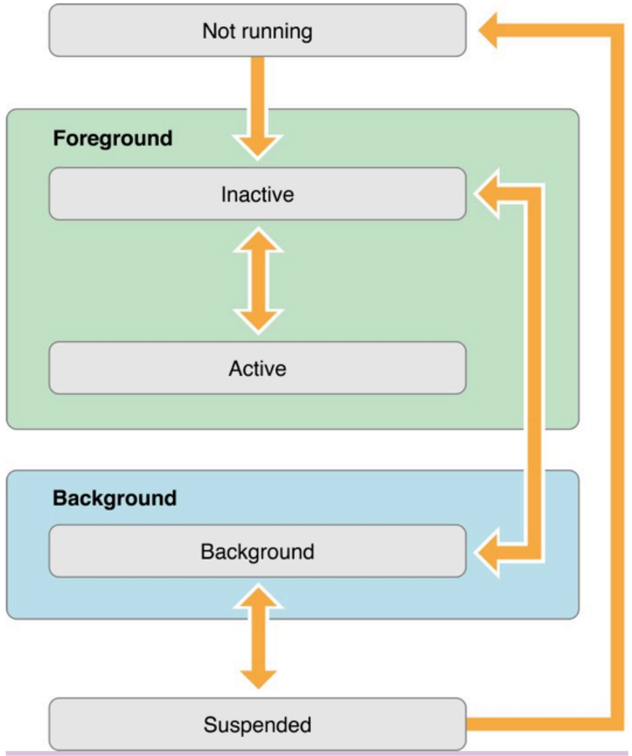
Memory management, Lifecycle

Multiscreen App, Table View

App Lifecycle

App State	Description
Not running	The app has not been launched or has been terminated, either by the user or by the system.
Inactive	The app is running in the foreground but isn't receiving touch events. (It may be executing other code though.) An app usually stays inactive only briefly as it transitions to a different state.
Active	The app is running in the foreground and receiving events. In the active state, an app has no special restrictions placed on it and should be responsive to the user.
Background	The app is executing code but is not visible onscreen. When the user quits an app, the system moves the app to the background state briefly before suspending it. At other times, the system may launch the app into the background (or wake up a suspended app) and give it time to handle specific tasks. For example, the system may wake an app so that it can process background downloads, certain types of location events, remote notifications, and other events. An app in the background state should do as little work as possible.



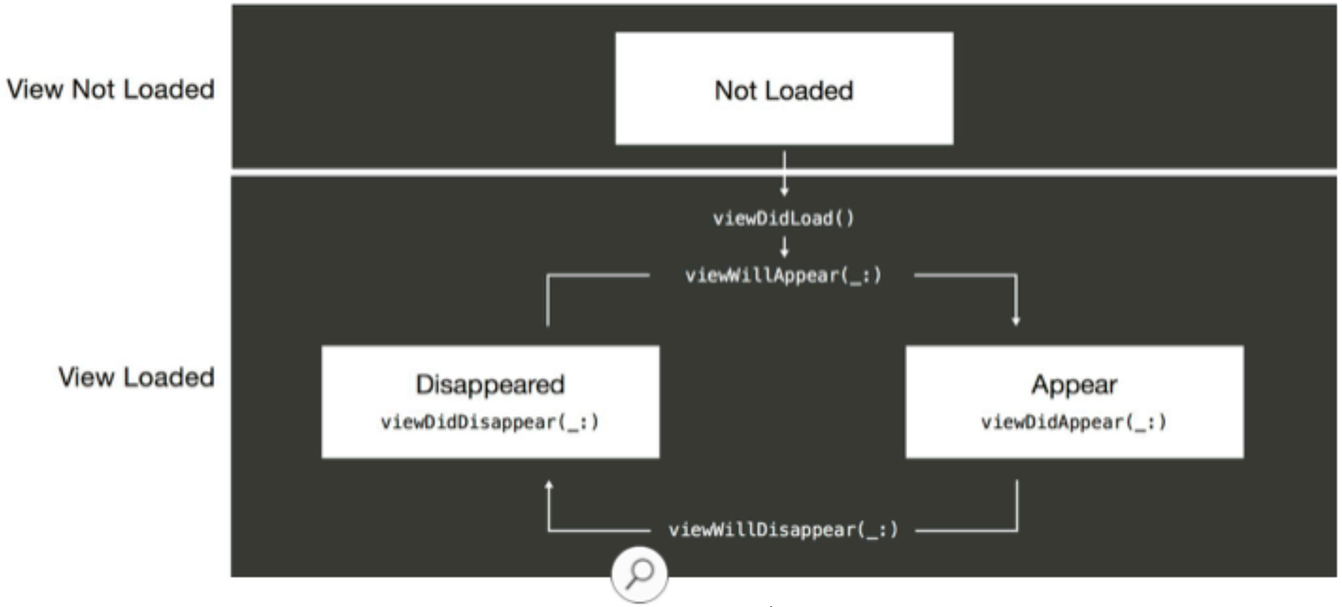
App Delegate methods

- Did Finish Launching
- Will Resign Active
- Did Enter Background
- Will Enter Foreground
- Did Become Active
- Will Terminate

View Controller Lifecycle

In iOS, view controllers can be found in one of several different states:

- View not loaded
- View appearing
- View appeared
- View disappearing
- View disappeared



viewDidLoad()

After you've instantiated a view controller, whether from a storyboard or programmatically, the view controller will load the view into memory.

