Lab 3 – Working with Lists and Searching

Overview:

The purpose of this lab is to examine how to create interactive lists of items within an iOS application; and how to search through them. The tasks for this lab will build the foundations for next week when we begin working with persistent data.

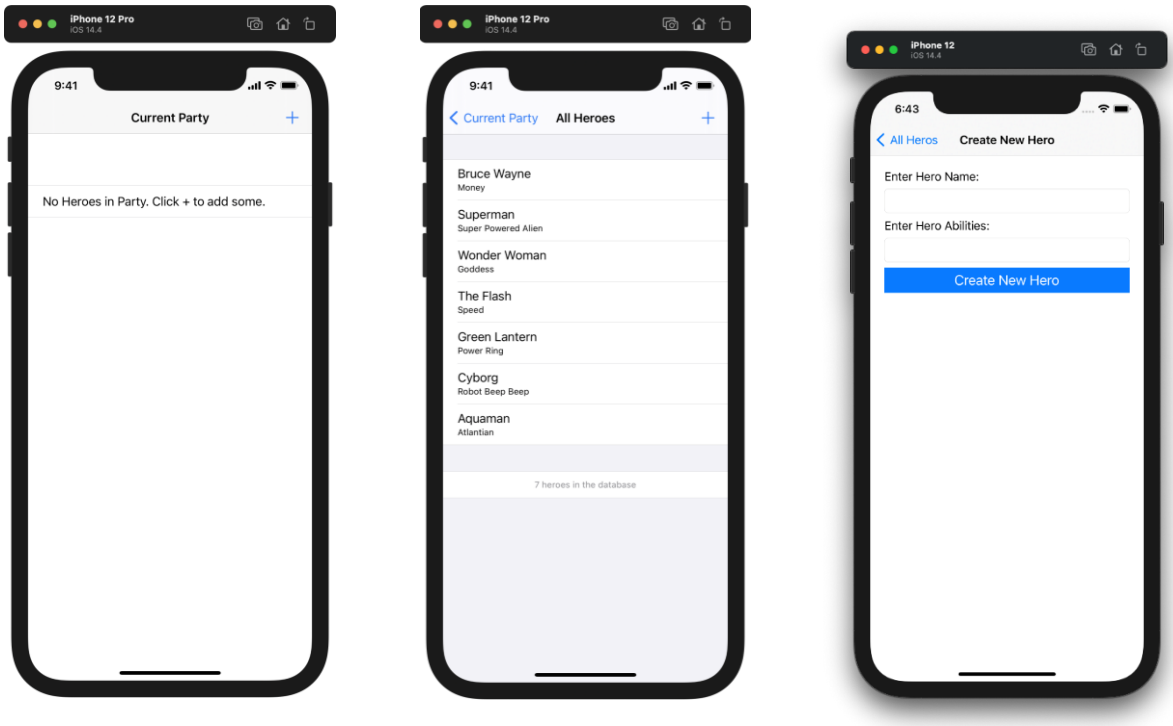
The Task:

For this lab you will create an application that allows users to build a party of Superheroes. These heroes will be from a combination of hard-coded entries as well as custom heroes created by the user.

- The initial screen of the application will be the user's current party of heroes
    - The user can only have up to 6 heroes in the party at any time.
    - The number of heroes in the party is displayed below the hero list.
    - The user can swipe to delete a hero from the party.
    - An Add button on the top navigation bar will allow the user to go to the second ("All Heroes") screen of the application and add heroes to the party.
    - If there are no heroes in the current party a different cell is shown suggesting the user add a party member
- The All Heroes screen displays a list of all the heroes currently present in the application.
    - This screen will have a create button on the navigation bar allowing users to go to a "Create Hero" screen where they can create heroes.
    - Tapping on a hero will add it to the current party so long as there are less than 6 heroes already.
    - Swiping down on the list will display a search bar. Users can type into the search bar to filter the hero list.
- The Create Hero screen allows the user to create their own new hero
    - The user must enter a hero name.
    - The user must enter at least one ability that the hero has.
    - Tapping "Add" will create the hero and add it to the list of all heroes.

Note: Constant testing of our applications is critically important. This lab will break down the development process into a number of steps to allow for individual testing of each screen before integrating it fully into the complete application.

An example for how the application may look is shown below.



Creating Our Project:

As with previous weeks, open Xcode and create a new project. Use the same default settings as in previously and save the application. Once the project has been created, delete the default "**ViewController.swift**" class (when prompted, select Move to Trash) as well as the default view controller on the main storyboard. We will be creating our own shortly.

Creating our initial classes:

With the project created, it is time to start building the first screen of our application. the **Current Party** screen. This will be done using a Table View but will first require us to create a couple class files.

Firstly, create a new **Cocoa Touch Class** for SuperHero. It should be a subclass of **NSObject** similar to the Person class from previous weeks. Ensure that it has the following properties:

- A name property that is a String optional
- An abilities property that is a String optional
- An initializer that takes in both properties and sets them correctly

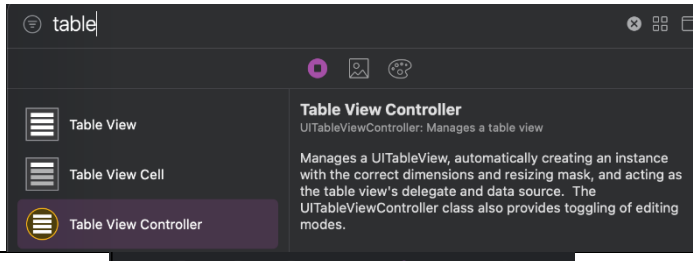
Note: We will revisit the reason why these are optional next week when persistent data is discussed.

Next create another **Cocoa Touch Class** for the Current Party screen. This should be a subclass of **UITableViewController**. Name it CurrentPartyTableViewController. This will create a class with stubs for several table view data source methods. We will fill these in later. For now, it's time to move onto creating the UI for the Current Party.

Building the Current Party Interface:

Open the Main storyboard and complete the following steps

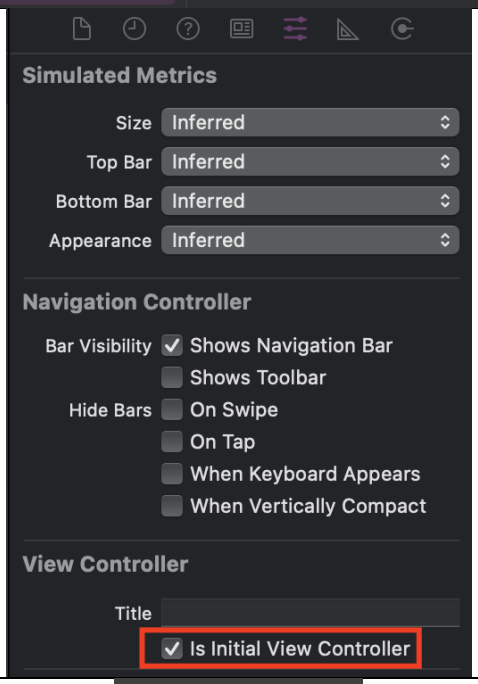
Drag a Table View Controller from the Library onto the Storyboard.



Select the new Table View Controller

Check the "Is Initial View Controller" in the Attributes Inspector for this view controller.

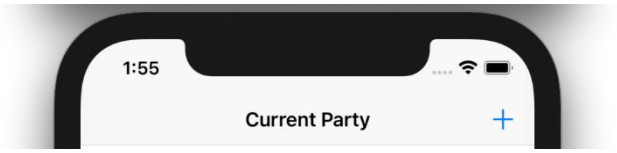
This means it will be our entry point for the storyboard.



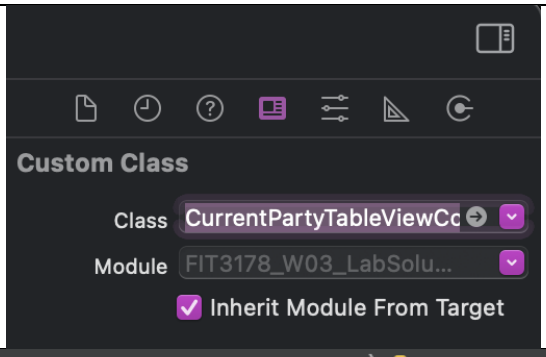
You can always check which View Controller is the entry point for the app by looking for the arrow shown to the left of the view controller.

Embed the Table View Controller in a Navigation Controller.

Set the Table View Controller's title to be "Current Party".

