# 2nd Day - iOS Swift Training in Barcelona

A constraint defines a relationship between two user interface objects.
Many UI object properties, including layout constraints, can be changed either from Interface Builder (IB) or programmatically.
Typically you will set a property by IB only if it is almost static. You use the programmatic approach instead if the property is going to be changed at runtime.

## Auto Layout

Auto Layout is the tool to handle constraints in IB.
Always use Auto Layout, otherwise you are going to write more code!

### Example: Layout app

Let’s create four square shaped views 80x80 placed at the four corners of a view, 50 point far from margins.

Now we are going to define spacing constraints to the view object.
Tip: before to apply the constraints, be sure to have positioned the objects correctly. Thereafter apply the constraints.

Any constraint between two UI objects is associated to a linear equation y = mx + q,

where
- y is the a vertical (horizontal) attribute of a UI object
- x is the a vertical (horizontal) attribute of another UI object

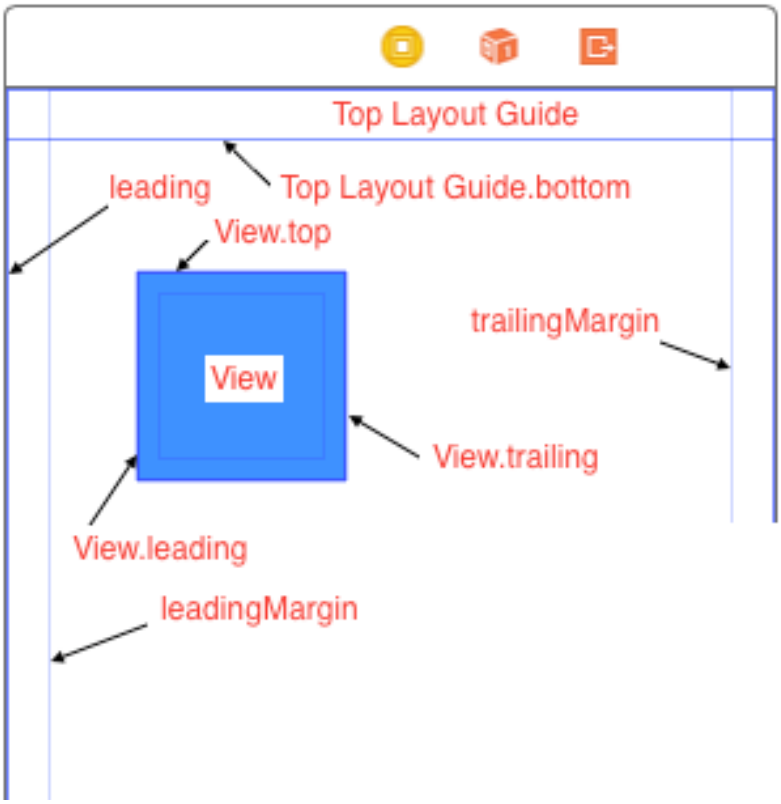
m and q are called “multiplier” and “constant”, respectively.
Constraints are not limited to equality relationships. They can also use greater than or equal to (>=) or less than or equal to (<=) to describe the relationship between the two attributes.

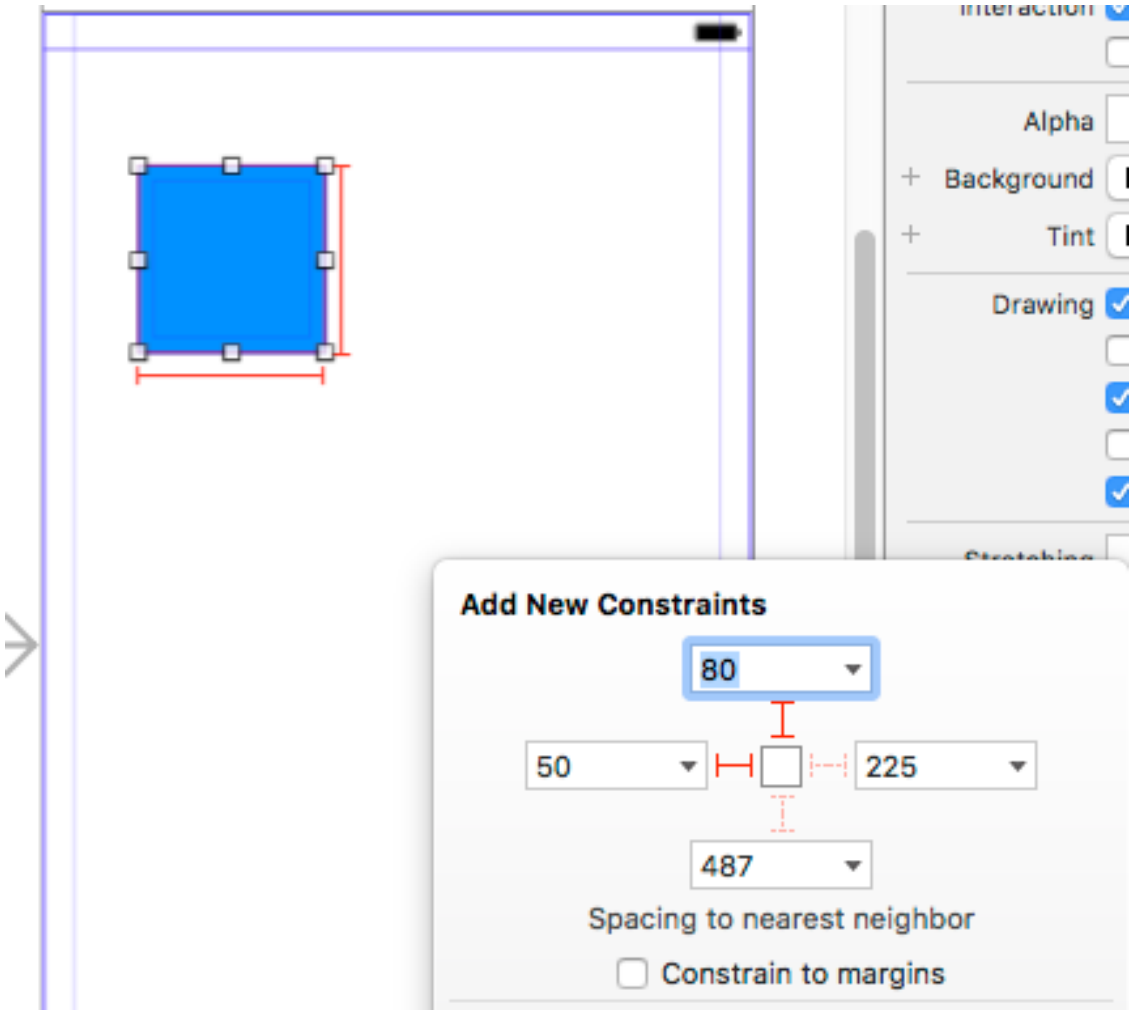
For each view object we need 2 constraint for x direction and 2 for y direction. Instead, for labels, buttons, text fields and lall objects containing text, you just need 2 constraints. This will allow the text to grow, for example when you use different languages to localise your app.

Tip: In order to see the distances of a view object select the view, move the cursor outside the view’s area, then press the Option key.

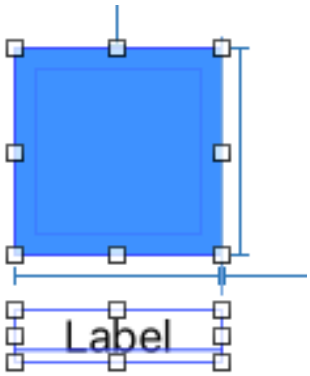
The light blue lines are called margins. They are reference lines that will change depending on the device model.
The ticker blue lines are called the layout guides.

