

Developing Applications for iOS



Lecture 4: Views, Autorotation and Gestures

Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

Content

- Views
- Drawing Paths
- Drawing Text
- Drawing Images
- Autorotation
- Protocols
- Gesture Recognizers

Views

- A view (i.e. `UIView` subclass) represents a rectangular area.
- It defines a coordinate space.
- Draws and handles events in that rectangle.

Hierarchical

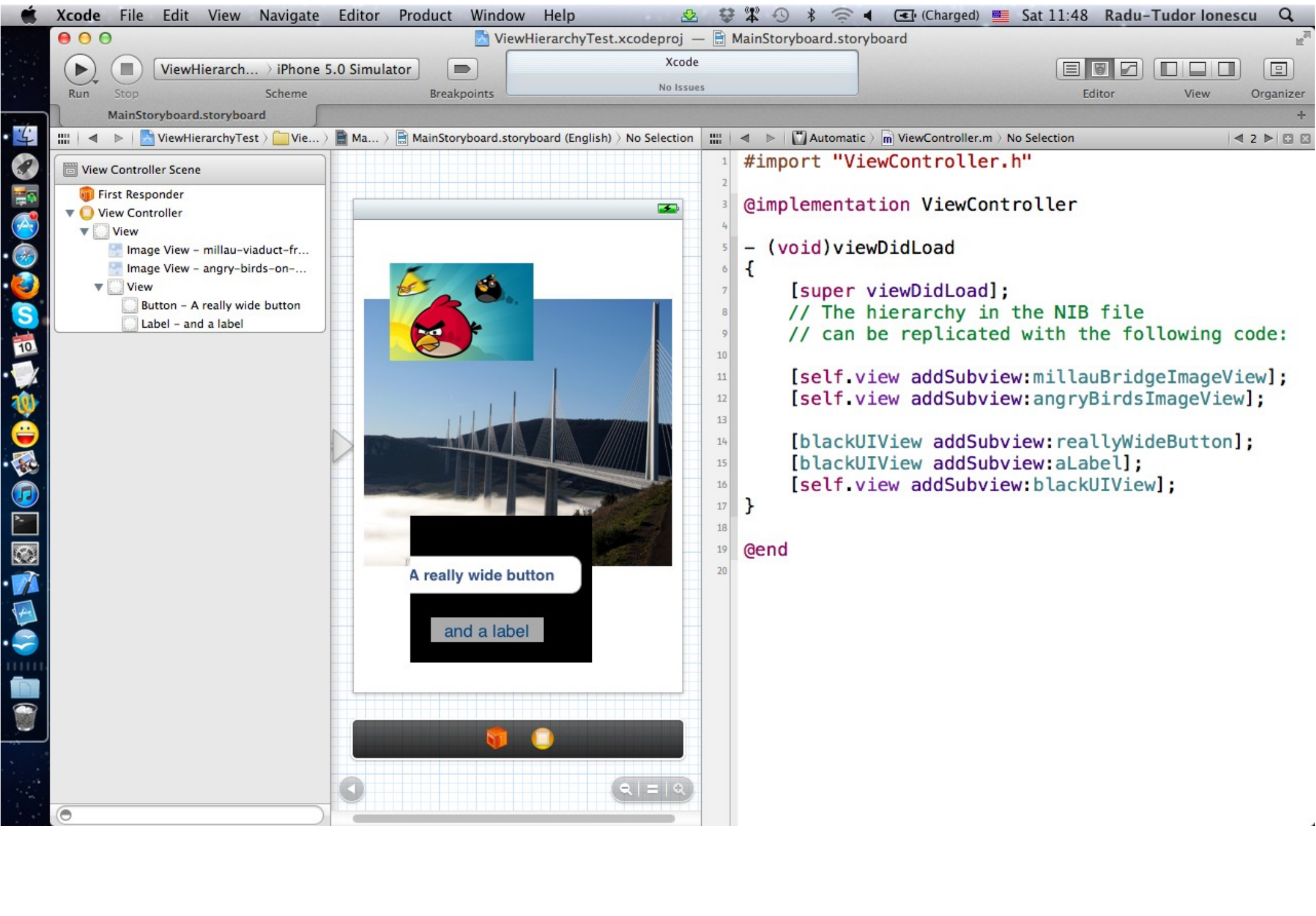
- A view has only one superview – `(UIView *)superview`.
- But can have many (or zero) subviews – `(NSArray *)subviews`.
- Subview order (in `subviews` array) matters: those later in the array are on top of those earlier.

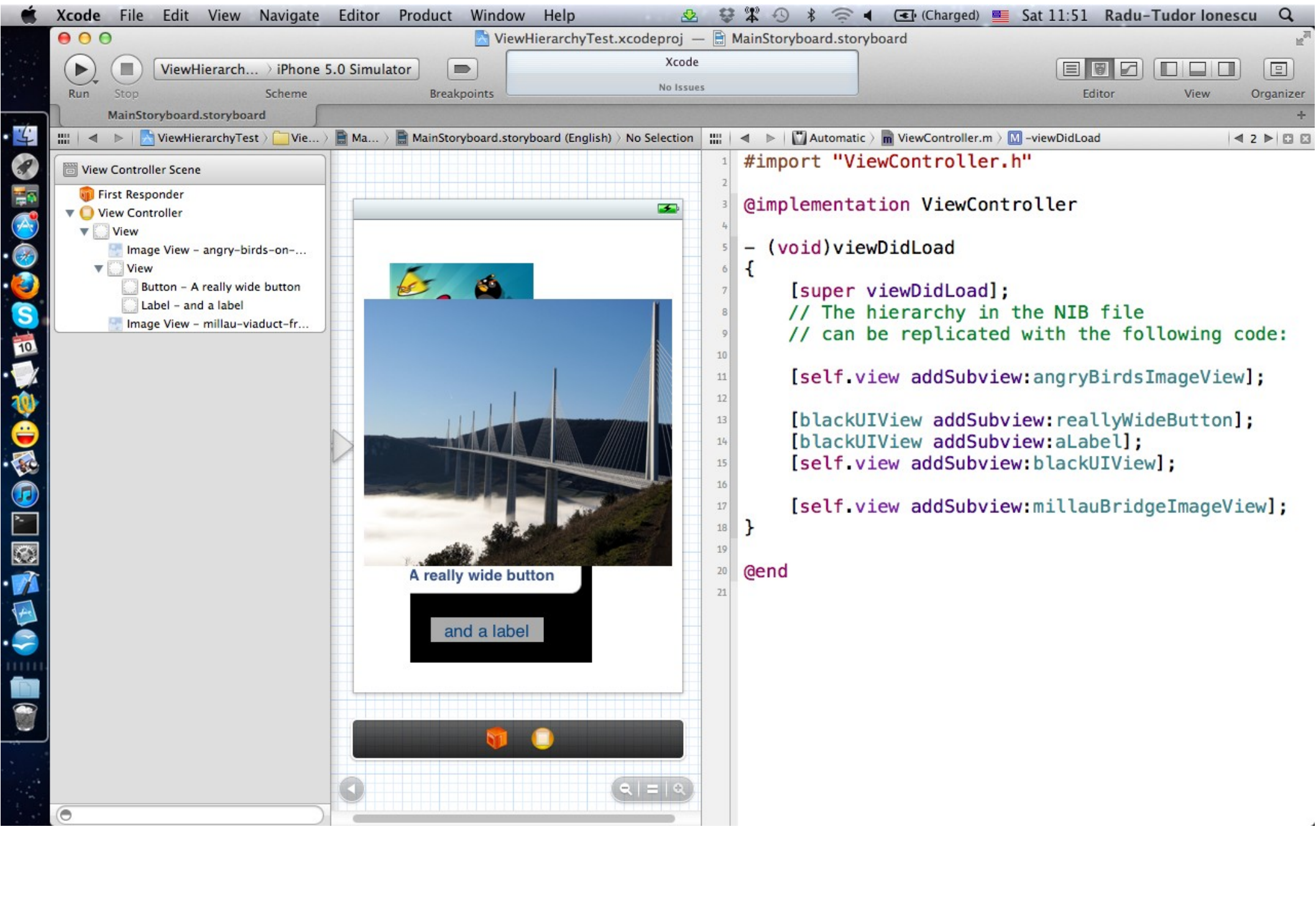
UIWindow

- The `UIView` at the top of the view hierarchy.
- Only have one `UIWindow` (generally) in an iOS application.
- It's all about views, not windows.