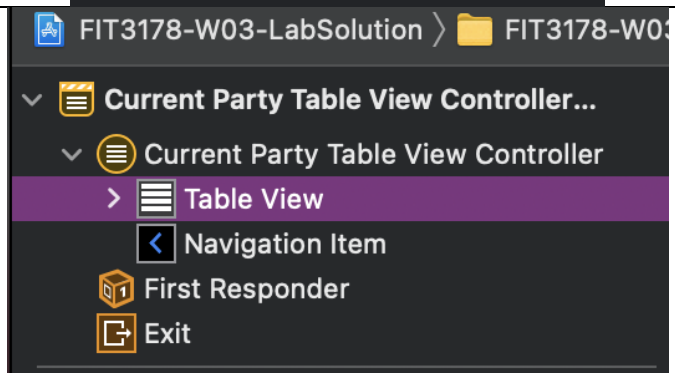


Making sure the Table View Controller is still selected, go to the Identity Inspector and set the custom Class to be the current party controller class that was created in the previous section.



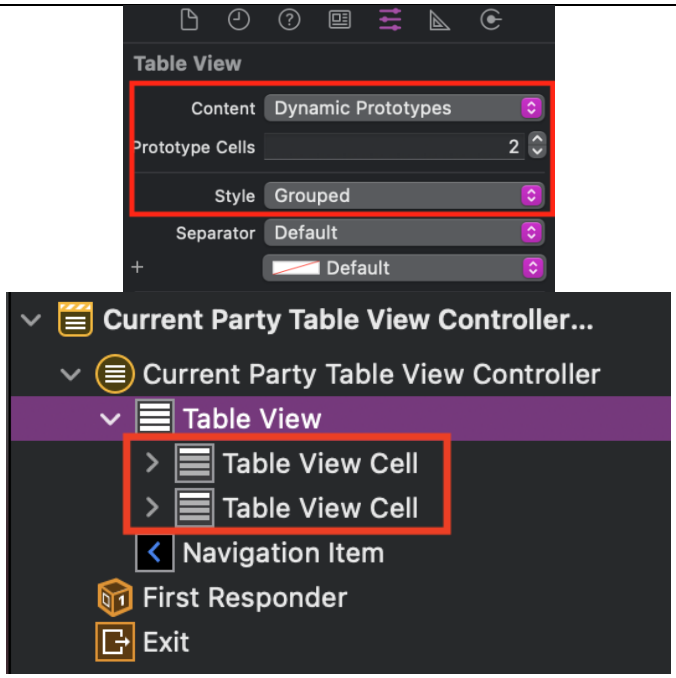
Select the Table View (not the view controller) and go to the Attributes Inspector.

(Optional: From the Attributes Inspector, set the Style to "Grouped". This changes the style of the table view to group different types of cells separately. You can experiment with the available Style and Separator properties to see what you prefer!)



Set the number of Prototype Cells to 2. This will create us two cells on the storyboard.

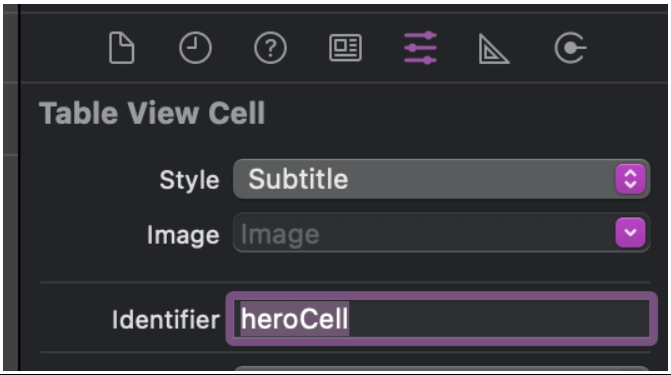
These prototype cells are the different cell types that can appear in our table. For this screen, we will have a Super Hero Cell and an Info Cell.



Select the first Table View Cell.

In the Attributes Inspector set the Style to Subtitle. This is a defaulion
t cell type that has a title and detail text labels.

Set the Reuse Identifier to “heroCell”. This is the identifier that will be used in code to reference this cell.

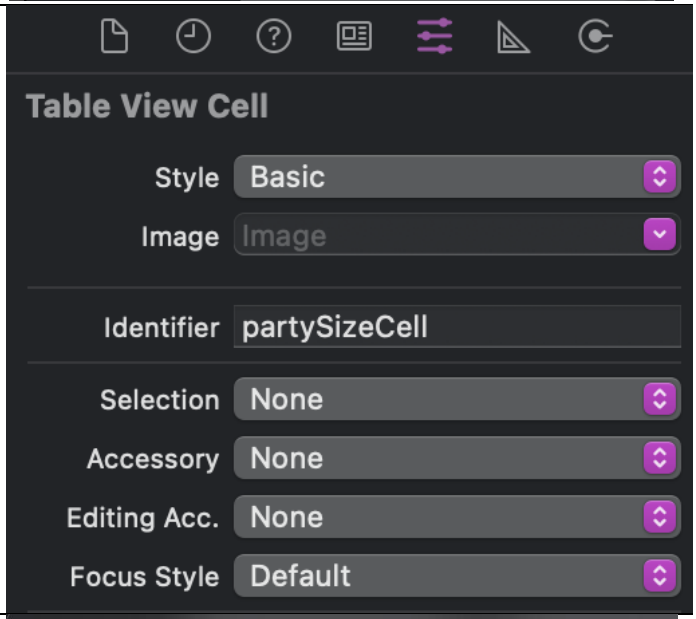


Select the second Table View Cell.

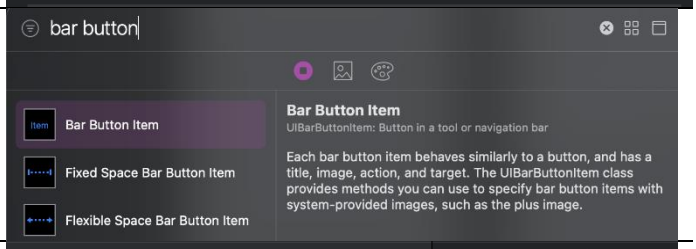
In the Attributes Inspector set the Style to Basic. This is a default cell type that has a title label.

Set the Reuse Identifier to “partySizeCell”.

Set the Selection type to None. This means that it will not be able to be selected by the user.

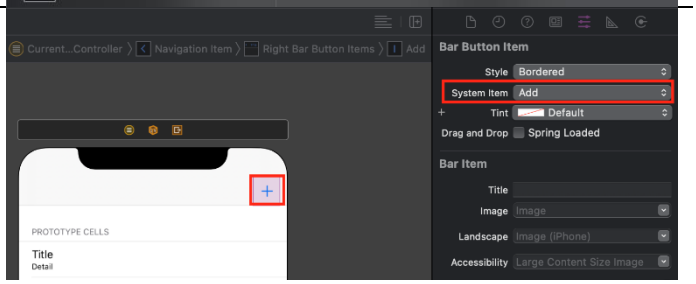


Select the Table View Controller and drag a Bar Button Item from the Objects Library onto the right side of the Navigation Bar on the Current Party View Controller.



In the Attributes Inspector, change its System Item property from “Custom” to “Add”. The button should become a Plus symbol.

This will be our button to go to the next screen.



With this the UI for the Current Party View Controller is complete. Next up is to finish the code for the controller.

Coding the Current Party Table View Controller Class:

Open the CurrentPartyTableViewController.swift file. We will make a series of additions to build out the required functionality. To begin, we need to define some constants and properties.

Table View Controllers have the concept of sections. Each section can have its own type of cell and number of cells. For this app we have two sections; one for heroes in our party and one for displaying current number of heroes in our party.

To avoid hard coding in section numbers it is important to define these at the top of the class as constants.

```
let SECTION_HERO = 0
let SECTION_INFO = 1
```

Much like array indices, section indices start at 0, with 0 being the first.

It is also good practice to create constants for any identifiers used. Create constants for each of the cell identifiers created in the previous section.

```
let CELL_HERO = "heroCell"
let CELL_INFO = "partySizeCell"
```

We will also need a property to store our party of heroes. Create an array of SuperHero objects and call it currentParty, initially empty.

```
var currentParty: [SuperHero] = []
```

With these done we can move onto setting up the required TableView methods. All of these are already included in the boilerplate code, though some are commented out. Let us examine each of them now.

Number of Sections in Table View

```
override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 0
}
```

The numberOfSections method, as its name implies, determines the number of sections in the Table View. We have two sections, so change this method to return 2. You can delete the warning comment.

