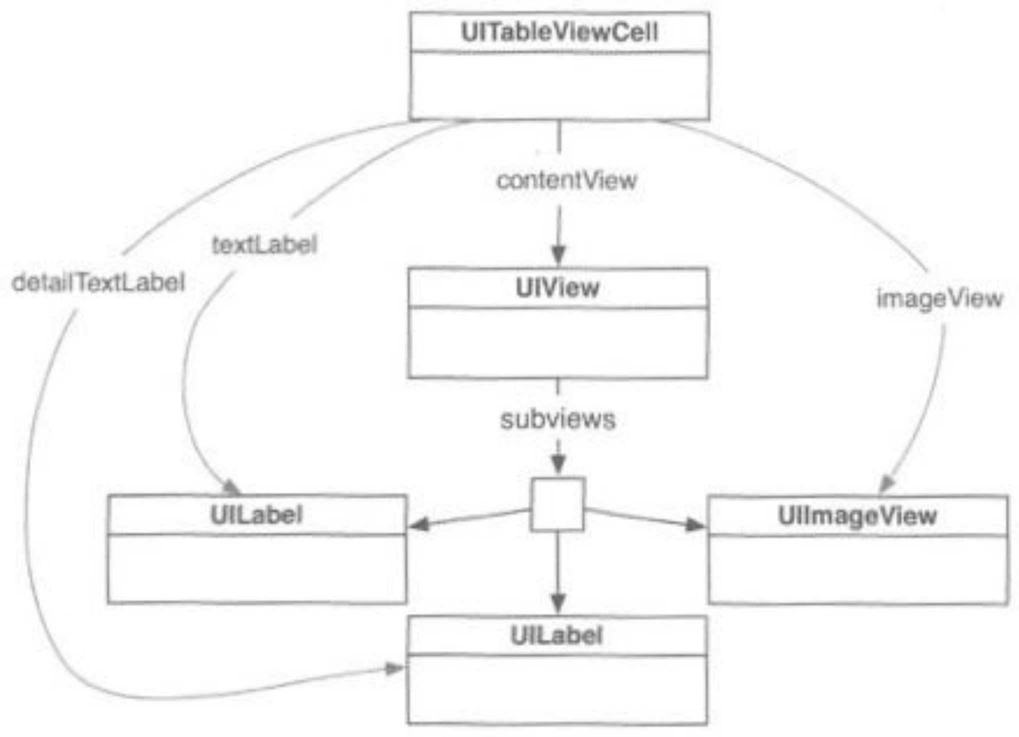
```
// Required methods for UITableViewDataSource

- (NSInteger) tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger) section
{
    return [[[INIItemStore sharedStore] allItems] count];
}
```

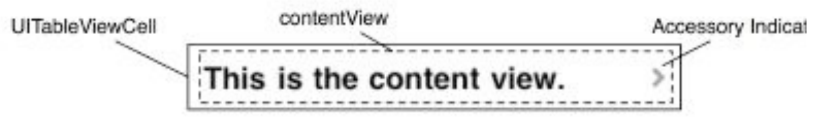


UITableViewCell

- tableView:cellForRowAtIndexPath will require that we learn more about UITableViewCell
- A table view in the UIKit framework is limited to a single column because it is designed for a device with a small screen.
- UITableView is a subclass of UIScrollView, which allows users to scroll through the table, although UITableView allows vertical scrolling only.
- The cells comprising the individual items of the table are UITableViewCell objects
- UITableView uses these UITableViewCell objects to draw the visible rows of the table.
- Cells have content—titles and images—and can have, near the right edge, accessory views.