Tip: When you set the spacing constraints in the “Add New Constraints” panel, deselect the “Constrain to margins”. In this way your constrain values will be relative to the layout guides

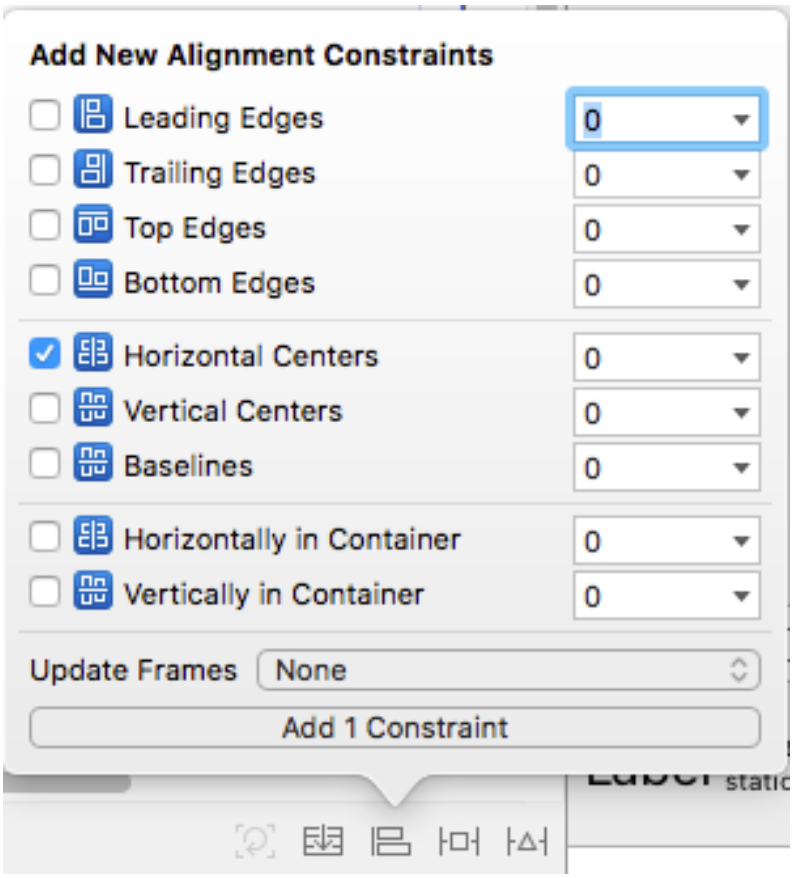




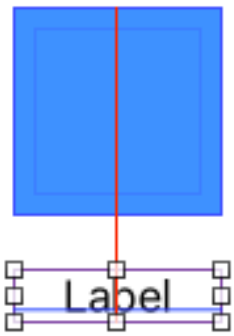

We want to place a label below the square, at a distance 20 and with horizontal centres aligned.
To do that, create the label and put it in place, then select both using the Cmd key



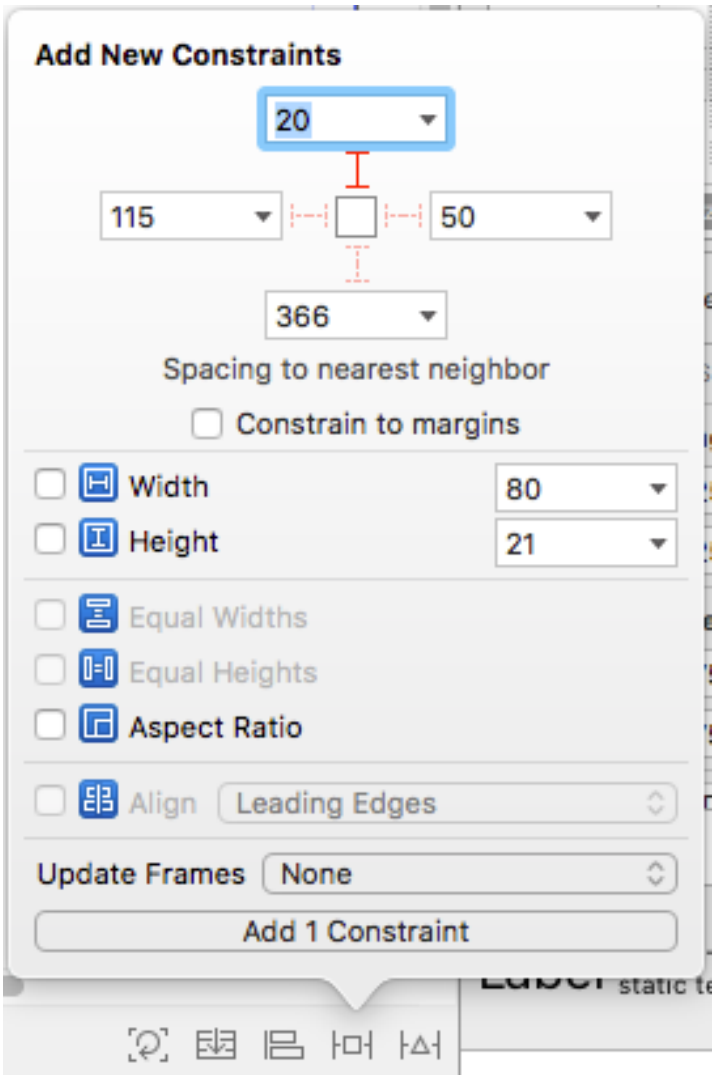

and then add the constraint

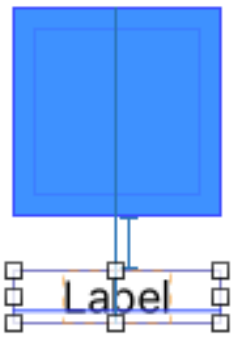



Finally select the label,



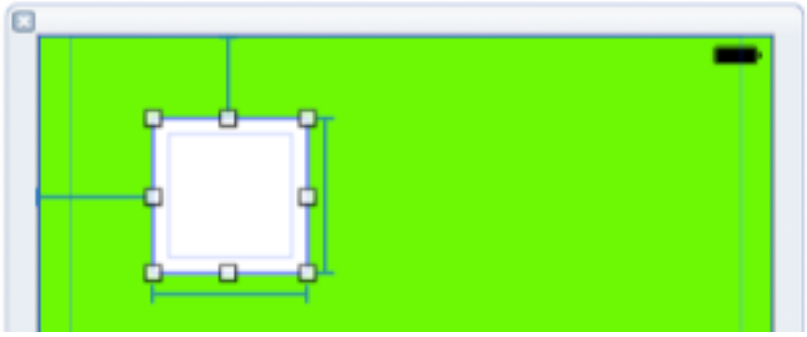

and then add the second constraint



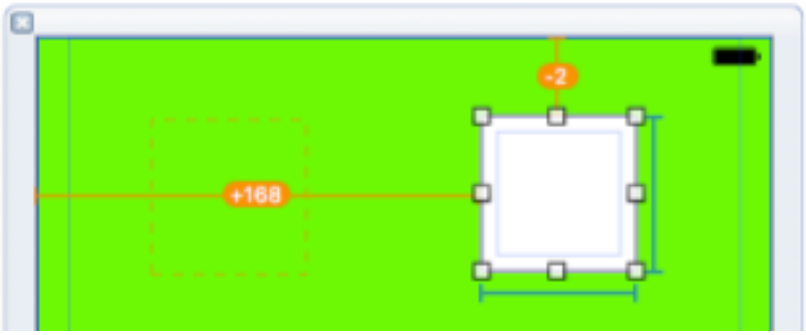




Constraints with more than one solution are ambiguous. Constraints with no valid solutions are conflicting.
The constrains  are non-ambiguous and non-conflicting when all of them appear in blue color. Anything red or orange marks an issue that has to be fixed. If, at any moment, you think that something is going wrong then it is better to clear all the constraints and start over. This is because Xcode doesn’t account for unnecessary constraints you may add, and there is the chance of a runtime error.

Once you have set some constraints

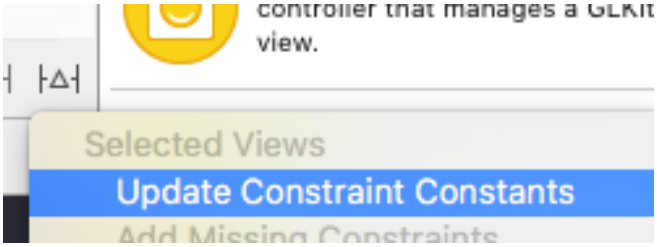


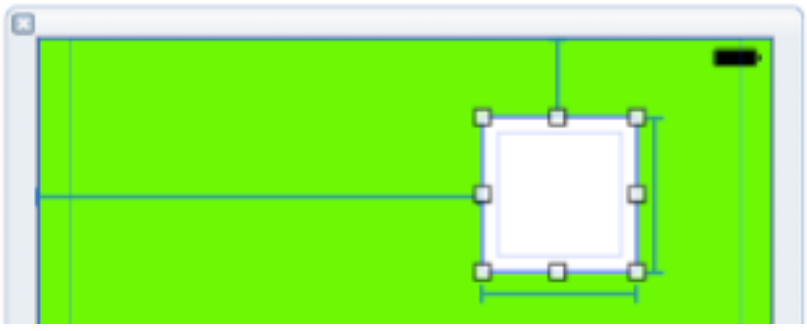
you may drag the view object in another position,



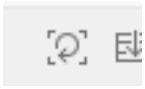

Then you have two options:

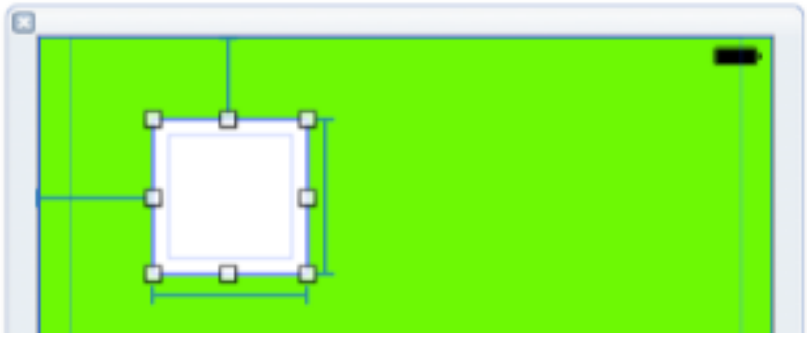
- Updating Constraint Constants -> will adjust constraints to the new situation