Table View Number of Rows In Section

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 0
}
```

The next series of methods look like they are named tableView, but each has different parameter labels and has a different purpose.

Note: Like we discussed in the last workshop, this is Apple's convention of delegate methods always passing to the delegate the class instance being delegated from, so one class can be the delegate for multiple instances of the same class and differentiate between them.

You can think of the above method as "tableViewNumberOfRowsInSection". Given a section it determines the number of rows in a specified section.

Some logic is needed here as we need to return a different value depending on if the section is for our current party or for our count. This can be done using a switch-statement or an if-statement, by comparing our defined sections to the section number passed into the method. A switch statement implementation could look as follows:

```
switch section {
    case SECTION_HERO:
        return currentParty.count
    case SECTION_INFO:
        return 1
    default:
        return 0
}
```

The Super Hero section will show a cell for each hero, the info section always shows a single cell. Whilst we should not ever reach this case, the switch-statement requires a default case.

Table View Cell For Row At Index Path

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
    ->UITableViewCell {

      let cell = tableView.dequeueReusableCell(withIdentifier: "reuseIdentifier", for:
indexPath)

      // Configure the cell...

      return cell
}
```

The **tableViewCellForRowAtIndexPath** method creates the cells to be displayed to the user. We call the **dequeReusableCell** method and provide it an identifier and the

indexPath to generate us a cell object. The identifier must match a Reuse Identifier we created on the storyboard. The index path specifies a section and row.

The section can help us work out the kind of cell that we need to generate. The row can help us work out which object we should associate with the cell.

As we have two different types of cells for each section the first step is determining which section we are currently in. Delete the current code within the **cellForRowAt** method and add the following.

```
if indexPath.section == SECTION_HERO {
    // Configure a hero cell
}

// Configure an info cell instead
```

If we are in the section for our current party, we will want to generate cells using our hero identifier then return a cell object. Otherwise, we will want to generate a cell for our information cell and return it.

```
if indexPath.section == SECTION_HERO {
    let heroCell = tableView.dequeueReusableCell(withIdentifier: CELL_HERO, for: indexPath)
    let hero = currentParty[indexPath.row]

    heroCell.textLabel?.text = hero.name
    heroCell.detailTextLabel?.text = hero.abilities
    return heroCell
}

let infoCell = tableView.dequeueReusableCell(withIdentifier: CELL_INFO, for: indexPath)

if currentParty.isEmpty {
    infoCell.textLabel?.text = "No Heroes in Party. Tap + to add some."
} else {
    infoCell.textLabel?.text = "\(currentParty.count)/6 Heroes in Party"
}

return infoCell
```

When dequeuing a new cell, the type generated will be whatever has been specified with the identifier. In the case of CELL_HERO, it means that our cell is a subtitle type, which is an instance of the default **UITableViewCell**. As such we do not need to do any casting. When using custom cells, we would need to do a cast here. We will examine this in the All Super Heroes View Controller later.

Did you know? As mentioned previously the indexPath contains the current row number which will correspond to a specific hero in our array, because we've told the table view the number of rows in this section is equal to the array length.

With this method done we can continue onto the remaining methods required by the table view.

Table View Can Edit Row At Index Path

```
override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool {
      return false
}
```

The **tableViewCanEditRowAtIndexPath** method allows us to specify whether a certain row can be edited at all during run time by the user. This could be updating, or in our case deleting! The default implementation just returns false, meaning that no cells can be edited. We can change this using the provided indexPath to say that Hero Cells can be edited by the Info Cells cannot.

```
if indexPath.section == SECTION_HERO {
    return true
}

return false
```

Determining whether or not any given row can be edited is one method. Handling the editing is done by another. Which brings us to…

Table View Commit Editing Style For Row At Index Path

```
override func tableView(_ tableView: UITableView, commit editingStyle:
    UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
}
```

This method allows us to handle deletion or insertion of rows into our table view. As with previous methods this provides us with an index path for a specified section and row. The method also provides an editing style which we will check to see if it is a deletion. Only Hero Cells can be deleted however.

```
if editingStyle == .delete && indexPath.section == SECTION_HERO {
}
```

Inside of the if statement the following steps must be done:

- Remove the Hero from the Current Party
- Delete the Row from the Table View
- Update the Info Section

To avoid any crashes these three steps need to be done in a single batch. Fortunately, the table view provides us with a method to do so.

```
if editingStyle == .delete && indexPath.section == SECTION_HERO {
    tableView.performBatchUpdates({
        self.currentParty.remove(at: indexPath.row)
        self.tableView.deleteRows(at: [indexPath], with: .fade)
        self.tableView.reloadSections([SECTION_INFO], with: .automatic)
    }, completion: nil)
}
```

performBatchUpdates allows us to provide a closure to execute in a single batch. This is necessary when multiple changes to a table view or collection view are needed at once.

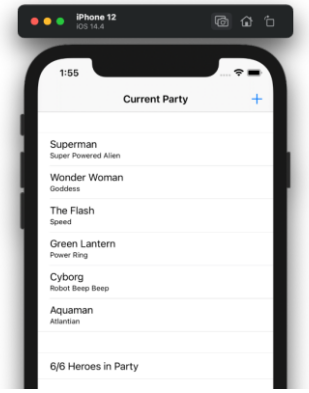
With the table view methods complete there is one more method that needs to be created for testing purposes. Create a method called **testHeroes** with no parameters and no return value. Inside this method, create and add 6 heroes to the current party array.

```
currentParty.append(SuperHero(name: "Superman", abilities: "Super Powered Alien"))
currentParty.append(SuperHero(name: "Wonder Woman", abilities: "Goddess"))
currentParty.append(SuperHero(name: "The Flash", abilities: "Speed"))
currentParty.append(SuperHero(name: "Green Lantern", abilities: "Power Ring"))
currentParty.append(SuperHero(name: "Cyborg", abilities: "Robot Beep Beep"))
currentParty.append(SuperHero(name: "Aquaman", abilities: "Atlantian"))
```

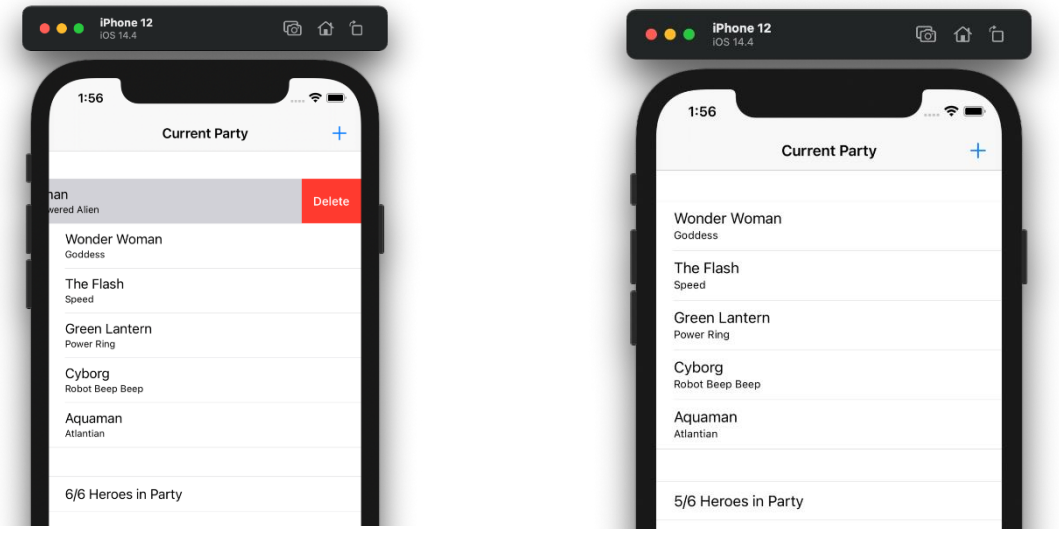
Add a call to the testHeroes method within the **viewDidLoad** method. We will be removing this later but for now include it to test the functionality of the current party screen.

Testing the Current Party Screen:

Compile and run the app in the simulator. Assuming everything is correct, the Current Party Screen should be visible with our 6 test heroes.



Swiping to the left on a hero should delete them from the list.



If you delete all heroes, the text of the info cell should display the “Tap + to add some.” text.

If everything is working then it is time to move onto the next step, creating the All Heroes Screen. If there are any issues, please discuss with your demonstrator.

Warning: Are you getting a blank screen? Try checking that the Navigation Controller is set to Initial View Controller and that the Table View Controller has its class set in the Identify Inspector.

