# iOS Dev Accelerator
# Week 3 Day 4

- Transforms
- WebView
- Regex and input validation
- Swift Extensions
- Scroll View & Autolayout
- App Ideas

# Transforms

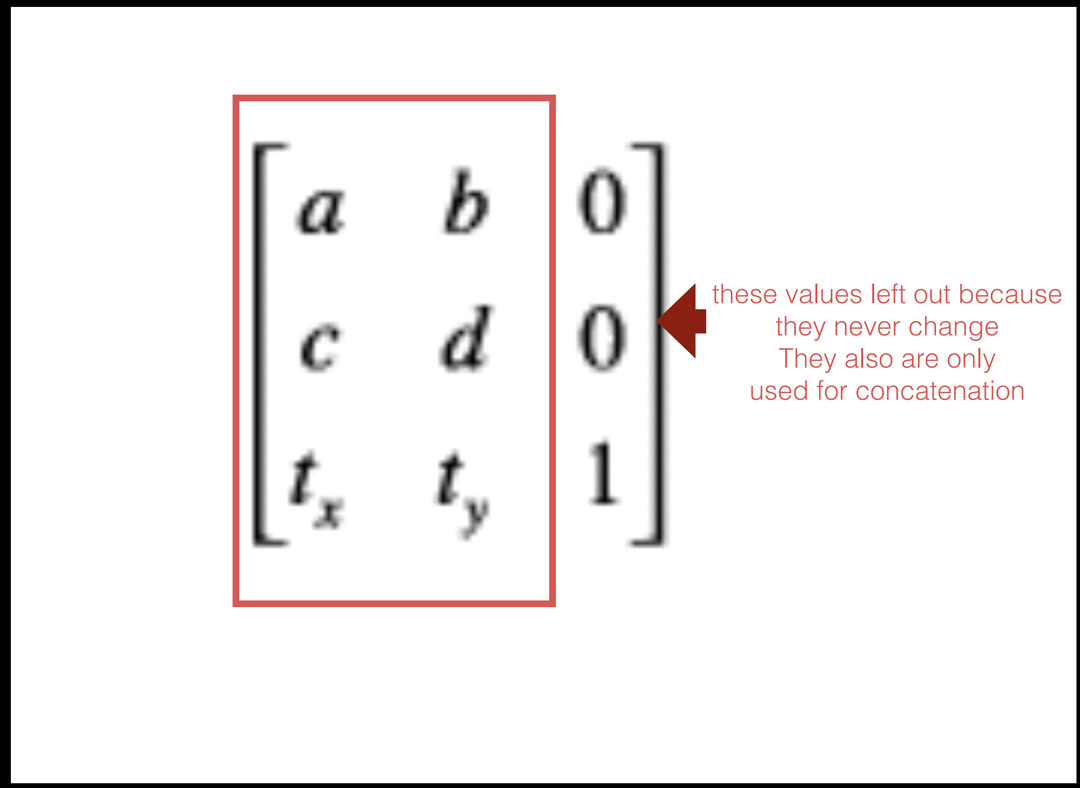
# Transform

- Transforms can be applied to views and layers to translate (move), scale, rotate, and make a number of other changes to themselves.
- Every view has a .transform property which has a type of CGAffineTransform. Its a struct.
- Transforms are represented my matrices.
- You can think of them as  two dimensional array of numbers.

# CGAffineTransform structure:

```
struct CGAffineTransform {
    CGFloat a;
    CGFloat b;
    CGFloat c;
    CGFloat d;
    CGFloat tx;
    CGFloat ty;
};
```

The data structure
we use to represent it




An actual affine transformation matrix

# Identity Matrix

- Every view (and layer) starts out with their transform set to the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- If a view's transform is set to the identity matrix, then we know no transforms have been applied to it.
- For a view that has a transform that is set to the identity matrix, the view will be drawn based only on its frame & bounds.
- Setting a view's transform back to the identity matrix will undo any transforms applied to it!