After a view controller has finished loading its particular views, its **viewDidLoad()** function is called

Other types of setup tasks to perform in **viewDidLoad()** include additional initialization of views, network requests, and database access.

View Will Appear and View Did Appear

After **viewDidLoad()**, the next method in the view controller life cycle is **viewWillAppear(_:)**. This is called right before the view appears on the screen.

This is an excellent place to add work that needs to be performed before the view is displayed.

viewDidAppear(_:) is called after the view appears on the screen.

Use the **viewDidAppear(_:)** method for starting an animation or for other long-running code, such as fetching data.

View Will Disappear and View Did Disappear

viewWillDisappear(_:) is called before the view disappears from the screen.

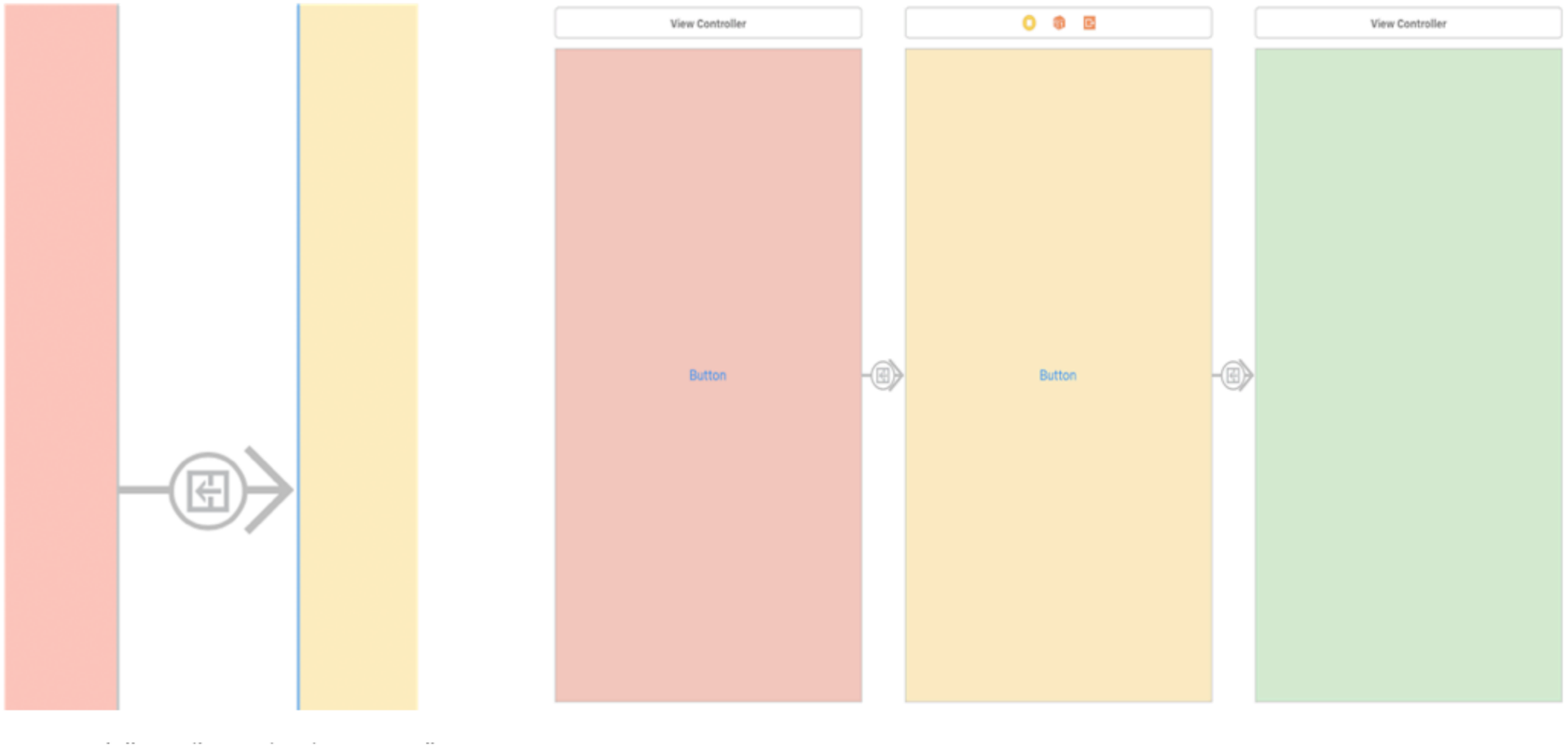
You can use the **viewWillDisappear(_:)** method for saving edits, hiding the keyboard, or canceling network requests.

viewDidDisappear(_:) is called after the view disappears from the screen.

This method gives you an opportunity to stop services related to the view, for example, playing audio or removing notification observers.