Creating Required Classes:

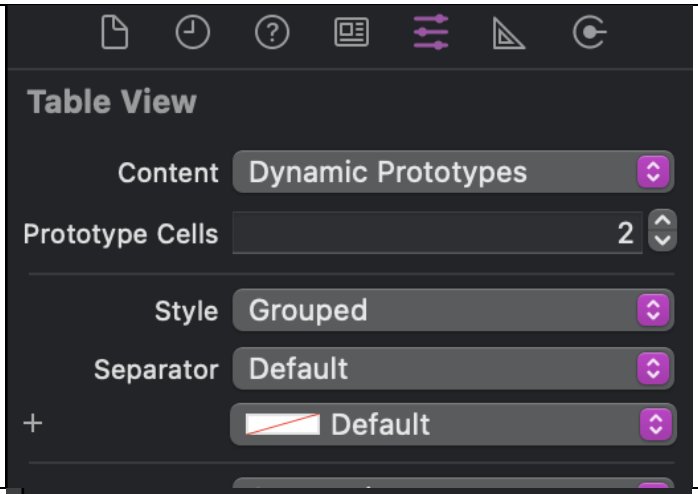
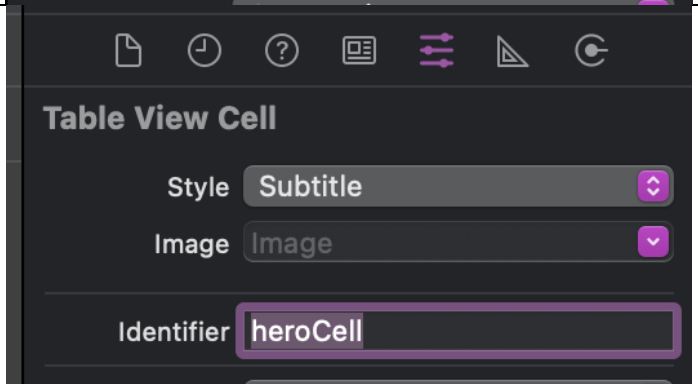
Before we begin building the interface, there are two additional classes that need to be created. The first is the Table View Controller for our All Heroes interface. The second is a custom Table View Cell for displaying our total number of heroes. The default cells provided to us are quite useful, but there is sometimes a need to create our own custom cells. We will look at doing so here.

Create a **Cocoa Touch Class** for the All Heroes screen. This should be a subclass of **UITableViewController**. Name it AllHeroesTableViewController.

Create a **Cocoa Touch Class** for the Information Cell screen. This should be a subclass of **UITableViewCell**. Name it HeroCountTableViewCell.

Building the All Heroes Interface:

Open the Main storyboard and complete the following steps:

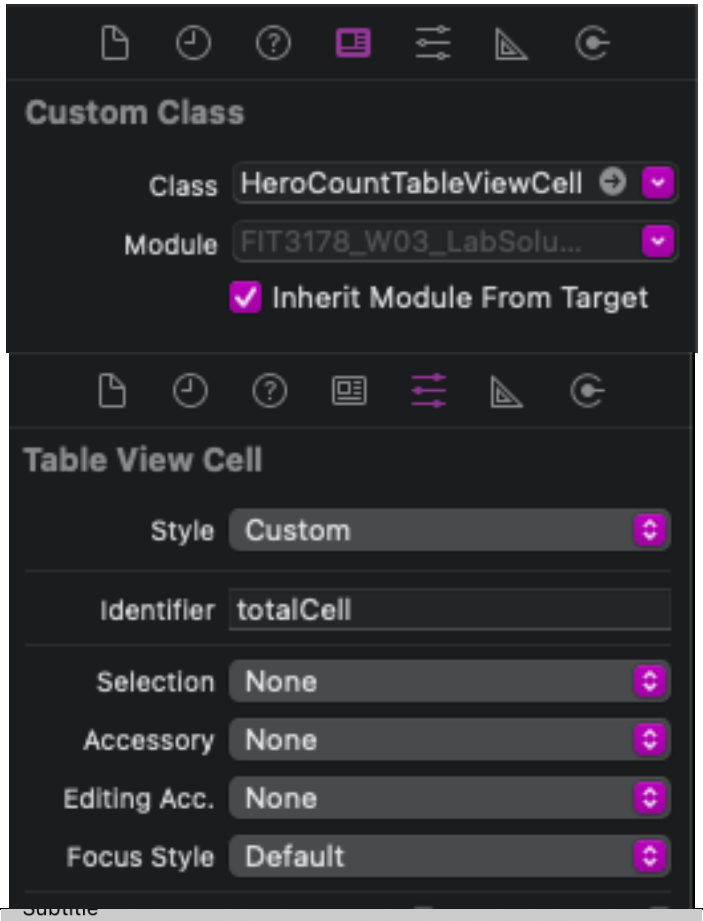
Drag a second Table View Controller from the Object Library onto the Storyboard. Select the + bar button on the Current Party Screen and create a segue to the new Table View Controller (Hold Control, then drag). Select the segue just created and go to the Attributed Inspector. Set the Identifier of this new segue to “allHeroesSegue”. Set the title to "All Heroes"	
Set the Custom Class to be the AllHeroesTableViewController	
In the Attributes inspector set the number of prototype cells for the Table View to be 2 and the style to grouped. You can change the style based on your preferences.	
Select the first Table View Cell. In the Attributes Inspector set the Style to Subtitle. Set the identifier to be heroCell. We have used this identifier before, but that’s okay, it was for a different table view.	

Select the second Table View Cell.

In the Identity Inspector set the Custom Class to be "HeroCountTableViewCell".

(If HeroCountTableViewCell is not an option, it means that you have not correctly selected the cell.)

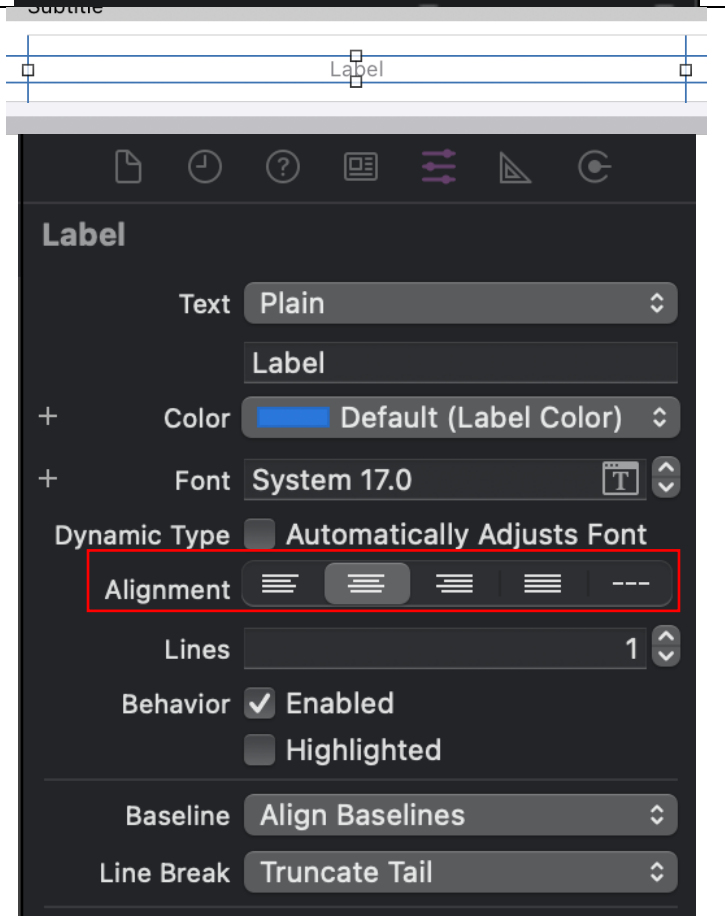
In the Attributes Inspector ensure that the Style is set to Custom, the Identifier is set to "totalCell" and that the Selection is set to "None"



Open the Object Library and drag a Label onto the second Table View Cell.

Constrain the label to the left, top, right and bottom cell margins with an offset of zero on all sides.

Set the Alignment property of the Label to Center in the Attributes Inspector.



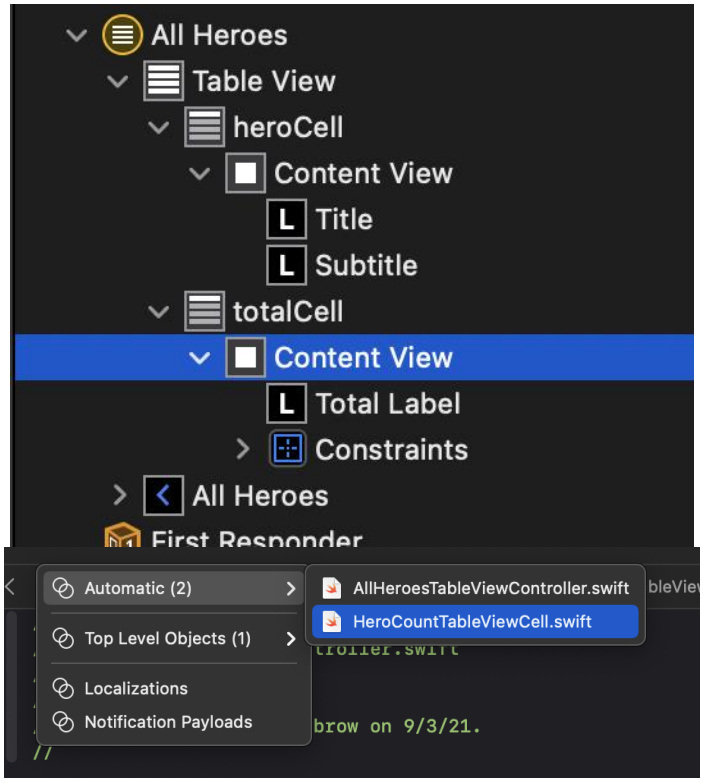
PAY CLOSE ATTENTION TO THIS STEP.

