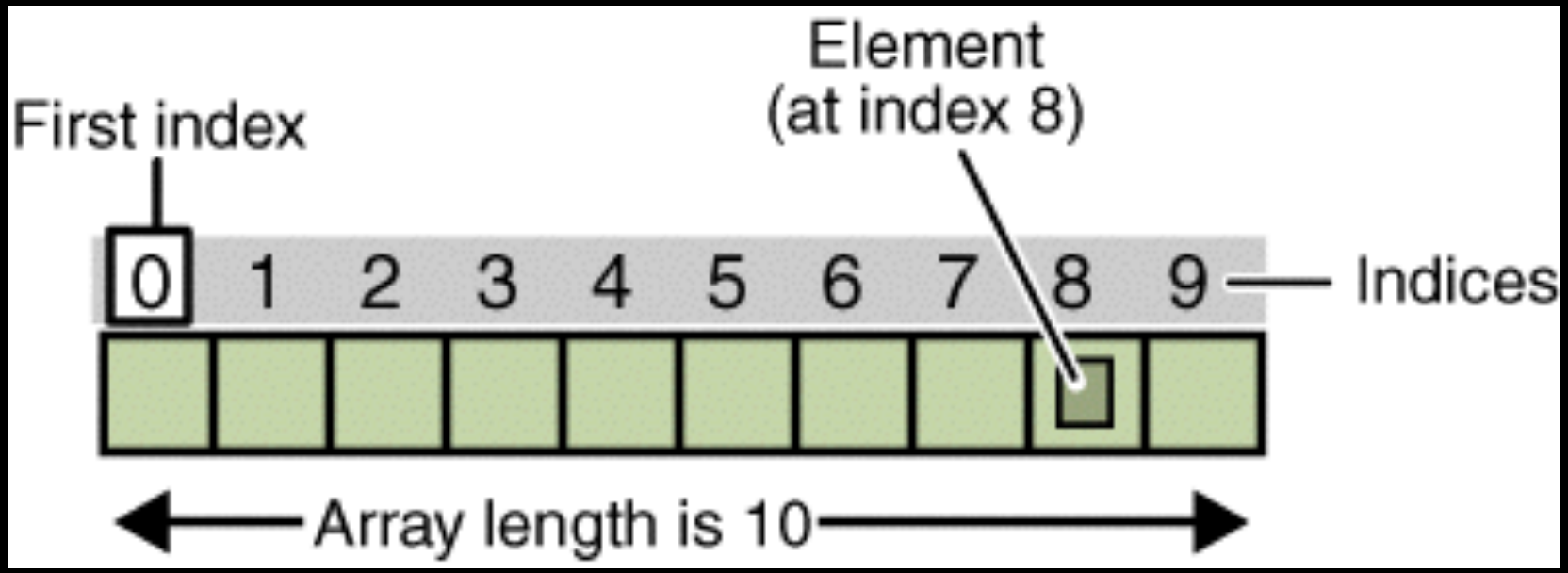
iOS Foundations II
Day 3

- Previous Class Review
- TableViews

Arrays

- Arrays are very important to understand
- Arrays are used in virtually ever app ever made!!!!
- You typically use arrays any time you have a collection of similar objects or data and you want to perform similar operations on them.
- So its considered a collection type.
- Arrays are ordered, which is important.

Arrays


Element
(at index 8)
First index
0
1
2
3
4
5
6
7
8
9
Indices
Array length is 10

Creating an Array in Swift

- An array is considered a type, and the way to signify an array type in Xcode is just []
- But thats not quite complete, because inside the brackets you need to also state the type of objects you are going to be putting inside the array.
- So [String] is the type of an array that holds Strings. and [UIImage] is the type of an array that holds images.
- To actually instantiate an array, you use () after the closing square bracket to create the array:

```
var myNames = [String]()
```

Demo

Adding things to an array

- There are two ways an object can get inside an array in Swift:
  - Arrays have a method called append, which takes in one parameter, the object you want to add to the end of the array:

```
var myNames = [String]()
myNames.append("Brad")
```