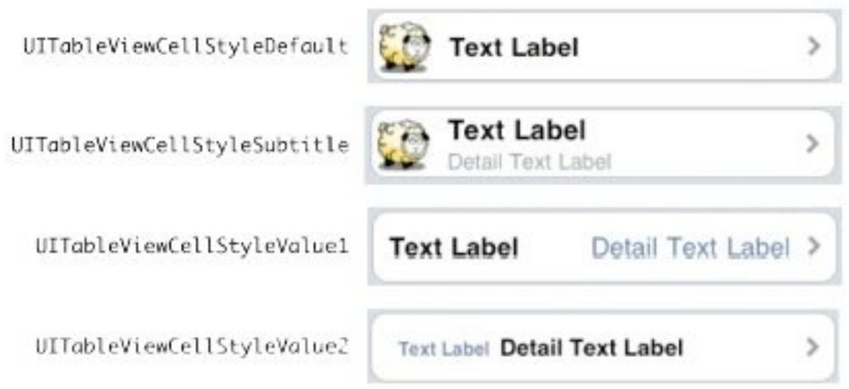
UITableViewCell Layout, Hierarchy, and Styles





Layout

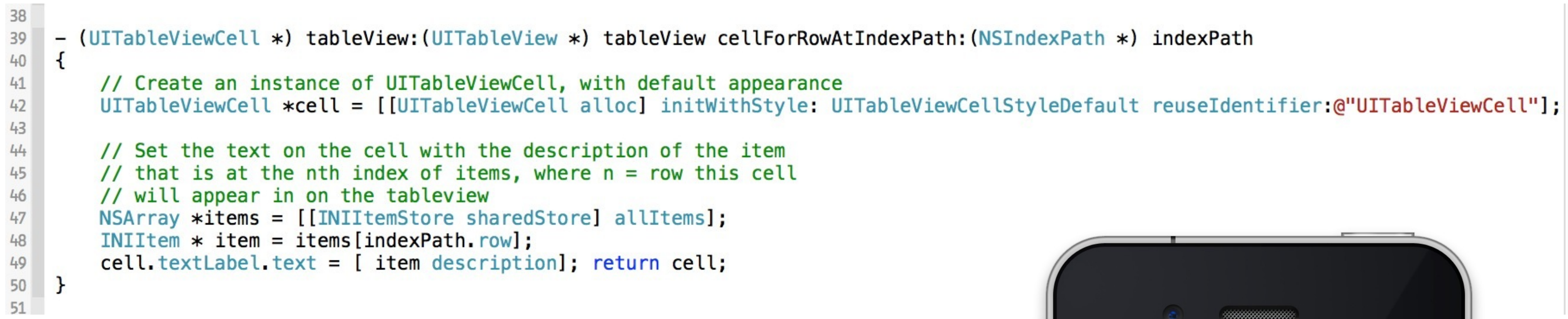
Hierarchy


UITableViewCellStyleDefault
Text Label
UITableViewCellStyleSubtitle
Text Label
Detail Text Label
UITableViewCellStyleValue1
Text Label
Detail Text Label
UITableViewCellStyleValue2
Text Label
Detail Text Label