Views

- The hierarchy is most often constructed in Xcode graphically.
Even custom views are added to the view hierarchy using Xcode (more on this later).
- But it can be done in code as well:
 - `(void)addSubview:(UIView *)aView;`
 - `(void)removeFromSuperview;`





View Coordinates

CGFloat

- Just a floating point number, but we always use it for graphics.

CGPoint

- C struct with two CGFloats in it: x and y.

```
CGPoint p = CGPointMake(34.5, 22.0);  
p.x += 20; // move right by 20 points
```

CGSize

- C struct with two CGFloats in it: width and height.

```
CGSize s = CGSizeMake(100.0, 200.0);  
s.height += 50; // make the size 50 points taller
```

CGRect

- C struct with a CGPoint origin and a CGSize size.

```
CGRect aRect = CGRectMake(45.0, 75.5, 300, 500);  
aRect.size.height += 45; //make the rect 45 points  
taller  
aRect.origin.x += 30; //move the rect to right 30  
points
```

Coordinates

(0,0)

increasing x

(400,38)

- Origin of a view's coordinate system is upper left.
- Units are "points" (not pixels).
- Usually you don't care about how many pixels per point are on the screen you're drawing on.
- Fonts and arcs and such automatically adjust to use higher resolution.
- However, if you are drawing something detailed (like a graph), you might want to know. There is a `UIView` property which will tell you:

```
@property CGFloat contentScaleFactor;  
/* Returns pixels per point  
   on the screen this view is on. */
```

- This property is not (readonly), but you should basically pretend that it is for this course.

increasing y

Coordinates

(0,0)

increasing x

(400,38)

- Views have 3 properties related to their location and size.
`@property CGRect bounds;`
- This is your view's internal drawing space's origin and size.
- The bounds `@property` is what you use inside your view's own implementation.
- It is up to your implementation as to how to interpret the meaning of `bounds.origin`.
`@property CGPoint center;`
- The center of your view in your superview's coordinate space.
`@property CGRect frame;`
- A rectangle in your superview's coordinate space which entirely contains your view's `bounds.size`.

increasing y

Coordinates

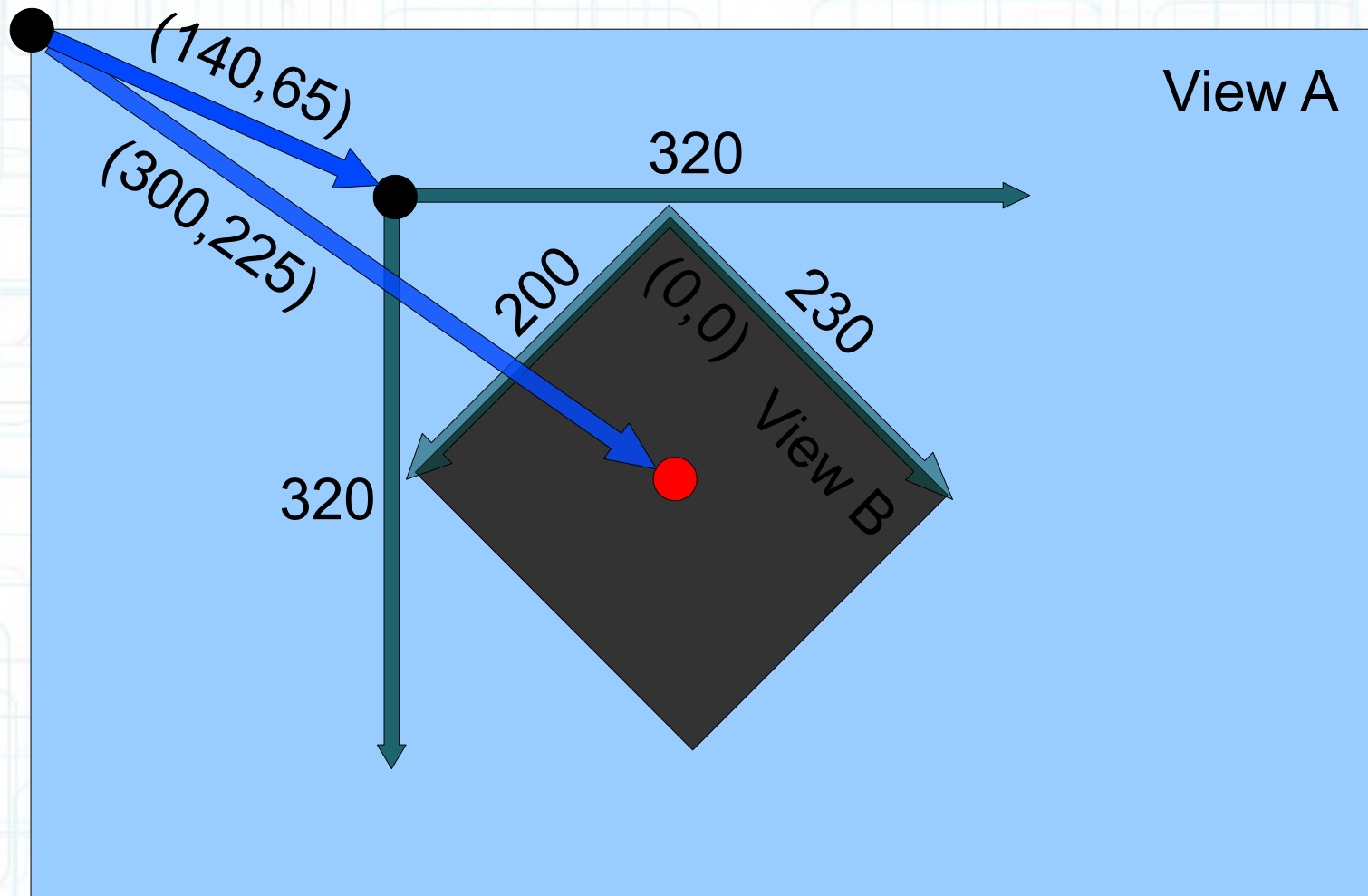
Use `frame` and `center` to position the view in the hierarchy

- These are used by `superviews`, never inside your `UIView` subclass's implementation.
- You might think `frame.size` is always equal to `bounds.size`, but you would be wrong because views can be rotated (and scaled and translated too).
- Views are rarely rotated, but don't misuse `frame` or `center` by assuming that.

Coordinates

Use frame and center to position the view in the hierarchy

- Let's take a look at the following example:



Coordinates

Use `frame` and `center` to position the view in the hierarchy

- Let's take a look at the following example:

View B's bounds = ((0,0),(230,200))

View B's frame = ((140,65),(320,320))

View B's center = (300,225)

View B's middle in its own coordinate space is

```
(bounds.size.width / 2 + bounds.origin.x,  
bounds.size.height / 2 + bounds.origin.y)
```

which is (115,100) in this case.

