Introducing

Auto Layout

Hands-On Challenges

Introduction to Auto Layout Lab Instructions

Unconstrained Constraints

You've seen how to set up your auto layout constraints from interface builder. Now it's time to apply that knowledge and try adding layout constraints from code!

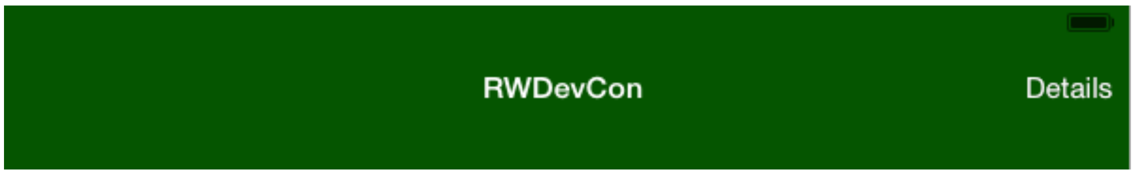
Layout constraints are just objects that you can create and attach to views. You can even connect constraints in your storyboard as outlets to your code, and modify them at runtime!

In this short lab, you'll create new views and then lay them out with constraints from code.

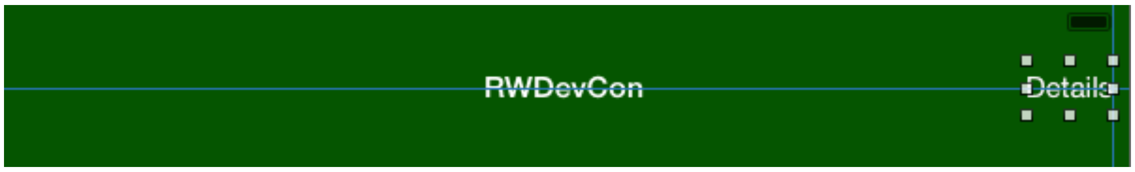
Button action

First, you'll need a button to trigger an action method to add the new views.

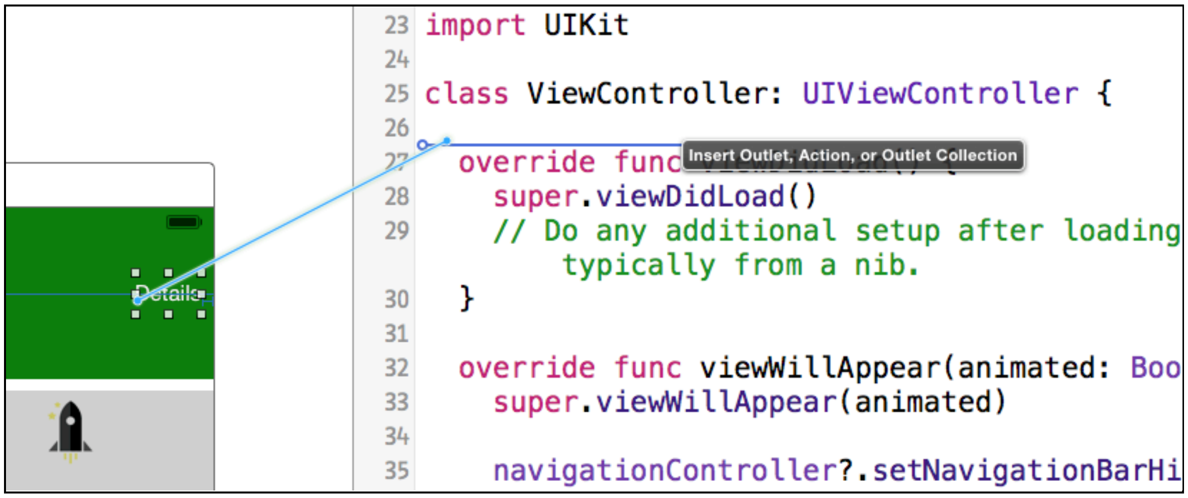
Add a new button to the top green view, and set the text to "Details" with white text color. Position it against the right margin and vertically centered.



The position is where you want it, so you can add the necessary constraints. From the menu, select **Editor\Pin\Trailing Space to Superview** to set the x-position. Next, select **Editor\Align\Vertical Center in Container** to set the y-position. You should see the joyful blue lines of happy constraints!