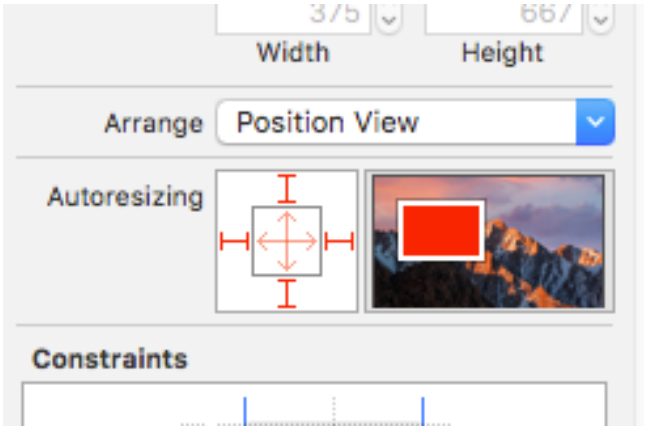



- Update Frames -> restore the object to its original position

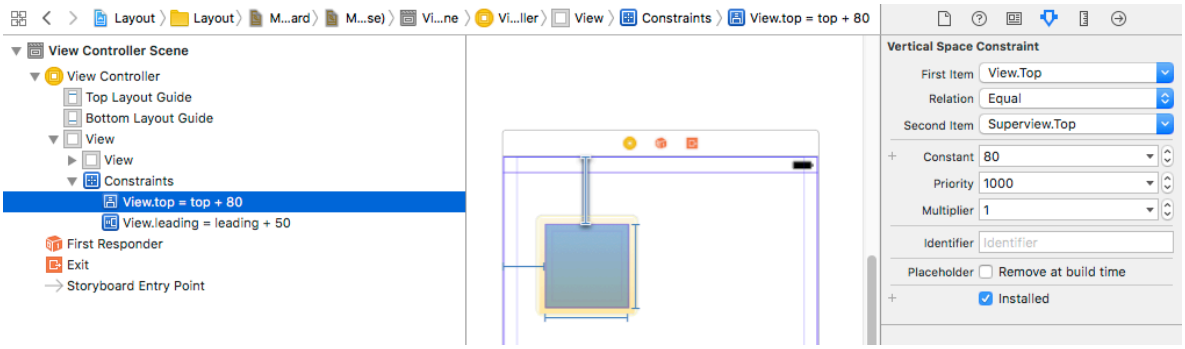




Tip: The following image shows another tool to set constraints. Do not use it. Use Auto Layout, which is more predictable and easy to debug




Constraints also have priorities between 1 and 1000. Constraints with a priority of 1000 are required. All priorities <1000 are optional. By default, all constraints are required.




When you add a view programmatically, at run time you can use NSLayoutConstraint and/or anchors (as done in the Container app..) in order to change its constraints

# ViewController Containment

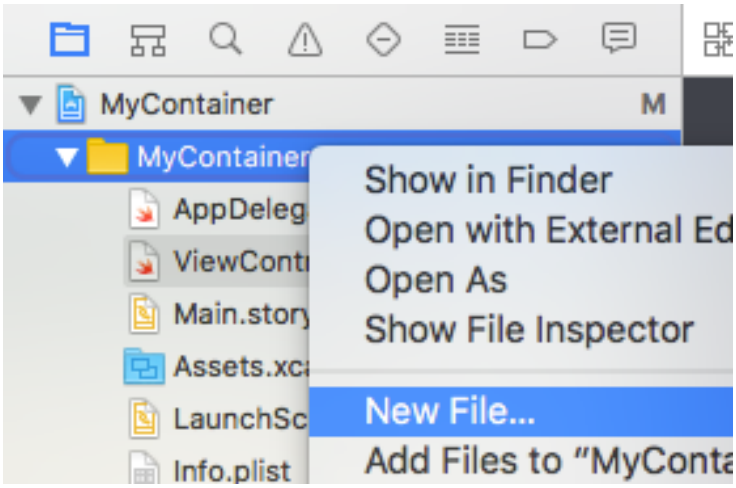
Every app contains at least one custom subclass of UIViewController.
More often, apps contain many custom VCs.
A Container VC is a way to combine the content from multiple VCs into a single UI.
A VC is the sole owner of its view (V) and any subviews it creates. You cannot share views between VCs.
Then you have 1 VC for 1 view.

But, suppose you have an iPad app with Map, Table, Buttons, Sliders, ..
A single VC for controlling everything would become gigantic and difficult to maintain.
A different approach is to add more VCs for handling different portions of the screen (subviews). Moreover, they can be assigned to different developers.

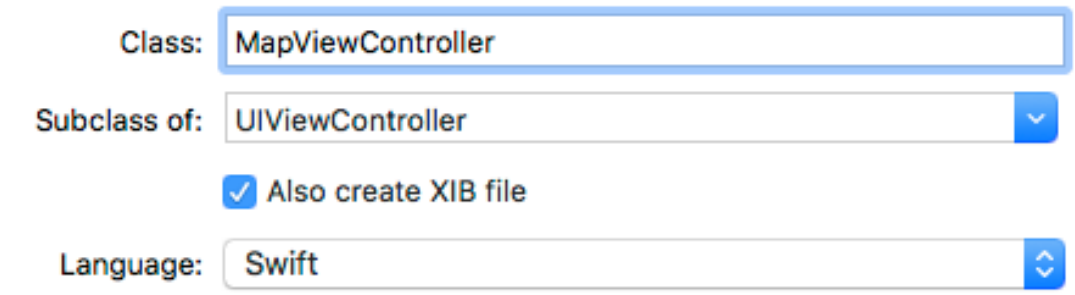
We create a parent-child relationship between VCs.
VC0 is the father, VC1,2,3,.. are the children
V0 is the super-view, V1,2,3,.. are the sub-views

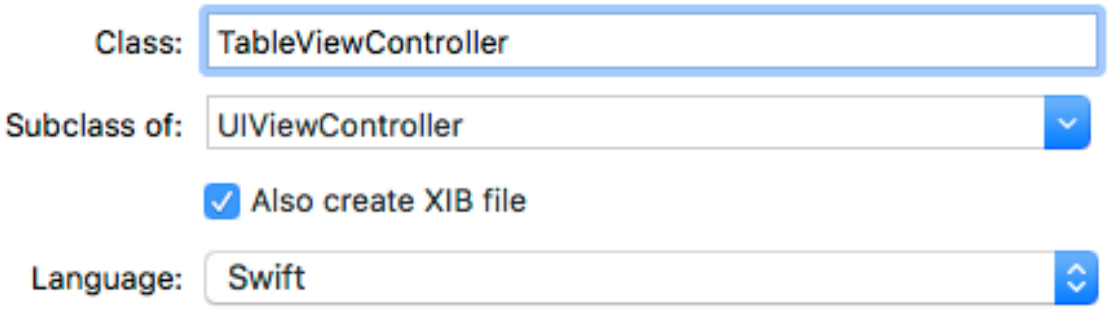
## Example: Container app

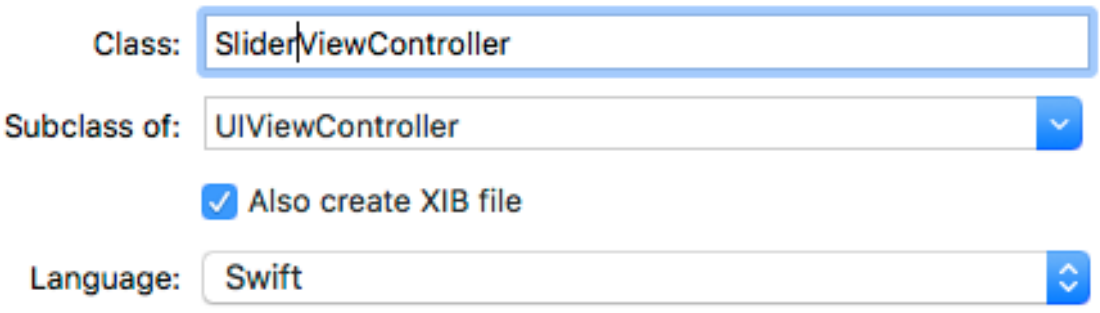
We use the default VC as the parent one (VC0).
We are now adding three more VCs and changing their background color.
Ctrl+Click over the main project folder to create a new file

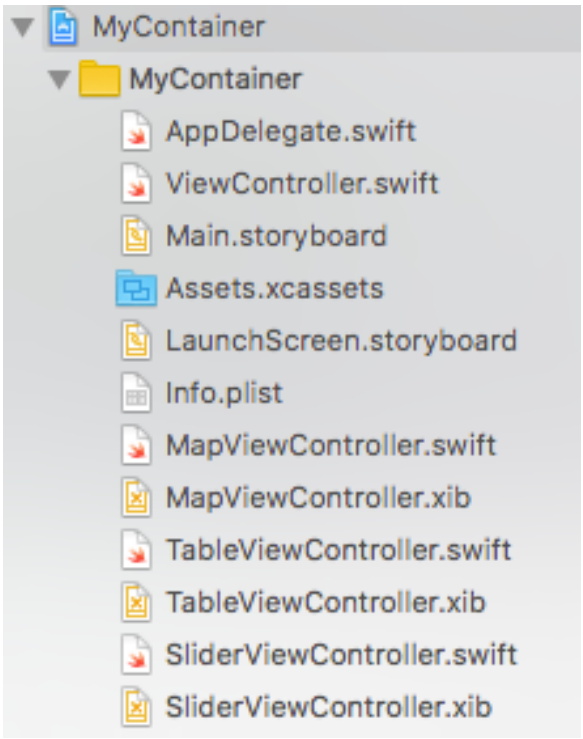



You can use either a storyboard or a nib file to store your view objects. In this case we use three nib files.