Creating Views

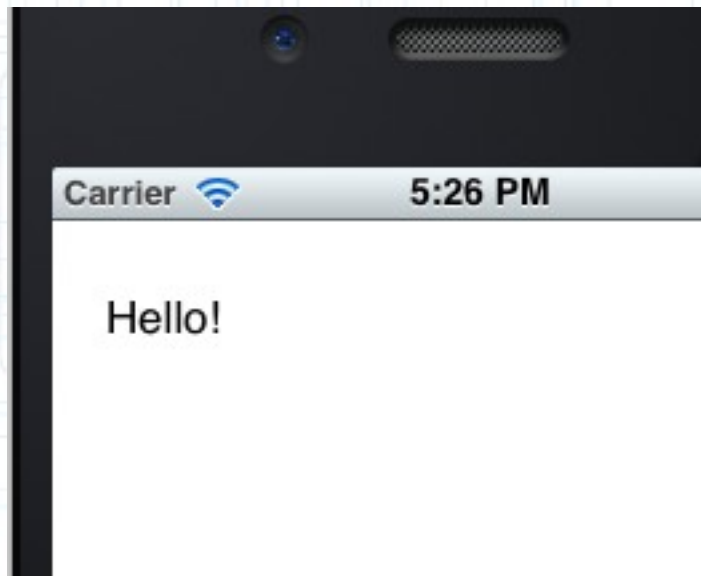
- Most often you create views in Interface Builder.
- Of course, Interface Builder's palette knows nothing about a custom view class you might create.
- In that case, you drag out a generic `UIView` from the palette and use the Inspector to change the class of the `UIView` to your custom class.
- How do you create a `UIView` in code (i.e. not in Interface Builder)?

Just use `alloc` and `initWithFrame:` (`UIView`'s designated initializer).

Creating Views

- Example:

```
CGRect rect = CGRectMake(20, 20, 50, 30);
UILabel *label = [[UILabel alloc] initWithFrame:rect];
label.text = @"Hello!";
[self.view addSubview:label];
// (self.view is a Controller's top-level view)
```



Custom Views

When would I want to create my own `UIView` subclass?

- I want to do some custom drawing on screen.
- I need to handle touch events in a special way (i.e. different than a button or slider does).
- We will talk about handling touch events later. For now we are focussing on drawing.

Drawing is easy

- Create a `UIView` subclass and override one method:
 - `(void)drawRect:(CGRect)aRect;`
- You can optimize by not drawing outside of `aRect` if you want (but not required).

Custom Views

NEVER call `drawRect:!`

- Instead, let iOS know that your view's visual is out of date with one of these `UIView` methods:
 - `(void)setNeedsDisplay;`
 - `(void)setNeedsDisplayInRect:(CGRect)aRect;`
- It will then set everything up and call `drawRect:` for you at an appropriate time.
- Obviously, the second version will call your `drawRect:` with only rectangles that need updates.

Custom Views

- So how do I implement my `drawRect:`?
- We use the Core Graphics framework.
- The API is C (not object-oriented).

Concepts

- Get a context to draw into (iOS will prepare one each time your `drawRect:` is called).
- Create paths (out of lines, arcs, etc).
- Set colors, fonts, textures, line widths, line caps, etc.
- Stroke or fill the above-created paths.

Context

The context determines where your drawing goes

- Screen (the only one we are going to talk about today)
- Offscreen Bitmap
- PDF
- Printer

For normal drawing, `UIKit` sets up the current context for you

- But it is only valid during that particular call to `drawRect:`.
- A new one is set up for you each time `drawRect:` is called. So never cache the current graphics context in `drawRect:` to use later!

How to get this magic context?

- Call the following C function inside your `drawRect:` method to get the current graphics context:

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

Define a Path

- Begin the path:

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path:

```
CGContextMoveToPoint(context, 75, 10);  
CGContextAddLineToPoint(context, 160, 150);
```



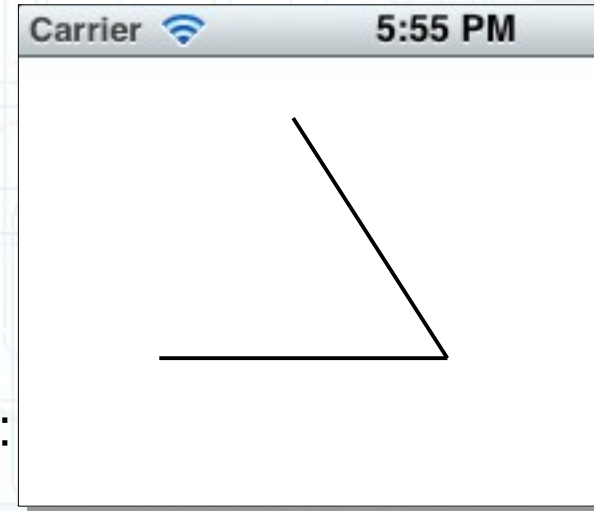
Define a Path

- Begin the path:

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path:

```
CGContextMoveToPoint(context, 75, 10);  
CGContextAddLineToPoint(context, 160, 150);  
CGContextAddLineToPoint(context, 10, 150);
```



Define a Path

- Begin the path:

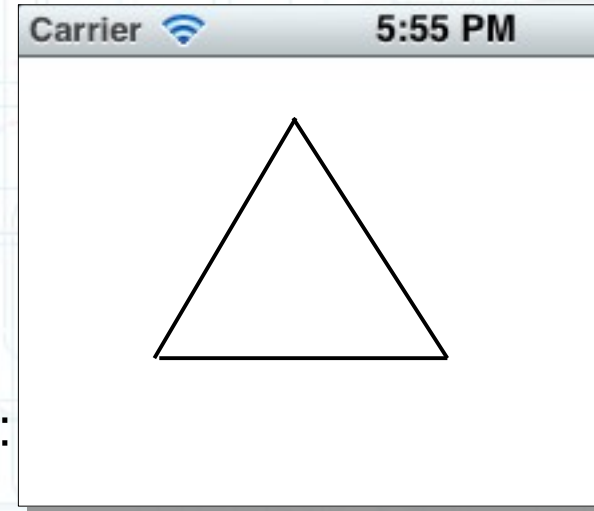
```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path:

```
CGContextMoveToPoint(context, 75, 10);  
CGContextAddLineToPoint(context, 160, 150);  
CGContextAddLineToPoint(context, 10, 150);
```

- Close the path (connects the last point back to the first):

```
CGContextClosePath(context); // not strictly required
```



Define a Path

- Begin the path:

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path:

```
CGContextMoveToPoint(context, 75, 10);  
CGContextAddLineToPoint(context, 160, 150);  
CGContextAddLineToPoint(context, 10, 150);
```

- Close the path (connects the last point back to the first):

```
CGContextClosePath(context); // not strictly required
```

- Actually the above draws nothing (yet)!
- You have to set the graphics state and then fill/stroke the above path to see anything.



Define a Path

- Begin the path:

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path:

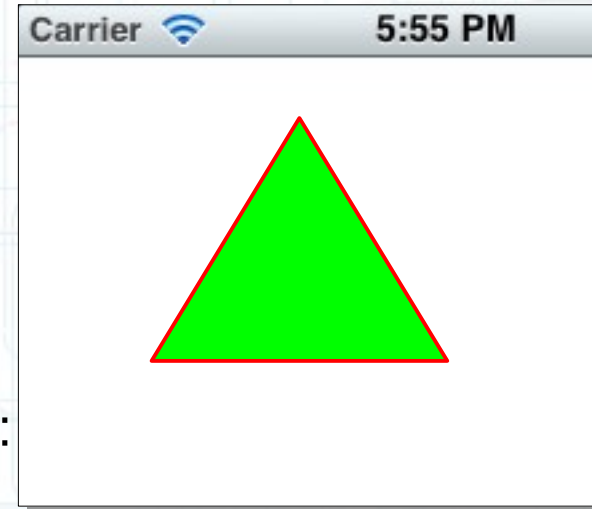
```
CGContextMoveToPoint(context, 75, 10);  
CGContextAddLineToPoint(context, 160, 150);  
CGContextAddLineToPoint(context, 10, 150);
```

- Close the path (connects the last point back to the first):

```
CGContextClosePath(context); // not strictly required
```

- Actually the above draws nothing (yet)!
- You have to set the graphics state and then fill/stroke the above path to see anything.