Next, you can connect the action to the button. Open the assistant editor and make sure **ViewController.swift** is in the assistant. Control-drag from the button to a spot inside the class. In the pop-up menu, remember to select **Action** rather that Outlet!

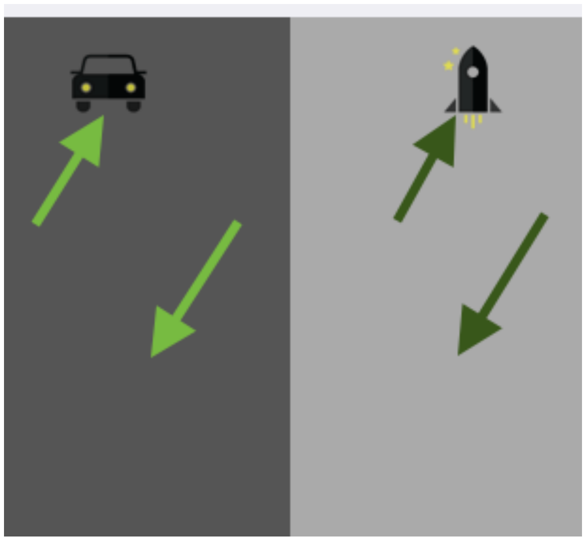


```
import UIKit

class ViewController: UIViewController {

  override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading
        typically from a nib.
  }

  override func viewWillAppear(animated: Boo
    super.viewWillAppear(animated)

    navigationController?.setNavigationBarHi
```

Name the action method **detailsButtonTapped**. You're all set with a place to create some views and constraints from code!

Outlets

While you have the assistant editor open, connect the following four outlets: the advanced and intermediate buttons, and the advanced and intermediate container views.



Name the outlets as follows:

- Advanced button > `advancedButton`
- Advanced view > `advancedView`
- Intermediate button > `intermediateButton`
- Intermediate view > `intermediateView`

You'll need these outlets to add the constraints to the containers, and to position the new views you'll be adding relative to the buttons.

Manual constraint relationships

Open **ViewController.swift** and add the following properties to the class:

```
var advancedLabel: UILabel!
var intermediateLabel: UILabel!

var advancedVerticalConstraint: NSLayoutConstraint!

let verticalMargin: CGFloat = 8.0
```

You'll be adding two new labels to the interface, so the two `UILabel` properties will hold them. You'll also want to hold a reference to one of the layout constraints, `advancedVerticalConstraint`, to modify it later.

Finally, `verticalMargin` is a constant that you'll use to keep the spacing consistent.

Next, add the following helper method to the class:

```
private func addAdvancedLabel() {
  // 1
  advancedLabel = UILabel()

advancedLabel.setTranslatesAutoresizingMaskIntoConstraints(false)
  advancedLabel.text = "Dive deep into a guided tour of more
advanced topics like functional programming, Scene Kit, and more!"
  advancedLabel.numberOfLines = 0
  advancedView.addSubview(advancedLabel)

  // 2
  advancedVerticalConstraint = NSLayoutConstraint(
    item: advancedLabel,
    attribute: .Top,
    relatedBy: .Equal,
    toItem: advancedButton,
    attribute: .Bottom,
    multiplier: 1.0,
    constant: verticalMargin)
  advancedVerticalConstraint.active = true

  // 3
  let leadingConstraint = NSLayoutConstraint(
    item: advancedLabel,
    attribute: .Leading,
    relatedBy: .Equal,
    toItem: advancedView,
    attribute: .LeadingMargin,
    multiplier: 1.0,
    constant: 0)
  leadingConstraint.active = true

  // 4
  let trailingConstraint = NSLayoutConstraint(
    item: advancedLabel,
    attribute: .Trailing,
    relatedBy: .Equal,
    toItem: advancedView,
    attribute: .TrailingMargin,
    multiplier: 1.0,
    constant: 0)
  trailingConstraint.active = true
}
```

This method will create a new label and set up its size and position with layout constraints. Here's what's going on, section by section:

1. Create a new label, set its properties, and add it to the view. Note the call to `setTranslatesAutoresizingMaskIntoConstraints(false)` – if you're setting up constraints in code, you need to be sure to call this method on your view so that it doesn't automatically create a default set of constraints. These constraints would conflict with your own, so you need to call this method to ensure you're starting off with a clean slate.