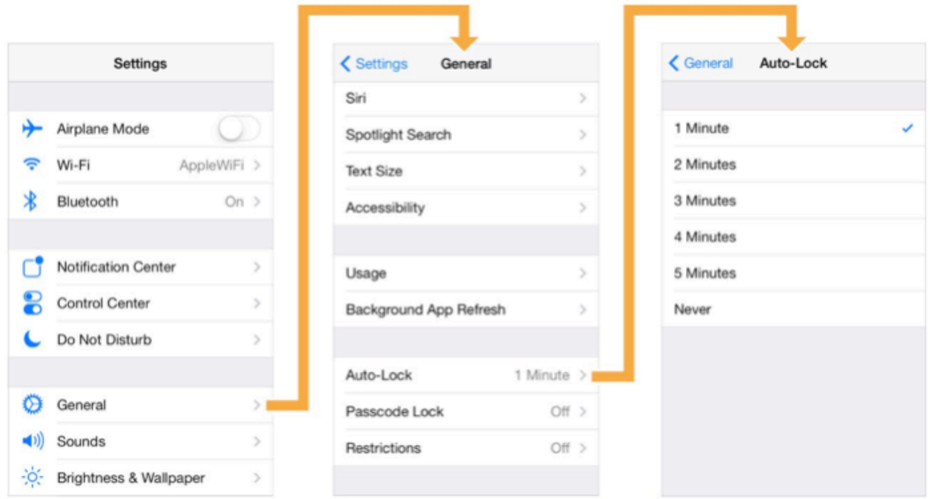
Segue

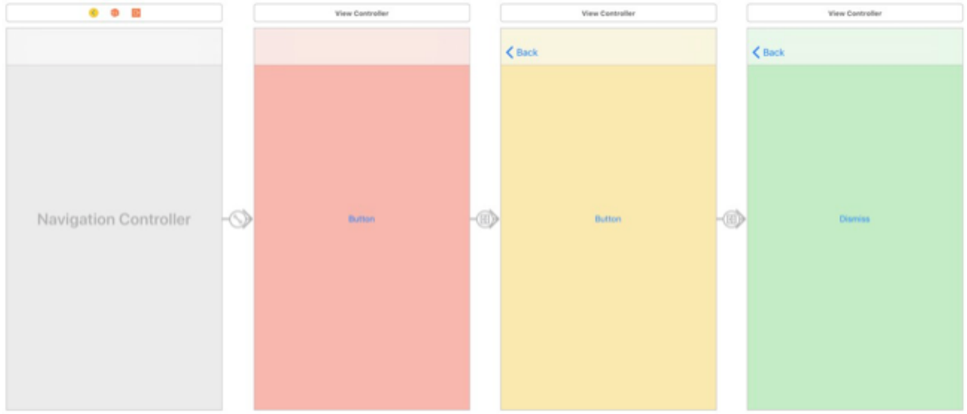
A segue defines a transition from one view controller to another



Navigation Controller

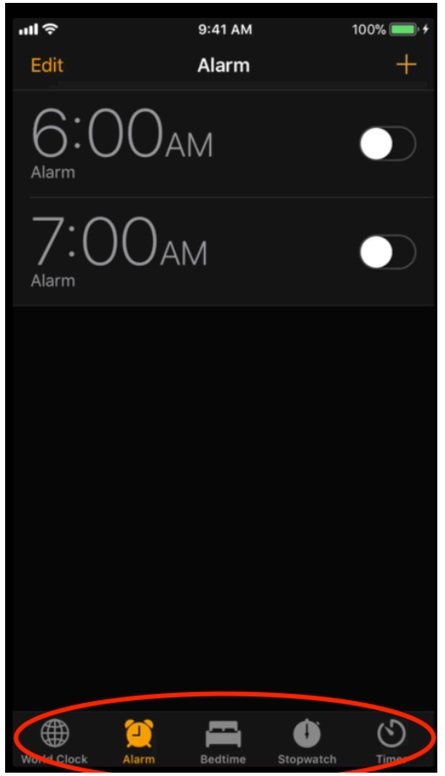
Navigation controllers manage the stack of view controllers and provide the animations when navigating between related views.