# Translation

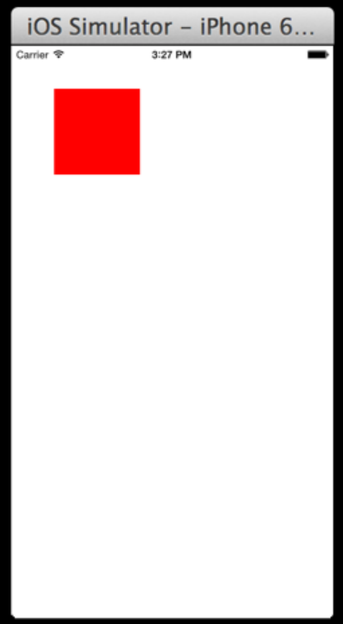
- Translation is just a fancy word for moving.
- If you a translate a view by 10,10 you just moved it 10 to the right and 10 down.
- This is the matrix that represents a translation operation:

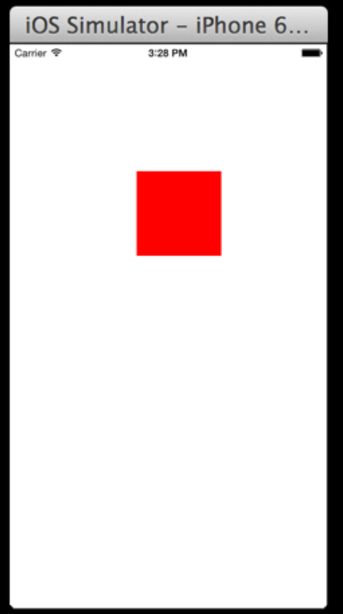
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

- So the equations that CoreGraphics uses to calculate the result are:
  - new X = old x + $t_x$
  - new Y = old Y + $t_y$

# Translation Example

- CoreGraphics provides relatively simple functions for all the primary transforms.
- Here is an example of using the translation function:



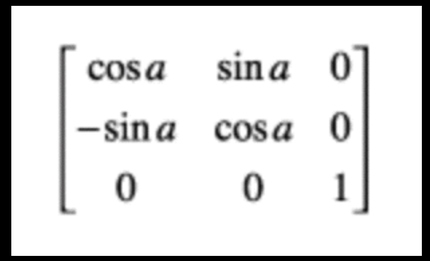

```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
```

```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
redView.transform = CGAffineTransformTranslate(redView.transform, 100, 100)
```

*Applying transforms is cumulative, so doing another translate just adds to the last one

# Rotation

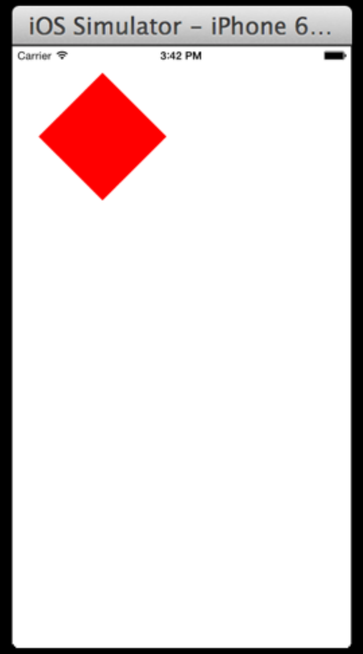
- Rotation is another of the primary transforms you can apply to views and layers.
- CoreGraphics provides the CGAffineTransformRotate() function that takes in two parameters:
  - the existing transform matrix
  - the angle or rotation in radians (Pi * degrees / 180.0)
- This is the matrix that describes rotation operations:




- This means the equations CoreGraphis uses to calculate the results are:
  - new x = old X * cos a - old Y sin a
  - new y = old X *sin a - old Y *cos a
- I hope you were paying attention in high school

# Rotation Example

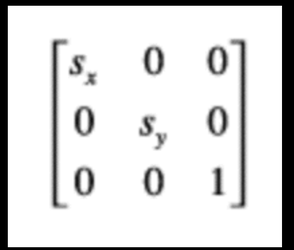
- Here is an example of rotating our red view 45 degrees clockwise




```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
redView.transform = CGAffineTransformRotate(redView.transform, CGFloat(M_PI * 45 / 180.0))
```