   Since the text is long, you're setting the label's number of lines to 0. This means the label will grow and add lines to fit the text.

2. Here's your first code constraint! You're saving this one to the `advancedVerticalConstraint` property so you can adjust it later, and adding it to the view with `addConstraint`. Notice how the initializer parameters for `NSLayoutConstraint` match what you see when you edit a constraint from interface builder.

   In this case, you're saying the position of the top of the label should be equal to the position of the bottom of the button. The constant is set to `verticalMargin`, so you'll get an 8-point vertical margin between the button and the label.

3. This constraint is for the leading (left) space between the label and its superview. You're relating the leading edge of the label to the leading margin of the superview.

4. The final constraint is similar to the previous one; this one is for the trailing (right) space between the label and its superview. You're relating the trailing edge of the label to the trailing margin of the superview.

After creating each constraint, you set its `active` property to `true`, which will install the constraint into the correct view.

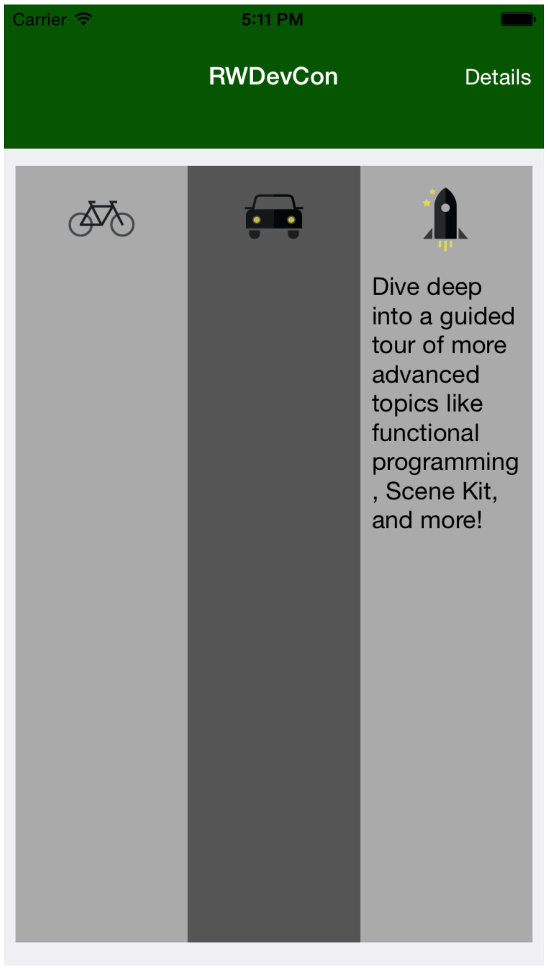
That's all you need! The label is set to 0 lines so it will automatically grow vertically to fit the amount of text. The constraints give the label its x- and y-positions, so you shouldn't have any layout ambiguity.

Add the following lines to `detailsButtonTapped`:

```
if advancedLabel == nil {
  addAdvancedLabel()
}
```

This will call the helper method to set up the label.

Build and run, and tap the Details button. You should see the label appear in the correct spot. If you rotate the device or simulator, you'll see the label resize properly to fit the constraints.



Dive deep into a guided tour of more advanced topics like functional programming, Scene Kit, and more!

Visual format language

Creating `NSLayoutConstraint` instances one at a time works but can be a lot of code. Instead, you can use the **visual format language**, which is an ASCII-art style way to define your layout constraints.

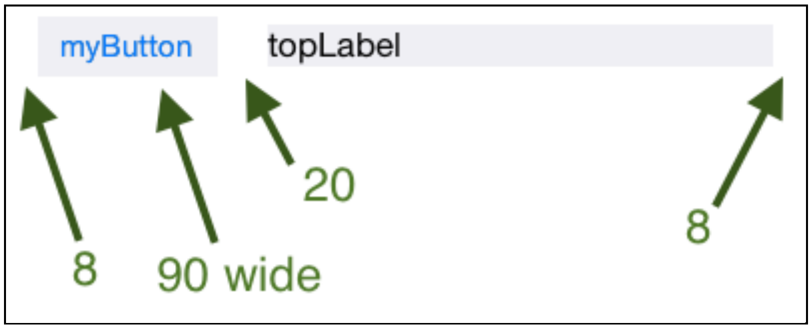
Here's an example of the visual format language:

```
H:|-8-[myButton(90)]-20-[topLabel]-8-|
```

The "H:" at the beginning means this is a representation of the horizontal layout.