  - When you initial create an array, you can use a special shorthand syntax where you place all the objects in the brackets of the array you are creating:

```
let names = ["Brad","David","Ryan"]
```

Demo

Retrieving objects from an array

- Retrieving objects from an array uses a special syntax that also involves []
- You retrieve objects from arrays by their index number.
- Remember the index starts at 0, not 1! (forgetting this fact is pretty common, and leads to the classic ‘off by one’ error)

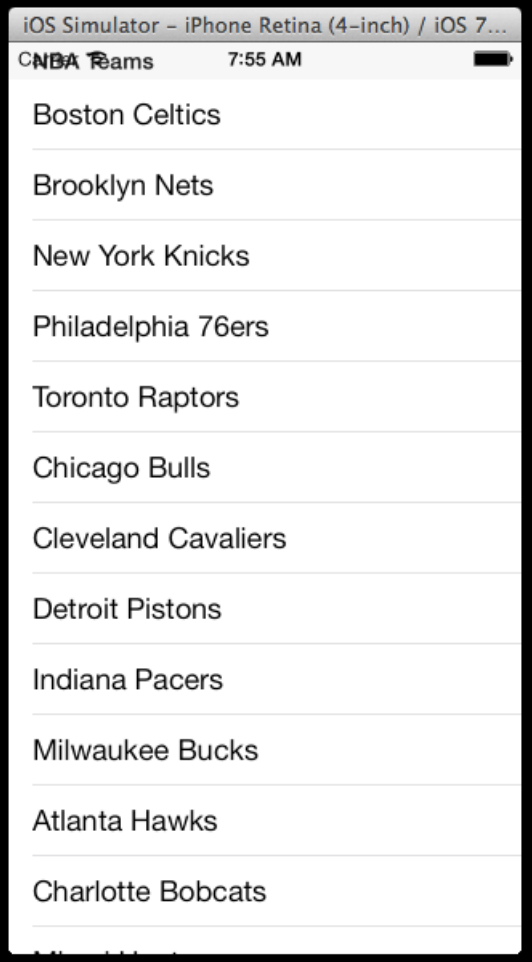
```
let names = ["Brad","David","Ryan"]

let brad = names[0]
let david = names[1]
let ryan = names[2]
```

Arrays with let vs var

- If you declare an array with a let, it makes it immutable
- This means you can only give it initial objets on the line the array is declared
- After that, the array can NEVER be modified again
- Immutable arrays are nice and safe. You have 100% confidence no one is going to mess with it, because the haters cant.
- When you declare an array with var, it is mutable, which means it can be modified at any time (ie objects removed or added to it)

UITableView



UITableView

- “A Tableview presents data in a scrollable list of multiple rows that may be divided into sections.”
- A Tableview only has one column and only scrolls vertically.
- A Tableview has any number of sections, and those sections have any number of rows. A lot of the time you will just have 1 section and its corresponding rows.
- Sections are displayed with headers and footers, and rows are displayed with Tableview Cells, both of which are just a subclass of UIView (Inheritance!)
- If you have used an iPhone or iPad, you have used a Tableview.