From the Document Outline select the Second Table View Cell's Content View.

With the Content View click the Adjust Editor Options and select the Assistant.

Click on "Automatic" and you should see two options. AllHeroesTableViewController and HeroCountTableViewCell. Select the latter.

(If the assistant shows “No Assistant Results”, this is a bug in Xcode. Close, and relaunch it and continue from this step.)



Create an outlet for the label. Name this outlet "totalLabel”.

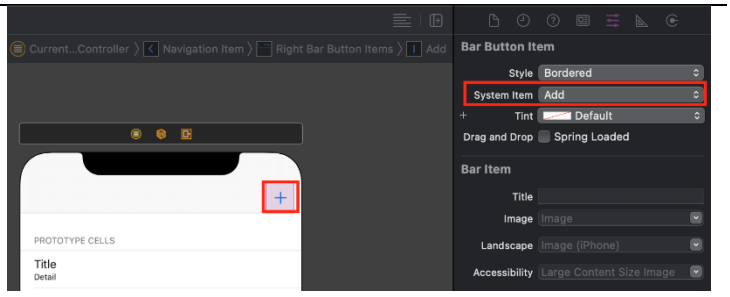
(Please check the type of this outlet is UILabel. If it says UIView, you accidently created outlets to the content view. In this case, undo and create them correctly.)

```
class HeroCountTableViewCell: UITableViewCell {

    @IBOutlet weak var totalLabel: UILabel!

    override func awakeFromNib() {
        super.awakeFromNib()
    }

    override func setSelected(_ selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)
    }
}
```

Select the All Heroes Table View Controller again. Go to the Objects Library and find "Bar Button Item".

Drag it onto the Navigation Bar of the second Table View Controller.

Again, set its System Item to “Add”.



With this, the UI updates are done. It is time to jump back into the code

Coding the Add Super Hero Delegate:

Before coding the next Table View Controller we need to create one more Swift file. This time it will be a protocol instead of a normal class. For a revision on protocols and delegates see Lecture 2.

Create a **Swift File** (not Cocoa Touch Class) for the delegate. This will be a protocol named AddSuperHeroDelegate. This will allow us to easily pass around Hero objects without having to worry about keeping unnecessary view controllers in memory.

For creating a delegate use the **protocol** keyword instead of class, provide a name and any inheritance needed. In our case we want to inherit from the **AnyObject** class

```
protocol AddSuperHeroDelegate: AnyObject {

}
```

Create a method stub for adding a new Hero. It should take in a Hero object and return a boolean. This will be for the delegate to say whether it can successfully add a Hero. For example, it might not be able to add a hero if the party already has 6 members.

```
func addSuperHero(_ newHero: SuperHero) -> Bool
```

Coding the All Heroes Table View Controller Class:

Open the AllHeroesTableViewController.swift file. We will be going through and making a series of additions to build out the required functionality.

To begin we need to define our class variables. They will be fairly similar to the Current Party Screen for now but we will be customising it further later.

For now, create two constants for the two sections: HERO and INFO.

```
let SECTION_HERO = 0
let SECTION_INFO = 1
```

Two identifiers for each of the two types of cells are also needed.

```
let CELL_HERO = "heroCell"
let CELL_INFO = "totalCell"
```

An array property to hold all the superheroes is also needed, named allHeroes.

```
var allHeroes: [SuperHero] = []
```

Lastly, we need to create a weak variable to hold a Add Super Hero delegate object.

```
weak var superHeroDelegate: AddSuperHeroDelegate?
```

The weak keyword prevents an object assigned to this property having its reference count increased (for Automatic Reference Counting). This is done to stop strong reference cycles, where the publisher and subscriber classes both hold a strong reference to each other. Mostly, delegate properties should be weak to prevent reference cycles, which cause memory leaks.

The default Table View methods need to be filled out similar to the Current Party View Controller.

The **numberOfSectionsInTableView** method should return 2. We have two sections again. One for the Hero Cells and one for the Info Cell.

The **tableViewNumberOfRowsInSection** method should return the count of the allHeroes array if the specified section is the Hero section and 1 if it is the Info section.

The **tableViewCellForRowAtIndexPath** method is again like the previous Table View with a few minor differences.

Setting up the Hero Cell is done the same.

```
if indexPath.section == SECTION_HERO {
    let heroCell = tableView.dequeueReusableCell(withIdentifier: CELL_HERO, for:
indexPath)
    let hero = allHeroes[indexPath.row]

    heroCell.textLabel?.text = hero.name
    heroCell.detailTextLabel?.text = hero.abilities
    return heroCell
}
```

For the Info Cell we need to make a few changes. Unlike the previous Table View we are using a custom cell type we have created ourselves. This means we will need to dequeue the cell and then cast it to its correct type (in order to use the totalLabel property).

```
let infoCell = tableView.dequeueReusableCell(withIdentifier: CELL_INFO, for:
indexPath) as! HeroCountTableViewCell

infoCell.totalLabel?.text = "\(allHeroes.count) heroes in the database"

return infoCell
```

The forced cast is done using the **as!** keyword and then specifying the class we want to cast it to.

Warning: If a forced cast fails the application will immediately crash. These should only be done when you know with 100% certainty that the cell (or other type) you are casting is a particular type.

The table view **canEditRowAt** and **commitEditingStyleForRowAtIndexPath** methods need to be identical to the CurrentPartyTableViewController. Copy and adapt the code you used earlier for these two methods (currentParty will need to be renamed allHeroes).

The All Heroes Table View Controller does need to include one more Table View Controller method to allow us to handle the action of selecting (tapping on) a Hero.

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

}
```

This **didSelectRowAtIndexPath** method allows us to provide behaviour for when the user selects a row within the Table View. This method will only be called for cells that have selection enabled. By default, all table view cells have this enabled. However, we disabled selection for the Info section back when creating the UI in the Storyboard. This means we only need to deal with the behaviour when a Hero cell is tapped.

We want this method to call the addSuperHero method of the delegate, passing it the selected hero. If this delegate call says the hero was added (i.e., it returns true), then the current screen is dismissed and the application returns back to the Current Party screen. If it fails, then we deselect the row and display an error message to the user.

Did you know? If you have not done so already, this is a good opportunity to include the displayMessage class extension into the project.

If you need a refresher, check out the supplementary on Class Extensions!

```
if let superHeroDelegate = superHeroDelegate {
    if superHeroDelegate.addSuperHero(allHeroes[indexPath.row]) {
        navigationController?.popViewController(animated: false)
        return
    }
    else {
```

```
        displayMessage(title: "Party Full", message: "Unable to add more members
to party")
    }
}
tableView.deselectRow(at: indexPath, animated: true)
```

The outer if-statement checks if there is a delegate and unwraps it. If there is, we call the addSuperHero on the delegate. If this returns true (meaning the hero was added), we pop back to the Current Party view controller and do nothing else. If the hero wasn't added, we show an error message. Finally, if we are remaining on this view controller, the tapped row is deselected.

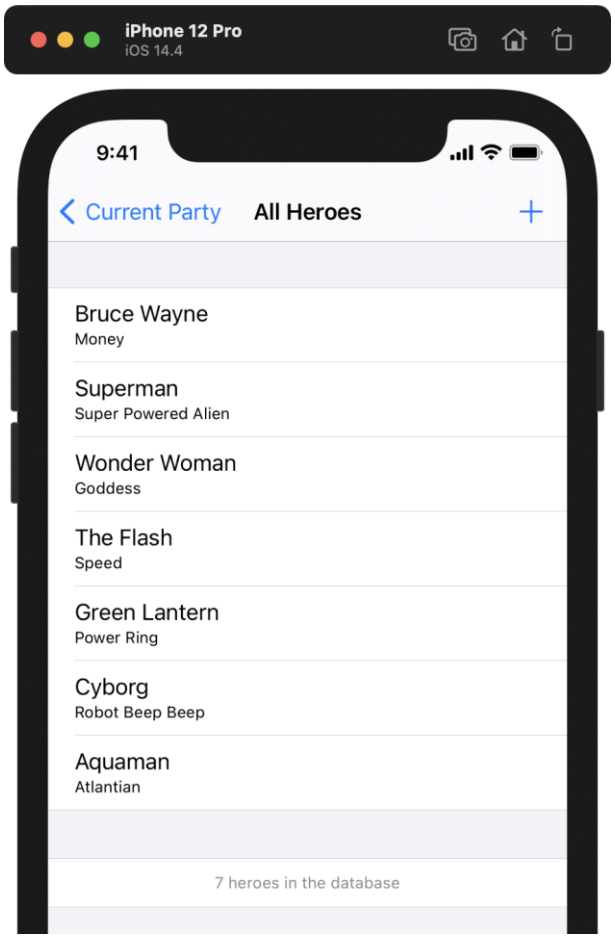
The last thing to do for this screen is to add some default heroes for testing purposes. Create a new method called createDefaultHeroes and add a bunch of heroes to the allHeroes array