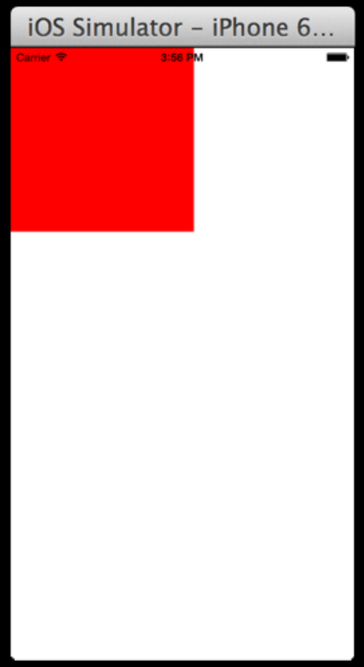
# Scaling

- Scaling is another primary transform that is simply changes the size of the view.
- CoreGraphics provides the CGAffineTransformScale function that takes in 3 parameters:
  - the transform matrix we are applying the scale to
  - the factor by which to scale the x-axis
  - the factor by which to scale the y-axis
- Here is the matrix that describes the scaling operation:



# Scaling Example

- Here is an example of double our red view's size:




```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
redView.transform = CGAffineTransformScale(redView.transform, 2.0, 2.0)
```

# Demo

# When to use transform or just regular frame/bounds

- Manipulate the transform when you want to make **temporary** visual changes to a view, or need to rotate the view.
- Manipulate the frame/bounds when you need to make permanent changes to a view's location or size

# WKWebView

- “Use the WKWebView class to embed web content in your application”
- The simple workflow of a web view:
    1. add web view to the view hierarchy
    2. send it a request to load web content
- Can have a delegate that tracks loading of content, this will come in handy when we do the ‘in-app’ way of doing OAuth
- It’s sort of like a mini browser in your app, and you can customize it to not allow the users to go back or forward.
- Prior to iOS8, we had to use UIWebView, and it was horrible. It leaked memory every time you used it.

# Demo

# Regex and Input Validation

# Input Validation

- “An app sometimes cannot accept the strings entered in textfields and text views without validating the value first”
- In our Github app, we cant have spaces in user’s search queries. We also don’t want them entering in symbols like = or $.
- So how can we ensure our users input doesn't break our app?

# Delegate methods on input views

- The go to methods for text validation are:
  - For UITextField  : textFieldShouldEndEditing:
  - For UITextView : textViewShouldEndEditing:
  - For UISearchBar : searchBar:shouldChangeTextInRange:
- These methods are called just before the textview/field is about to resign first responder or for the search bar, when a character is about to be entered.
- Returning false prevents that from happening. So return false when the text isn't valid. Also perhaps pop up an alert view or show some indicator explaining why its invalid.
- So how do we check if the text is valid?

# Regex

- Regular Expressions are a pattern-matching standard for text validation.
- The regular expression is a pattern that you compare with the text you are validating.
- Regex’s can be used for finding patterns, replacing text, and extracting substrings in strings.

# The simplest Regex

- The most simple regex is just a string. Like “seahawks”.
- The regex “seahawks” will find a match on the string “go seahawks”
- The regex “sea hawks” will not find a match on the string “go seahawks”
- In these two examples, we are only using literal characters. So its basically just running a find operation looking for our regex string.
- There are special reserved characters we can use in regex to make it much more powerful

# + and *

- + is used to match the previous character 1 or more times
- so the regex “sea+” will match sea and seaaaaaaaaaa and seaaaa
- * works the same way, except it matches 0 or more times
- so “sea*” matches seaaaaaa, sea, and also se

# (Capturing parens)

- Parentheses are used to create a capturing group.
- Capturing groups capture the text matched by the regex inside the parens and put them into a group that can be referenced together.

# ? optional

- The question mark makes the preceding token in the regular expression optional.
- So “seahawks?” matches both “seahawks” and “seahawk"
- You can use grouping to make groups optional
- So “sea(hawks)?” matches seahawks and sea

# [Character Classes]

- With a character class, sometimes called character set, you can have the regex only match one of several characters.
- If you want to match a t or d you will use [td]
- so “foo[td]” will match “foot” and “food”
- You can use a hypen inside the character classes to specify ranges
- “[0-9]” matches any single number
- You can combine ranges “[0-9a-zA-Z]” will match any regular literal character

# [Negating Character Classes]

- Adding a caret after the opening square bracket negates the character class.
- This makes it so the character class matches any character that is NOT in the character class.
- So [^0-9a-zA-Z] will match any non regular literal character.
- We also need to add \n in that range because that is what is entered into a textfield/view/searchBar when you hit return/done/search. \n is ascii/unicode for end of line.
- Hey I think we can use [^0-9a-zA-Z] in our app!

