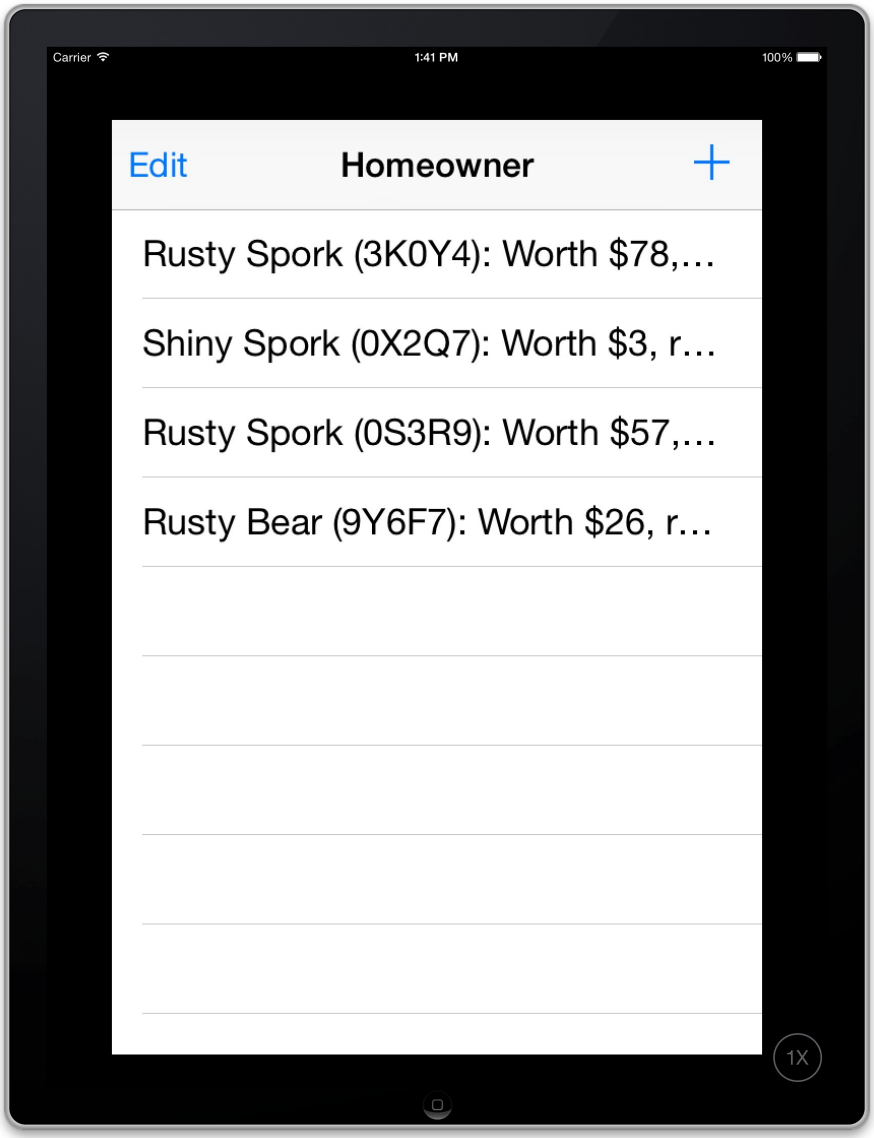
Chapter 15
Introduction to Auto Layout

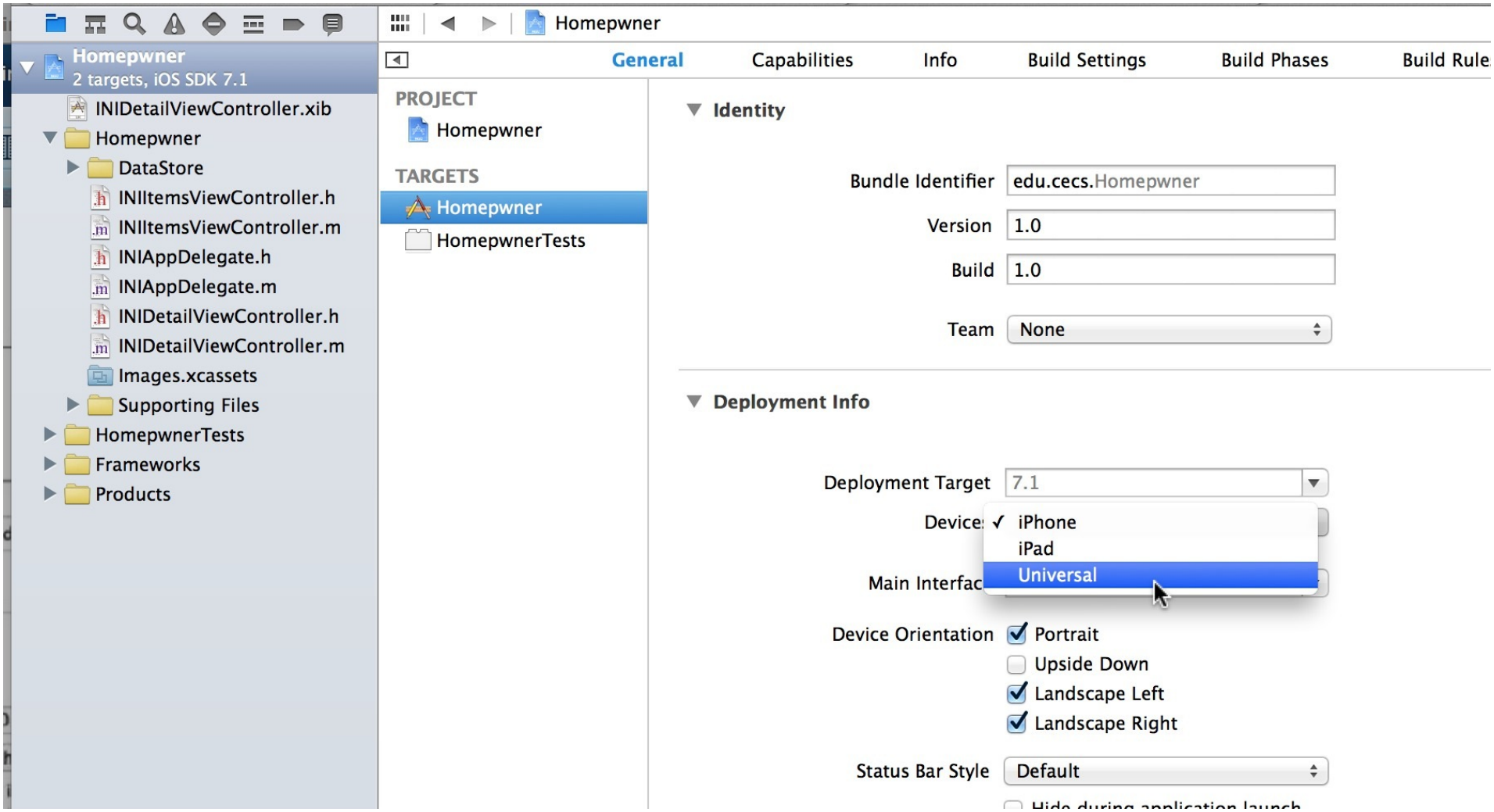
- The Auto Layout System
- Alignment rectangle and layout attributes
- Constraints
- Priorities
- Debugging Constraints

Current Homeowner Applications

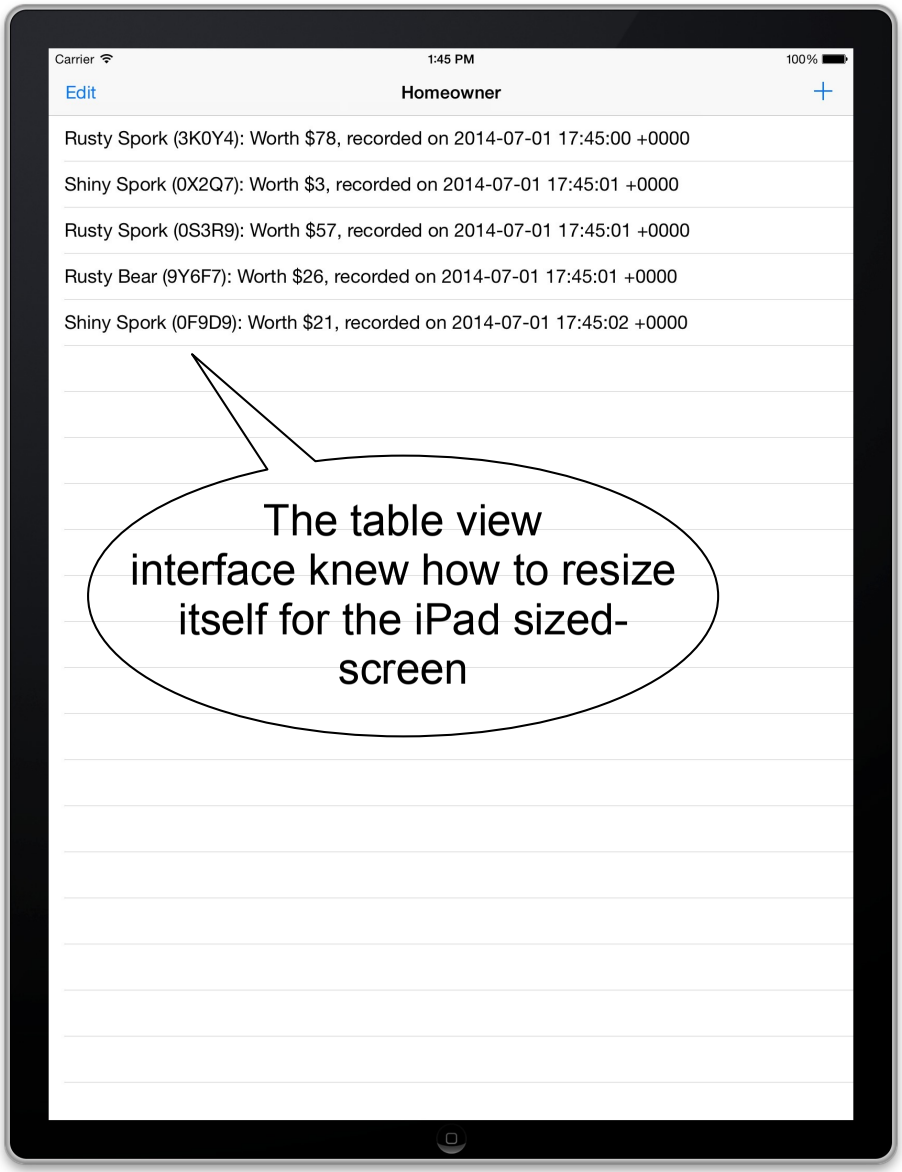
- Our Homeowner application can be run on the iPad simulator, but it will not look right.
- You want Homeowner to run natively on the iPad so that it will look like an iPad app. A single application that runs natively on both the iPad and the iPhone is called a universal application.