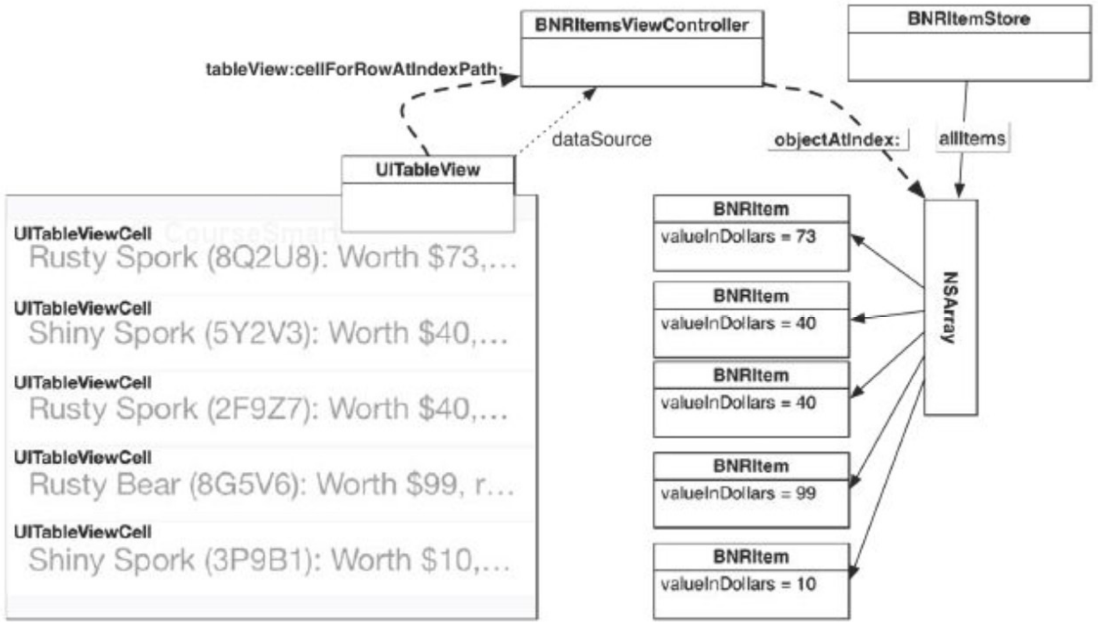
Styles

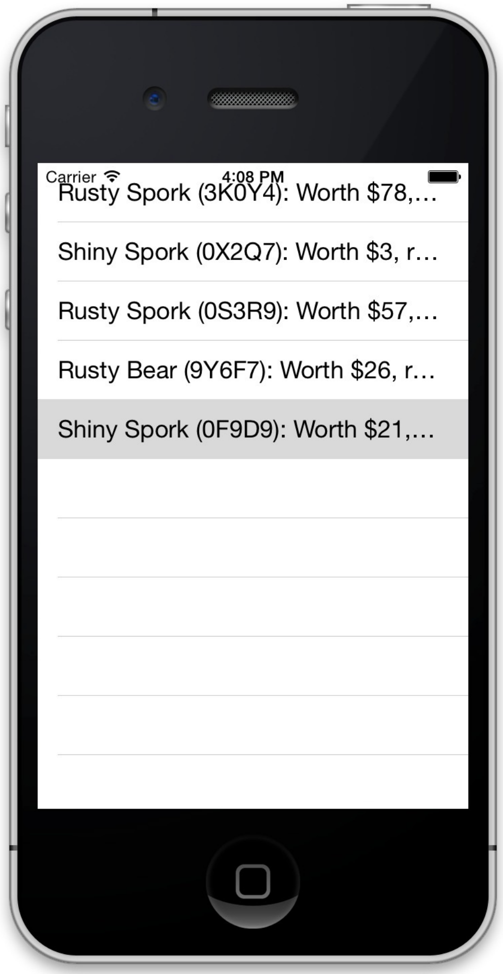
Creating and retrieving UITableViewCells

- tableView:cellForRowAtIndexPath: will
  - create a cell
  - set the texLabel to the description of the corresponding INIItem
  - return it to the UITableView
- The corresponding INIItem is determined by obtaining the row value from the NSIndexPath instance passed to the method
- The other component of the NSIndexPath instance is the section of the table the row is in. In our case we only have one section.
- Every cell has a reuse identifier, which is the cell class “UITableViewCell” in our case.
- The reuse identifier is used to identify and reuse cell that move of the screen when the user scrolls the table view. These cells are kept in a reuse pool