# Regex and iOS

- You can use the NSRegularExpression class to natively use regex in your app.
- You instantiate an instance of NSRegularExpression with your regex pattern, options, and an error pointer:

```
let regex = NSRegularExpression(pattern: "[^0-9a-zA-Z]", options:
    nil, error: nil)
```

# Search for matches

- NSRegularExpression has methods for returning the total count of matches, enumerating through each match, returning an array of matches, returning the first match, and the range of the first match.
- In our app we can just use the number of matches method, and if its greater than 0 we know the user typed in something invalid:

```
let regex = NSRegularExpression(pattern: "[^0-9a-zA-Z]", options:
    nil, error: nil)
let match = regex.numberOfMatchesInString(self, options: nil, range:
    NSRange(location:0, length: countElements(self)))
if match > 0 {
    return false
}
return true
```

# Demo

# Swift Extensions

# Extensions

- Extensions add new functionality to pre-existing classes, structs, or enums.
- You can even do this on types you do not have access to the source code.
- Similar to categories in Objective-C, except extensions dont have names.

# Things you can add with extensions:

- computed properties
- instance methods
- type methods
- new inits
- subscripts
- nested types
- make an existing type conform to a protocol

# Extension Syntax

```
extension SomeType {
    // new functionality to add to SomeType goes
        here
}
```

# Demo

# UIScrollView

Sourced From www.objc.io, issue 3, Understanding Scroll Views

# UIScrollView

- A UIScrollView is technically a subclass of UIView, but its really just a UIView with a few simple added features.
- So to understand how scroll views work, you must first have a great understanding of how UIViews and the view hierarchy works in iOS.

# Rasterization and Composition

- Rasterization and Composition combine to make up the rendering process of your iOS app.
- Rasterization is simply just a set of drawing instructions that produces an image.
- So a UIButton draws an image with a rounded rectangle and title in the center as its rosterization process.
- But these images are not drawn onto screen during the rasterization process, they are held onto by their views, in preparation for…