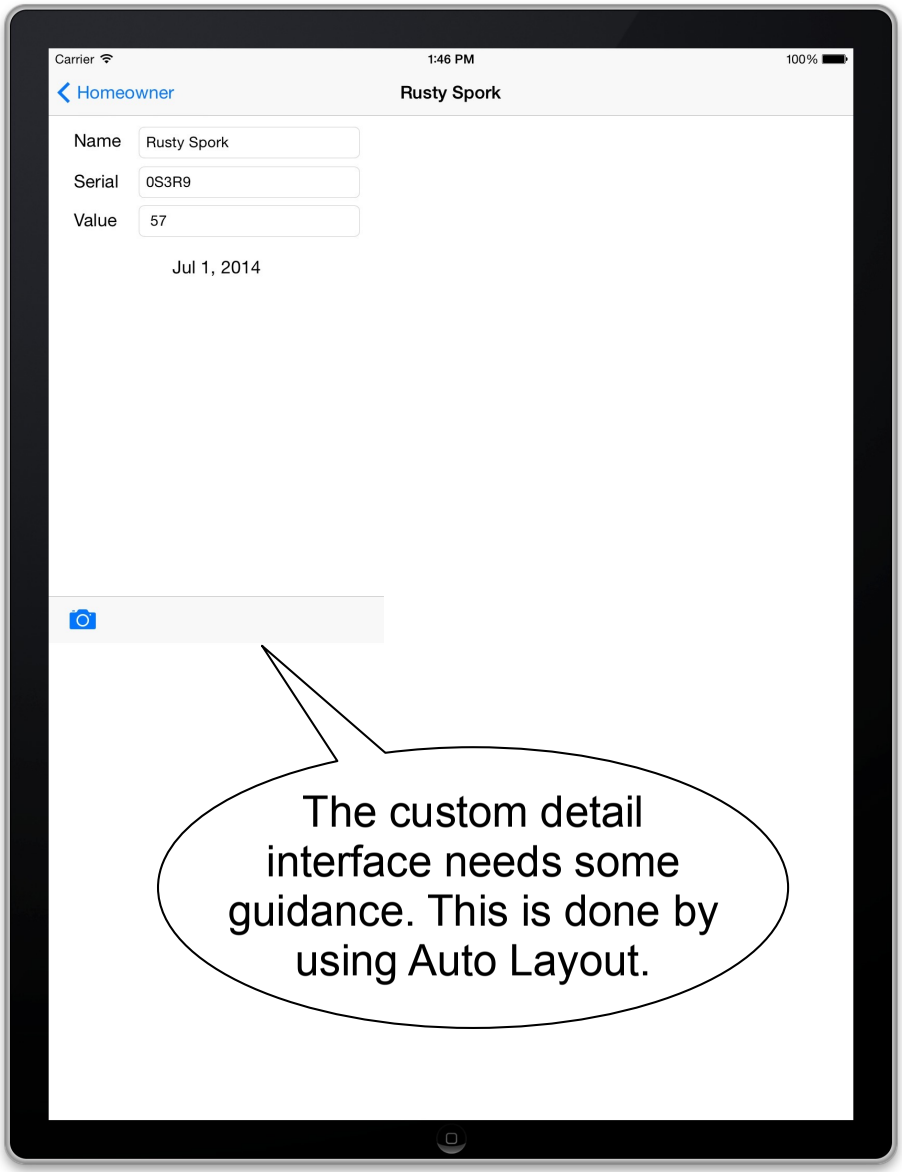


Universalizing Applications



The New Look


Carrier
1:45 PM
100%
Edit
Homeowner
Rusty Spork (3K0Y4): Worth $78, recorded on 2014-07-01 17:45:00 +0000
Shiny Spork (0X2Q7): Worth $3, recorded on 2014-07-01 17:45:01 +0000
Rusty Spork (0S3R9): Worth $57, recorded on 2014-07-01 17:45:01 +0000
Rusty Bear (9Y6F7): Worth $26, recorded on 2014-07-01 17:45:01 +0000
Shiny Spork (0F9D9): Worth $21, recorded on 2014-07-01 17:45:02 +0000
The table view interface knew how to resize itself for the iPad sized-screen


Carrier
1:46 PM
100%
Homeowner
Rusty Spork
Name
Rusty Spork
Serial
0S3R9
Value
57
Jul 1, 2014
The custom detail interface needs some guidance. This is done by using Auto Layout.

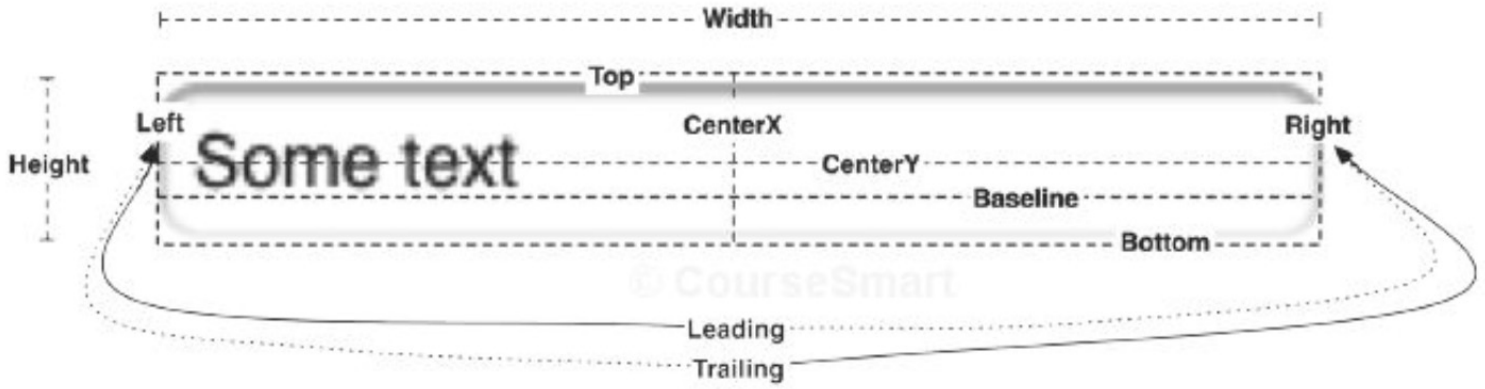
The Auto Layout System

- Up until now we used view's frame to specify its size and position relative to its superview.
- We have defined the frames of our views with absolute coordinates.
- Absolute coordinates make our layout fragile because they assume that we know the size of the screen ahead of time.
- Using Auto Layout, we will describe the layout of our views in a relative way that allows the frames to be determined at runtime so that the frames' definitions can take into account the screen size of the device that the application is running on.

Device	Width x Height (points)
iPhone/iPod (4S and earlier)	320 x 480
iPhone/iPod (5 and later)	320 x 568
All iPads	768 x 1024

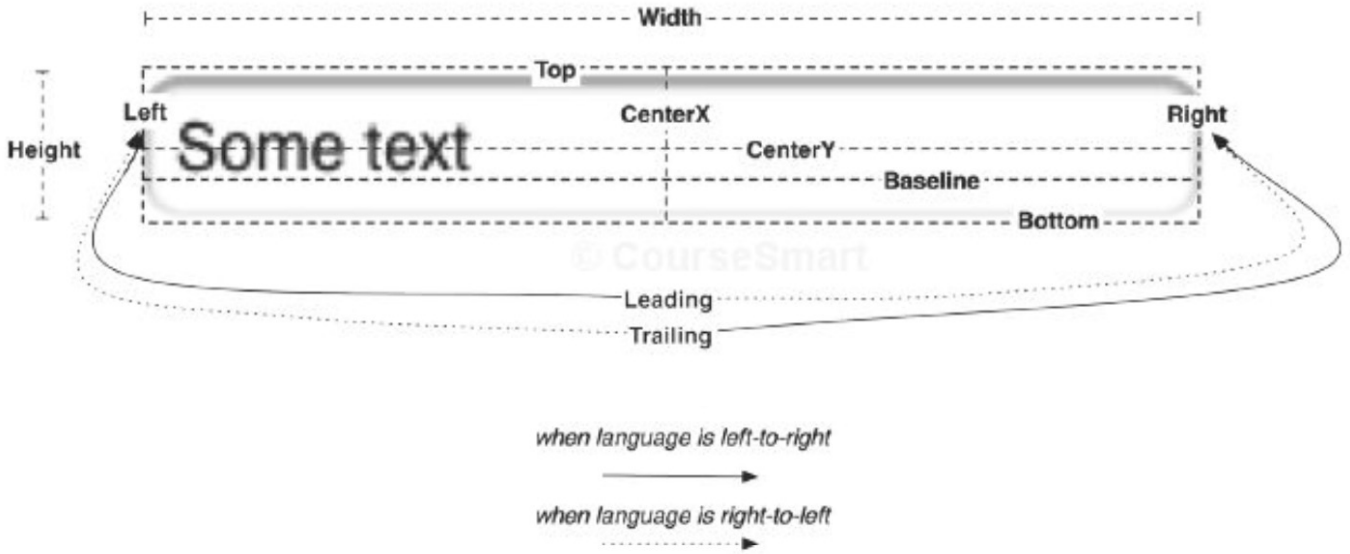
Alignment rectangle and layout attributes

- The Auto Layout system works with yet another rectangle for a view – the alignment rectangle. This rectangle is defined by several layout attributes



Width/Height	These values determine the alignment rectangle's size.
Top/Bottom/Left/Right	These values determine the spacing between the given edge of the alignment rectangle and the alignment rectangle of another view in the hierarchy.
CenterX/CenterY	These values determine the center point of the alignment rectangle.
Baseline	This value is the same as the bottom attribute for most, but not all, views. For example, UITextField defines its baseline to be the bottom of the text it displays rather than the bottom of the alignment rectangle. This keeps " descenders" (letters like ' g' and ' p' that descend below the baseline) from being obscured by a view right below the text field.

Alignment rectangle and layout attributes (cont.)



Leading/Trailing	These values are used with text- based views like UITextField and UILabel. If the language of the device is set to a language that reads left-to- right (e. g., English), then the leading attribute is the same as the left attribute and the trailing attribute is the same as the right attribute. If the language reads right-to- left (e. g., Arabic), then the leading attribute is on the right and the trailing attribute is on the left.

Alignment rectangle and layout attributes (cont.)

- By default, every view in a XIB file has an alignment rectangle, and every view hierarchy uses Auto Layout.
- But the default will not always work as we want. This is when you need to step in.
- We do not define a view's alignment rectangle directly. We do not have enough information (screen size!) to do that.
- We provide a set of constraints. Taken together, these constraints allow the system to determine the layout attributes, and thus the alignment rectangle, for each view in the view hierarchy.