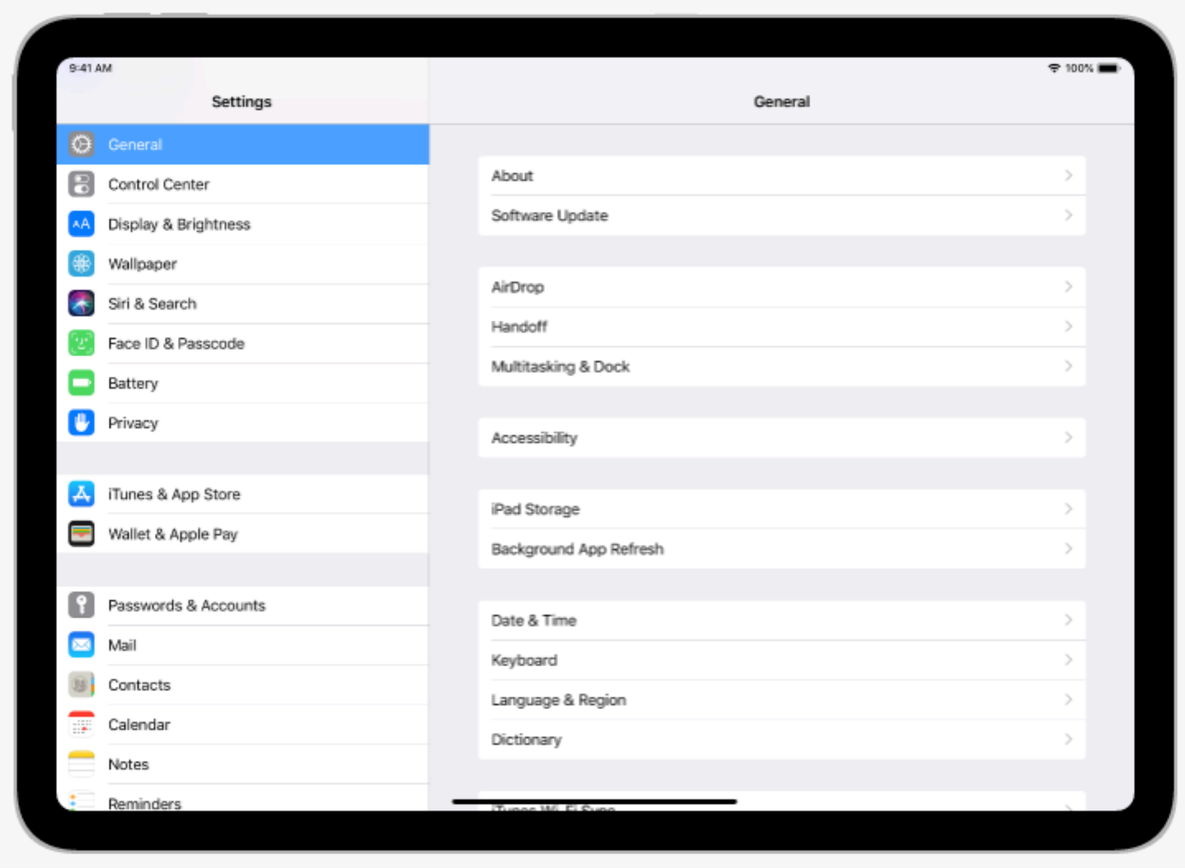
Tab Bar Controller

A tab bar controller allows you to arrange your app according to distinct modes or sections



Split View

A split view manages the presentation of two side-by-side panes of content, with persistent content in the primary pane and related information in the secondary pane



ARC & Memory management

Swift uses *Automatic Reference Counting* (ARC) to track and manage your app's memory usage.

- Reference counting applies only to instances of classes.

- Structures and enumerations are value types, not reference types, and are not stored and passed by reference.

ARC tracks how many properties, constants, and variables are currently referring to each class instance.

ARC will not deallocate an instance as long as at least one active **strong** reference to that instance still exists.

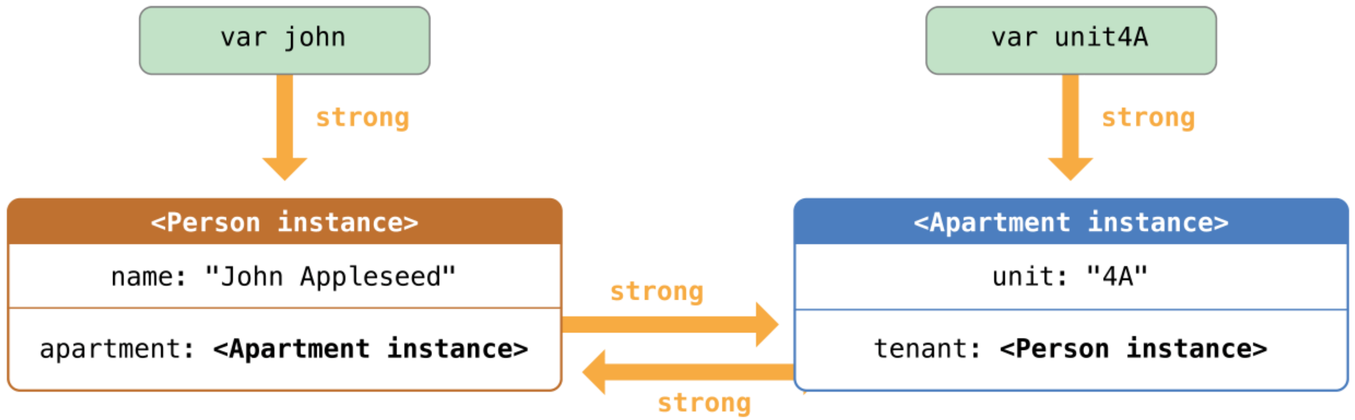
- **Strong reference** - The reference is called a "strong" reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains.

Strong Reference Cycles

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

```swift
var john: Person?
var unit4A: Apartment?
```

```swift
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

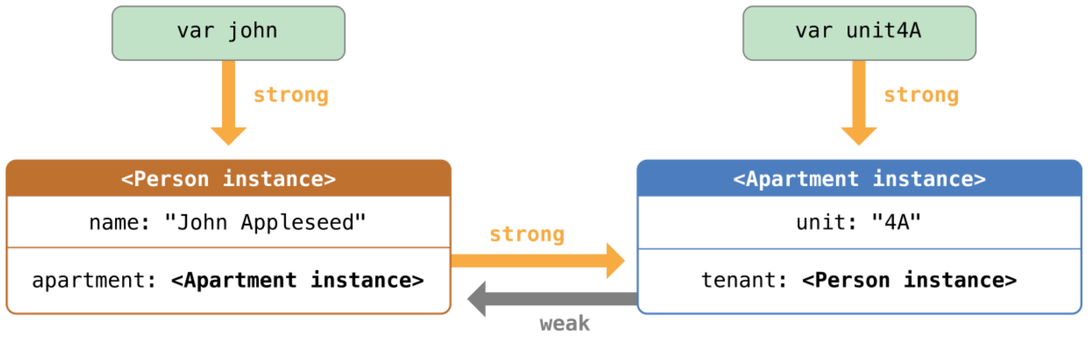


Resolving Strong Reference Cycles Between Class Instances

Swift provides two ways to resolve strong reference cycles : weak **references** and **unowned** references.

- A ***weak reference*** is a reference that does not keep a strong hold on the instance it refers to.
- ARC automatically sets a weak reference to **`nil`** when the instance that it refers to is deallocated
- They are always declared as variables of an **optional** type.