The vertical bar characters represent the outer bounds of the superview. The numbers surrounded by minus signs are the margins between views. The named elements in square brackets are the views themselves.

So this means `myButton` would be 90 points wide, and have 8 points of leading space to the superview. Then there would be a 20 point margin to `topLabel`, and then 8 points from the trailing edge of `topLabel` to the superview.



You would then need a similar bit of VFL for the vertical constraints.

Let's see this in real life. Add the following helper method to the class to set up the intermediate track label:

```
private func addIntermediateLabel() {
  intermediateLabel = UILabel()
  intermediateLabel.setTranslatesAutoresizingMaskIntoConstraints(
    false)
  intermediateLabel.text = "This track is for Objective-C
developers who are not yet fully up-to-speed with Swift."
  intermediateLabel.numberOfLines = 0
  intermediateView.addSubview(intermediateLabel)

  // 1
  let views = ["intermediateLabel": intermediateLabel,
    "intermediateView": intermediateView,
    "intermediateButton": intermediateButton]
  let metrics = ["margin": 4, "verticalMargin": verticalMargin]

  // 2
  let horizontalConstraints =
    NSLayoutConstraint.constraintsWithVisualFormat(
      "H:|-margin-[intermediateLabel]-margin-|",
      options: nil, metrics: metrics, views: views)
  // 3
  let verticalConstraints =
    NSLayoutConstraint.constraintsWithVisualFormat(
      "V:[intermediateButton]-verticalMargin-[intermediateLabel]",
      options: nil, metrics: metrics, views: views)

  // 4
  NSLayoutConstraint.activateConstraints(
    horizontalConstraints + verticalConstraints)
}
```

The first bit of code is similar to the advanced label in setting up the `UILabel` and its properties. Things get a little different after that point:

1. In the sample VFL you saw names like "myButton" in there, but how does iOS know which view has which name? This is where you set that up! You can pass in a dictionary to map strings to view objects, so that's what you're setting up here.

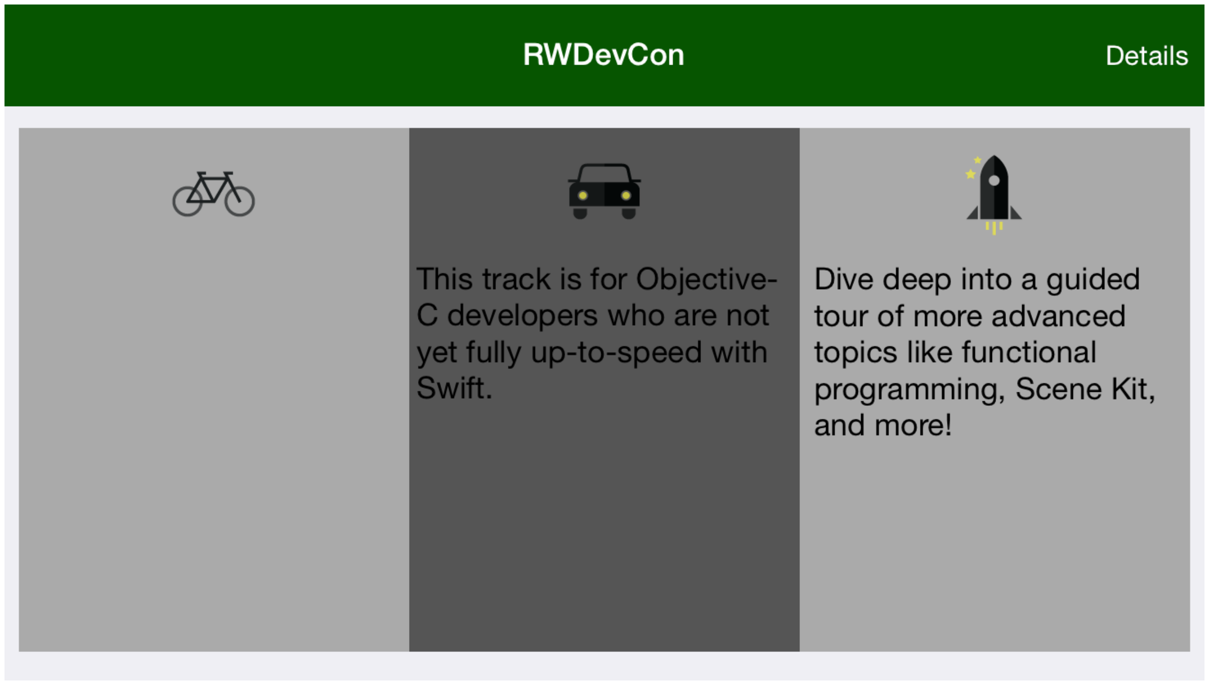
   You can do a similar thing with metrics; rather than hard-code numbers, you define another dictionary with numeric values that you can refer to later.
2. Here are the first set of visual format language constraints. This is a horizontal constraint that will center the label horizontally in the superview, with 4-point leading and trailing margins.
3. The vertical constraint here sets the y-position of the label. Since the button's position is already set, you don't need to refer to the superview. Instead, you're setting the label to be 8 points (that's the `verticalMargin`) below the button.
4. If you have many constraints to add at the same time, you can pass in an entire array of NSLayoutConstraint objects to the `activateConstraints` class method. This is more efficient than setting each `active` property one at a time.