Constraints

- A constraint defines a specific relationship in a view hierarchy that can be used to determine a layout attribute for one or more views.
- We might add a constraint like

  the vertical space between these two views should always be 8 points

  these views must always have the same width
- A constraint can also be used to give a view a fixed size

  this view's height should always be 44 points.
- We do not need to have a constraint for every layout attribute. Some values may come directly from a constraint; others will be computed by the values of related layout attributes.
- Ambiguous or missing value for a layout attribute will generate errors and warnings from Auto Layout

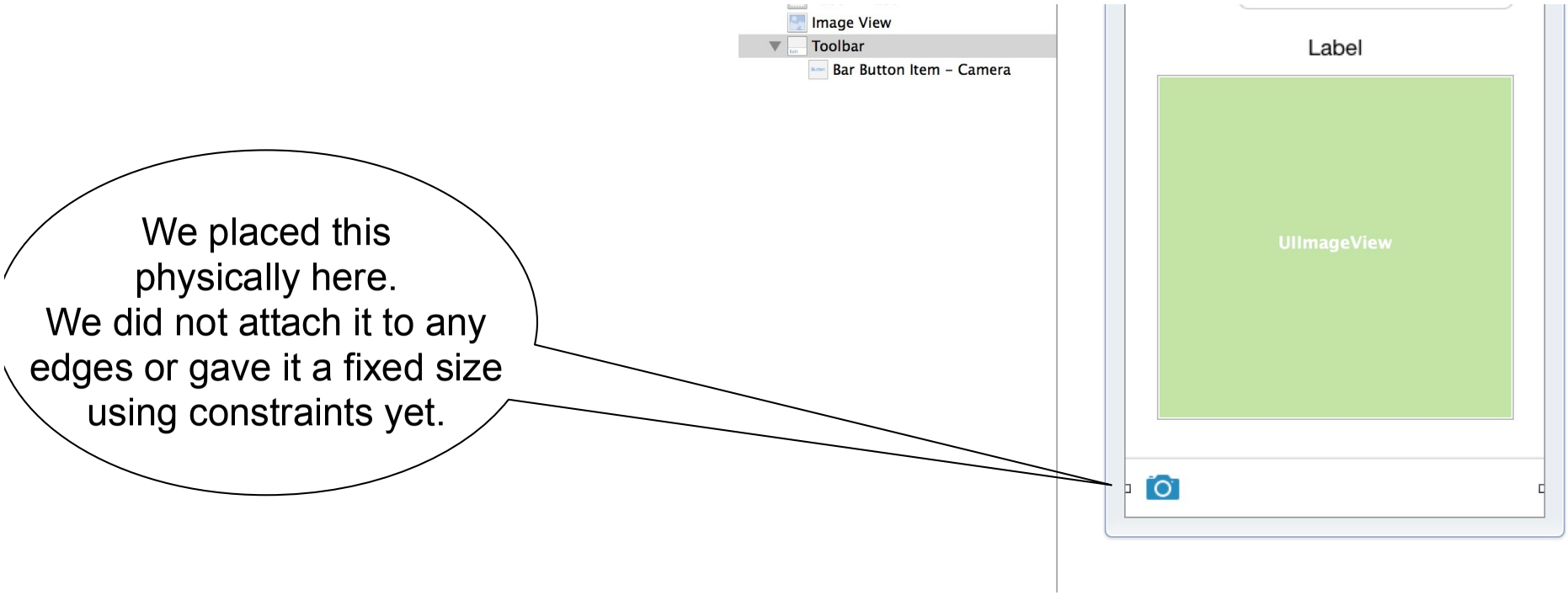
Constraining the Toolbar

- We must decide on how we want the view to look like regardless of screen size.
- For the toolbar:

  should sit at the bottom of the screen.

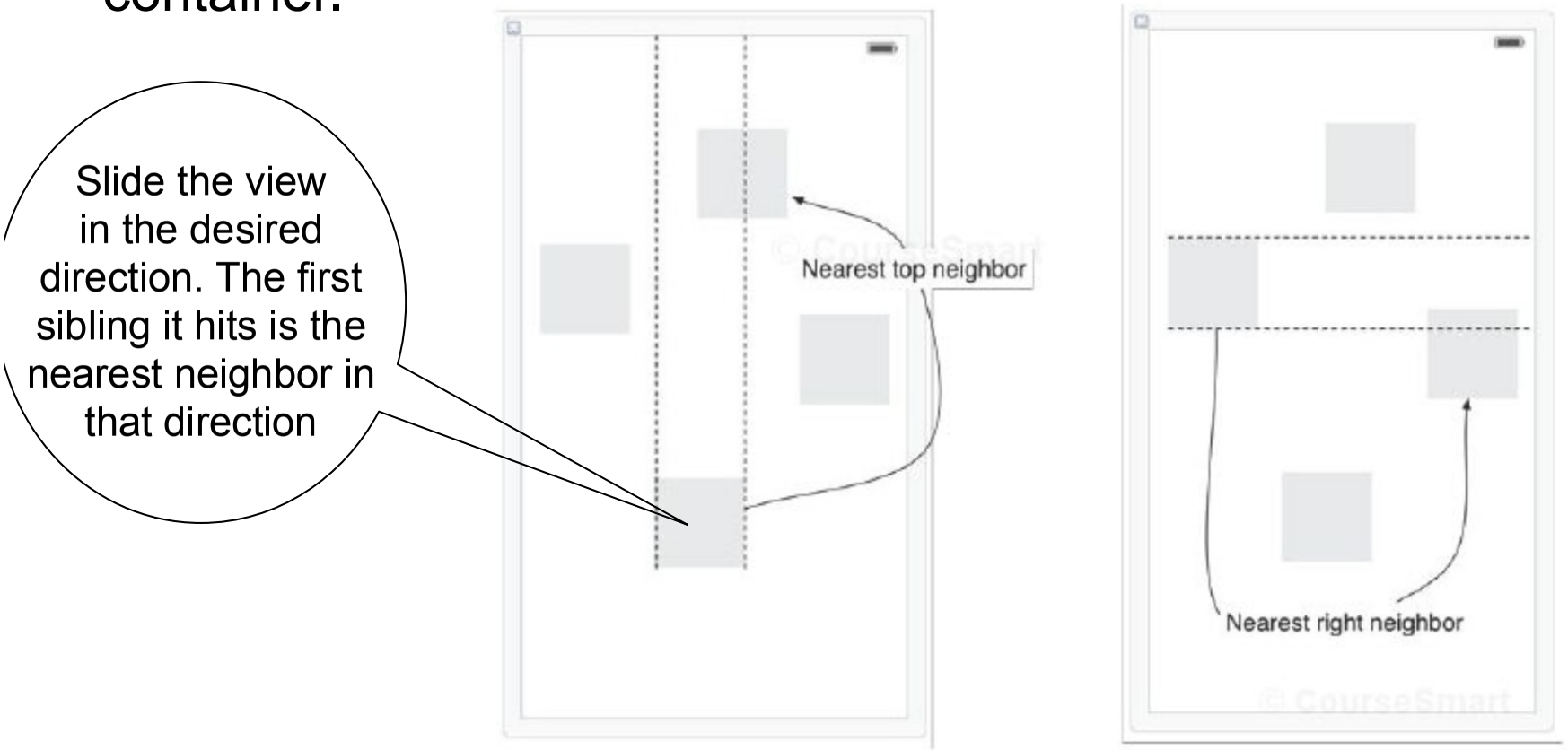
  should be as wide as the screen.

  has height of 44 points. (This is Apple's standard for UIToolbar.)



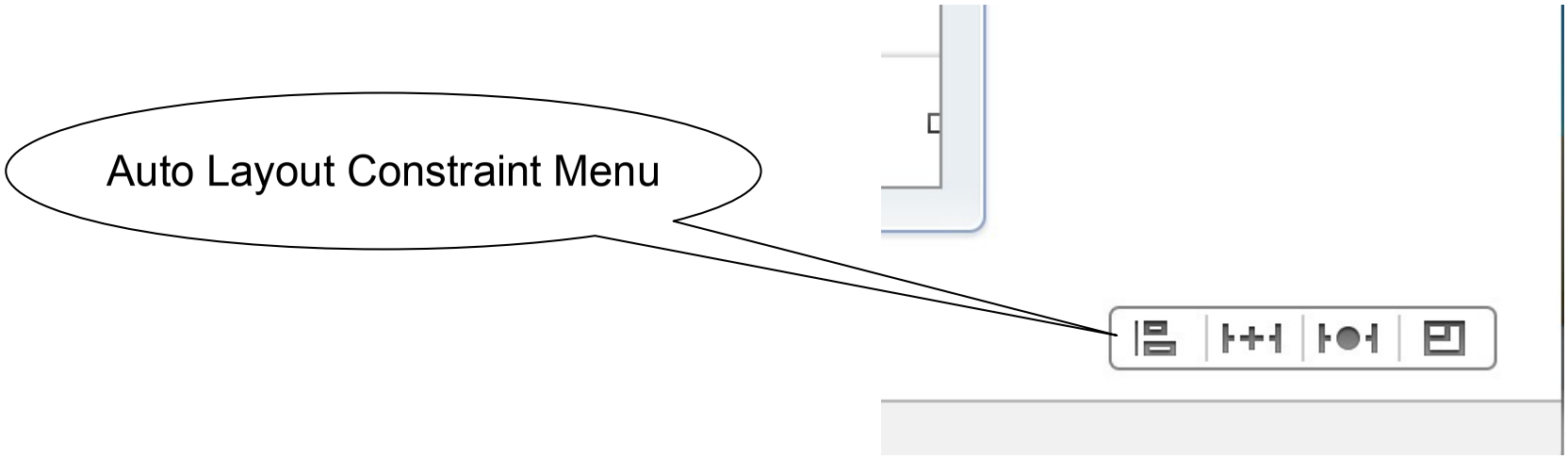
Nearest Neighbor

- The nearest neighbor is the closest sibling view in the specified direction.
- If a view does not have any siblings in the specified direction, then the nearest neighbor is its superview, also known as its container.