```
[[UIColor greenColor] setFill];  
// object-oriented convenience method (more in a moment)  
  
[[UIColor redColor] setStroke];  
  
CGContextDrawPath(context, kCGPathFillStroke);  
// kCGPathFillStroke is a constant
```



Define a Path

- It is also possible to save a path and reuse it.
- There are similar functions to the previous slide that let you do this, but starting with `CGPath` instead of `CGContext`.
- We won't be covering those, but you can certainly feel free to look them up in the documentation.

Graphics State

UIColor class for setting colors

- It has class methods for creating most common colors:

```
UIColor *red = [UIColor redColor];  
UIColor *invisible = [UIColor clearColor];
```

- Custom colors can be created using initializers:

```
UIColor *custom = [[UIColor alloc]  
                  initWithRed:(CGFloat)red //0.0 to 1.0  
                  green:(CGFloat)green  
                  blue:(CGFloat)blue  
                  alpha:(CGFloat)alpha];
```

```
UIImage *woodTexture = ...;  
UIColor *pattern = [[UIColor alloc]  
                  initWithPatternImage:woodTexture];
```

- To set the fill/stroke color in current graphics context:

```
[red setFill]; // stroke color not set  
[[UIColor grayColor] setStroke];  
[custom set]; /* sets both stroke and fill color to  
               custom (would override the previous) */
```

View Transparency

Drawing with transparency in `UIView`

- Note the alpha component of `UIColor`s. This is how you can draw with transparency in your `drawRect:`.
- `UIView` also has a `backgroundColor` property which can be set to transparent values.
- Be sure to set `@property BOOL opaque` to `NO` in a view which is partially or fully transparent.
- If you don't, results are unpredictable (this is a performance optimization property, by the way).
- The `UIView @property CGFloat alpha` can make the entire view partially transparent.

View Transparency

What happens when views overlap?

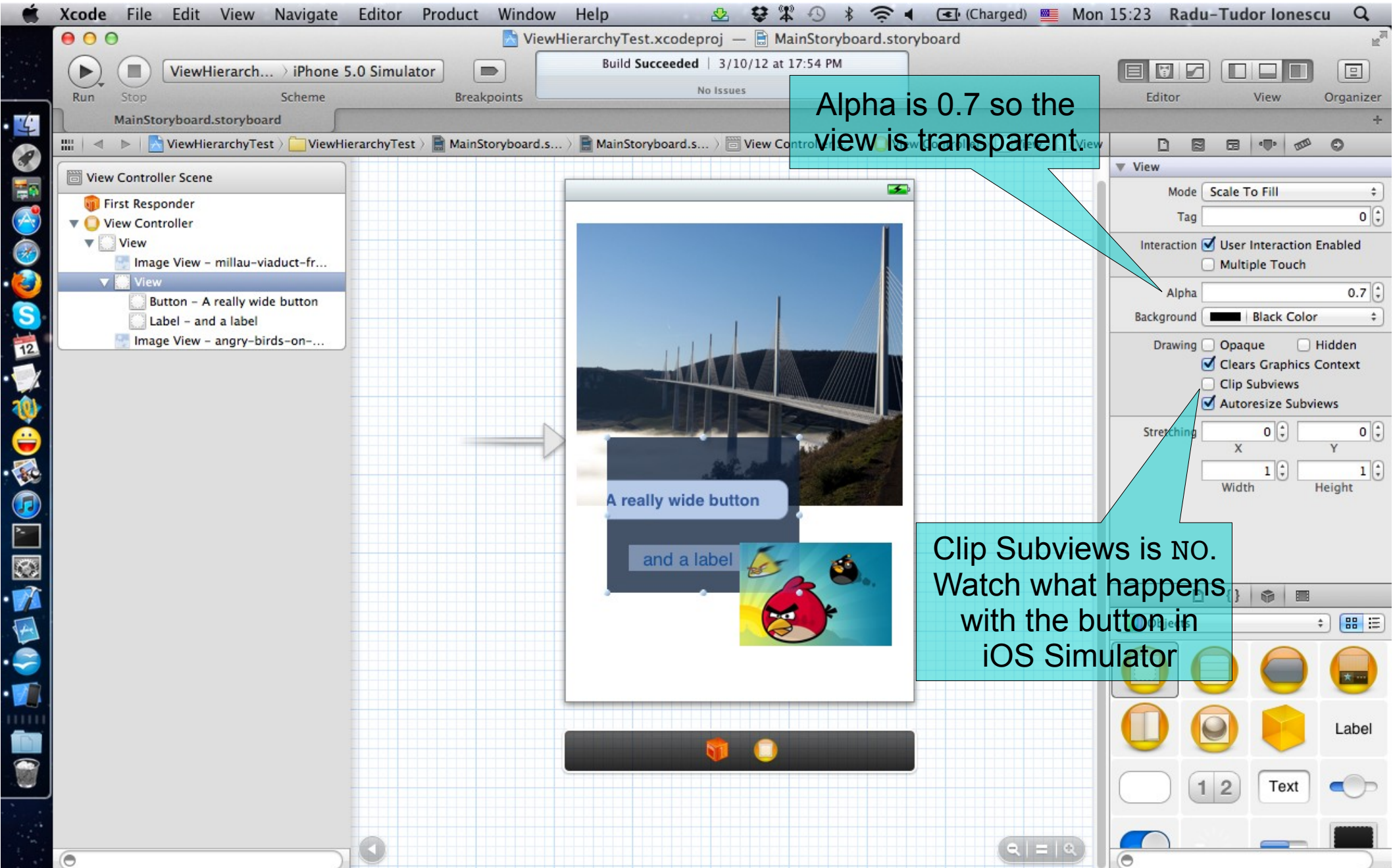
- As mentioned before, `subviews` list order determines who's in front.
- Lower ones (earlier in `subviews` array) can “show through” transparent views on top of them.
- Default drawing is opaque. Transparency is not cheap when you think of performance.
- Also, you can hide a view completely by setting `hidden` property:

```
@property (nonatomic) BOOL hidden;
```

- The view will not be on screen and will not handle events if you set:

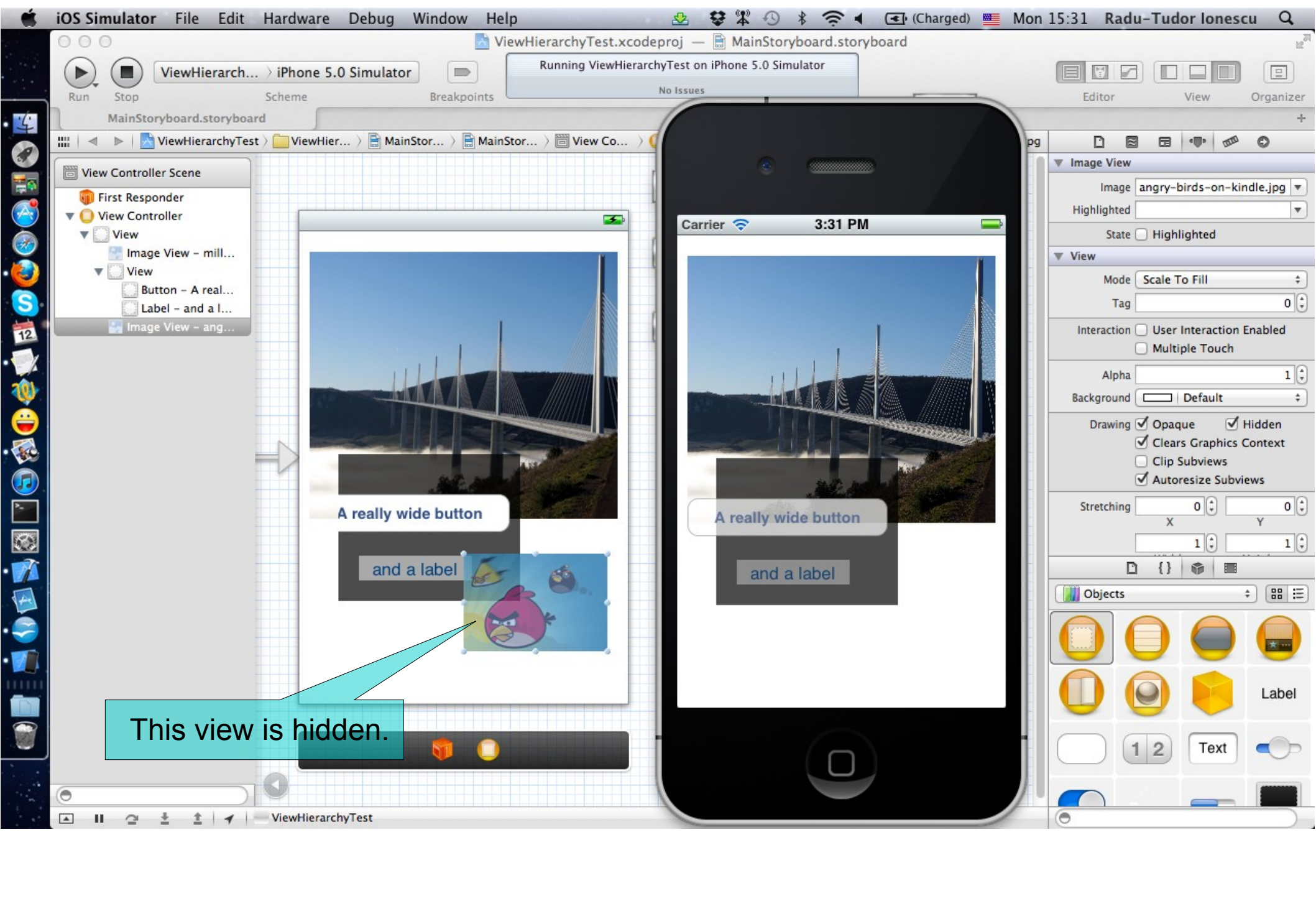
```
myView.hidden = YES;
```

- This is not as uncommon as you might think. On a small screen, keeping it de-cluttered by hiding currently unusable views make sense.

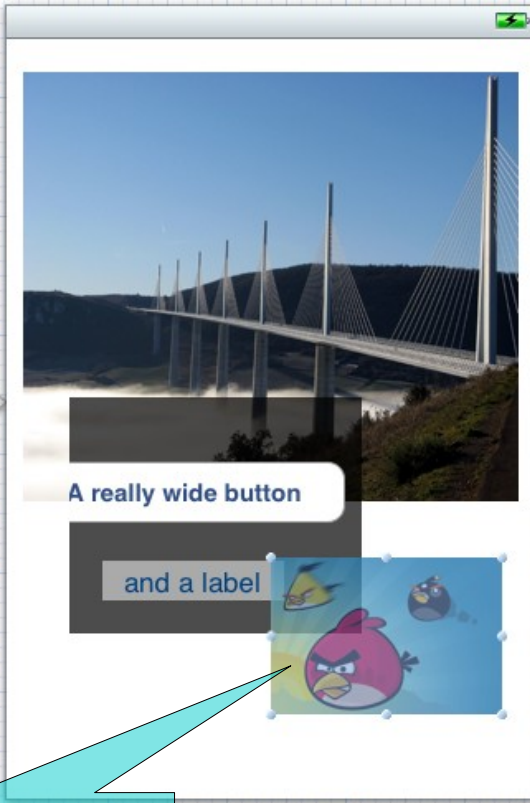


Alpha is 0.7 so the view is transparent

Clip Subviews is NO. Watch what happens with the button in iOS Simulator



- View Controller Scene
 - First Responder
 - View Controller
 - View
 - Image View - mill...
 - View
 - Button - A real...
 - Label - and a l...



This view is hidden.

Inspector panel showing properties for an Image View:

- Image: angry-birds-on-kindle.jpg
- Highlighted: [dropdown]
- State: Highlighted
- View
 - Mode: Scale To Fill
 - Tag: 0
 - Interaction: User Interaction Enabled, Multiple Touch
 - Alpha: 1
 - Background: Default
 - Drawing: Opaque, Hidden, Clears Graphics Context, Clip Subviews, Autoresize Subviews
 - Stretching: X: 0, Y: 0; X: 1, Y: 1

Objects palette showing various UI elements:

- Buttons, Labels, Text, and other UI components.

Graphics State

Some other graphics state set with C functions

- Set line width (in points, not pixels):

```
CGContextSetLineWidth(context, 1.0);
```

- Set the fill pattern in specified graphics context:

```
CGContextSetFillPattern(context,  
                        (CGPatternRef)pattern,  
                        (CGFloat[])components);
```

Graphics State

Special considerations for defining drawing “subroutines”

- What if you wanted to have a utility method that draws something?
- You don't want that utility method to mess up the graphics state of the calling method.
- Use push and pop context functions:

```
- (void)drawGreenCircle:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    [[UIColor greenColor] setFill];
    // draw my circle
    UIGraphicsPopContext();
}

- (void)drawRect:(CGRect)aRect
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    [[UIColor redColor] setFill];
    // do some stuff
    [self drawGreenCircle:context];
    // do more stuff and expect fill color to be red
}
```

Drawing Text

- Use UILabel to draw text, but if you feel you must ...
- Use UIFont object in UIKit to get a font.

```
UIFont *myFont = [UIFont systemFontOfSize:12.0];  
UIFont *h = [UIFont fontWithName:@"Helvetica" size:36.0];  
NSArray *availableFonts = [UIFont familyNames];
```

- Then use special NSString methods to draw the text

```
NSString *text = ...;  
[text drawAtPoint:(CGPoint)pt  
    withFont:myFont]; // NSString instance method
```

- How much space will a piece of text will take up when drawn?

```
CGSize textSize = [text sizeWithFont:myFont];  
// NSString instance method
```


Drawing Text

- You might be disturbed that there is a Foundation method for drawing (which is a UIKit thing).
- But actually these `NSString` methods are defined in UIKit via a mechanism called **categories**.
- Categories are an Objective-C way to add methods to an existing class without subclassing.
- Usually, categories have names.
- Class extensions (that we used to declare private `@propertys` and methods in our labs) are like anonymous categories, except that the methods they declare must be implemented in the main `@implementation` block for the corresponding class.
- We'll cover how (and when) to use categories a bit later in this course.

Drawing Images

- Use `UIImageView` to draw images, but if you feel you must ...
(Note that we will cover `UIImageView` later in the course)

- Create a `UIImage` object from a file in your Resources folder:

```
UIImage *image = [UIImage imageNamed:@"foo.jpg"];
```

- Or create one from a named file or from raw data:

(of course, we haven't talked about the file system yet, but ...)