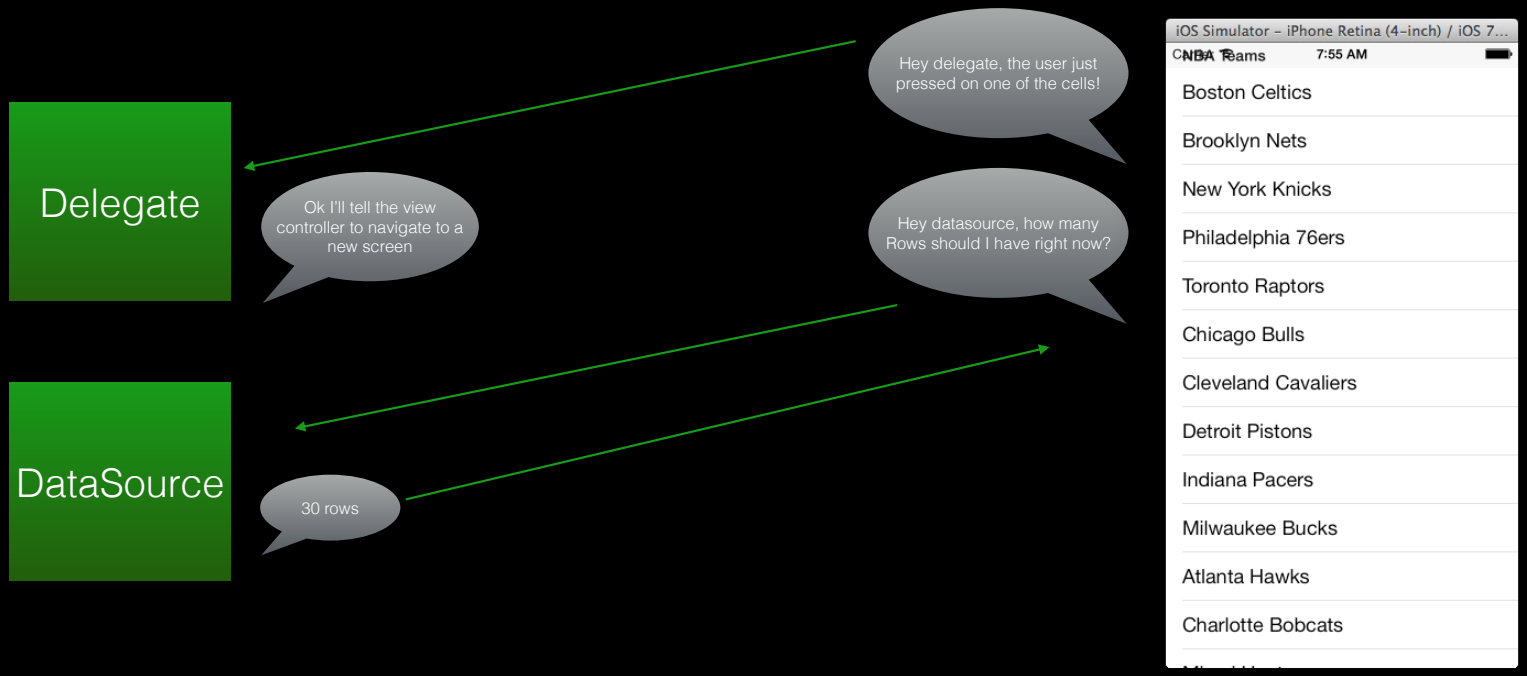
How do Tableviews work?

- Tableview's rely on, or **delegate**, onto other objects in order to function.
- A Tableview asks another object how many rows and sections it should display, and what to display in each row.
- A Tableview notifies another object when it is interacted with, like when the user selects a certain cell (aka row), or when the user scrolls the list.
- So what is this pattern of one object relying on another called?

Delegation

- Tableviews rely on the design pattern of delegation to get their job done.
- Picture time:

TableViews and Delegates

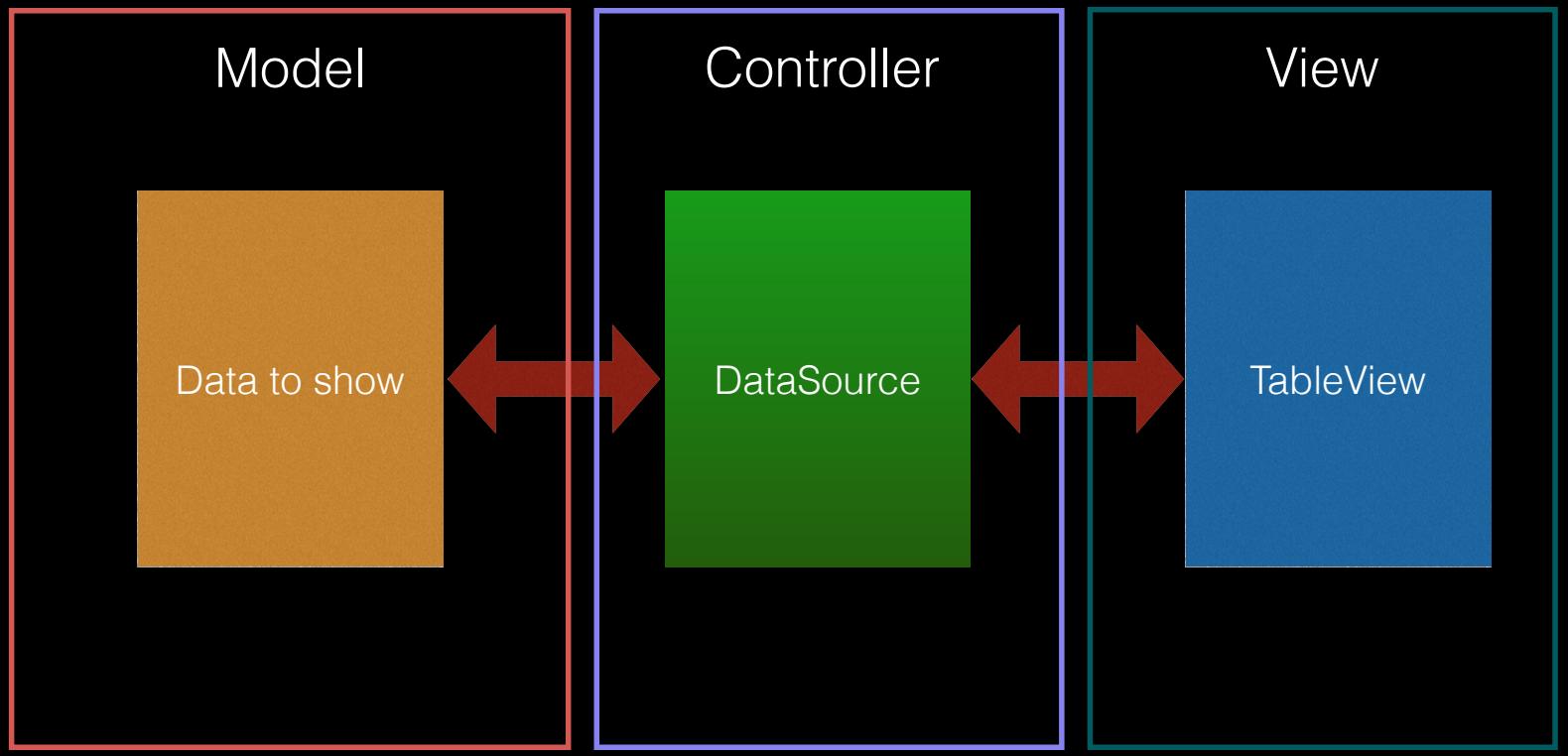


A Tableview has 2 separate delegate objects. One actually called **delegate** and one called **data source**. The data source is just a specialized form of a delegate that is for data retrieval only.

A note on delegation vs datasource

- The design pattern is called **Delegation**.
- **Delegation** is defined as “a simple and powerful design pattern in which one object in a program acts on behalf, or in coordination with, another object”
- So the object that is acting on behalf, or in coordination with another object, is called the **delegate**.
- For example, the table view’s **delegate** acts on behalf on the table view whenever the user interacts with the table view.
- A **data source** is almost identical to a delegate. The key difference is that the **data source** is control of data, while a **delegate** is normally in control of some sort of user interface attribute or event.