```
allHeroes.append(SuperHero(name: "Bruce Wayne", abilities: "Money"))
allHeroes.append(SuperHero(name: "Superman", abilities: "Super Powered Alien"))
allHeroes.append(SuperHero(name: "Wonder Woman", abilities: "Goddess"))
allHeroes.append(SuperHero(name: "The Flash", abilities: "Speed"))
allHeroes.append(SuperHero(name: "Green Lantern", abilities: "Power Ring"))
allHeroes.append(SuperHero(name: "Cyborg", abilities: "Robot Beep Beep"))
allHeroes.append(SuperHero(name: "Aquaman", abilities: "Atlantian"))
```

Make sure to call the method from **viewDidLoad** and with that it is ready for testing again.

Testing the All Heroes Screen:

Compile and run the app in the simulator. Assuming everything so far is correct the Current Party Screen should be visible with our 6 test heroes. Clicking on the + sign should transition over to the All Heroes Screen.



Since there is no delegate set, selecting a super hero will have no effect (other than deselecting the cell).

Updating the Current Party Table View Controller:

The first important step is to ensure that our class inherits and conforms to the AddSuperHeroDelegate protocol that we created before. This will require us updating the class definition and including a new method within the class.

```
class CurrentPartyTableViewController: UITableViewController, AddSuperHeroDelegate
{
```

Once this is added Xcode will give an error saying the class does not conform to the protocol. Add the following method to the class to fix this issue.

```
func addSuperHero(_ newHero: SuperHero) -> Bool {

}
```

This method needs to first check to see if there is enough room in the party. If there is, then add the hero to the party, insert a row into the table view and update the info section. Once again this is done through a batch update. If we can successfully add a hero, we need to return true, otherwise we return false.

```
if currentParty.count >= 6 {
    return false
}

tableView.performBatchUpdates({
    currentParty.append(newHero)
    tableView.insertRows(at: [IndexPath(row: currentParty.count - 1, section:
SECTION_HERO)],
        with: .automatic)
    tableView.reloadSections([SECTION_INFO], with: .automatic)
}, completion: nil)

return true
```

Currently the only validation being done is making sure the party is not full. We are also using a magic number for the maximum party size! As an additional task, add extra validation that stops the user from adding the same hero a second time. Also fix the hard-coded number by defining a constant variable.

The final addition to the Current Party Table View Controller is to add in the prepareForsegue method and set up the All Heroes' superHeroDelegate. We will be making this class the delegate so when the user selects a hero on the All Heroes screen, the Current Party view controller will handle the addition.

Create the prepareForsegue method as follows:

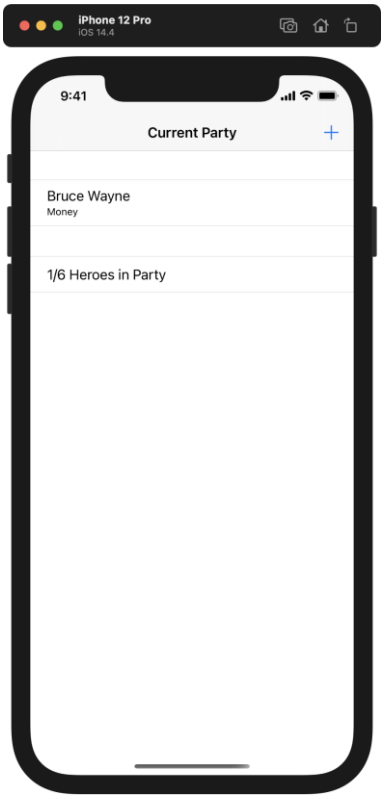
```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "allHeroesSegue" {
        let destination = segue.destination as! AllHeroesTableViewController
        destination.superHeroDelegate = self
    }
}
```

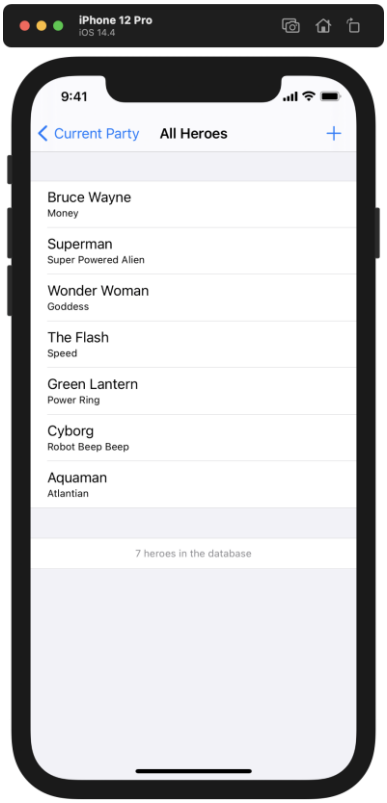
Ensure that the segue identifier matches the one you entered in the Storyboard.

Lastly, remove the **testHeroes** method from the Current Party Table View Controller. This way we have an empty party for testing.

Testing adding Heroes to Party:

Compile and run the app in the simulator. Go to the All Heroes screen and select a hero. You should see the application return to the Current Party screen and include the selected hero within the party. Continue adding heroes to confirm that you are unable to add more than 6 heroes.





Adding Search Functionality to All Heroes:

With the number of heroes we currently have it is fairly easy to look through them all. But what if we had a larger list and wanted to filter through them. We can define our own searching using the UISearchResultsUpdating protocol provided by UIKit.

Firstly we need to add the protocol to the class definition

```
class AllHeroesTableViewController: UITableViewController, UISearchResultsUpdating
{
```

This will cause an error due to not including the required method. Add the method into the class to fix the issue.

```
func updateSearchResults(for searchController: UISearchController) {

}
```

The **updateSearchResults** method will be called every time a change is detected in the search bar. We will set this up shortly to filter heroes based on an entered text string. Firstly, we need to set up the filtered list and the search bar itself.

Create a new property, an array of SuperHero objects called filteredHeroes. This will be our filtered list that is displayed to the user. We will also need to change multiple table view methods so that they use filteredHeroes instead of allHeroes. Replace mentions of allHeroes in the following methods:

- numberOfRowsInSection
- cellForRowAtIndexPath
- commitEditingStyleforRowAtIndexPath
- didSelectRowAtIndexPath

This is as simple as replacing any mention of allHeroes with filteredHeroes inside of these methods.

The **commitEditingStyleforRowAtIndexPath** method will require some additional changes to handle deletion. When we delete a row now, we need to ensure that the hero is deleted in both the allHeroes list and the filteredHeroes list.

```
if let index = self.allHeroes.firstIndex(of: filteredHeroes[indexPath.row]) {
    self.allHeroes.remove(at: index)
}
self.filteredHeroes.remove(at: indexPath.row)
```

Inside of the **viewDidLoad** method set filteredHeroes to be equal to allHeroes after the default heroes have been generated. This just allows us to easily start with a populated list.

```
createDefaultHeroes()
filteredHeroes = allHeroes
```

Next, we need to create the **UISeachController** and assign it to the View Controller. This can be done easily by adding the following to the **viewDidLoad** method:

```
let searchController = UISearchController(searchResultsController: nil)
searchController.searchResultsUpdater = self
searchController.obscuresBackgroundDuringPresentation = false
searchController.searchBar.placeholder = "Search All Heroes"
navigationItem.searchController = searchController

// This view controller decides how the search controller is presented
definesPresentationContext = true
```

The first step is initializing a **UISearchController**. At this stage we do not give a specialised view controller for displaying search results. This is something we will look at in future weeks.

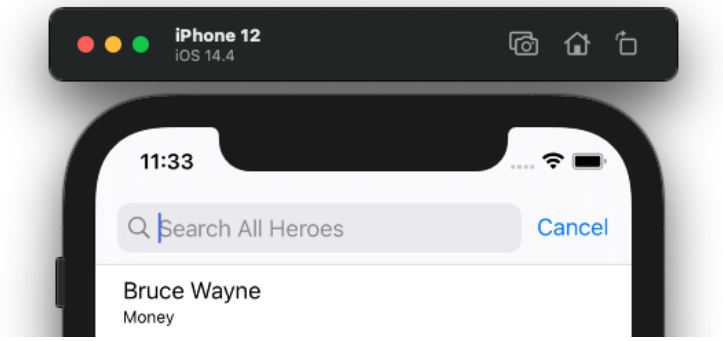
The next step is setting the **searchResultsUpdater (**delegate), which will update based on the search. This is the current class, hence self.

The **obscuresBackgroundDuringPresentation** property is set to false. Its name is self-explanatory. Try turning it off to see the effect it has.

We give the search bar placeholder text telling the user to enter a search term.

Finally, we tell our **navigationItem** that its search controller is the one we just created. This adds the search bar to the view controller. The search bar is hidden initially. Swiping down will reveal it!

If you run the application now and swipe down on the All Heroes Screen you should see the search bar appear. However, there is still no functionality for searching. We need to complete the **updateSearchResults** method.