```
UIImage *img1 = [[UIImage alloc]
                 initWithContentsOfFile:(NSString *)path];
UIImage *img2 = [[UIImage alloc]
                 initWithData:(NSData *)imageData];
```

- Or you can even create one by drawing with `CGContext` functions:

```
UIGraphicsBeginImageContext(CGSize);
// draw with CGContext functions
UIImage *myImage = UIGraphicsGetImageFromCurrentContext();
UIGraphicsEndImageContext();
```

Drawing Images

- To draw the UIImage's bits into the current graphics context:

```
UIImage *image = ...;  
[image drawAtPoint:(CGPoint)pt];  
// pt is the upper left corner of the image
```

- Scale the image to fit in a CGRect:

```
[image drawInRect:(CGRect)aRect];
```

- Tile the image into a CGRect:

```
[image drawAsPatternInRect:(CGRect)patternRect];
```

- You can also get a PNG or JPG data representation of UIImage:

```
NSData *jpgData = UIImageJPEGRepresentation(  
    (UIImage *)myImage, (CGFloat)quality);
```

```
NSData *pngData = UIImagePNGRepresentation(  
    (UIImage *)myImage);
```

Autorotation

- What goes on in your Controller when the device is rotated?
You can control whether the user interface rotates along with it.
- Implement the following method in your Controller:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:  
    (UIInterfaceOrientation)orientation  
{  
    return UIInterfaceOrientationIsPortrait(orientation);  
    // only support portrait  
  
    return YES;  
    // support all orientations  
  
    return  
(orientation != UIInterfaceOrientationPortraitUpsideDown);  
    // anything but  
}
```

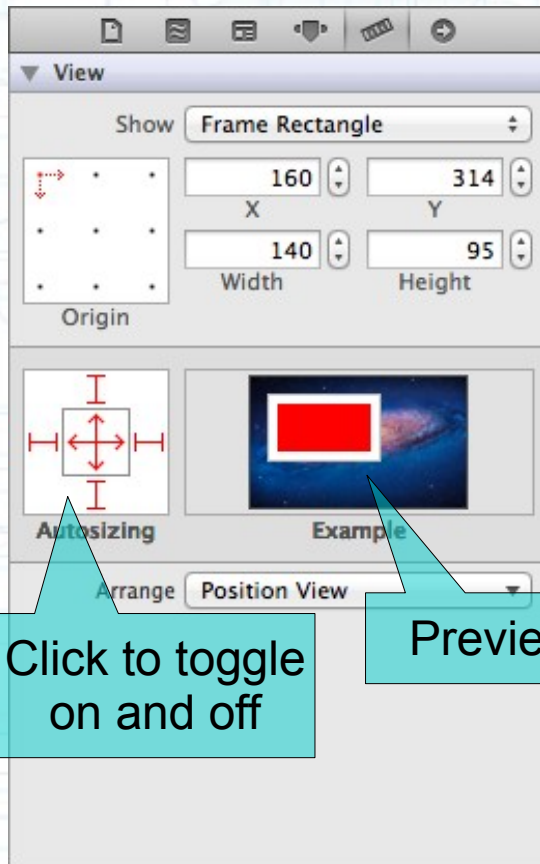
Autorotation

If you support an orientation, what will happen when rotated?

- The frame of all subviews in your Controller's View will be adjusted.
- The adjustment is based on their "struts and springs".
- You set "struts and springs" in Interface Builder.
- When a view's bounds changes because its frame is altered, does `drawRect:` get called again? No.

Struts and Springs

Set a view's struts and springs in Size Inspector in Xcode

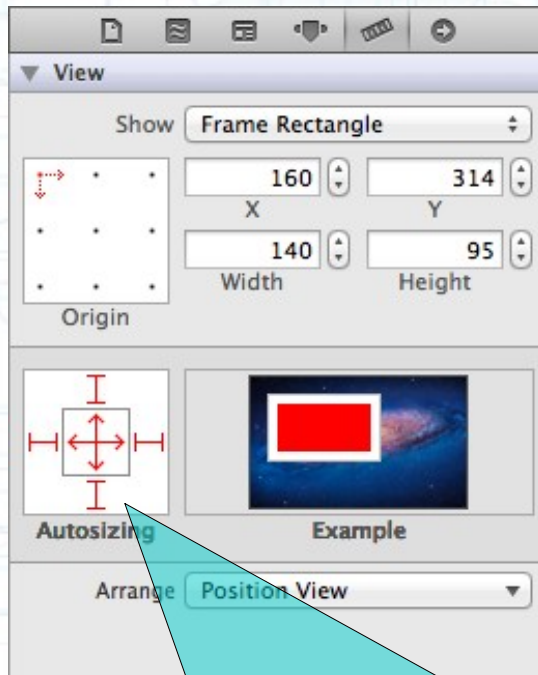


Click to toggle
on and off

Preview

Struts and Springs

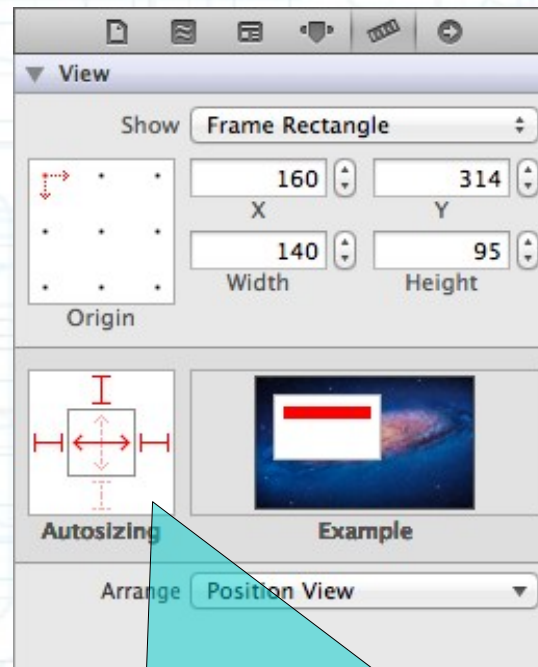
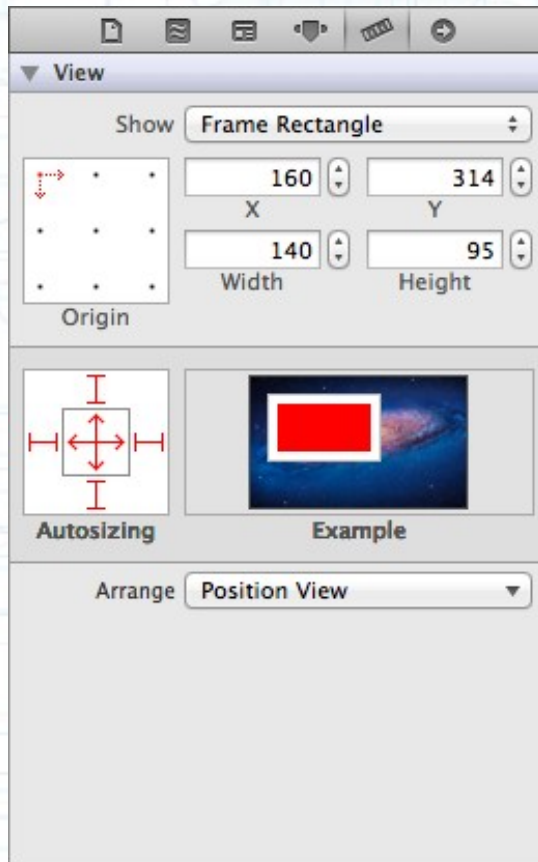
Set a view's struts and springs in Size Inspector in Xcode



Grows and shrinks as its superview's bounds grow and shrink because struts fixed to all sides and both springs allow expansion.