TableViews and its data source



Delegation helps greatly with the separation of concerns in our MVC environment

Delegation

- Another benefit of delegation is to allow you to implement custom behavior without having to subclass.
- Apple could have designed UITableView's api so you would have to subclass UITableView, but then you would have to understand UITableviews in a lot more detail(which methods can I override? Do I have to call super? omfg?)
- Instead, you can just create a custom delegate/datasource object, which will do the customizations for the table view.
- Delegation is used extensively in a large portion of Apple's frameworks.
- **Delegates/Datasources must adopt the protocol of the object they are being delegated from.**

Protocols

- When an object wants to be something's delegate (or datasource), it needs to first conform to a protocol.
- Conforming to a protocol is like saying "hey, i can do all of these things, so let me be your delegate/datasource!"
- Sort of like signing a contract.
- To have your custom class conform to a protocol, just add the name of the protocol after the classes superclass:

```
class RootVC: UIViewController, UIPageViewControllerDelegate {
```

TableView Datasource Protocol

- A tableview requires 2 questions to be answered (aka methods to be implemented) by their datasource. At the very least the tableview needs data to display. It doesn't have to respond to user interaction, so the delegate object is completely optional.
- `tableView(numberOfRowsInSection:) How many rows am I going to display?`
- `tableView(cellForRowAtIndexPath:) What cell do you want for the row at this index?`
- Number of sections is actually optional, and is 1 by default.

The Bottom Line

- So the bottom line is, if you want your view controller to be the data source for your table view, you must do these 2 things:
  1. Declare that your view controller class conforms to the UITableViewDataSource protocol.
  2. Implement the required methods from the UITableViewDataSource protocol that we just looked at.

Demo

tableView(cellForRowAtIndexPath:)

- lets look at the data source cellForRow method in depth:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
  NSIndexPath) -> UITableViewCell {

  let cell = tableView.dequeueReusableCellWithIdentifier("MyCell",
    forIndexPath: indexPath) as! UITableViewCell

  cell.backgroundColor = UIColor.blueColor()
  cell.textLabel?.text = "Showing cell for row \(indexPath.row)"