```
guard let searchText = searchController.searchBar.text?.lowercased() else {
    return
}
```

Before starting any filtering, we want to make sure that there is search text to be accessed. The likelihood of it being an empty optional is slim but we should never assume that variables we have no control over will have a value. The search text is converted to lowercase so that we do not have to worry about case sensitivity.

The next thing to check is to see if there is a search term. This is easily done by checking to see if the string length (count) is greater than 0. If it is, then we will apply a filter, otherwise we will set the filteredHeroes to allHeroes.

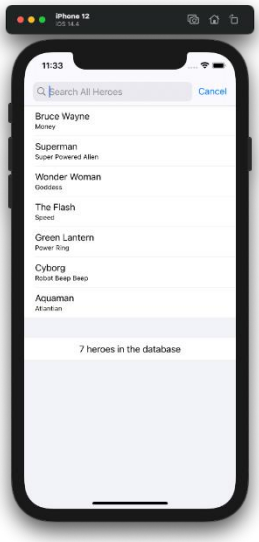
```
if searchText.count > 0 {
    filteredHeroes = allHeroes.filter({ (hero: SuperHero) -> Bool in
        return (hero.name?.lowercased().contains(searchText) ?? false)
    })
} else {
    filteredHeroes = allHeroes
}

tableView.reloadData()
```

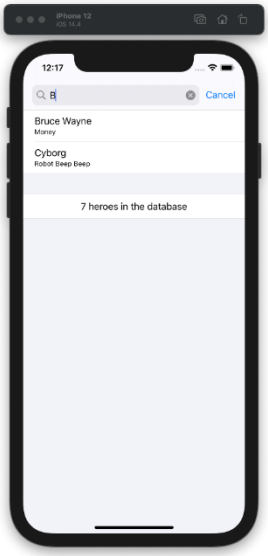
The filter method of the array class allows us to set custom filtering using a closure. In this case we are including a row into our filtered list if it contains the search text (after also making it lowercase). We use the nil-coalescing operator (??) since the name property is an optional. This allows us to provide a default case if the name is nil, and no boolean will be returned (since the contains method won't be called).

For more information on the filter method, check the documentation at the following link https://developer.apple.com/documentation/swift/sequence/3018365-filter

Once changes have been made to the filtered list, the table view is reloaded, causing the UI to be updated.

With that done we can now test the search functionality.



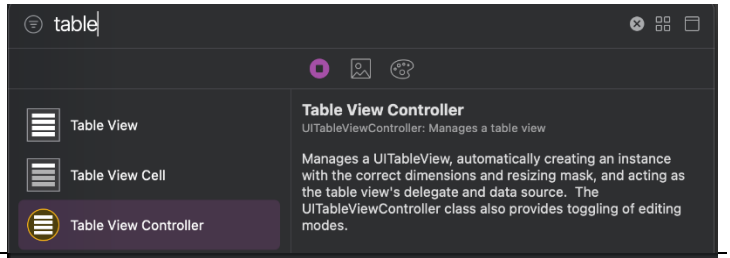
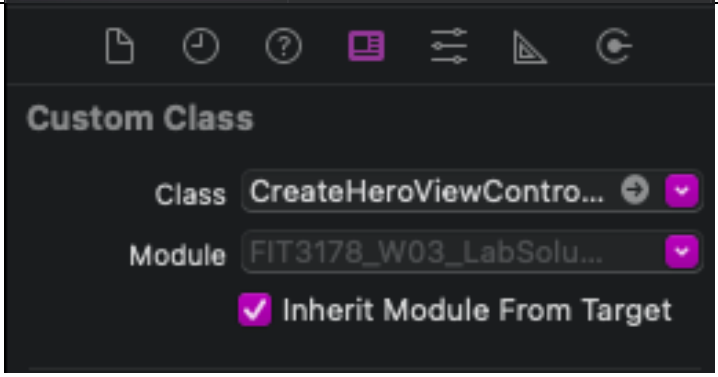


Building the Create Hero Screen:

With the All Heroes Screen done the final Screen to create is the "Create Hero Screen".

Create a **Cocoa Touch Class** for the Create Hero screen. This should be a subclass of **UIViewController**. Name it CreateHeroViewController.

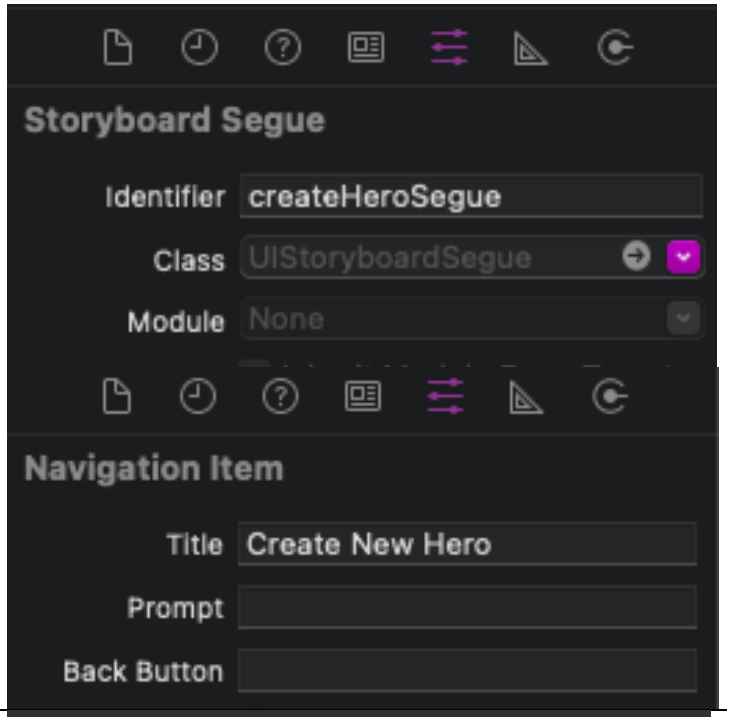
Open the Main storyboard and complete the following steps:

Go to the Objects Library. Drag a View Controller onto the Storyboard	
Select the new View Controller. Change its custom class to "CreateHeroViewController".	

Create a segue from the Add bar button on the All Heroes table view controller to the new View Controller.

Select the segue and change its Identity to "createHeroSegue"

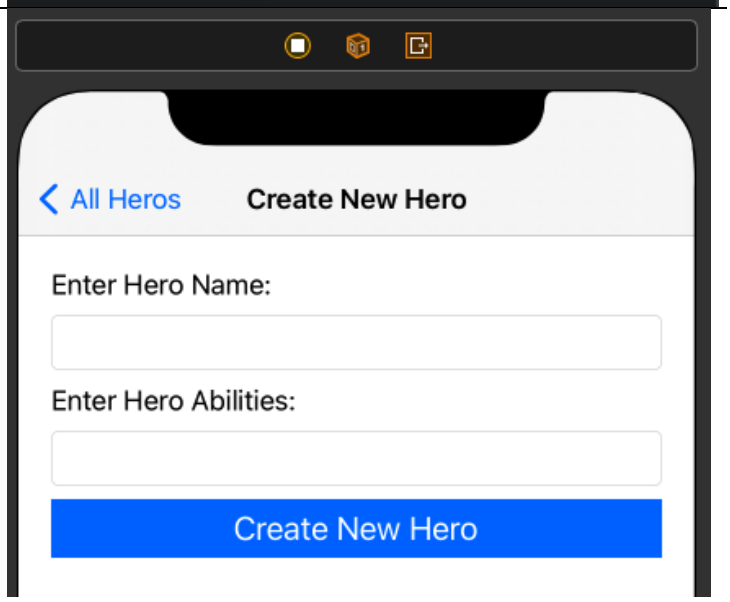
Select the Navigation Item in the new View Controller and set its title to “Create New Hero”.



Add the following to the View Controller:

Label with the text "Name"
A Text Field
Label with the text "Abilities"
Another Text Field
A button with the text "Create Hero"

Set up appropriate constraints for each of these UI elements



Finally select the View Controller again.

Open the Assistant Editor.

Link the first text field as an outlet with the name "nameTextField"

Link the second text field as an outlet with the name "abilitiesTextField"

Link the button as an action with the name "createHero"

Coding the CreateHeroViewController Class:

Open the CreateHeroViewController.swift file and add one additional property:

```
weak var superHeroDelegate: AddSuperHeroDelegate?
```

The remainder of the class additions are within the createHero action method that is linked to the UI Button. For this method we should validate the user input and ensure they have entered valid information. Currently we will just ensure name and abilities are not empty.