Class: MapViewController
Subclass of: UIViewController
Also create XIB file
Language: Swift


Class: TableViewController
Subclass of: UIViewController
Also create XIB file
Language: Swift


Class: SliderViewController
Subclass of: UIViewController
Also create XIB file
Language: Swift




The .xib is an xml file. You cannot edit this xml directly.
When compiled, the .xib it becomes a .nib binary

First, let's assign a different background color to each nib file.
Now we need to instantiate the three child VCs within the viewDidLoad of the parent VC.
Then we are going to create the parent-child relationships using the method *addChildViewController*.
Then we add the corresponding views to the parent view by using the method *addSubview*.
The last step is to call didMove(toParentViewController:) method of each child VC.
We need this method to inform the child views that everything is in place and they can start to do what they want to do.
Actually, inside this method Apple calls the willMove(ToParentViewController:).
The same method has to be called also after the VCs are removed from the container VC.

```
13      override func viewDidLoad() {
14          super.viewDidLoad()
15
16          let mapVC = MapViewController()
17          addChildViewController(mapVC)
18
19          let tableVC = TableViewController()
20          addChildViewController(tableVC)
21
22          let sliderVC = SliderViewController()
23          addChildViewController(sliderVC)
24
25          view.addSubview(mapVC.view)
26          view.addSubview(tableVC.view)
27          view.addSubview(sliderVC.view)
28
29          mapVC.didMove(toParentViewController: self)
30          tableVC.didMove(toParentViewController: self)
31          sliderVC.didMove(toParentViewController: self)
```

If you need to remove a VC from the container VC, then you have to execute the reverse operation, as follows

```
34          mapVC.willMove(toParentViewController: nil)
35          mapVC.view.removeFromSuperview()
36          mapVC.removeFromParentViewController()
```

Note: we can omit self. in front of the methods call.
Note: you could also complete the four steps for each sub VC separately, but respect the same order.
Warning: The sequence of the commands is very  important.

Now if you run the app only the color of the last subview you added is visible. That is because the views were added on top of each other.
Then we have to define the layout of the subviews, this time programmatically. You typically do this right after the addSubview method.
Let's create a function for each subview for setting the layout constraints.
For the Map subview,

```
42      func addConstraintsForMapVC(to subview: UIView) {
```

As soon as you create a view programmatically, iOS applies on it a set of Auto Layout constraints. The first thing to do is to override them by setting to false the view property *translatesAutoresizingMaskIntoConstraints*.

```
43          subview.translatesAutoresizingMaskIntoConstraints = false
```

Then we  use the NSLayoutConstraint to create a constraint

```
45          let aConstraint = NSLayoutConstraint(item: Any , attribute: NSLayoutAttribute , relatedBy: NSLayoutRelation , toItem:
                Any? , attribute: NSLayoutAttribute , multiplier: CGFloat , constant: CGFloat )
```

Arguments item and toItem represents the two UI objects we are constraining. The attribute arguments are of type NSLayoutAttribute, which is an enumeration. Multiplayer and constant are m and q. After having created the constraint, set its isActive property to true.

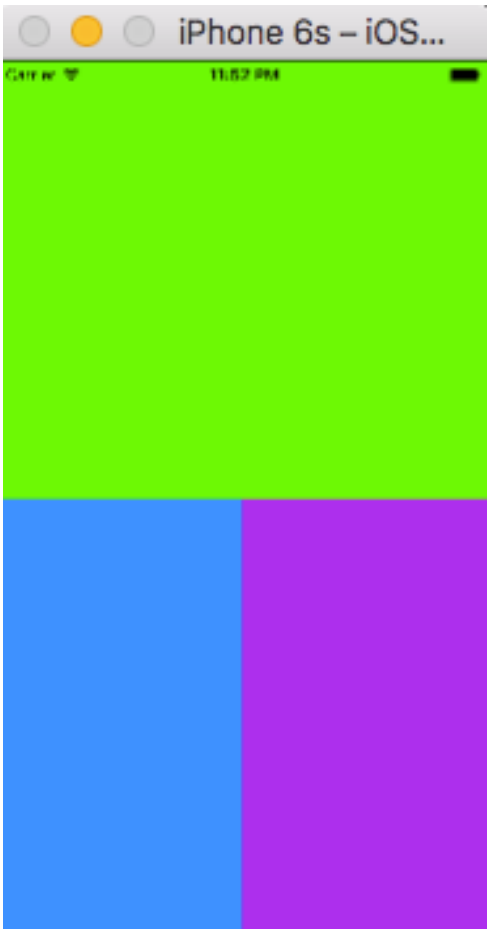
Suppose we want the MapView occupies the upper half of the screen area, while the TableView and the SliderView occupy the lower left and lower right portion of the screen. Then,

```
    func addConstraintsForMapVC(to subview: UIView) {
        subview.translatesAutoresizingMaskIntoConstraints = false
        let topConstraint = NSLayoutConstraint(item: subview, attribute: .top, relatedBy: .equal, toItem: self.view, attribute: .top, multiplier: 1, constant: 0)
        topConstraint.isActive = true

        let bottomConstraint = NSLayoutConstraint(item: subview, attribute: .bottom, relatedBy: .equal, toItem: self.view, attribute: .centerY, multiplier: 1, constant: 0)
        bottomConstraint.isActive = true

        let leftConstraint = NSLayoutConstraint(item: subview, attribute: .left, relatedBy: .equal, toItem: self.view, attribute: .left, multiplier: 1, constant: 0)
        leftConstraint.isActive = true

        let rightConstraint = NSLayoutConstraint(item: subview, attribute: .right, relatedBy: .equal, toItem: self.view, attribute: .right, multiplier: 1, constant: 0)
        rightConstraint.isActive = true
    }

    func addConstraintsForTableVC(to subview: UIView) {
        subview.translatesAutoresizingMaskIntoConstraints = false
        let topConstraint = NSLayoutConstraint(item: subview, attribute: .top, relatedBy: .equal, toItem: self.view, attribute: .centerY, multiplier: 1, constant: 0)
        topConstraint.isActive = true

        let bottomConstraint = NSLayoutConstraint(item: subview, attribute: .bottom, relatedBy: .equal, toItem: self.view, attribute: .bottom, multiplier: 1, constant: 0)
        bottomConstraint.isActive = true

        let leftConstraint = NSLayoutConstraint(item: subview, attribute: .left, relatedBy: .equal, toItem: self.view, attribute: .left, multiplier: 1, constant: 0)
        leftConstraint.isActive = true

        let rightConstraint = NSLayoutConstraint(item: subview, attribute: .right, relatedBy: .equal, toItem: self.view, attribute: .centerX, multiplier: 1, constant: 0)
        rightConstraint.isActive = true
    }

    func addConstraintsForSliderVC(to subview: UIView) {
        subview.translatesAutoresizingMaskIntoConstraints = false
        let topConstraint = NSLayoutConstraint(item: subview, attribute: .top, relatedBy: .equal, toItem: self.view, attribute: .centerY, multiplier: 1, constant: 0)
        topConstraint.isActive = true

        let bottomConstraint = NSLayoutConstraint(item: subview, attribute: .bottom, relatedBy: .equal, toItem: self.view, attribute: .bottom, multiplier: 1, constant: 0)
        bottomConstraint.isActive = true

        let leftConstraint = NSLayoutConstraint(item: subview, attribute: .left, relatedBy: .equal, toItem: self.view, attribute: .centerX, multiplier: 1, constant: 0)
        leftConstraint.isActive = true

        let rightConstraint = NSLayoutConstraint(item: subview, attribute: .right, relatedBy: .equal, toItem: self.view, attribute: .right, multiplier: 1, constant: 0)
        rightConstraint.isActive = true
    }
```