  return cell
}
```

1) dequeue the cell

2) configure the cell

3) return the cell

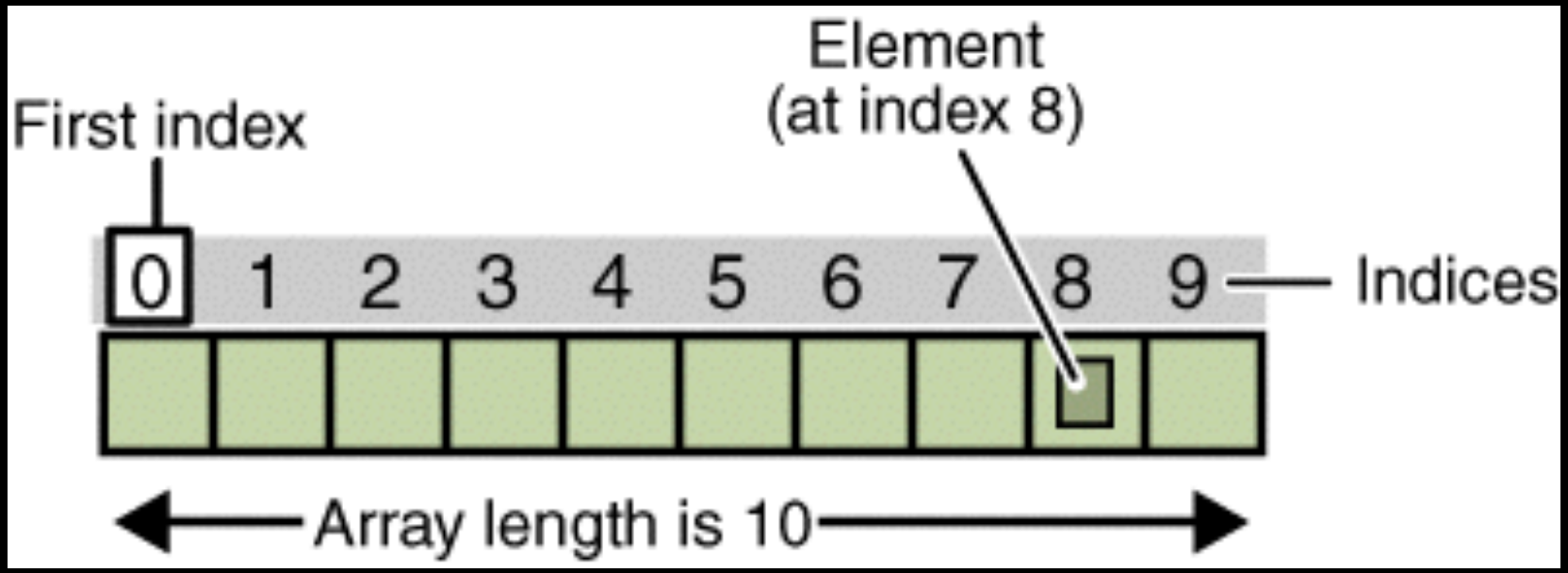
These are the 3 basic steps you will always follow when dequeueing a cell

Table View and arrays

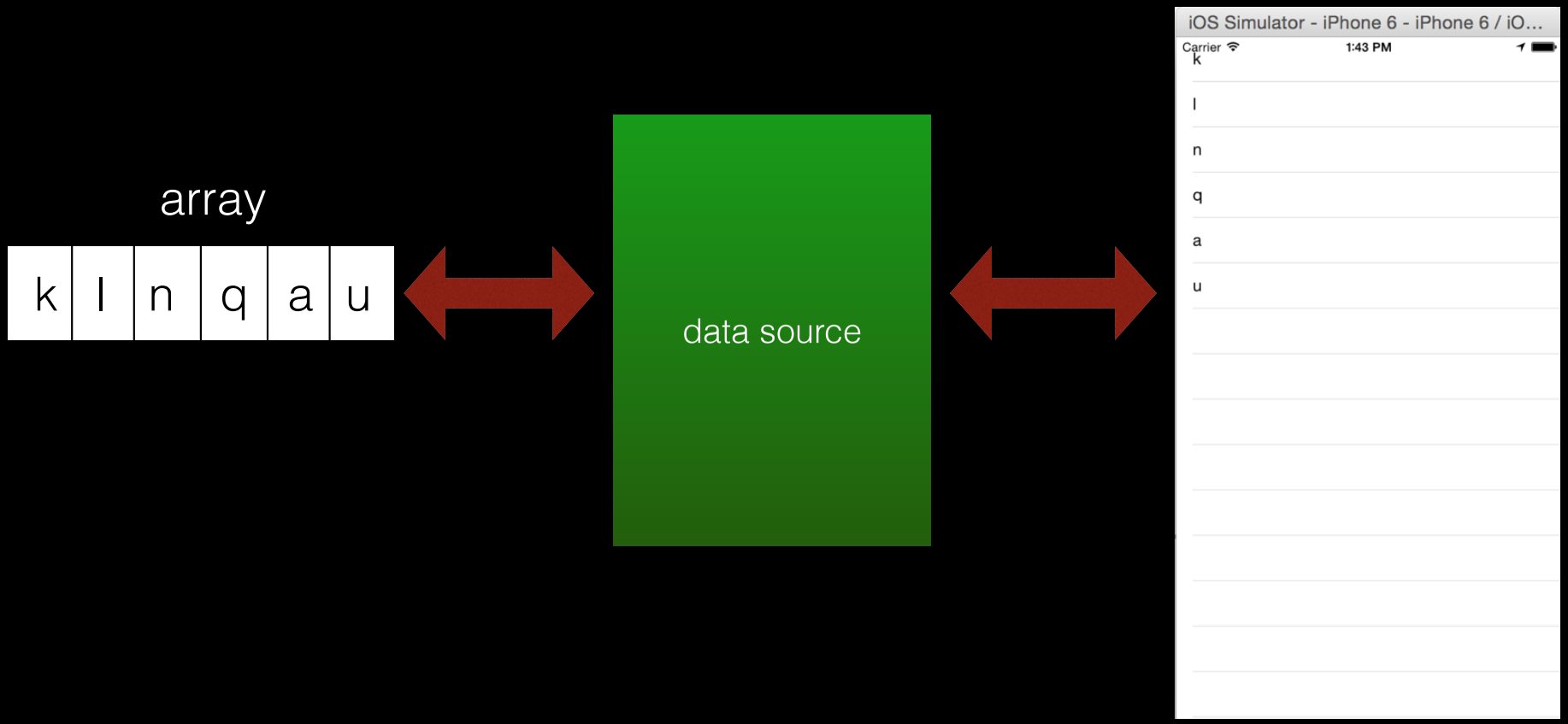
TableView+Array

- 99% of the time, table views are ‘backed’ by an array.
- So basically the table view’s data source is the middle man between the array, and where its going to be shown, the table view
- The data source uses the count of the array to figure out how many cells its going to display
- And then for each cell, it displays whatever is at that index of the array

Array Visual



TableView+Array



Demo

Cell Reuse

Reusing Cells

- Table views are very efficient
- They can be asked to show lists of thousands, and even millions of objects
- They do this by only generating enough cells to fill the screen, and then reusing cells as they are scrolled off the screen, immediately placing them back on screen.
- So showing an array of ten thousand strings, the table view only has to generate 7 or 8 cells (enough to fill the screen, depends on screen size and row height)

Properly reusing cells