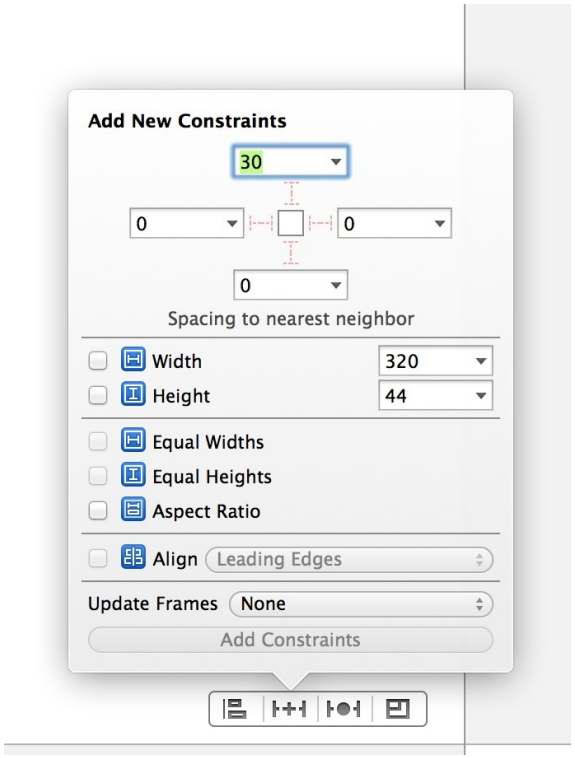


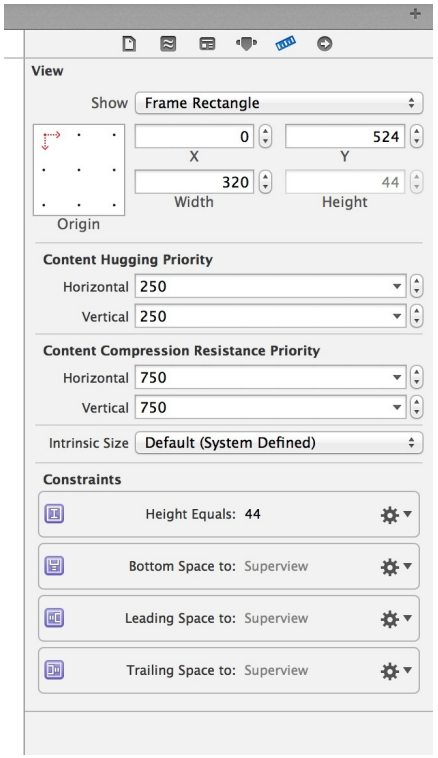
Spelling Out The Constraints For The toolbar

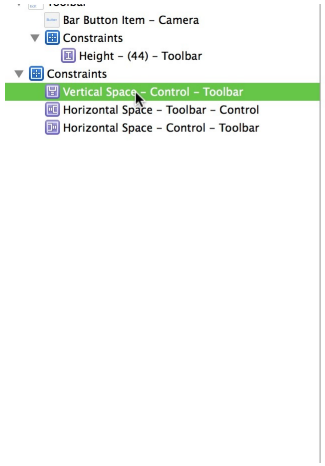
- The toolbar's bottom edge should be 0 points away from its nearest neighbor (which is its container – the view of the INIDetailViewController).
- The toolbar's left edge should be 0 points away from its nearest neighbor.
- The toolbar's right edge should be 0 points away from its nearest neighbor.
- The toolbar's height should be 44 points.

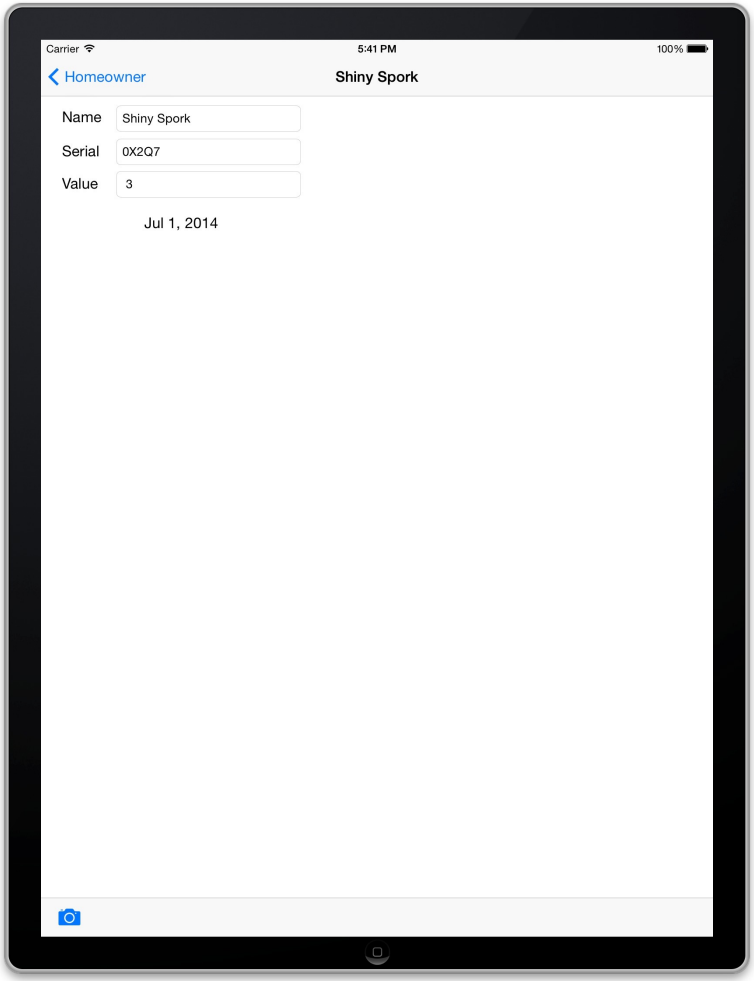


Pinning The toolbar

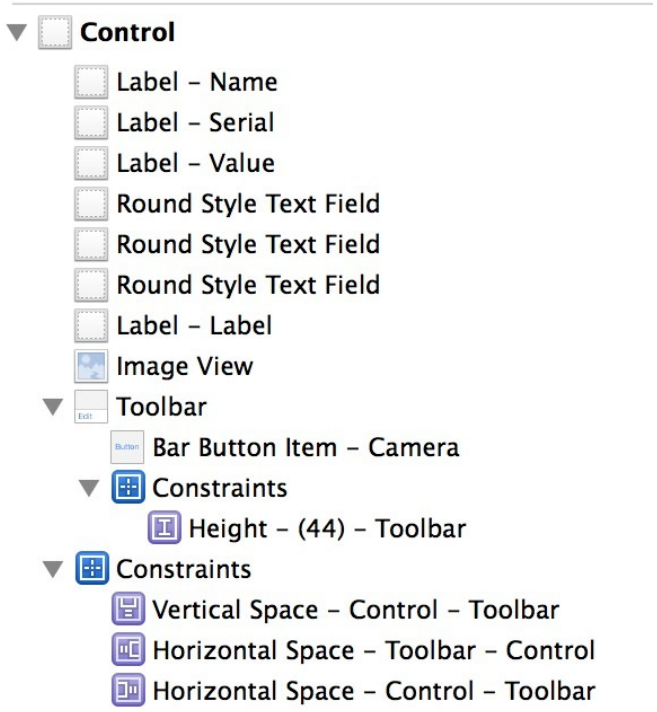






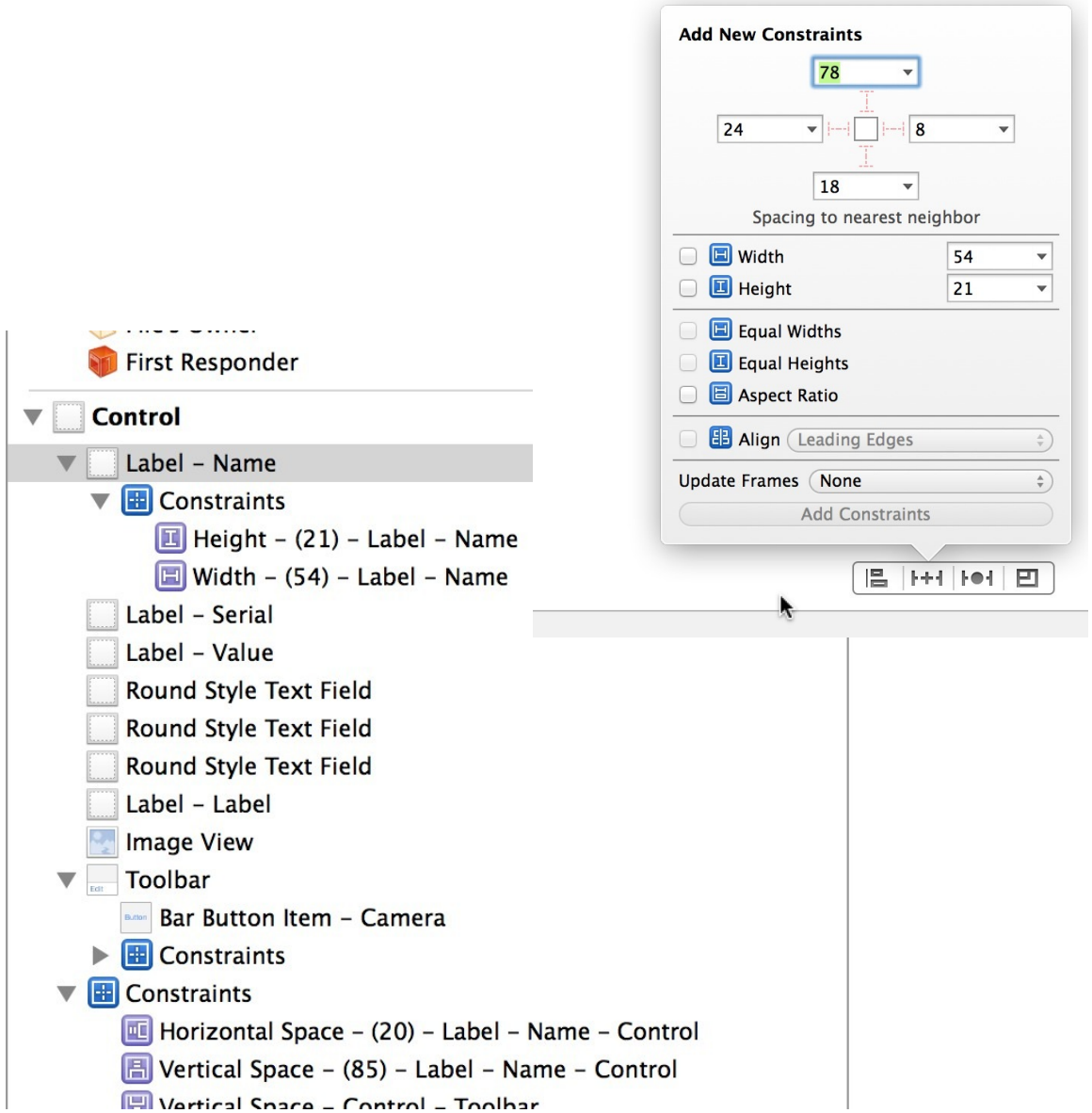


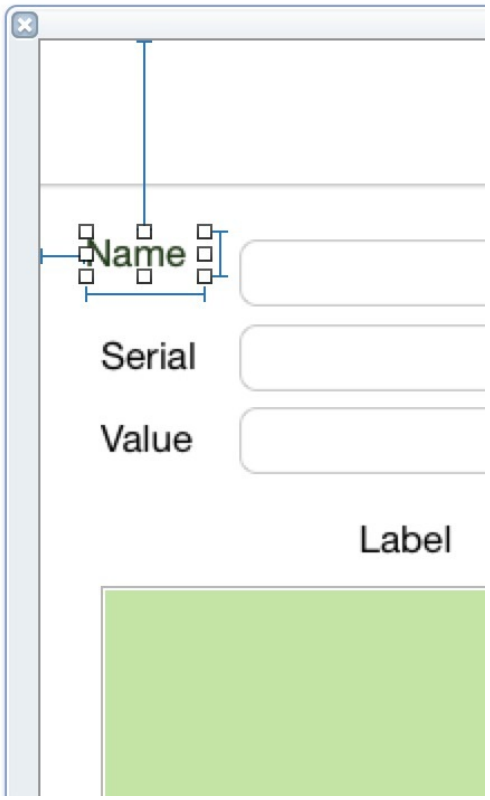
Constrains Display in The Dock



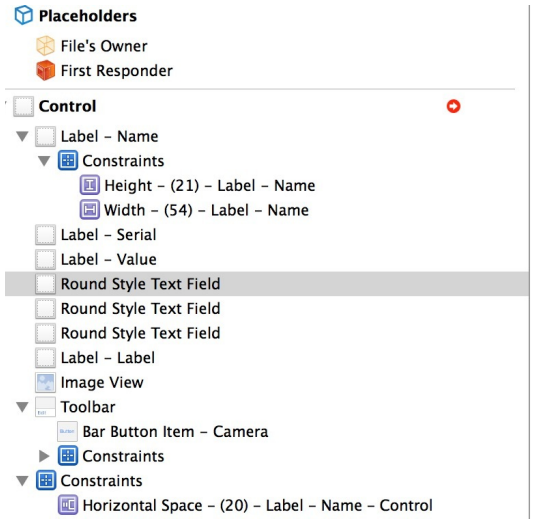
- You can see the constraints that you just added in the dock to the left of the canvas.
- The are three constraints in one constraints section
  - These are ehe edge constraints.
  - They are added to the Control because they apply to both the toolbar and its superview
- The fourth constraint, the fixed height of the toolbar, is in a separate Constraints section underneath Toolbar