The three functions have to be called right after the subviews have been added to the container view,

```
            view.addSubview(mapVC.view)
            view.addSubview(tableVC.view)
            view.addSubview(sliderVC.view)

            addConstraintsForMapVC(to: mapVC.view)
            addConstraintsForTableVC(to: tableVC.view)
            addConstraintsForSliderVC(to: sliderVC.view)
```

The UI appearance looks like this, both in portrait and landscape mode,




A shortcut is as follows

```
NSLayoutConstraint(item: subview, attribute: .top, relatedBy: .equal, toItem: self.view, attribute: .top, multiplier: 1,
    constant: 0).isActive = true
```

An alternative way is to use the anchor properties of the views,

```
func addConstraintsForMapVC(to subview: UIView) {
    subview.translatesAutoresizingMaskIntoConstraints = false
    subview.topAnchor.constraint(equalTo: self.view.topAnchor).isActive = true
    subview.bottomAnchor.constraint(equalTo: self.view.centerYAnchor).isActive = true
    subview.leftAnchor.constraint(equalTo: self.view.leftAnchor).isActive = true
    subview.rightAnchor.constraint(equalTo: self.view.rightAnchor).isActive = true
```

This is a more compact form when you don't need to change the multiplier (but there is also a version of constraint to change the constant).
You can also have a mixed solution, using both NSLayoutConstraint and anchors.

## Size Classes (and trait collections)

When I rotate the device, the app switches from landscape to portrait mode, or viceversa. You may want to dynamically change the layout of the app according to the current perspective. You may even want to customise the layout depending on the device model.
For example you may desire a bigger font when you move from iPhone to iPad.

The solution is to use the Size Classes.

We have a 2 x 2 matrix of size classes:

iPhone in portrait (also Plus models)
- compact width (wC)
- regular height (hR)

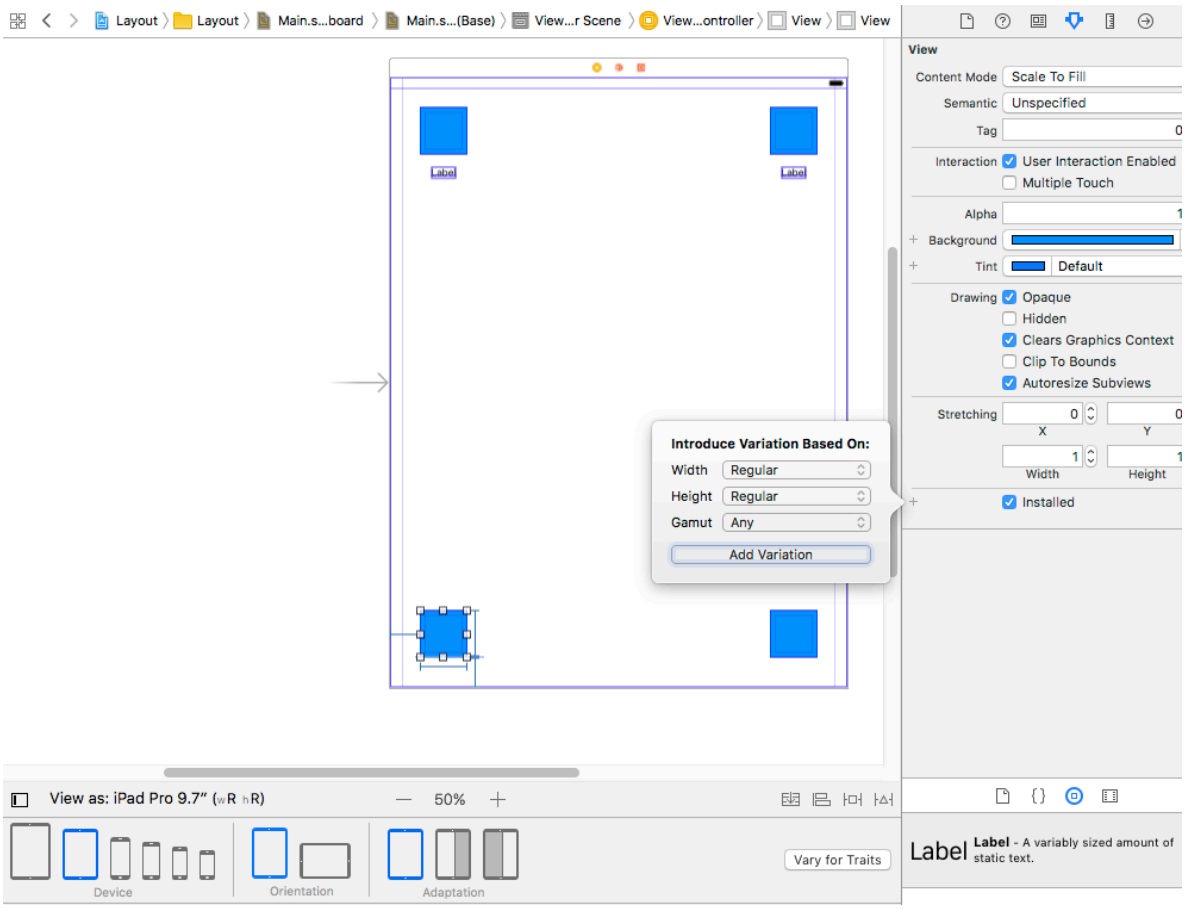
iPhone in landscape
- compact width (wC)
- compact height (hC)

iPhone Plus in landscape
- regular width (wR)
- compact height (hC)

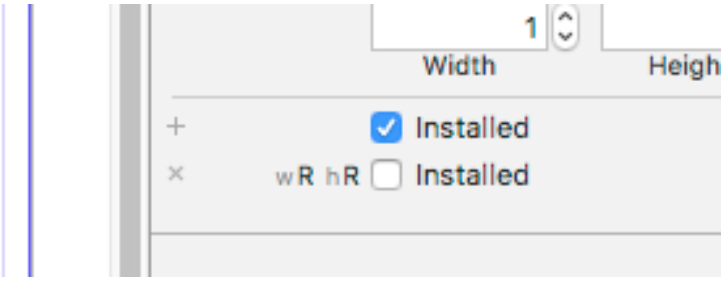
iPad (portrait or landscape) and Apple TV
- regular width (wR)
- regular height (hR)

Tip: When you start a new app select the Universal device mode, because you may need to create the app for several devices.

In order to remove a view object in a particular size class or device, select the view object, then from the attribute inspector select and click on the “+” symbol near the “Installed” property. This will create a duplicate for the desired size class. Then add a variation to the duplicate, or remove it.




Finally deselect the Installed checkbox for the variation just created




Tip: Starting from the iPad Air you can run two apps at the same time. When you do it each app takes half screen (Adaptation) which is organised as a iPhone in portrait mode (wC hR). This means that when you create an app for iPad you must also prepare the UI for the iPhone!! In other words, today makes no sense to have two different apps, one for iPad and one for iPhone.

Each VC has a property of type UITraitCollection.
A trait collection describes the iOS interface environment for your app, including traits such as horizontal and vertical size class, display scale, and user interface idiom. At any moment you can ask many things programmatically about the device by using the following two UITraitCollection methods

```
    //executed just before the device rotation
    override func willTransition(to newCollection: UITraitCollection, with coordinator:
        UIViewControllerTransitionCoordinator) {
        //you can ask about many different device configuration you are going to
        if newCollection.horizontalSizeClass == .compact && newCollection.verticalSizeClass == .compact &&
            newCollection.displayScale == 2.0 {
            print("======== DO SOMETHING ==========")
        }

    }

    //executed just after the device rotation happened
    override func traitCollectionDidChange(_ previousTraitCollection: UITraitCollection?) {
        //...
    }
```

In the example above I'm going to do something when the device goes in Compact Width and Compact Height mode, and the device is a Retina display one (displayScale is 2).
To create an adaptive interface, write code to adjust your app's layout according to changes in these traits.