```
weak var tenant: Person?
```

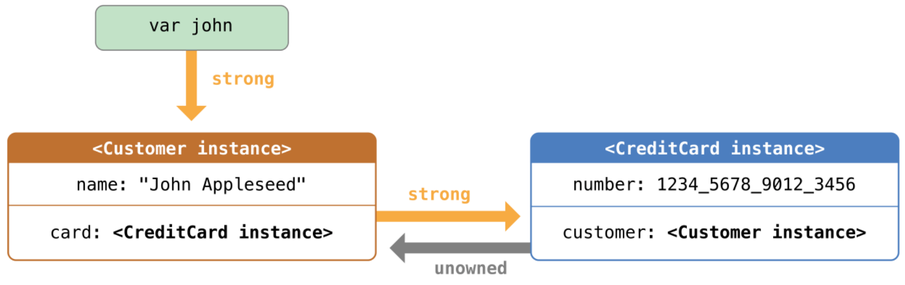


Unowned References

unowned reference does not keep a strong hold on the instance it refers to,

- Is used when the other instance has the same lifetime or a longer lifetime.
- An unowned reference is expected to always have a value.
- ARC **never** sets an unowned reference's value to **`nil`**.

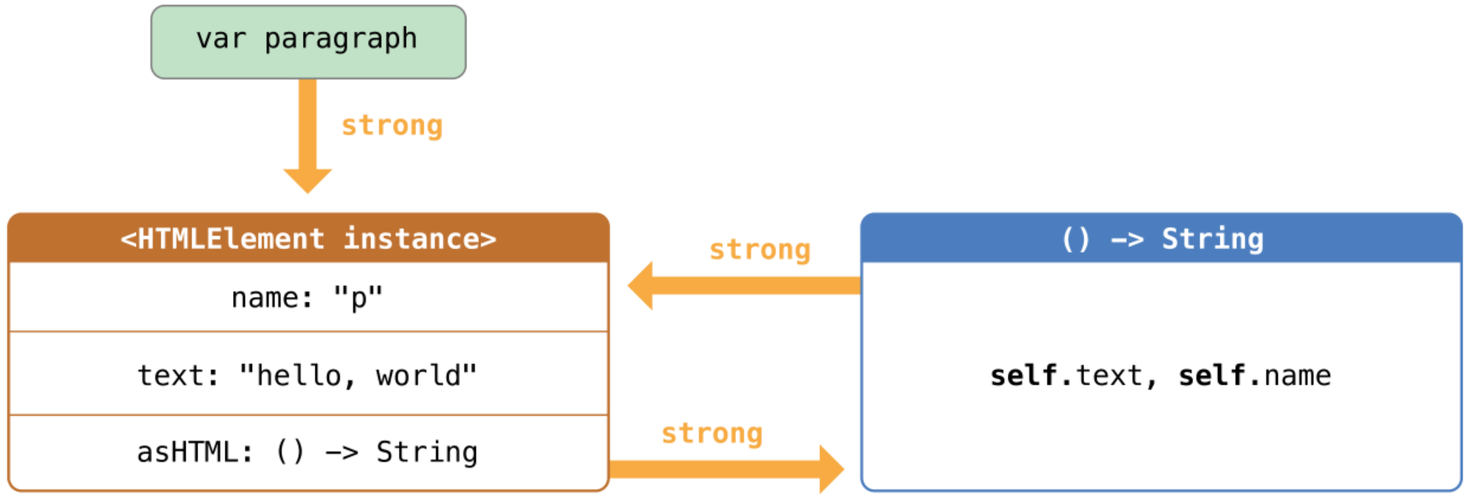
```swift
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}
```



Resolving Strong Reference Cycles for Closures

```swift
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

```

```swift
let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<\(heading.name)>\(heading.text ?? defaultText)</\
  (heading.name)>"
}
print(heading.asHTML())
// Prints "<h1>some default text</h1>"
```



Capture list

You resolve a strong reference cycle between a closure and a class instance by defining a ***capture list***