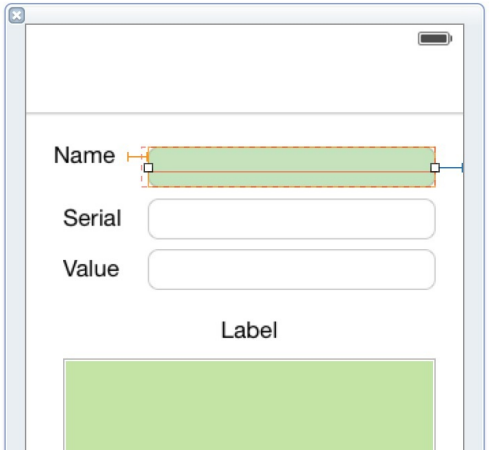
Adding more constraints (Name Label)

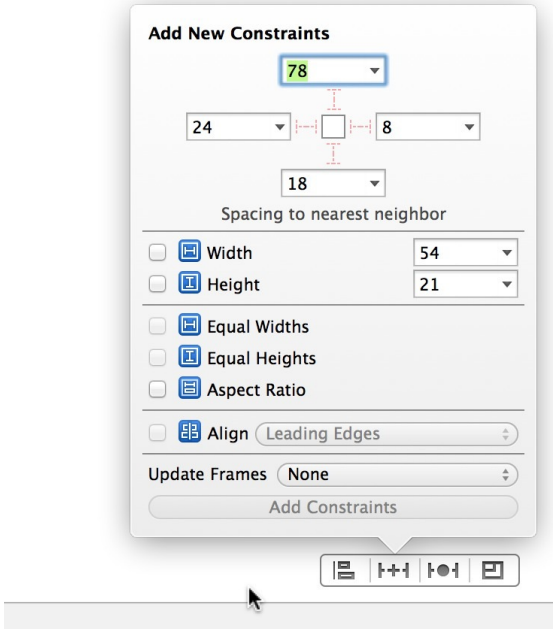


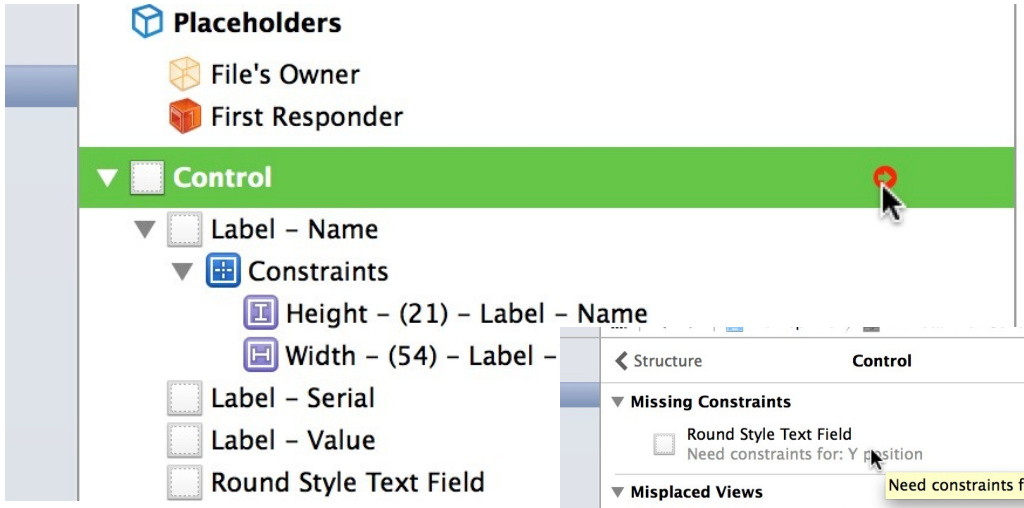


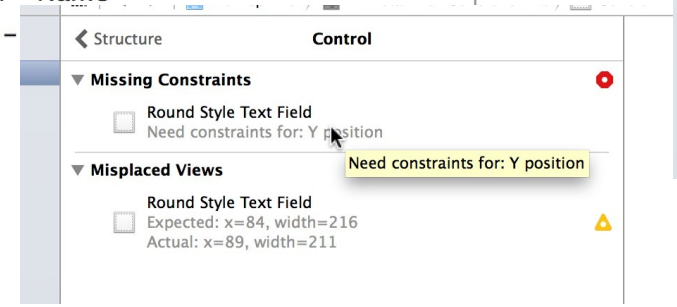
Adding more constraints (Name Field)

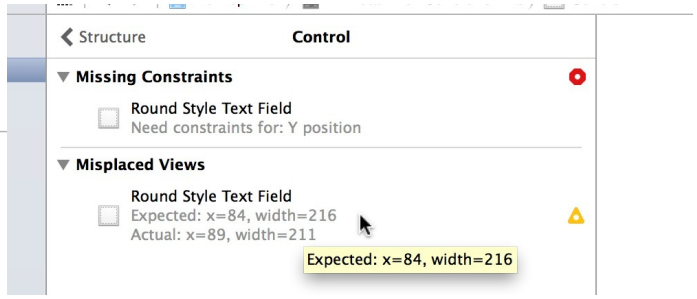




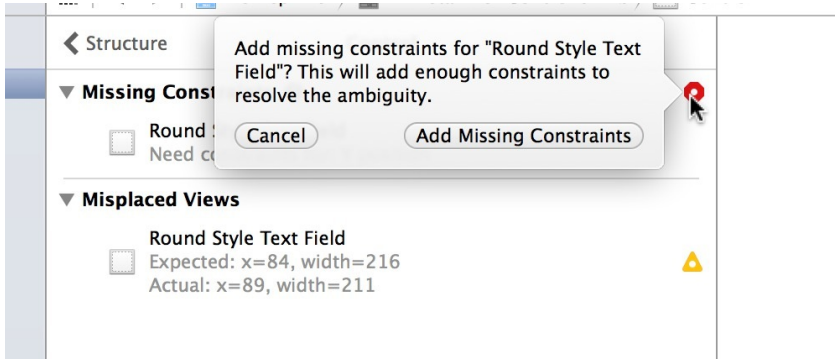


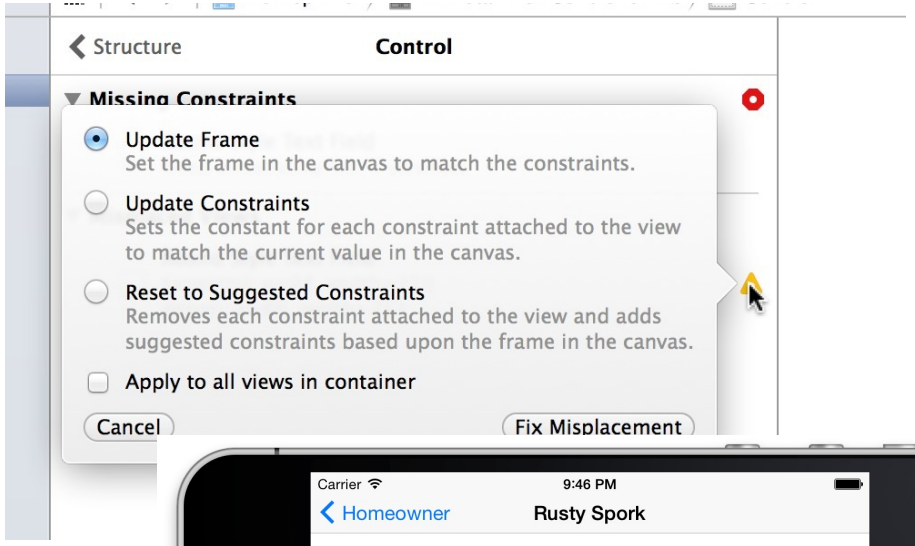


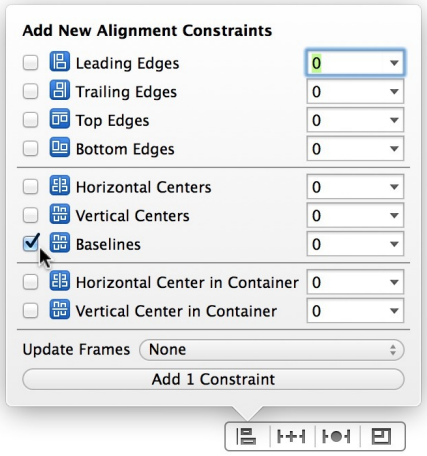


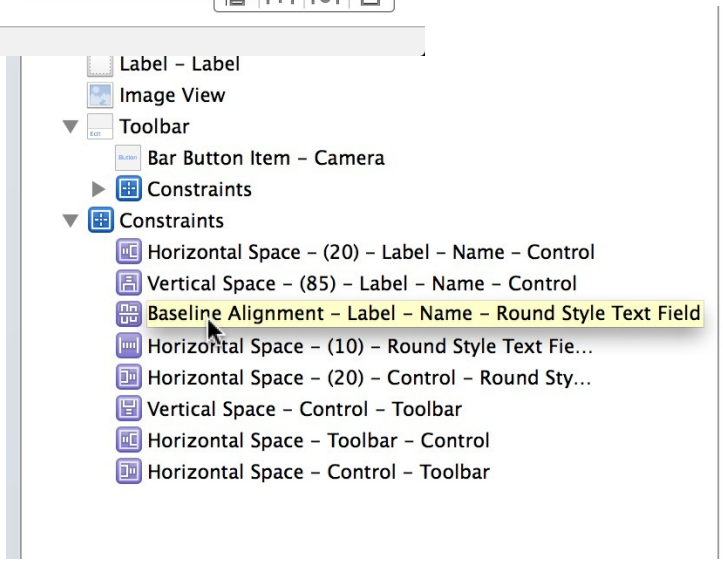


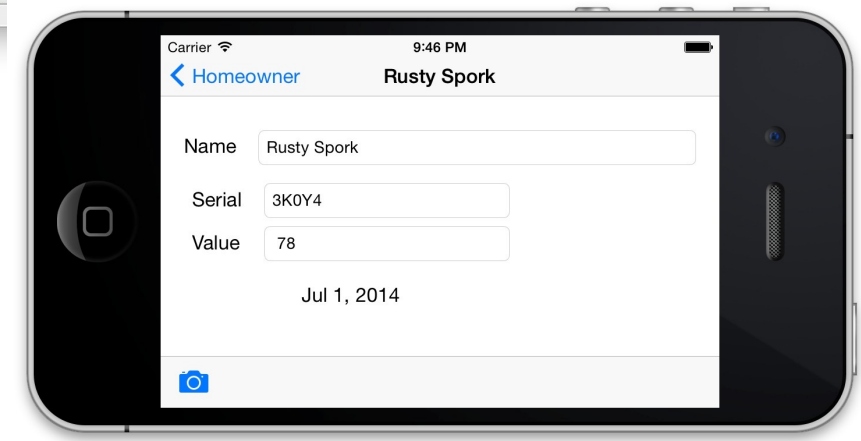
Fixing missing and bad constraints (Name Field)