```
let cell = tableView.dequeueReusableCellWithIdentifier("MyCell",
  forIndexPath: indexPath) as! UITableViewCell
```

- When you dequeue a cell, you are asking the table view for a cell. The table view pulls a cell from its ‘reuse’ queue.
- The identifier you pass in must match the reuse identifier you set on storyboard
- You need the as! UITableViewCell because the method dequeueReusableCellWithIdentifier returns a type of AnyObject. as! is a forced downcast, which is like saying “I know this method is actually going to return something of this type. In fact, I’m so confident, crash the app if it doesn't”

Demo

Index Paths

- Table view use Index Paths to describe a section and row.
- The class NSIndexPath is a simple struct that has 2 important properties:
  - .section : An Int that contains the section number
  - .row: An Int that contains the row number

Demo

TableViews Gotchas

1. Did you give the tableview a data source? (by setting up an outlet to it and then settings it datasource property, or wiring it up via storyboard)
2. Does your view controller conform to the data source protocol?
3. Did you implement the required 2 methods? (how many rows, and what cell for each row)
4. Did you drag out a tableview cell, and then set its reuse identifier?
5. Does the reuse identifier in your storyboard match the one in your code?

GitHub

- Github is a repository web-based hosting service.
- Github hosts people's repositories, and those repositories can be public or private.
- The repositories on Github are just like the repositories that you have on your own machine, except Github's web application has additional features that makes working in a team much easier.

- https://help.github.com/articles/set-up-git/

Remotes

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere.
- So when you have a repository on Github, it is considered a remote repository.
- When you import a directory into a git project with git init, eventually you may want to create a remote repository on Github and push to it. This gives you a backup in the cloud, and also lets your work be seen by others.
- Or you can clone an already existing remote repository (your own or someone elses) to get a local copy of the remote repository on your machine.

GitHub and Git

- Github and your local git install have 2 ways of communicating:
  - https (recommended)
  - ssh
- Both of these forms of communication require authentication.
- For https, it is recommended you use a credential helper so you don't have to enter in your credentials every time you interact with a remote repository.
- For SSH, you can generate SSH keys on your computer and then register those SSH keys to your Github account.
- Lets all follow the steps at https://help.github.com/articles/set-up-git/ together to get our github & git properly setup.

demo

Git Remote Commands

- git remote -v shows you all the remotes you have configured for your local repository on your machine
- use git remote add <nickname> <url> to add a remote repo
- after committing, use git push <nickname> <branch> to push your committed changes to your remote
- use git pull to automatically fetch and merge a remote branch into your current local branch

Demo