```
if let name = nameTextField.text, let abilities = abilitiesTextField.text,
!name.isEmpty, !abilities.isEmpty {
    let hero = SuperHero(name: name, abilities: abilities)

    let _ = superHeroDelegate?.addSuperHero(hero)
    navigationController?.popViewController(animated: true)
    return
}
```

With this code, we do not currently check to ensure that the hero is successfully added. With the current logic, there is no way for this action to fail. Try adding extra functionality to stop heroes with the same name being added!

If the name or abilities fields are empty we should return an error to the user stating what the issue is and how to fix it.

```
var errorMsg = "Please ensure all fields are filled:\n"
if nameTextField.text == "" {
      errorMsg += "- Must provide a name\n"
}
if abilitiesTextField.text == "" {
      errorMsg += "- Must provide abilities"
}
displayMessage(title: "Not all fields filled", message: errorMsg)
```

With that, a final change is needed in All Heroes Table View Controller to bring the application together!

Extending the AllHeroesTableViewController Class:

Open the AllHeroesTableViewController.swift file. The last couple of changes that need to be made are conforming to the AddSuperHeroDelegate and setting up the prepare for segue method.

Firstly, the class definition.

```
class AllHeroesTableViewController: UITableViewController,
UISearchResultsUpdating,
        AddSuperHeroDelegate {
```

Second, implement the addSuperHero method.

```
tableView.performBatchUpdates({
    // Safe because search can't be active when Add button is tapped.
    allHeroes.append(newHero)
    filteredHeroes.append(newHero)

    tableView.insertRows(at: [IndexPath(row: filteredHeroes.count - 1, section:
SECTION_HERO)],
            with: .automatic)
    tableView.reloadSections([SECTION_INFO], with: .automatic)
}, completion: nil)
return true
```

We return true always. No validation is done here. Consider adding additional functionality to stop duplicate heroes being created.

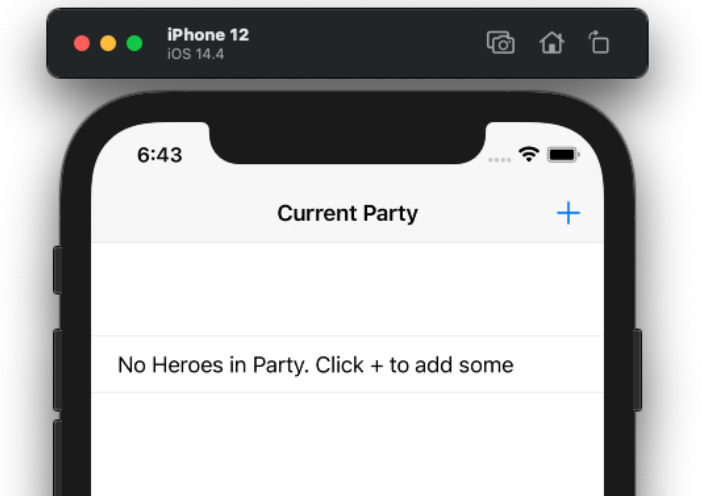
Finally, the prepareForSegue method.

```
if segue.identifier == "createHeroSegue" {
    let destination = segue.destination as! CreateHeroViewController
    destination.superHeroDelegate = self
}
```

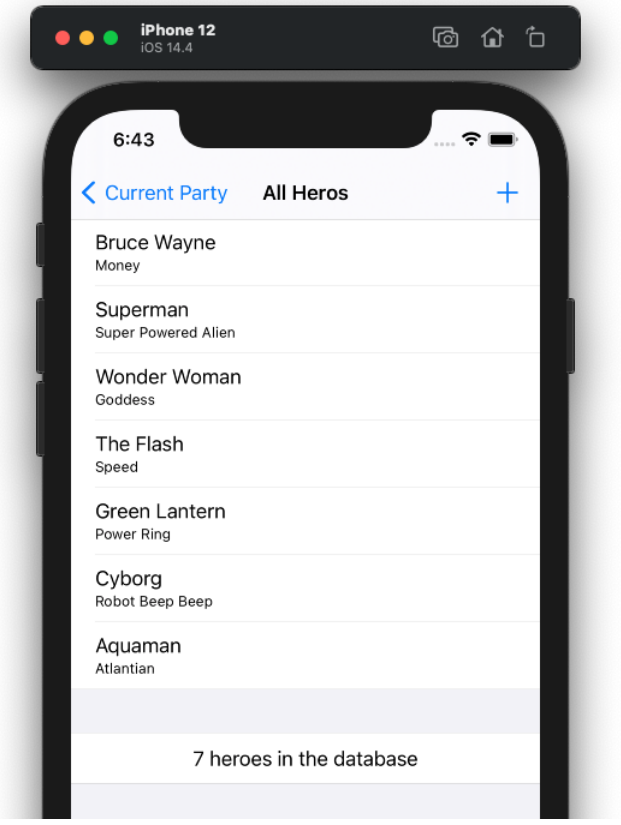
And with that all code changes are done! It is time for final testing of the application.

Testing the Application:

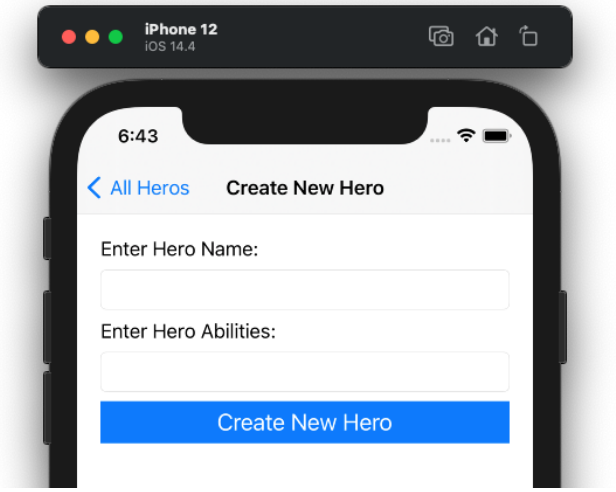
At this point we should be ready to test the application. Compile and run the app in the simulator. Assuming all is correct you should be greeted by the initial Current Party Screen.



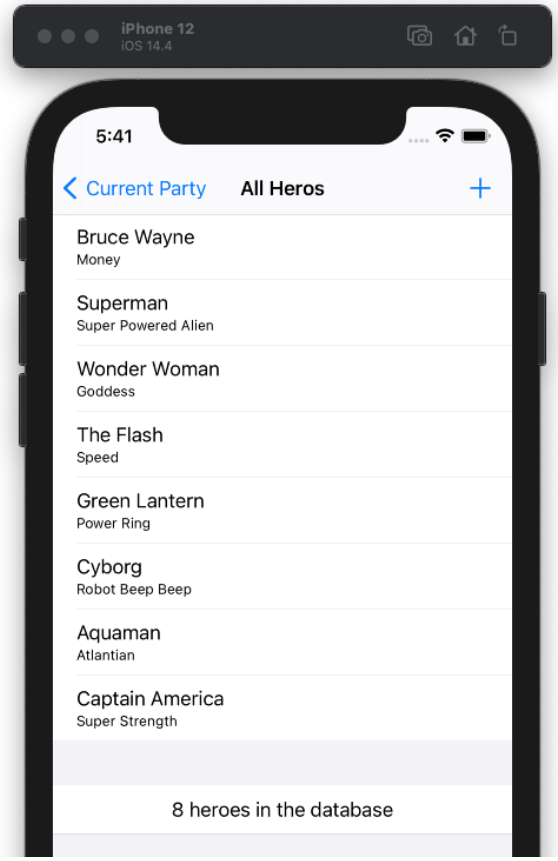
Click Add in the top right corner to go to the All Heroes screen.



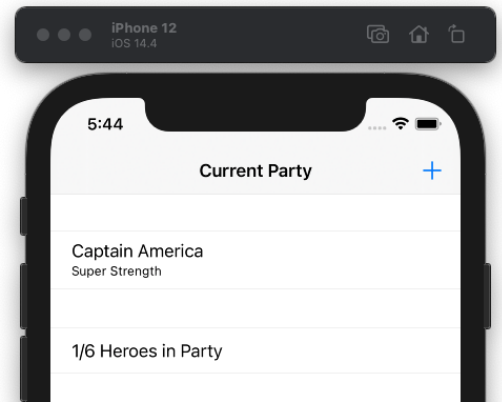
Clicking Add again will take you to the Create Hero screen.



Create a hero. If the name and abilities are entered, you will be taken back to All Heroes with the new hero now displayed.



Clicking on any hero will take you back to the current party with that hero now present.



You may also notice that when returning to All Super Heroes the Captain America hero we created is gone! When the All Heroes screen disappears the user-added heroes are lost. When we tap Add again it reloads only the default heroes. This is the expected behaviour (for now). It is something we will fix in next week's lab.