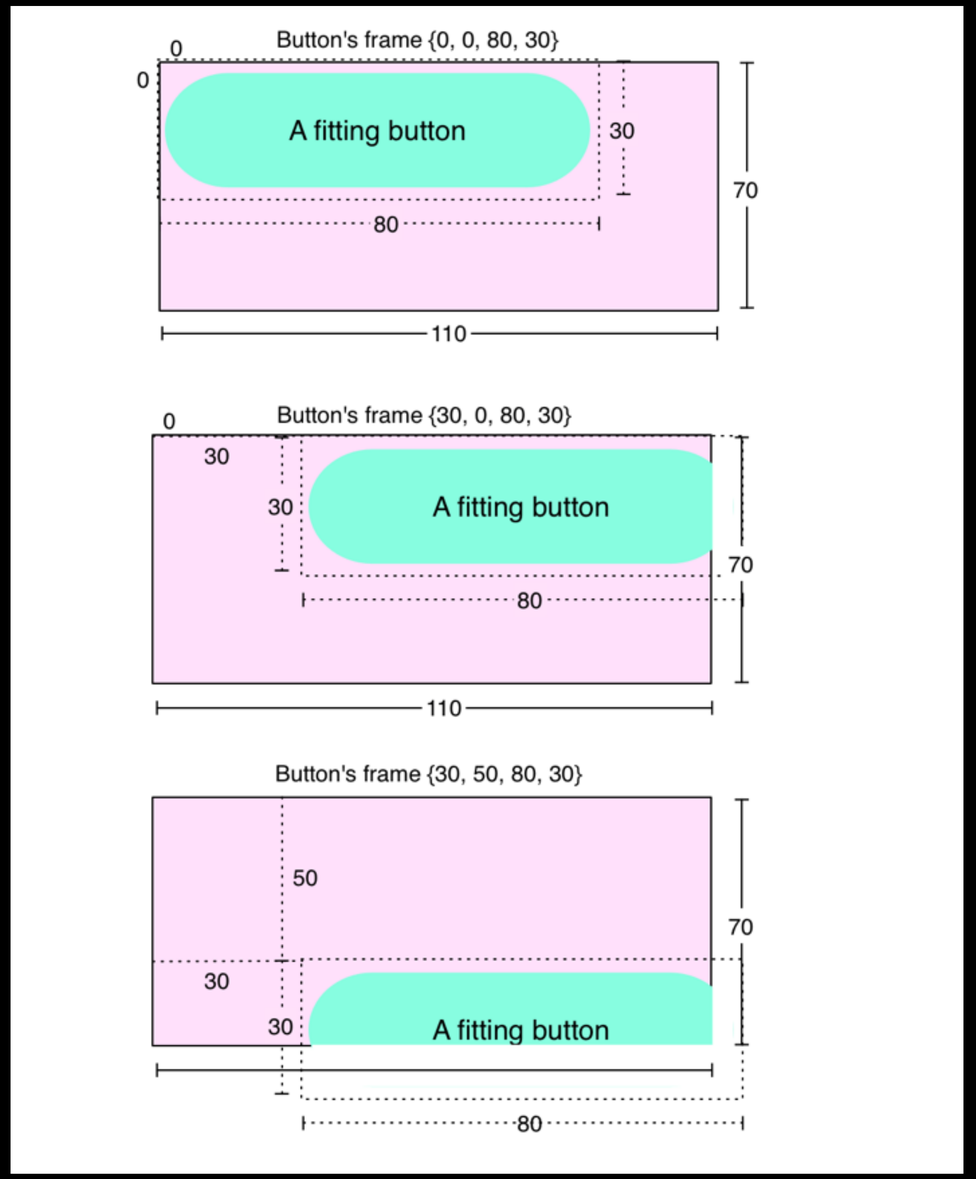
# Composition

- The composition step involves all of the rosterized images being drawn on top of each other to produce one screen-sized image.
- A view's image is composited on top of its superview's image.
- Then the composited image is composited onto of the super-superview's image, and so on.
- The view at the very top of the hierarchy is the window, and its composited image is what the user sees.
- So basically, you are layering independent images on top of each other to produce a final, flat image.

# Bounds vs Frame

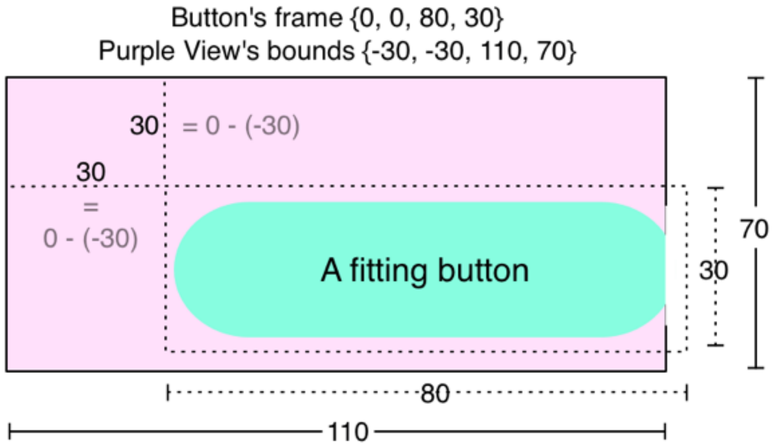
- So this is where bounds and frames come in to play.
- When the images are being laid out, the frame is used to figure out exactly where the image should be placed relative to its superview. So the frame is used in the composition step.
- During the rasterization step, a view doesn't care about its frame. The only thing it cares about is drawing its own content. Anything inside of its current bounds rect gets drawn onto the image. Anything that is outside of the bounds is discarded.
- Lets look at some pics!