```
lazy var asHTML: () -> String = {
    [unowned self] in
    if let text = self.text {
        return "<\(self.name)>\(text)</\(self.name)>"
    } else {
        return "<\(self.name) />"
    }
}
```

capture list

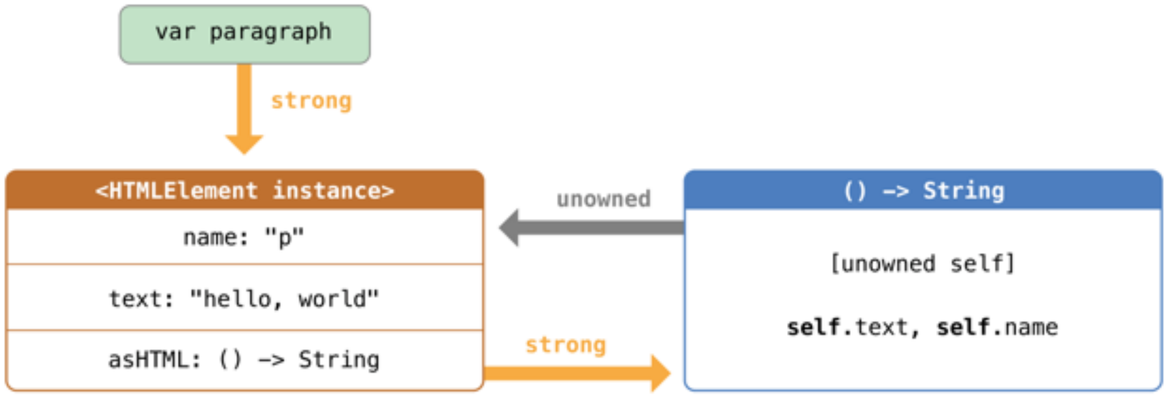
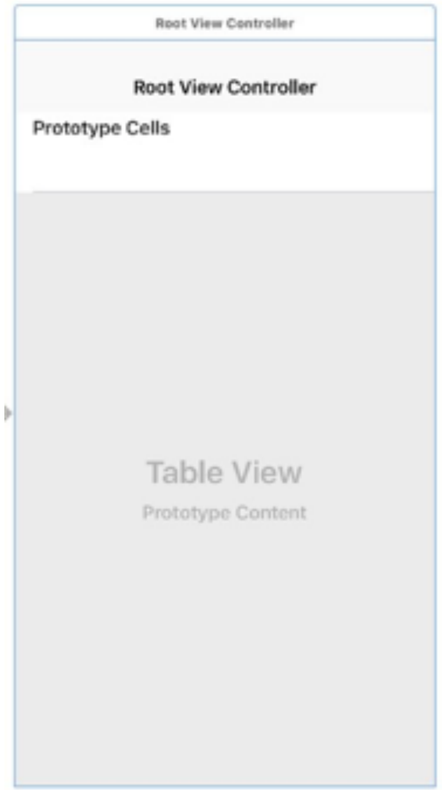


Table View

A table view is responsible for displaying one data object or tens of thousands of data objects.

- Table views present data as a scrolling, single-column list, with the option to divide the rows into sections or groups
- Each section can have a header above the first item and a footer below the last item
- The table view itself can have its own header and footer



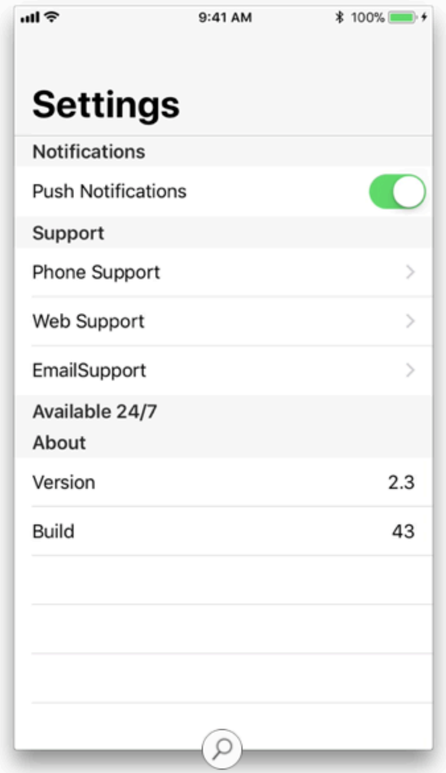


Table View Cell

Cells are reusable views that can display text, images, or any other UIView.

Each cell has an optional accessory view

In editing mode:

The UITableViewCell class defines three properties for cell content:

- *textLabel*, a UILabel for the title
- *detailTextLabel*, a UILabel for the subtitle (if there's additional detail)
- *imageView*, a UIImageView for an image

Reusable Cells

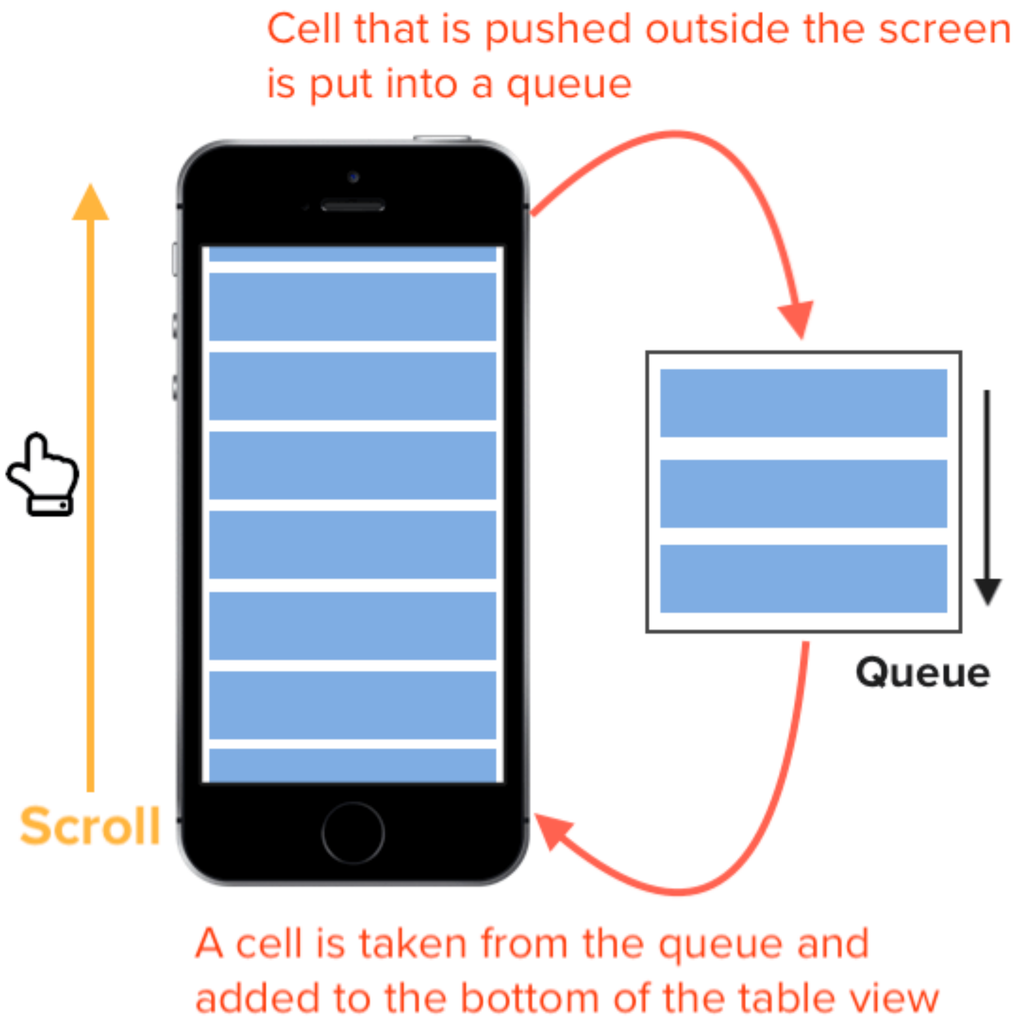


Table View Data Source

- The data source, which adopts the UITableViewDataSource protocol.
- Responsible for providing the necessary data to the table view object.

To provide this information, you'll implement methods in the data source:

```
optional func numberOfSections(in tableView: UITableView) ->
  Int
func tableView(_ tableView: UITableView, numberOfRowsInSection
  section: Int) -> Int
func tableView(_ tableView: UITableView, cellForRowAt
  indexPath: IndexPath) -> UITableViewCell
```

Table View Delegate

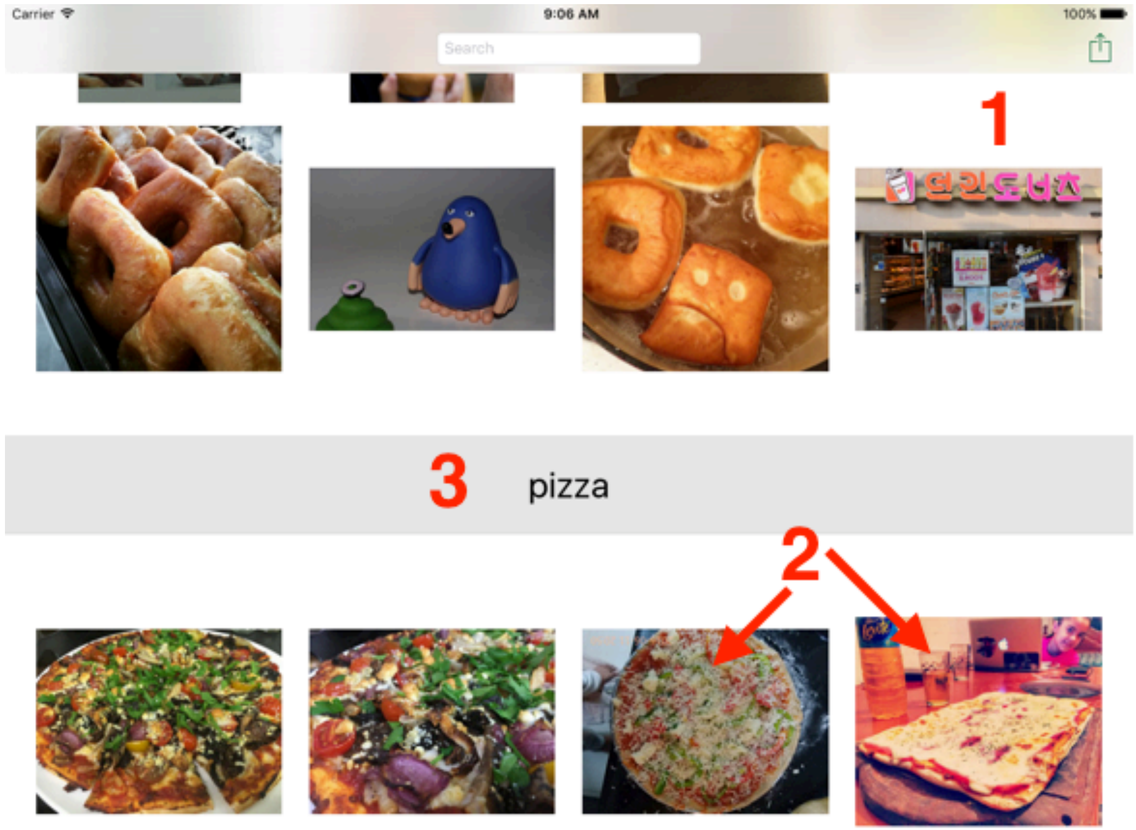
The second protocol in the table view API is the delegate.

- The delegate object, which conforms to the UITableViewDelegate protocol
- Implements methods to modify visible aspects of the table view
- Manage selections
- Support an accessory view
- Support editing of individual rows in a table behavior

Unlike the data source protocol, the delegate protocol has no required methods.