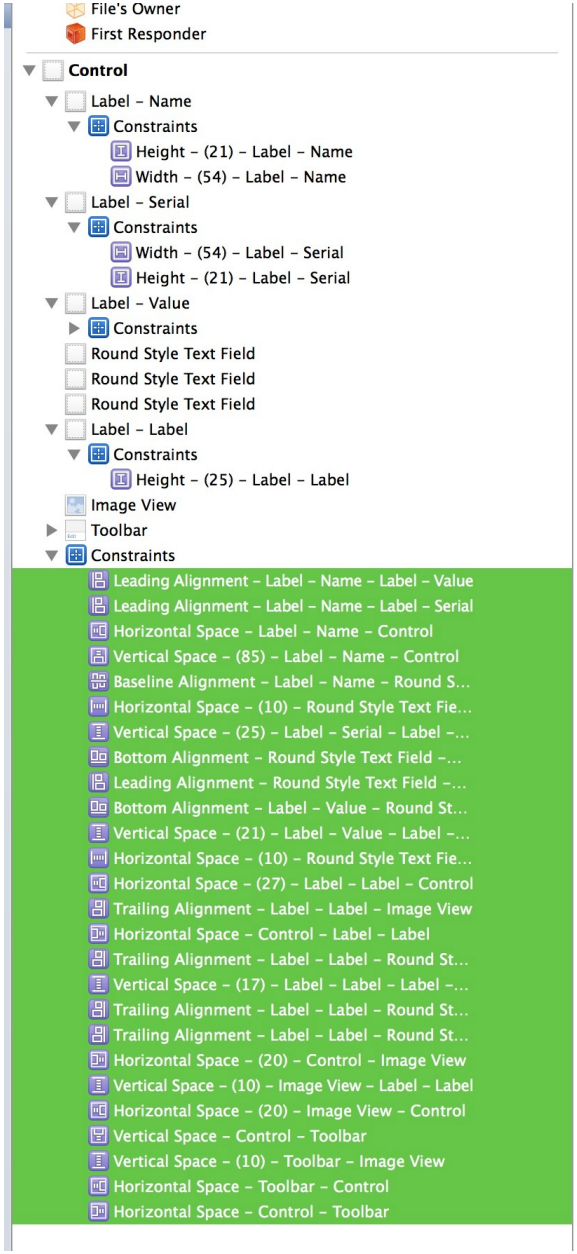


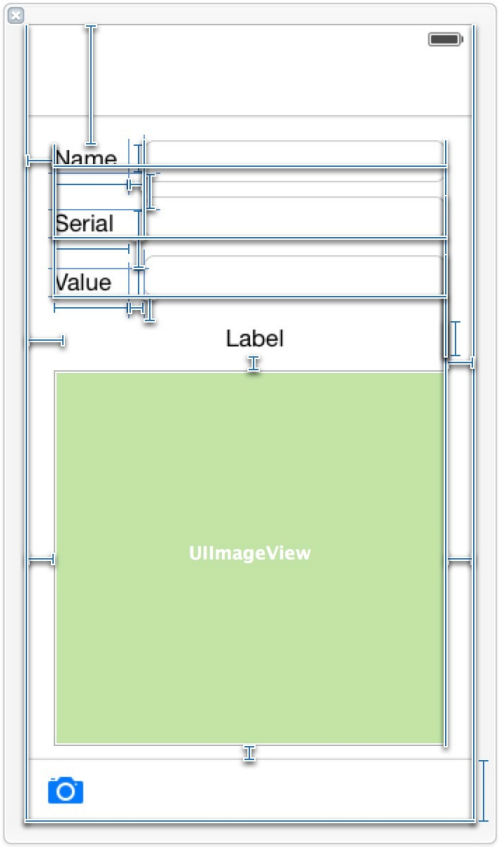




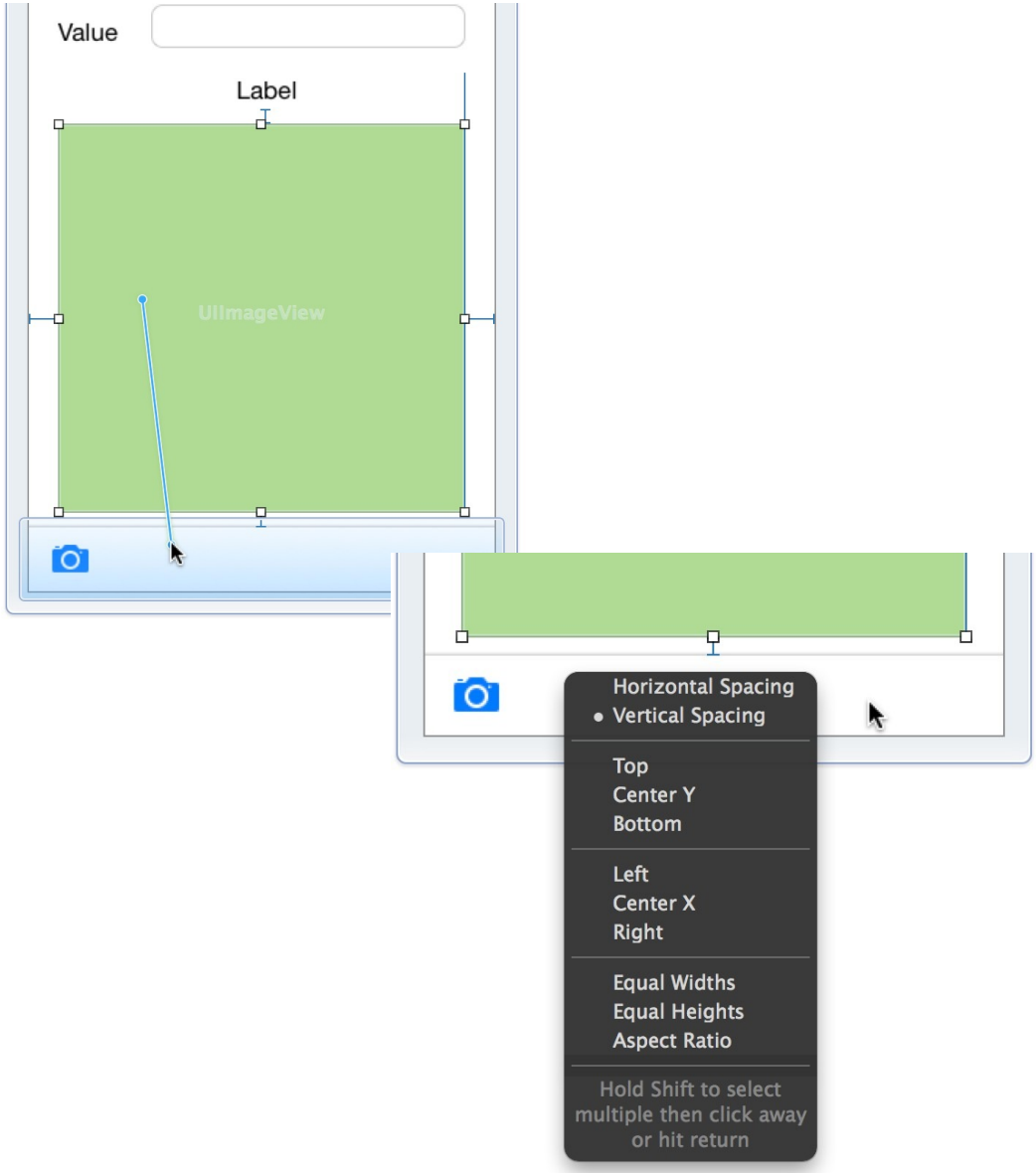


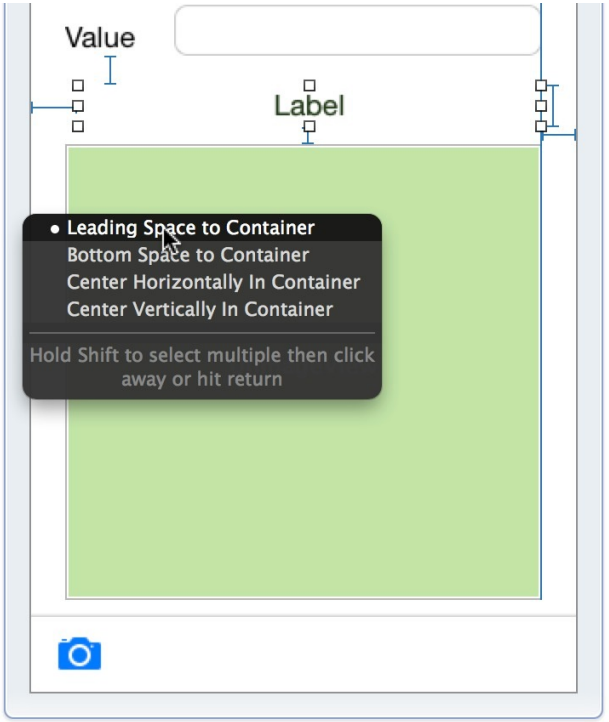
Adding even more constraints



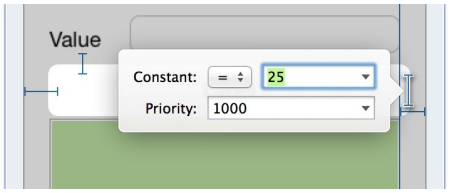


Adding constraint by Control- dragging between views





Priorities



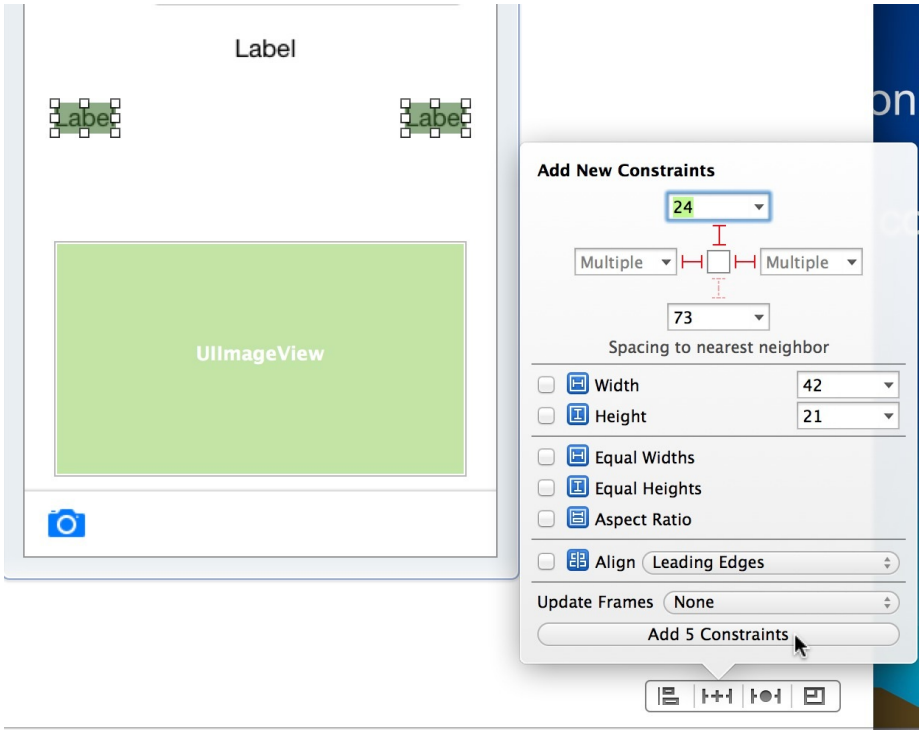
- Each constraint has a priority level that is used to determine which constraint wins when more than one constraint conflicts.
- A priority is a value from 1 to 1000, where 1000 is a required constraint.
- By default, constraints are required
- This means that the priority level would not help if you had conflicting constraints. Instead, Auto Layout would report an issue regarding unsatisfiable constraints.
- Typically, you find the constraints that conflict and then either remove one or reduce the priority level of a constraint to resolve the conflict but keep all the constraints in play