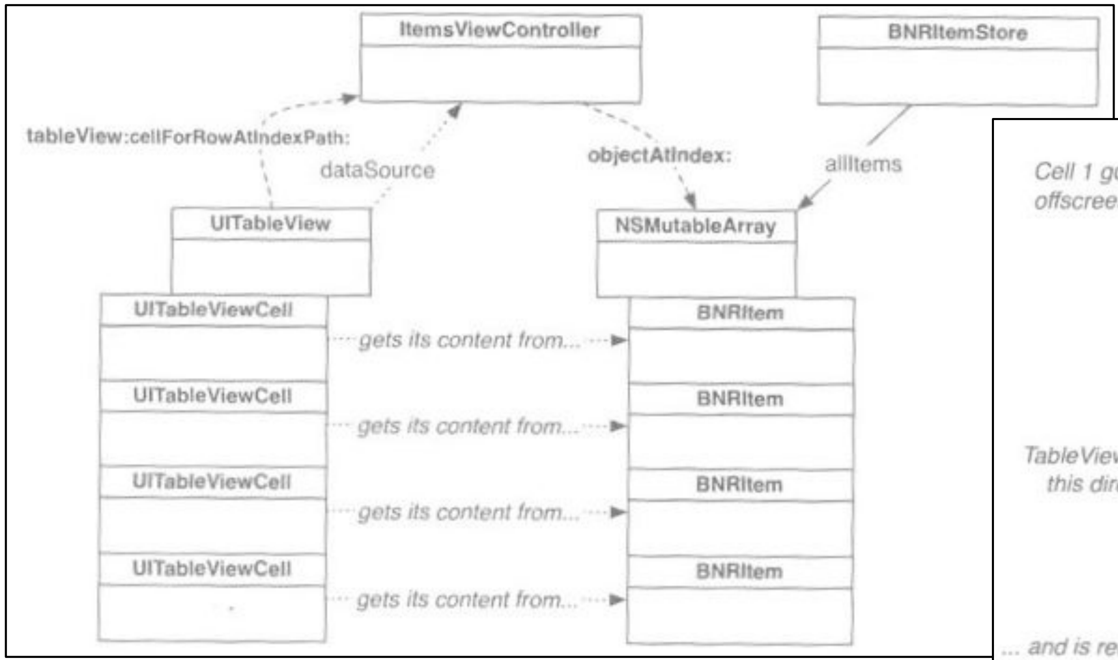
```
- (UITableViewCell *) tableView:(UITableView *) tableView cellForRowAtIndexPath:(NSIndexPath *) indexPath
{
    // Create an instance of UITableViewCell, with default appearance
    UITableViewCell *cell = [[UITableViewCell alloc] initWithStyle: UITableViewCellStyleDefault reuseIdentifier:@"UITableViewCell"];

    // Set the text on the cell with the description of the item
    // that is at the nth index of items, where n = row this cell
    // will appear in on the tableview
    NSArray *items = [[INIItemStore sharedStore] allItems];
    INIItem * item = items[indexPath.row];
    cell.textLabel.text = [ item description]; return cell;
}
```