```
- tableView(_:accessoryButtonTappedForRowWith:)

- tableView(_:didSelectRowAt:)
```

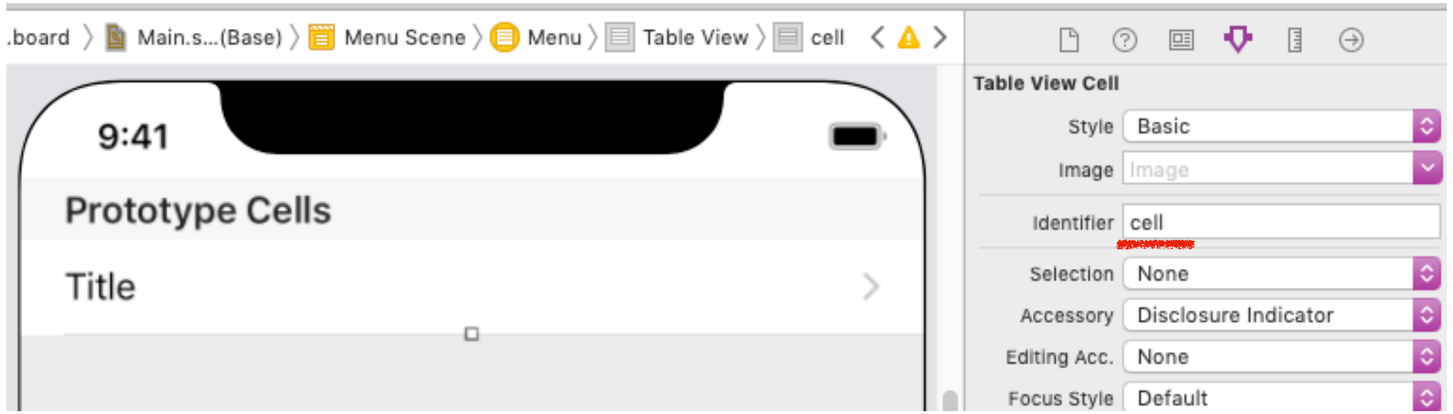
Collection View



1. **`UICollectionView`**: The main view in which the content is displayed, similar to a `UITableView`.
2. **`UICollectionViewCell`**: This is similar to a `UITableViewCell` in a table view.
3. **`Supplementary Views`**: These are commonly used for headers or footers.

Using Cell

First of all, you must set cell **identifier** in storyboard



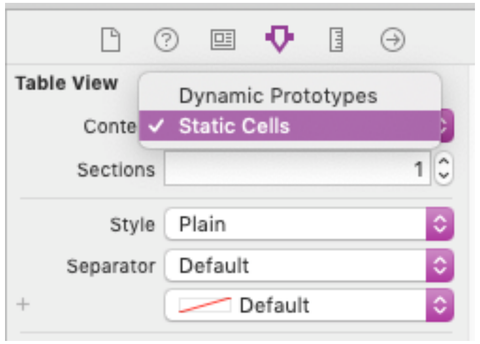
Then you should use that **identifier** to dequeue cell in **cellForRowAt**

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "cell", for: indexPath)

    cell.textLabel?.text = "Hello"

    return cell
}
```

Static Table View

Static table views are great for forms, settings, or anything where the number of rows doesn't change.

For static table views, you'll always use a table view controller.



Passing Data

Data can be passed forward to the new screen and back to the previous screen

Common ways to pass data to the new screen:

- **Prepare for segue**
- **Patterns elements (Coordinator, Router, Fabric Method)**

Common ways to pass data to previous screen:

- **Delegation**
- **Closures**
- **Notifications**
- **Sharing reference**
- **Singleton**
- **Patterns elements (Coordinator…)**

To New Screen

Via segue:

```
class FirstViewController: UIViewController {

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        guard segue.identifier == "toSecondController" else { return }

        let secondController = segue.destination as! SecondViewController
        secondController.user = User()
    }
```

Via factory method:

```
static func create(with user: User) -> SecondViewController {

    let storyboard = UIStoryboard(name: "Main", bundle: nil)

    let controller = storyboard.instantiateViewController(withIdentifier: "SecondViewController") as! SecondViewController

    controller.user = user

    return controller
}
```

```
let secondVC = SecondViewController.create(with: User())
present(secondVC, animated: true)
```

To Previous Screen.
Delegate

```
protocol SecondViewControllerDelegate: class {
    func secondViewController(_ secondViewController: SecondViewController, didLoadUser user: User)
}

class SecondViewController: UIViewController {

    var user: User!
    weak var delegate: SecondViewControllerDelegate?

    func loadUser() {
        let newUser = User()
        user = newUser
        delegate?.secondViewController(self, didLoadUser: newUser)
    }

}
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard segue.identifier == "toSecondController" else { return }

    let secondController = segue.destination as! SecondViewController
    secondController.delegate = self
}
```

```
extension FirstViewController: SecondViewControllerDelegate {

    func secondViewController(_ secondViewController: SecondViewController, didLoadUser user: User) {
        print(user)
    }

}
```

To Previous Screen. Closure

```
class SecondViewController: UIViewController {

    var user: User!
    var userDidLoad: ((User) -> Void)?

    func loadUser() {
        let newUser = User()
        user = newUser
        userDidLoad?(newUser)
    }

}
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard segue.identifier == "toSecondController" else { return }

    let secondController = segue.destination as! SecondViewController

    secondController.userDidLoad = { user in
        print(user)
    }
}
```