Struts and Springs

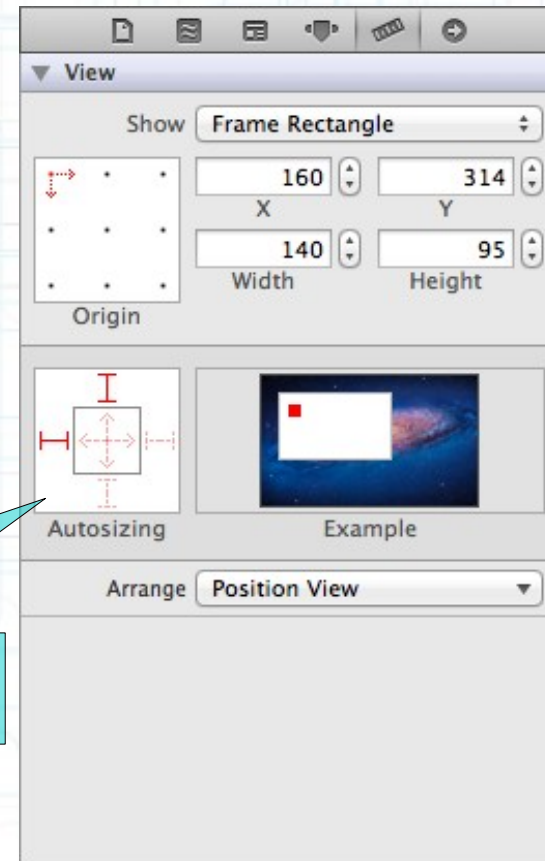
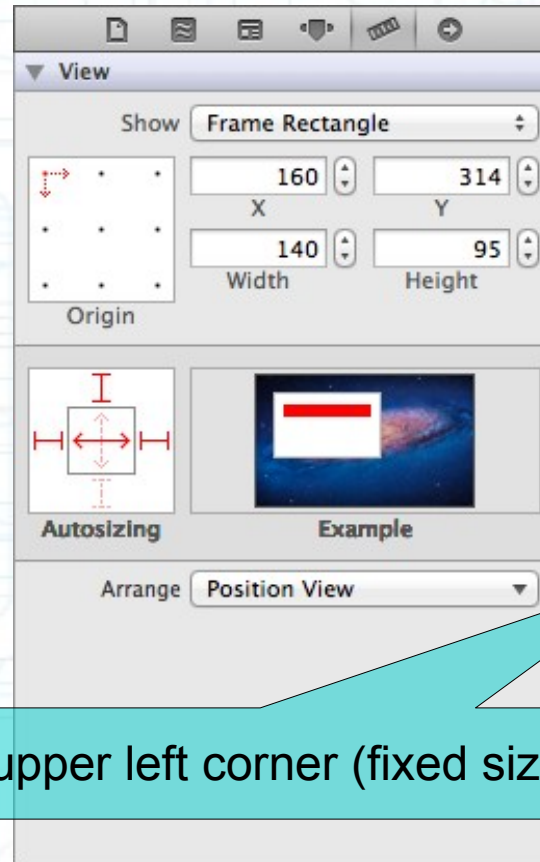
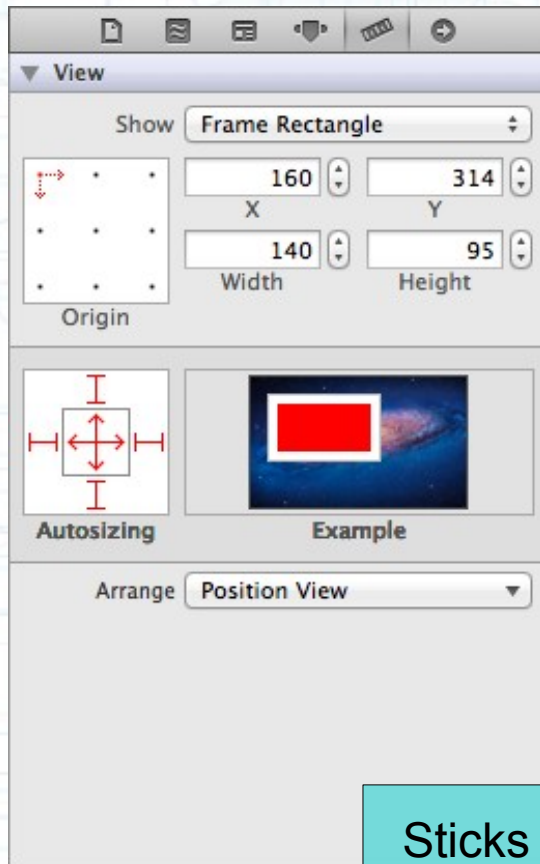
Set a view's struts and springs in Size Inspector in Xcode



Grows and shrinks only horizontally as its superview's bounds grow and shrink and sticks to the top in its superview.

Struts and Springs

Set a view's struts and springs in Size Inspector in Xcode



Sticks to upper left corner (fixed size).

Redraw on bounds change

- By default, there is no redraw when your `UIView`'s bounds change.
- Instead, the “bits” of your view will be stretched or squished or moved.
- Often this is not what you want. Luckily, there is a `UIView` `@property` to control this:

```
@property (nonatomic) UIViewContentMode contentMode;  
UIViewContentMode{Left, Right, Top, Bottom,  
    BottomLeft, BottomRight, TopLeft, TopRight, ...}
```

- The above is not springs and struts! This is after springs and struts have been applied!
- These content modes move the bits of your drawing to that location.

Redraw on bounds change

- For bit stretching or shrinking set `contentMode` to:
`UIViewContentModeScale{ToFill, AspectFill, AspectFit}`
- But many times you want just to call `drawRect:.` For this use:
`UIViewContentModeRedraw`
- Default is `UIViewContentModeScaleToFill`.

You can control which of your bits get stretched

```
@property (nonatomic) CGRect contentStretch;
```

- Rectangle of `((0, 0), (1, 1))` stretches all the bits.
- Something smaller stretches only a portion of the bits. If width/height is 0, duplicates a pixel.

More Complex Autorotation

- When Interface Builder is not enough we can do things programmatically.
- The View Controller can implement this method:

```
- (void)willAnimateRotationToInterfaceOrientation:  
    (UIInterfaceOrientation)orientation  
    duration:  
    (NSTimeInterval)duration  
{  
    if (orientation == UIInterfaceOrientationPortrait)  
    {  
        // Set up views for portrait mode.  
    }  
    else if (orientation == UIInterfaceOrientationLandscapeLeft  
            || orientation == UIInterfaceOrientationLandscapeRight)  
    {  
        // Set up views for landscape mode.  
    }  
}
```

Initializing a UIView

- Yes, you can override `initWithFrame:`. Use previously explained syntax:

```
self = [super initWithFrame:aRect]
```

- But you will also want to set up stuff in `awakeFromNib`.

This is because `initWithFrame:` is **NOT** called for a `UIView` coming out of a storyboard!

But `awakeFromNib` is. It's called "awakeFromNib" for historical reasons.