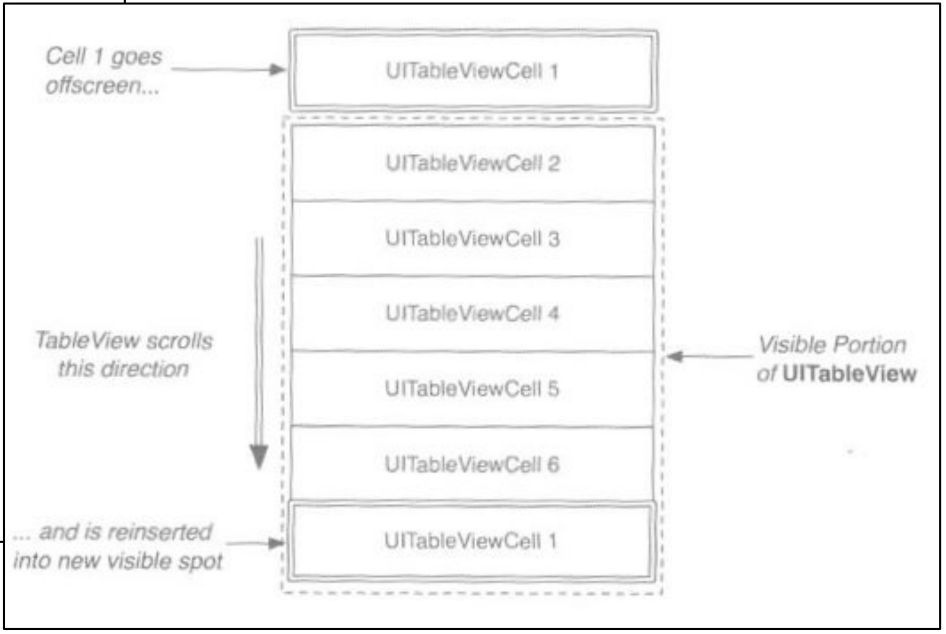


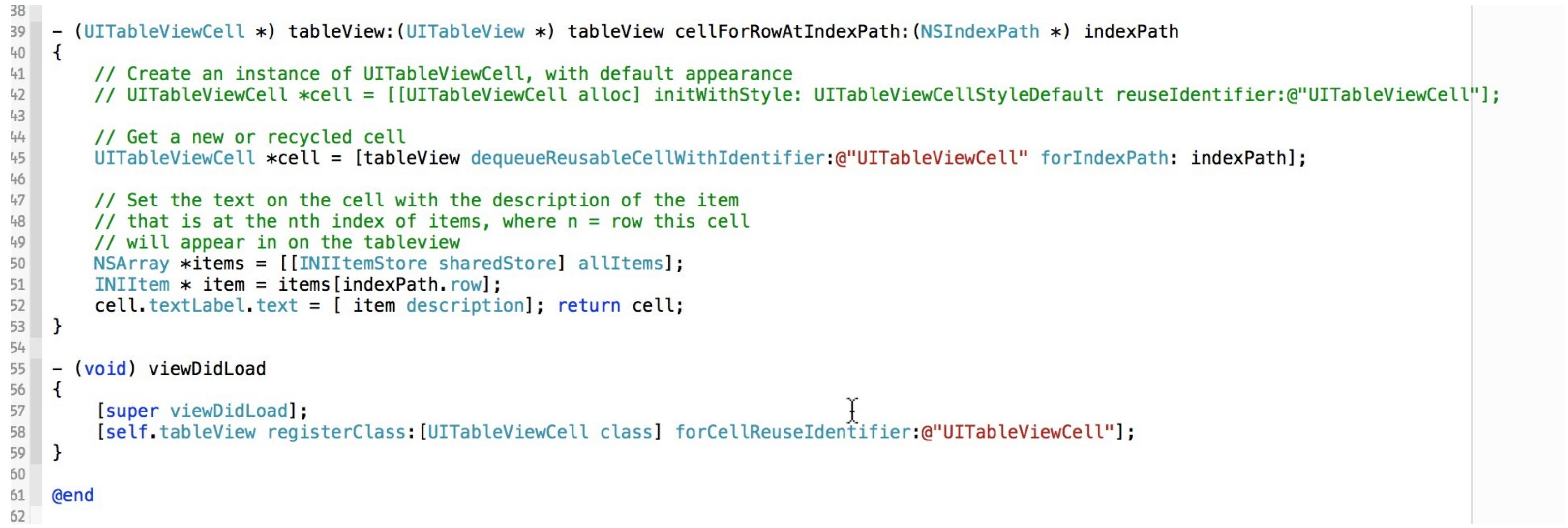


Rusty Spork (3K0Y4): Worth $78,...
Shiny Spork (0X2Q7): Worth $3, r...
Rusty Spork (0S3R9): Worth $57,...
Rusty Bear (9Y6F7): Worth $26, r...
Shiny Spork (0F9D9): Worth $21,...

Reusing UITableViewCells







```
- (UITableViewCell *) tableView:(UITableView *) tableView cellForRowAtIndexPath:(NSIndexPath *) indexPath
{
    // Create an instance of UITableViewCell, with default appearance
    // UITableViewCell *cell = [[UITableViewCell alloc] initWithStyle: UITableViewCellStyleDefault reuseIdentifier:@"UITableViewCell"];

    // Get a new or recycled cell
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell" forIndexPath: indexPath];

    // Set the text on the cell with the description of the item
    // that is at the nth index of items, where n = row this cell
    // will appear in on the tableview
    NSArray *items = [[INIItemStore sharedStore] allItems];
    INIItem * item = items[indexPath.row];
    cell.textLabel.text = [ item description]; return cell;
}

- (void) viewDidLoad
{
    [super viewDidLoad];
    [self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier:@"UITableViewCell"];
}

@end
```