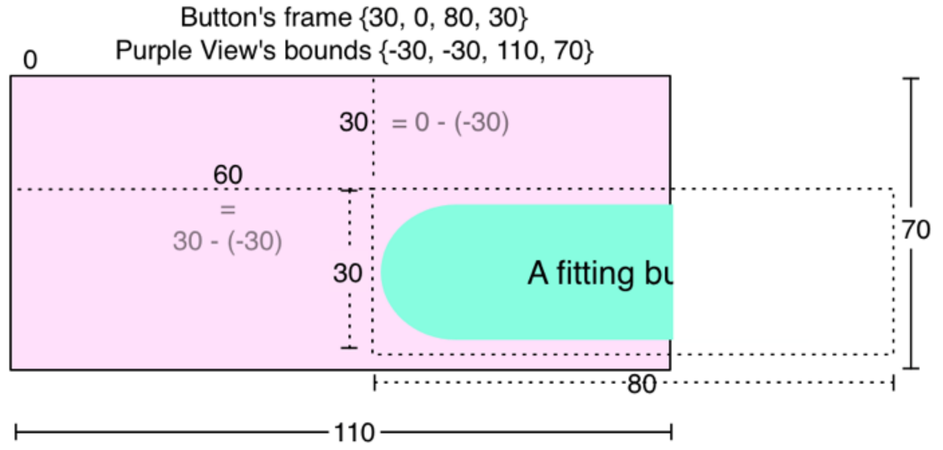
# Frame

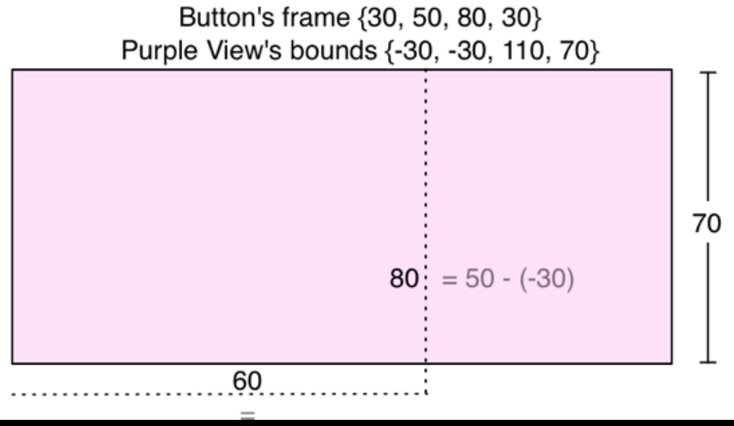

Button's frame {0, 0, 80, 30}
0
0
A fitting button
30
70
80
110
0
Button's frame {30, 0, 80, 30}
30
30
A fitting button
70
80
110
Button's frame {30, 50, 80, 30}
50
70
30
30
A fitting button
80

# ScrollView

- So what does this have to do with scroll views? Everything.
- When you scroll a scroll view, the offset value changes. Under the hood, this is just changing the origin of the bounds of the scroll view.
- You could accomplish the same effect by changing the frames of all the scroll views subviews as it scrolls, but you would have to loop through every single subview every time the scroll view scrolls, which would suck.
- More Visuals!


Button's frame {0, 0, 80, 30}
Purple View's bounds {-30, -30, 110, 70}
30
= 0 - (-30)
30
=
0 - (-30)
A fitting button
70
30
80
110


Button's frame {30, 0, 80, 30}
Purple View's bounds {-30, -30, 110, 70}
0
30
= 0 - (-30)
60
=
30 - (-30)
30
A fitting bu
70
80
110


Button's frame {30, 50, 80, 30}
Purple View's bounds {-30, -30, 110, 70}
70
80
= 50 - (-30)
60
=