Note that `constraintsWithVisualFormat()` has an "s" at the end of the word constraint**s** since it returns an array of constraints. You could have a whole set of views in your VFL string, so one string could map to many constraints.

Now that the helper method is set up, update `detailsButtonTapped` to add the new method call:

```
if advancedLabel == nil {
  addAdvancedLabel()
  addIntermediateLabel()
}
```

Build and run, tap the Details button, and you'll see both labels in place!



This track is for Objective-C developers who are not yet fully up-to-speed with Swift.

Dive deep into a guided tour of more advanced topics like functional programming, Scene Kit, and more!

Modifying constraints

Tapping the Details button will show the labels, but it then makes sense for it to toggle them on/off.

You could change the `alpha` (transparency) value or the `hidden` property of a view to hide it. Or, you could update its position to move it off-screen. The effect is the same, but the exact technique could make a difference, say if there were animations involved.

Add the following helper method to toggle the two labels on and off:

```
private func toggleLabel() {
  if intermediateLabel.alpha == 0 {
    intermediateLabel.alpha = 1
    advancedVerticalConstraint.constant = verticalMargin
  } else {
    intermediateLabel.alpha = 0
    advancedVerticalConstraint.constant =
      CGRectGetHeight(advancedView.frame)
  }
}
```

You're changing the intermediate label's `alpha` to 1 to show it, and 0 to hide it.

For the advanced label, you saved its vertical position constraint when you set it up. To show the label, you're changing the constant to its initial value of `verticalMargin`, currently 8.0. Remember, that's the y-position for the number of points below the button.

To hide the label, you're setting the constant to the height of the container frame. That will push the label down below the bounds of the screen and effectively hide it.

Finally, update the implementation of detailsButtonTapped with the following:

```
if advancedLabel == nil {
  addAdvancedLabel()
  addIntermediateLabel()
} else {
  toggleLabel()
}
```

If this is the first button tap, the method will add the two labels. On subsequent taps, the method will call your new `toggleLabel()` method.

Build and run the app, and tap the Details button a few times. You've now created constrains two different ways from code, and also updated an existing constraint on the fly!

Where to go from here?

You've seen many different ways to set up auto layout constraints: the Pin button on your storyboard, from the menu, control-dragging between views, and from code. iOS offers many ways for you to customize and modify your layout and you should take advantage of this flexible technology to get your apps looking good on all screen sizes.

There's still one more challenge ahead in this session! Given a screenshot of a completed view, you'll need to create and lay it out from scratch. Check out the challenge document for all the details!