## Example: MyContainer app

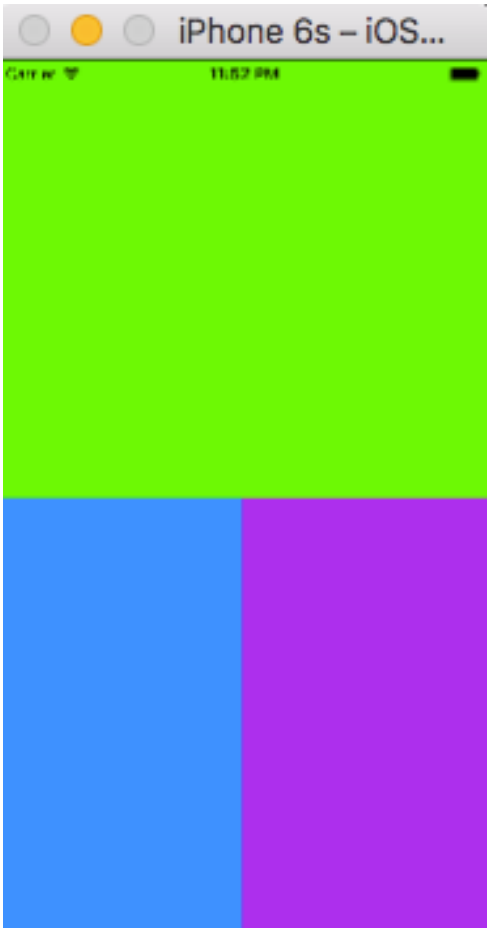
This app is almost the same as the Container app described above. The only difference is that here I needed a solution to preserve as much as possible the shapes of the subviews. We defined 7 class properties of type NSLayoutConstraint to create a UI layout which adapts to rotation of the iPhone (all models).

```
class ViewController: UIViewController {

    let mapVC = MapViewController()
    let tableVC = TableViewController()
    let sliderVC = SliderViewController()
    var mapViewRightAnchor: NSLayoutConstraint?
    var mapViewBottomAnchor: NSLayoutConstraint?
    var tableViewTopAnchor: NSLayoutConstraint?
    var tableViewLeftAnchor: NSLayoutConstraint?
    var tableViewRightAnchor: NSLayoutConstraint?
    var sliderViewTopAnchor: NSLayoutConstraint?
    var sliderViewLeftAnchor: NSLayoutConstraint?
```

```
    override func viewDidLoad() {
        super.viewDidLoad()
        addChildViewController(mapVC)
        addChildViewController(tableVC)
        addChildViewController(sliderVC)
        view.addSubview(mapVC.view)
        view.addSubview(tableVC.view)
        view.addSubview(sliderVC.view)
        addConstraintsForPortraitMapVC()
        addConstraintsForPortraitTableVC()
        addConstraintsForPortraitSliderVC()
        mapVC.didMove(toParentViewController: self)
        tableVC.didMove(toParentViewController: self)
        sliderVC.didMove(toParentViewController: self)
    }
```

```
40      override func willTransition(to newCollection: UITraitCollection, with coordinator:
            UIViewControllerTransitionCoordinator) {
41          if (newCollection.verticalSizeClass == .compact && newCollection.horizontalSizeClass == .compact) ||
                (newCollection.verticalSizeClass == .compact && newCollection.horizontalSizeClass == .regular) {
42              print("moving to wC&hC or wR&hC ..")
43              mapViewRightAnchor?.isActive = false
44              mapViewRightAnchor = mapVC.view.rightAnchor.constraint(equalTo: view.centerXAnchor)
45              mapViewRightAnchor?.isActive = true
46              mapViewBottomAnchor?.isActive = false
47              mapViewBottomAnchor = mapVC.view.bottomAnchor.constraint(equalTo: view.bottomAnchor)
48              mapViewBottomAnchor?.isActive = true
49              tableViewTopAnchor?.isActive = false
50              tableViewTopAnchor = tableVC.view.topAnchor.constraint(equalTo: view.topAnchor)
51              tableViewTopAnchor?.isActive = true
52              tableViewLeftAnchor?.isActive = false
53              tableViewLeftAnchor = tableVC.view.leftAnchor.constraint(equalTo: view.centerXAnchor)
54              tableViewLeftAnchor?.isActive = true
55              tableViewRightAnchor?.isActive = false
56              tableViewRightAnchor = tableVC.view.rightAnchor.constraint(equalTo: view.centerXAnchor, constant: view.frame.
                    height/4)
57              tableViewRightAnchor?.isActive = true
58              sliderViewTopAnchor?.isActive = false
59              sliderViewTopAnchor = sliderVC.view.topAnchor.constraint(equalTo: view.topAnchor)
60              sliderViewTopAnchor?.isActive = true
61              sliderViewLeftAnchor?.isActive = false
62              sliderViewLeftAnchor = sliderVC.view.leftAnchor.constraint(equalTo: view.centerXAnchor, constant: view.frame.
                    height/4)
63              sliderViewLeftAnchor?.isActive = true
64          }
65          else if newCollection.verticalSizeClass == .regular && newCollection.horizontalSizeClass == .compact {
66              print("moving to wC&hR ..")
```

The result looks like this




## Grand Central Dispatch (GCD)

Grand Central Dispatch is a low-level framework in OS X that allows you to handle multithreading. Itmanages concurrent and asynchronous execution of tasks across the operating system.

- thread: a separate path of execution for code. The underlying implementation for threads in OS X is based on the POSIX threads API.
- process: a running executable, which can encompass multiple threads.
- task: the abstract concept of work that needs to be performed

Here we consider the Swift 3 version of GCD. In Objective-C and in Swift 2 the APIs were completely different.
The GCD works on queues of works, not threads. You can create a closure, and then pass it to the queue for execution.
The “main queue” is a very important queue. It is the main event loop the app enters after **applicationDidBecomeActive()**. The main queue is in charge of the UI rendering, the user interaction, etc. UIKit is executed on the main queue.
Warning: Whatever is not related to UI (e.g data processing, data parsing, etc.) should run on a different queue (secondary queue).
You can create serial queues or concurrent queues.

```
16          //let's create a queue
17          let queue = DispatchQueue(label: "com.invasivecode.queue", attributes: .concurrent)
18
19          //create a closure and pass it to the queue
20          queue.async {
21              for i in 0..<1000{
22                  print("CL1: \(i)")
23          }
```

The “com.invasivecode.queue” name is only for debugging purposes.
To dispatch a work to a queue you have two options, sync or async. Do not use sync queues because they will stop the main queue (and then the UI) until your work execution completes.

You can create as many queues as you want but the system is able to run up to four simultaneously.
Tip: The iPhone and the iPad has two cores. The Mac Pro has 32 cores.

After having processed the data in a secondary queue, you may want to present the data to the UI.
Tip: If you have to do something in the UI, don't do that outside the main queue.
Then you need a method to reference the main queue

```
26          DispatchQueue.main.async {
27                  self.view.backgroundColor = .red
28              }
29          }
```

Serial dispatch queues are a good alternative to using timers for synchronization.

Tip: ()->() is a closure without any argument or return.

## GCD QoS

GCD can also utilise Quality of Service classes for task prioritisation. This concept is also available on Mac.
There are four primary QoS classes:

- UI (User Interactive): it is the higher priority (is the main queue)
- IN (User Initiated): it is executed as fast as possible, but does not tops the main queue.
- UT (Utility): those tasks typically have a progress bar
- BG (Background): it handles works that takes a long time (such as minutes or hours)

Suppose you want to get data from the network. It takes some time and you won't block the UI.
You can create a UI QoS queue to do that

```swift
let queue = DispatchQueue(label: "com.cosmed.queue", qos: .userInitiated)

queue.async {
    for i in 0..<10000 {
        print("work_UI\(i*i)")
    }
}

let queue1 = DispatchQueue(label: "com.cosmed.queue", qos: .background)

queue1.async {
    for i in 0..<10000 {
        print("work_B\(i*i)")
    }
}
```

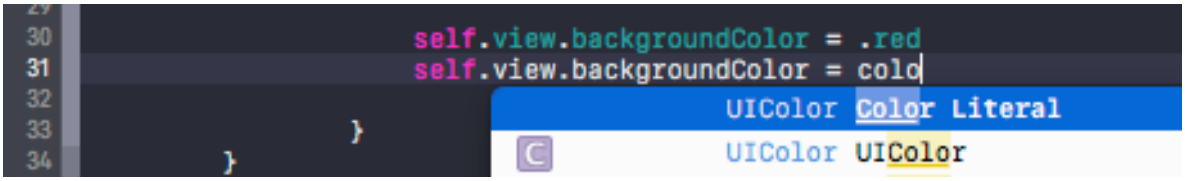
GCD allows you to create groups of queue, and send a notification to the main queue upon completion of all the secondary queues.
A DispatchGroup allows you to submit multiple different works and track when they all complete, even though they might run on different queues. This behaviour can be helpful when progress can't be made until all of the specified tasks are complete.

