Programmatic Views and Constraints

`UIViewController` lifecycle methods

UIViewController lifecycle methods are called in the following order:

- `loadView()`
- `viewDidLoad()`
- `viewWillAppear()`
- `viewDidAppear()`
- `viewWillDisappear()`
- `viewDidDisappear()`

`loadView()`

- creates the view that the controller manages. It is only called when the view controller is created programmatically. This makes it a good place to create your views in code.
- called when you create a new instance of your view.
- only called once during the life of the view.
- best place to do any view initialization or setup, such as adding subviews or AutoLayout constraints.
- when this method is called, your view will likely be displayed soon, but there's no guarantee. Consequently, don't try to do too much in this method.

`viewDidLoad()`

- called when the view is FIRST loaded into memory
- only called once during the life of the object
- just because the view has been loaded into memory, it doesn't necessarily mean that it's going to be displayed soon
- remember to call `super.viewDidLoad()` in your implementation to make sure your superclass's `viewDidLoad()` gets a chance to do its work. (Good practice to do this right at the start of `viewDidLoad()`.)

`viewWillAppear()`

- called after `viewDidLoad()` is called, and just before the view appears on the screen, as well as any time the view is about to be displayed again. This means it can be called multiple times during the life of the view object:
  - when a VC's tab is selected in a tab VC
  - when a user taps the back button from another VC
  - when a user closes a modal alert
- used to refresh the screen with new data before redisplaying it, kick off network requests, etc.
- remember to call `super.viewWillAppear()` in your implementation to make sure your superclass's `viewWillAppear()` gets a chance to do its work. (Good practice to do this right at the start of `viewWillAppear()`.)

`viewDidAppear()`

- called after a view is displayed or redisplayed on the screen, as well as any time the view is displayed again. Like `viewWillAppear()`, this means it can be called multiple times during the life of the view object.
- great place to start animations, load external data from an API, etc.: updates that you don't want to start until the screen is actually being displayed.

There are also

```
viewWillDisappear()
viewDidDisappear()
```

that work in a similar way.

Guidelines for creating View Hierarchies using AutoLayout

- Initialize your views with the frame `.zero` .
- Set `translatesAutoresizingMaskIntoConstraints` to `false`. By default, all views automatically translate their initial *auto resizing masks* into layout constraints, which can conflict with the constraints we've defined in our code, so we disable this.
- Add your view to the view hierarchy using `addSubview()` .
- Create and activate your layout constraints using `NSLayoutConstraint.activate()` .
- Use `loadView()` instead of `viewDidLoad()` for creating views with constraints.
- Set every other property (background color, etc.) in `viewDidLoad()` .
- Take care of memory management by using `weak` properties.

The old way of defining constraints

Defining a constraint that forces the height of `label` to be the same as the height of `button`:

```
let constraint = NSLayoutConstraint(
    item:         label,
    attribute:    .height,
    relatedBy:    .equal,
    toItem:       button,
    attribute:    .height,
    multiplier:   1,
    constant:     0 )
```

The new way: using “anchors”

Every UIView in iOS contains a series of *anchors* that can be used to automatically create constraints relative to other views.

```
let constraint = label.heightAnchor.constraint(
    equalTo: button.heightAnchor )
```

In both styles, you then *activate* the constraint using:

```
NSLayoutConstraint.activate([constraint])
```

where the argument of `NSLayoutConstraint.activate` is an array of constraint objects.

Commonly used anchors

```
heightAnchor
widthAnchor
leftAnchor
rightAnchor
leadingAnchor
trailingAnchor
centerXAnchor
centerYAnchor
topAnchor
bottomAnchor
```

Example

```
NSLayoutConstraint.activate([

    // Place the button at the center of its parent
    button.centerXAnchor.constraint(equalTo:
         parent.centerXAnchor),
    button.centerYAnchor.constraint(equalTo:
         parent.centerYAnchor),

    // Give the label a minimum width based on the
    // width of the button
    label.widthAnchor.constraint(greaterThanOrEqualTo:
         button.widthAnchor),

    // Place the label 20 points beneath the button
    label.topAnchor.constraint(equalTo:
         button.bottomAnchor, constant: 20),
    label.centerXAnchor.constraint(equalTo:
         button.centerXAnchor)

])
```

Templates for commonly used constraints

Set a view's height or width to a fixed amount:

```
myView.widthAnchor.constraint(equalToConstant: 320)
myView.heightAnchor.constraint(equalToConstant: 240)
```

Set a view's aspect ratio:

```
myView.widthAnchor.constraint(equalToConstant: 64)
myView.widthAnchor.constraint(equalTo:
            myView.heightAnchor, multiplier: 16/9)
```

Center vertically or horizontally within a parent view:

```
myView.centerXAnchor.constraint(equalTo:
            self.view.centerXAnchor)
myView.centerYAnchor.constraint(equalTo:
            self.view.centerYAnchor)
```

Templates for commonly used constraints (cont.)

Stretch/fill with padding:

```
myView.topAnchor.constraint(equalTo:
            self.view.topAnchor, constant: 32),
myView.leadingAnchor.constraint(equalTo:
            self.view.leadingAnchor, constant: 32),
myView.trailingAnchor.constraint(equalTo:
            self.view.trailingAnchor, constant: -32),
myView.bottomAnchor.constraint(equalTo:
            self.view.bottomAnchor, constant: -32)
```

Proportional width and height:

```
myView.widthAnchor.constraint(equalTo:
            self.view.widthAnchor, multiplier: 1/3)
myView.heightAnchor.constraint(equalTo:
            self.view.heightAnchor, multiplier: 2/3)
```

Using constraints with the “safe area”

The more recent iPhones have a “notch” at the top of the screen. Those phones have a “safe area” called `safeAreaLayoutGuide` that reflects the maximum size of the screen that doesn’t overlap with the notch.

```
myView.topAnchor.constraint(equalTo:
     self.view.safeAreaLayoutGuide.topAnchor)

myView.leadingAnchor.constraint(equalTo:
     self.view.safeAreaLayoutGuide.leadingAnchor)

myView.trailingAnchor.constraint(equalTo:
     self.view.safeAreaLayoutGuide.trailingAnchor)

myView.bottomAnchor.constraint(equalTo:
     self.view.safeAreaLayoutGuide.bottomAnchor)
```