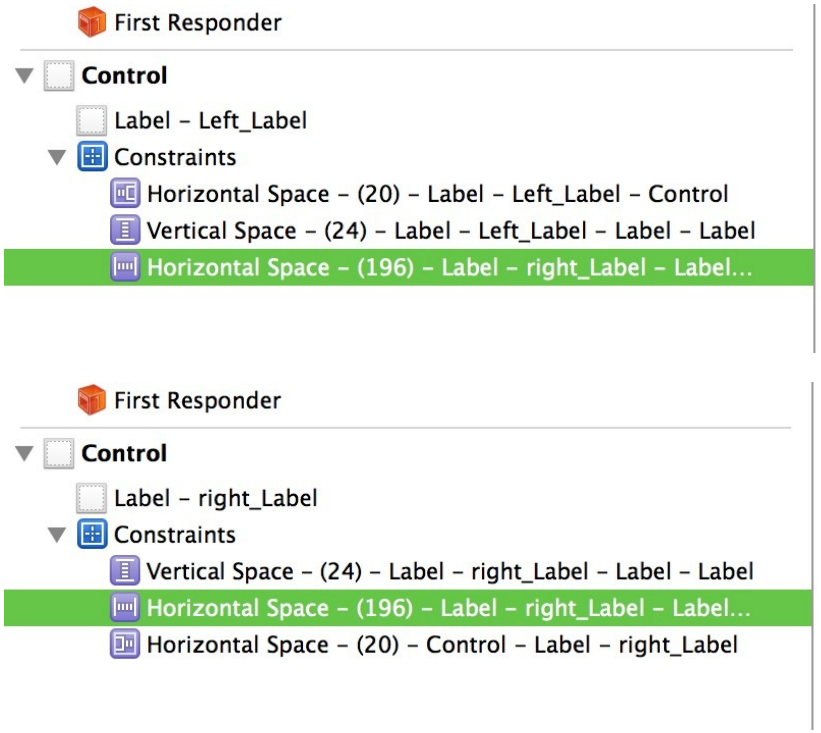
Debugging Constraints

- Given the sheer number of constraints, it is easy to introduce problems:
  - Miss a constraint
  - Introduce constraints that conflict with each other
  - Introduce constraint that conflicts with how a view appears on the canvas.

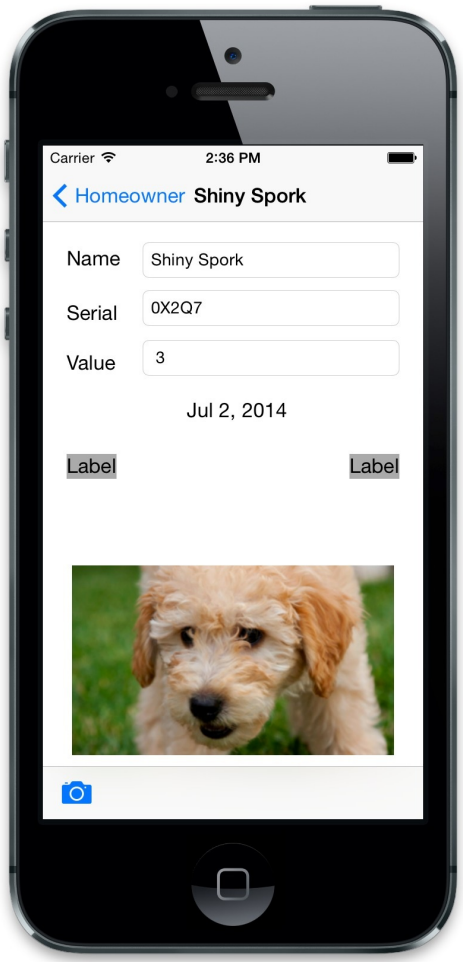
Ambiguous layout

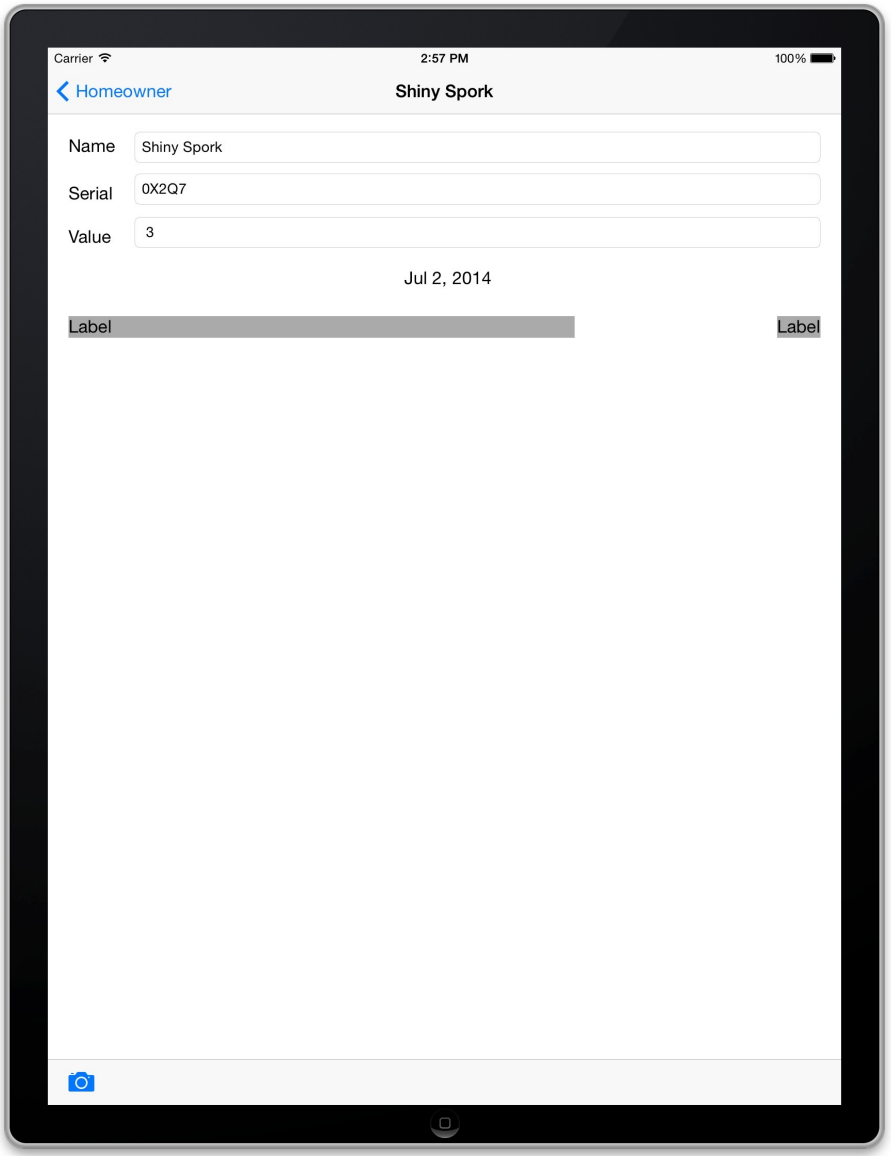
- An ambiguous layout occurs when there is more then one way to fulfill a set of constraints.
- Typically, this means that you are missing at least one constraint.





Labels Do Not Look Right





Testing for Ambiguous Constraints

- viewDidLayoutSubviews gets called any time the view changes in size (for example, when the device is rotated) or when it is first presented on the screen.