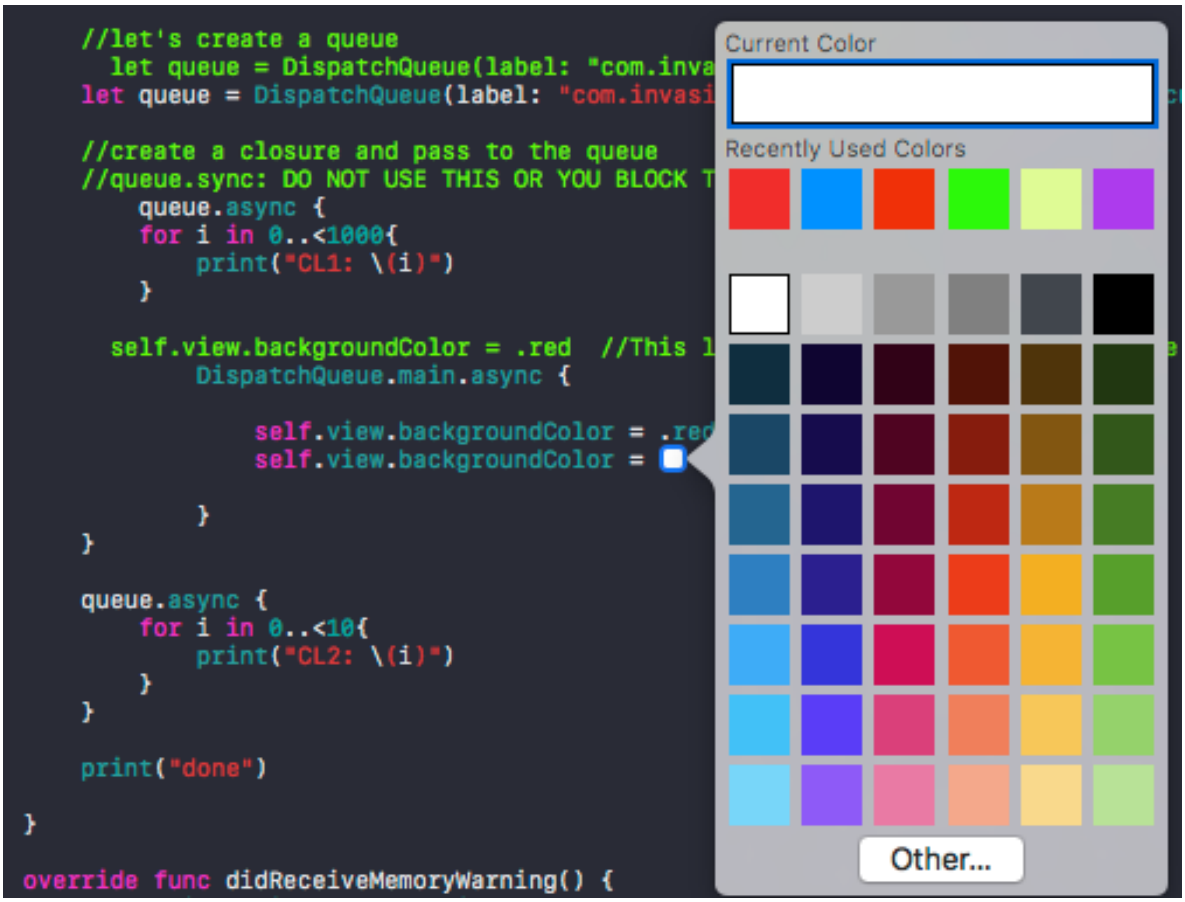
For example if you are reading a very big file from the filesystem, you can speedup it by splitting the operation among multiple threads on a dispatchIO queues.
Again, dispatchSources is a special queue to monitor if a network socket has new data to get. GCD automatically will dispatch a work for this.

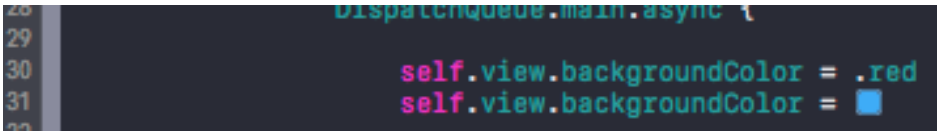
```swift
let work = DispatchWorkItem {
    for i in 0...1000 {
        print("work: \(i+3)")
    }
}

let dispatchGroup = DispatchGroup()
dispatchGroup.notify(queue: queue, work: work)
```

go to GCD app...

Tip: A different way to peek a UI color






## Table Views

They are the only way to display a lot of data on a small screen.
UITableView is a special View able to handle a lot of smaller views of type UITableViewCells.
In iOS you can have only one column for a TableView. For each row there is a cell. Each cell can have a different height.
You can create one or more sections, each with its customisable header and footer.
To select a cell you need to know the section and the cell number. To this aim you will use IndexPath. IndexPath is a structure in Swift 3.0 (in Obj-C it is still a NSIndexPath class)

IndexPath.section
IndexPath.cell

You need a dataSource object (DS).
The client is the ViewController (VC) that creates the TableView (TV) that will

interact with the DS. The TV looks for the DS asking for the number of sections, the number of rows for each section and the cell to display for each row. Then the TV will render everything.
There is a complex cache mechanism working behind the scene.

go to Table app...

```
import UIKit

class ViewController: UIViewController, UITableViewDataSource {

    @IBOutlet var tableView: UITableView! {
        willSet { //property observer. We can use it to allocate the required cache for the Table View
            newValue.dataSource = self //this is the same of Ctrl dragging in the storyboard
            newValue.register(UITableViewCell.self, forCellReuseIdentifier: "com.invasivecode.cell")
        }
    }


    //let's create fake data. We use an array with a closure to fill it
    lazy var dataList: [String] = {
        var tempArray = [String]()
        for i in 0..<1000 {
            tempArray.append("\(i)")
        }
        return tempArray
    }()

    //let's implement the methods of UITableViewDataSource
    func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return dataList.count
    }
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
//        let cell = UITableViewCell()
        let cell = tableView.dequeueReusableCell(withIdentifier: "com.invasivecode.cell", for: indexPath)
        cell.textLabel?.text = dataList[indexPath.row]
        return cell
    }
```

## Delegation

Subclassing is discouraged in Cocoa Touch. They prefer to use a kind of helper object to provide the additional functionality, the delegate object. It is widely used, for example in CoreMotion, CoreLocation, etc..
Those helper objects use protocols.
A protocol is a list of empty methods.
Given a class, the usual naming convention for its delegate protocol is:
<main class name>Delegate.
Some apps require more than one protocol. For example a TableView may need of UITableViewDelegate, UITableViewDataSource, UITableViewPrefetchingDataSource.
Sometimes you need to create your own protocol.
To implement a protocol

- set the delegate. To do that from the storyboard use Ctrl + Click and drag
- add the name of the protocol close to name of the parent class

```
class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
```

- add the delegate method implementation (some are required, some are optional)

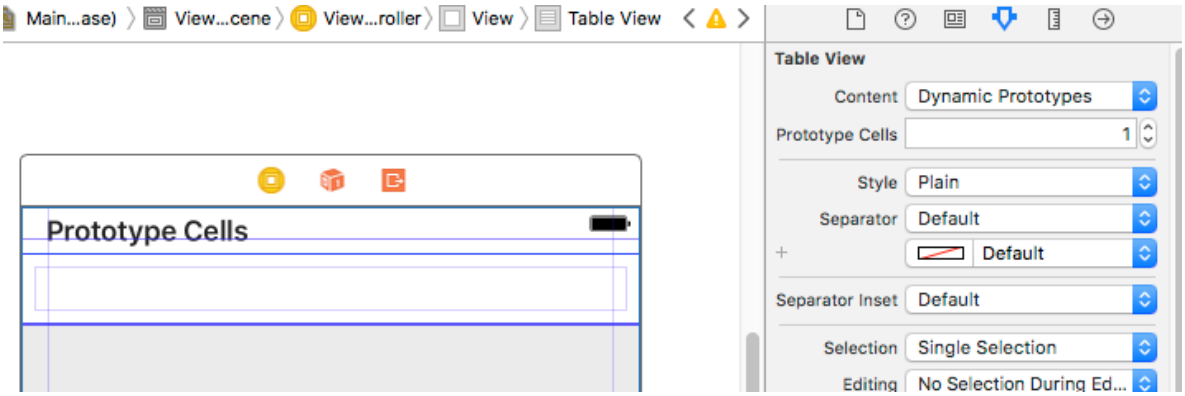
We do not use UITableViewController. It is not flexible enough for adding constraints, for example.

Tip: READ THE PROTOCOL REFERENCE DOCUMENTATION! AT LEAST ONE TIME INT THE LIFE. DO NOT LOOK IN THE INTERNET

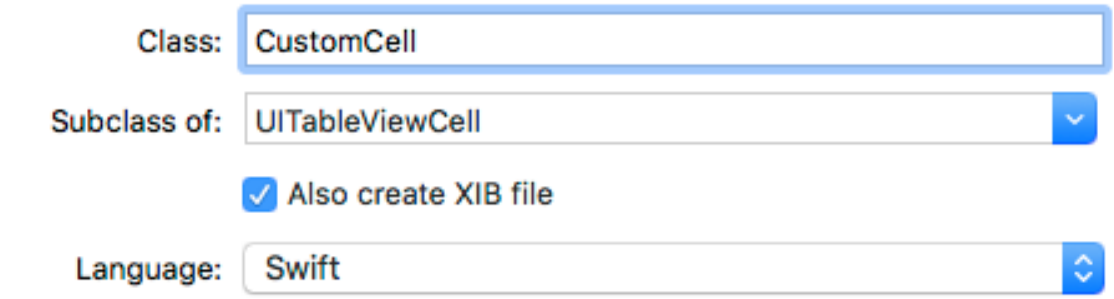
## Customize the Cell

You can do that either programmatically or via IB.