# Scroll View’s contentSize

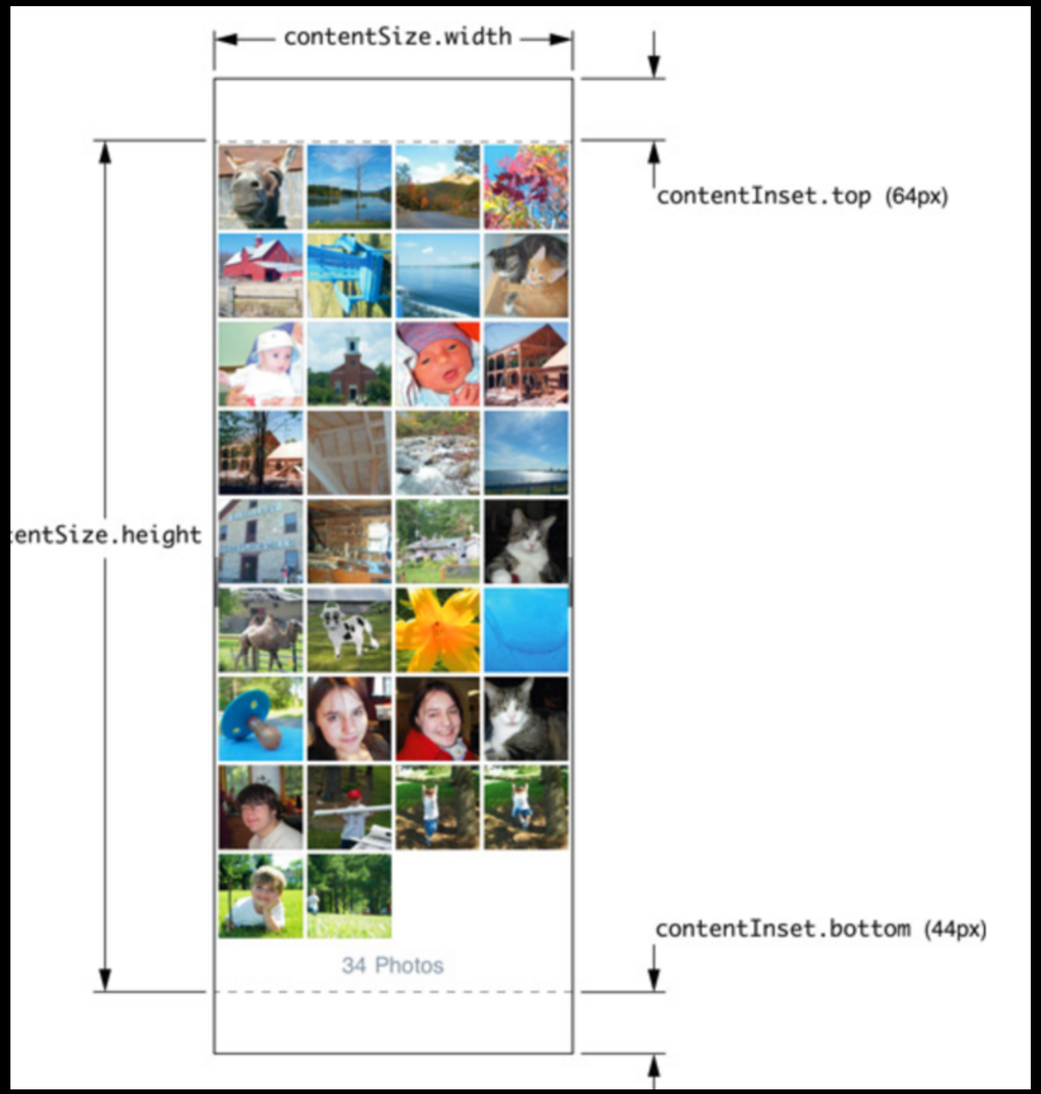
- A scroll view’s contentSize is the just the entire scrollable area.
- It doesn't effect the frame or bounds of a scroll view.
- By default it is set to 0,0
- When the content size is larger than the scroll view’s bounds, the user is allowed to scroll.
- Think of the bounds of the scroll view as a window into the scrollable area defined by the content size.

# Scroll View's contentOffset

- "The point at which the origin of the content view is offset from the origin of the scroll view"
- The contentOffset is just how much the scroll view has scrolled.
- In addition to panning, you can set it programmatically with setContentOffset:animated:

# Scroll View's contentInset

- content inset is basically just padding around your content area



# Scroll View's delegate

- scroll view has an awesome delegate that is constantly notified of the state of the scroll view:
  - scrollViewDidScroll:
  - scrollViewWillBeginDragging:
  - scrollViewWillEndDragging:withVelocity:targetContentOffset:
  - scrollViewDidZoom:
  - etc

# Scroll View's and storyboard

- Scroll views and storyboards are worst enemies
- This is partly because scroll views and autolayout have a special rule:
  - If you use autolayout with a scroll view, you don't sent the content size manually, you let the items inside the scroll view, with their constraints, dictate the content size.
- So autolayout and scroll views can work together, but getting it working via storyboard is a nightmare
- It took a team of senior apple engineers 40 minutes to get a scroll view and storyboard working
- Luckily some smart people on the interwebs have mostly cracked the code

# Scroll View’s and storyboard Steps

1. Drag out your scroll view onto your view controller’s view
2. Drag out another view and place it onto the scroll, this will effectively become the content view of your scroll view
3. Pin the scroll view to its super view on all 4 sides
4. Pin the ‘content view’ to the scroll view on all 4 sides
5. Pin the ‘content view”s width and height to be equal to the view controller’s view (not the scroll view! its parent)
6. Lower the priority of the constraint that is pinning the view controller’s view and the content view’s height to 250
7. Now layout your interface however you want in the scroll view (add text fields, image views, etc)

# Demo

# App Ideas

- discrete math calc
- marriage calendar
- baby names
- restaurant app, connecting everybody
- Flickr app
- Metacritic
- music, artists concert finder
- karaoke practice thing
- workout help
- location based events
- background check
- credit check
- game with sprite kit
- party game
- trivia game
- sports app
- SportsBall