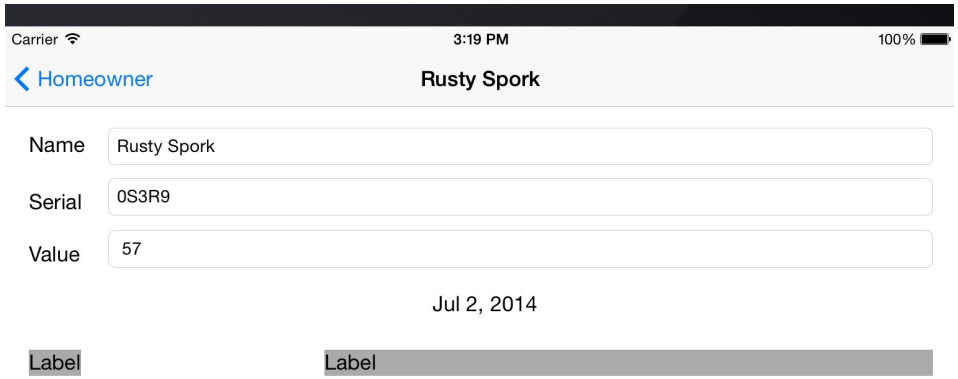
```
- (BOOL)textFieldShouldReturn:(UITextField *) textField
{
    [textField resignFirstResponder];
    return YES;
}

- (void) viewDidLayoutSubviews {

    NSLog(@"%@", NSStringFromSelector(_cmd));
    for ( UIView *subview in self.view.subviews) {
        if ([subview hasAmbiguousLayout]) {
            NSLog(@"AMBIGUOUS: %@", subview);
        }
    }
}
```

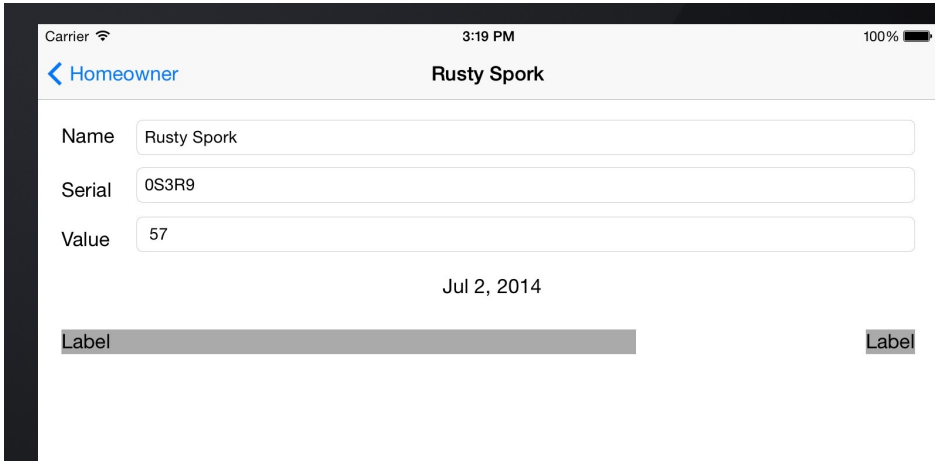
Homepwner

```
2014-07-02 15:10:09.633 Homepwner[895:60b] viewDidLayoutSubviews
2014-07-02 15:10:09.634 Homepwner[895:60b] AMBIGUOUS: <UILabel: 0x8f8ded0; frame = (20 260; 490 21); text = 'Label'; clipsToBounds = YES;
opaque = NO; autoresize = RM+BM; userInteractionEnabled = NO; layer = <CALayer: 0x8f8d8d0>>
2014-07-02 15:10:09.635 Homepwner[895:60b] AMBIGUOUS: <UILabel: 0x8f8e090; frame = (706 260; 42 21); text = 'Label'; clipsToBounds = YES;
opaque = NO; autoresize = RM+BM; userInteractionEnabled = NO; layer = <CALayer: 0x8f8e140>>
2014-07-02 15:10:10.145 Homepwner[895:60b] viewDidLayoutSubviews
2014-07-02 15:10:10.145 Homepwner[895:60b] AMBIGUOUS: <UILabel: 0x8f8ded0; frame = (20 260; 490 21); text = 'Label'; clipsToBounds = YES;
opaque = NO; autoresize = RM+BM; userInteractionEnabled = NO; layer = <CALayer: 0x8f8d8d0>>
2014-07-02 15:10:10.146 Homepwner[895:60b] AMBIGUOUS: <UILabel: 0x8f8e090; frame = (706 260; 42 21); text = 'Label'; clipsToBounds = YES;
opaque = NO; autoresize = RM+BM; userInteractionEnabled = NO; layer = <CALayer: 0x8f8e140>>
```

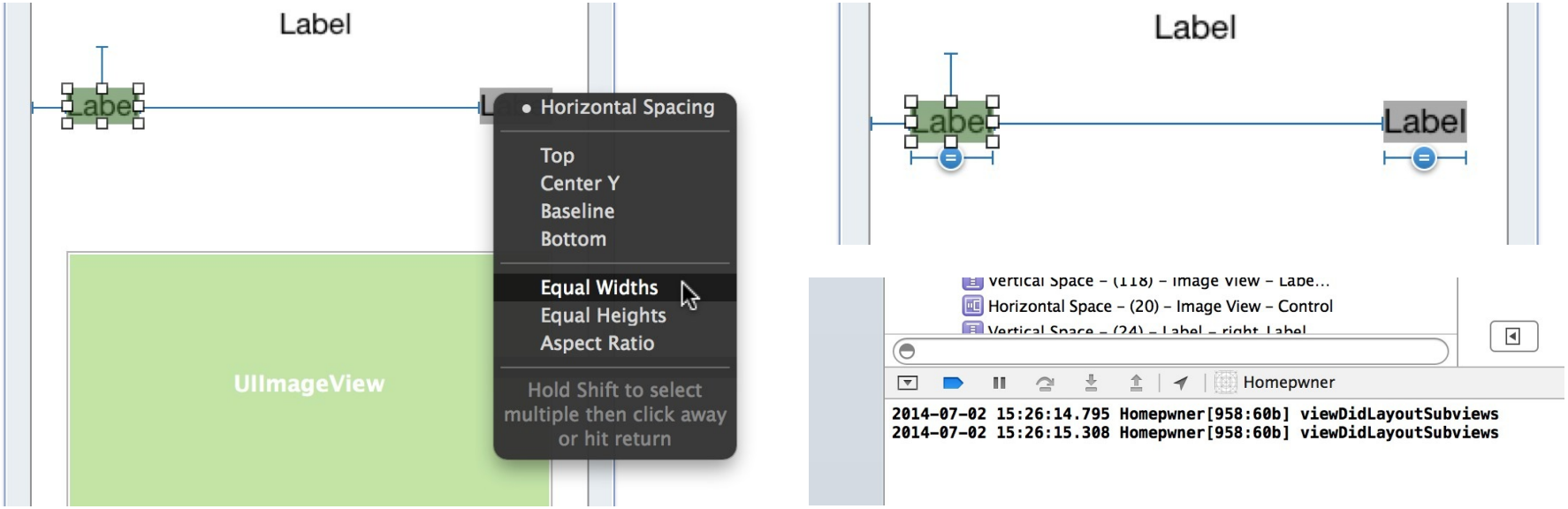
Demonstrating the other possible layout

```
30  }
31
32  - (IBAction)backgroundTapped:(id)sender {
33      [self.view endEditing:YES];
34      for ( UIView *subview in self.view.subviews) {
35          if ([subview hasAmbiguousLayout]) {
36              [subview exerciseAmbiguityInLayout];
37          }
38      }
39  }
```



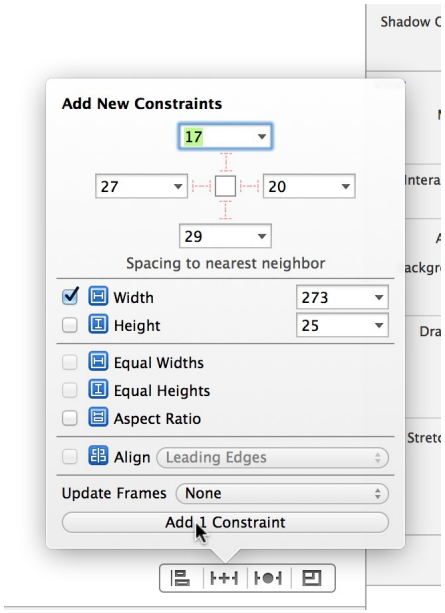


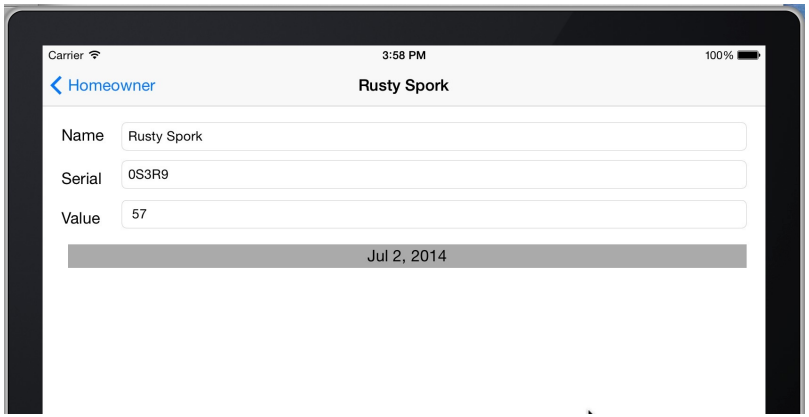
Removing Ambiguity

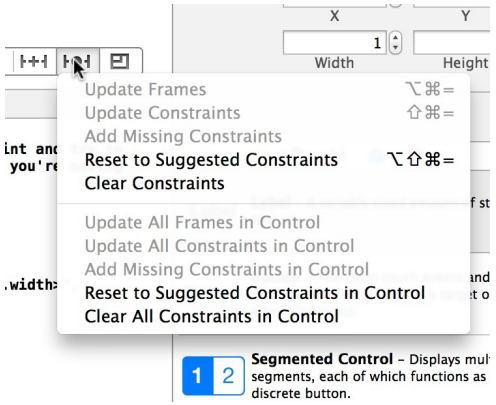


Unsatisfiable Constraints

- The problem of unsatisfiable constraints occurs when two or more constraints conflict.
- This often means that a view has too many constraints.








```
2014-07-02 15:58:08.272 Homepwner[1406:60b] Unable to simultaneously satisfy constraints.
    Probably at least one of the constraints in the following list is one you don't want. Try this: (1) look at each constraint and try to
figure out which you don't expect; (2) find the code that added the unwanted constraint or constraints and fix it. (Note: If you're seeing
NSAutoresizingMaskLayoutConstraints that you don't understand, refer to the documentation for the UIView property
translatesAutoresizingMaskIntoConstraints)
(
    "<NSLayoutConstraint:0x8ca7c60 H:[UILabel:0x8ca7ac0(273)]>",
    "<NSLayoutConstraint:0x8caa8b0 H:|-(27)-[UILabel:0x8ca7ac0]   (Names: '|':UIControl:0x8c975c0 )>",
    "<NSLayoutConstraint:0x8caa910 H:[UILabel:0x8ca7ac0]-(NSSpace(20))-|   (Names: '|':UIControl:0x8c975c0 )>",
    "<NSAutoresizingMaskLayoutConstraint:0x8cbf300 h=--& v=--& UIControl:0x8c975c0.width == _UIParallaxDimmingView:0x8c97510.width>",
    "<NSAutoresizingMaskLayoutConstraint:0x8cc0140 h=--& v=--& H:[_UIParallaxDimmingView:0x8c97510(768)]>"
)

Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x8ca7c60 H:[UILabel:0x8ca7ac0(273)]>

Break on objc_exception_throw to catch this in the debugger.
The methods in the UIConstraintBasedLayoutDebugging category on UIView listed in <UIKit/UIView.h> may also be helpful.
```

Misplaced views