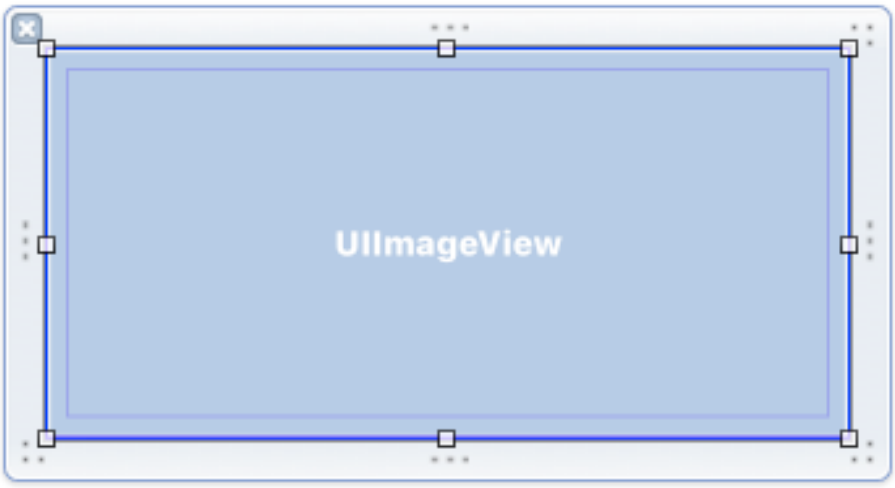
Tip: Do not use the following method:



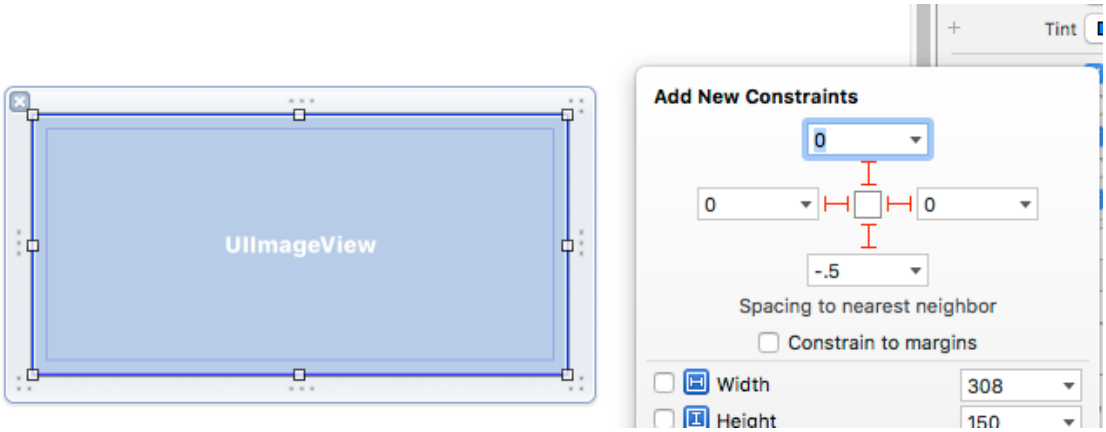

instead use a nib file. It is more portable.




Now add an ImageView to the xib file




Add the constraints




Download an image from the internet and use it as the cell background

```
48
49          //load the cell image
50          //let image = UIImage(named: "cellImage.jpg")
51          //cell.landscapeImageView.image = image
52
53          cell.landscapeImageView.image = cell
54                                  CustomCell cell
55                                  cellImage.jpg
56      }
```

## Exercise:

go in the documentation site, look for “TableView Programming Guide for iOS”, Find the section “Inserting and deleting Rows and Sections”, read that and implement the deletion of the cell.

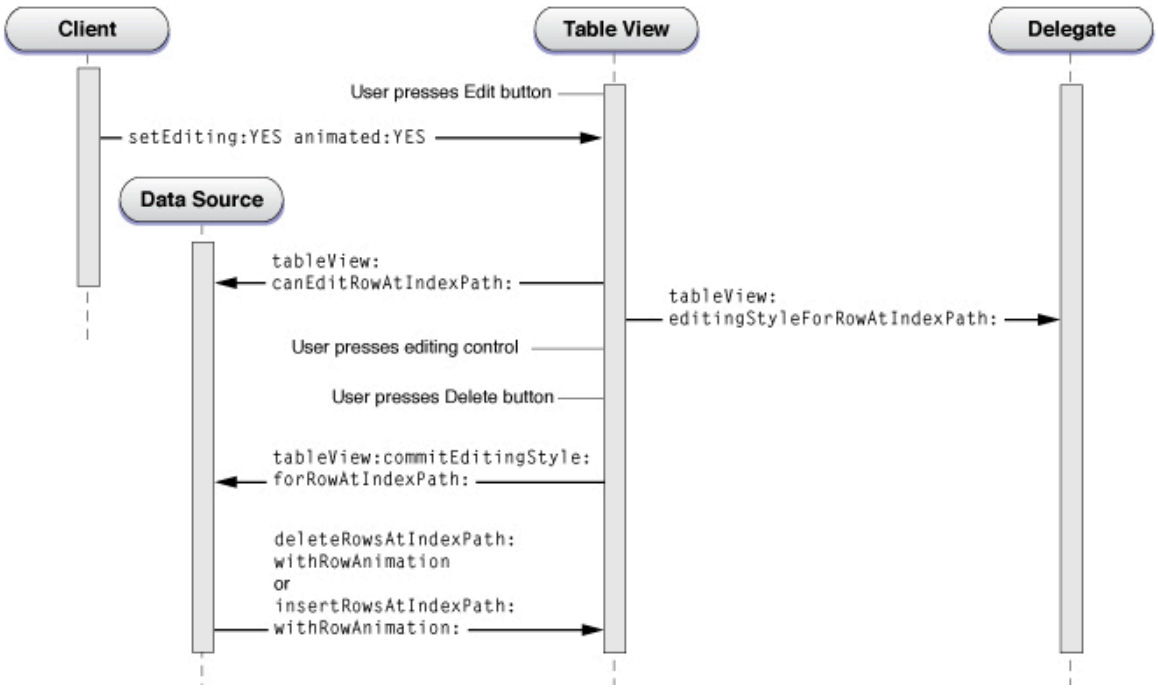
1. Go to the Documentation site

2. Look for “TableView Programming Guide for iOS”
3. Search for section on deleting cells

## Inserting and Deleting Rows and Sections

A table view has an editing mode as well as its normal (selection) mode. When a table view g
editing and reordering controls associated with its rows. The editing controls, which are in th

4. I found out there are two editing controls which allow to insert new rows and delete existing rows in a table view.




```swift
// by implementing this method I say "Yes, you can edit any rows"
func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
    return true
}

// this method will be called when the user tap the delete button. Here I have to actually delete the cell
func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt
    indexPath: IndexPath) {
    // insert (or remove) the cell in (from) the list. Do the same for the corresponding dataList item !
    if (editingStyle == UITableViewCellEditingStyle.delete) {
        self.dataList.remove(at: indexPath.row)
        tableView.deleteRows(at: [indexPath], with: UITableViewRowAnimation.fade)
    } else if (editingStyle == UITableViewCellEditingStyle.insert) {
        tableView.insertRows(at: [indexPath], with: UITableViewRowAnimation.fade)
        self.dataList.insert("500", at: indexPath.row)
    }
}

// Implementing this method I enable the cell editing mode (deletion)
func tableView(_ tableView: UITableView, editingStyleForRowAt indexPath: IndexPath) ->
    UITableViewCellEditingStyle {
    // Only for the last cell of the TV we enable the insert editing option
    if indexPath.row == self.dataList.count - 1 {
        return UITableViewCellEditingStyle.insert
    } else {
        return UITableViewCellEditingStyle.delete
    }
}
```

## Collection Views

It is a generalisation of a TableView. An example is Calendar in iPhone.

You can do any layout.
The protocols are similar, but there is an additional one
UICollectionViewLayout helps you to organise the different views on the screen

To populate and interact wit cell is the same as in TableViews.

We have again sections, but instead of having header and footer we supplementary cells, small views that you can place anywhere.

There is a sub-protocol that is more easy to use, UICollectionViewFlowLayout, which is a grid of views. They can be scrolled either vertically or horizontally.
And there is a lot of beautiful animations for free.
Each cell need attributes (frame, bounds, center, size, alpha, ...)

It supports DynamicAnimations (e.g. spring constants between cells when scrolling or moving)

You can also subclass to create custom layouts different from a grid.

go to Collection app..

## Scroll View

can contain more content than the size of the screen.
UIScrollView needs a delegate
There is a contentOffset
You can switch on or off the bounces
you can zoom in and out with the pinch gesture (bouncesZoom)

go to ScrollView app...