- Typical code

```
- (void)setup { /* set up your custom view here */ }  
- (void)awakeFromNib { [self setup]; }  
- (id)initWithFrame:(CGRect)aRect  
{  
    self = [super initWithFrame:aRect];  
    [self setup];  
    return self;  
}
```

Protocols

Similar to @interface, but someone else does the implementing

```
@protocol Foo <Other, NSObject>
// implementors must implement Other and NSObject too
- (void)doSomething;
/* implementors must implement this
   (methods are @required by default) */
@optional
- (int)getSomething;
- (void)doSomethingOptionalWith:(NSString *)argument;
// implementors do not have to implement these methods
@required
- (NSArray *)getManySomethings:(int)howMany;
// back to being "must implement"
@property(nonatomic, strong) NSString *fooProp;
/* note that you must specify strength
   (unless the property is readonly) */
@end
```

- The NSObject protocol includes most of the methods implemented by NSObject. Many protocols require whoever implements them to basically “be an NSObject” by requiring the NSObject protocol as a “sub-protocol” using this syntax.

Protocols

Similar to @interface, but someone else does the implementing

```
@protocol Foo <Other, NSObject>
// implementors must implement Other and NSObject too
- (void)doSomething;
/* implementors must implement this
   (methods are @required by default) */
@optional
- (int)getSomething;
- (void)doSomethingOptionalWith:(NSString *)argument;
// implementors do not have to implement these methods
@required
- (NSArray *)getManySomethings:(int)howMany;
// back to being "must implement"
@property(nonatomic, strong) NSString *fooProp;
/* note that you must specify strength
   (unless the property is readonly) */
@end
```

- Protocols are defined in a header file. Either its own header file (e.g. Foo.h) or the header file of the class which wants other classes to implement it. For example, the UIScrollViewDelegate protocol is defined in UIScrollView.h.

Protocols

- Classes then say in their `@interface` if they implement a protocol:

```
#import "Foo.h"  
/* importing the header file that  
   declares the Foo @protocol */  
  
@interface MyClass : NSObject <Foo>  
/* MyClass is saying it implements  
   the Foo @protocol */  
  
...  
  
@end
```

- We say that `MyClass` conforms to the `Foo` protocol.
- You must implement all non-`@optional` methods (or the compiler will show warnings if you do not).

Protocols

- We can then declare `id` variables with a protocol requirement:

```
id <Foo> obj = [[MyClass alloc] init];  
//compiler will love this!
```

```
id <Foo> obj = [NSArray array];  
//compiler will not like this one bit!
```

- Also can declare arguments to methods to require a protocol:
- `(void)giveMeObject:(id <Foo>)objectImplementingFoo;`

- Properties too:

```
@property (nonatomic, weak) id <Foo> myFooProperty;
```

- If you call these and pass an object which does not implement the `Foo` `@protocol` you will get a compiler warning!
- Just like static typing, this is all just compiler-helping-you stuff. It makes no difference at runtime.
- Think of it as documentation for your method interfaces. It's another powerful way to leverage the `id` type.

Protocols

The first use of protocols in iOS: delegates and data sources

- A delegate or dataSource is pretty much always defined as a weak @property, by the way.

```
@property(nonatomic, weak) id <UIObjectDelegate> delegate;
```

- This assumes that the object serving as delegate will outlive the object doing the delegating.
- Especially true in the case where the delegator is a View object (e.g. UIScrollView) and the delegate is that View's Controller.
- Controllers always create and clean up their View objects.
- Thus the Controller will always outlive its View objects.
- dataSource is just like a delegate, but, as the name implies, we are delegating provision of data.
- Views commonly have a dataSource because Views cannot own their data!

Protocols

```
@protocol UIScrollViewDelegate
@optional
-(UIView *)viewForZoomingInScrollView:(UIScrollView *)sender;
-(void)scrollViewDidEndDragging:(UIScrollView *)sender
willDecelerate:(BOOL)decelerate;
...
@end

@interface UIScrollView : UIView
@property(nonatomic,weak) id <UIScrollViewDelegate> delegate;
@end
```

Protocols

```
@interface MyViewController : UIViewController
    <UIScrollViewDelegate>
@property(n nonatomic, weak) IBOutlet UIScrollView *scrollView;
@end

@implementation MyViewController

@synthesize scrollView = _scrollView;

- (void)setScrollView:(UIScrollView *)scrollView
{
    _scrollView = scrollView;
    self.scrollView.delegate = self; //compiler won't complain
}

- (UIView *)viewForZoomingInScrollView:(UIScrollView *)sender
{
    return ...;
}

@end
```

UIGestureRecognizer

We've seen how to draw in our `UIView`, but how do we get touches?

- We can get notified of the raw touch events (touch down, moved, up).
- Or we can react to certain, predefined “gestures”. This latter is the way to go.

Gestures are recognized by the class `UIGestureRecognizer`

- This class is “abstract”. We only actually use “concrete subclasses” of it.
- There are two sides to using a gesture recognizer:
 1. Adding a gesture recognizer to a `UIView` to ask it to recognize that gesture.
 2. Providing the implementation of a method to “handle” that gesture when it happens.

UIGestureRecognizer

Usually #1 is done by a Controller

- Though occasionally a `UIView` will do it to itself if it just doesn't make sense without that gesture.

Usually #2 is provided by the `UIView` itself

- But it would not be unreasonable for the Controller to do it.
- Or for the Controller to decide it wants to handle a gesture differently than the view does.

UIGestureRecognizer

Adding a gesture recognizer to a UIView from a Controller

```
- (void)setPannableView:(UIView *)pannableView
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *panGRec =
        [[UIPanGestureRecognizer alloc]
         initWithTarget:pannableView
         action:@selector(pan:)];
    [pannableView addGestureRecognizer:panGRec];
}
```

This is a concrete subclass of `UIGestureRecognizer` that recognizes “panning” (moving something around with your finger). There are, of course, other concrete subclasses (for swipe, pinch, tap, etc).

UIGestureRecognizer

Adding a gesture recognizer to a `UIView` from a Controller

```
- (void)setPannableView:(UIView *)pannableView
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *panGRec =
        [[UIPanGestureRecognizer alloc]
         initWithTarget:pannableView
         action:@selector(pan:)];
    [pannableView addGestureRecognizer:panGRec];
}
```

Note that we are specifying the view itself as the target to handle a pan gesture when it is recognized. Thus the view will be both the recognizer and the handler of the gesture. The `UIView` does not have to handle the gesture. It could be, for example, the Controller that handles it. The View would generally handle gestures to modify how the View is drawn. The Controller would have to handle gestures that modify the Model.

UIGestureRecognizer

Adding a gesture recognizer to a UIView from a Controller

```
- (void)setPannableView:(UIView *)pannableView
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *panGRec =
        [[UIPanGestureRecognizer alloc]
         initWithTarget:pannableView
                  action:@selector(pan:)];
    [pannableView addGestureRecognizer:panGRec];
}
```

This is the action method that will be sent to the target (the `pannableView`) during the handling of the recognition of this gesture. This version of the action message takes one argument (which is the `UIGestureRecognizer` that sends the action), but there is another version that takes no arguments if you would prefer it. We'll look at the implementation of this method in a moment.

UIGestureRecognizer

Adding a gesture recognizer to a `UIView` from a Controller

```
- (void)setPannableView:(UIView *)pannableView
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *panGRec =
        [[UIPanGestureRecognizer alloc]
         initWithTarget:pannableView
         action:@selector(pan:)];
    [pannableView addGestureRecognizer:panGRec];
}
```

If we don't do this, then even though the `pannableView` implements `pan:`, it would never get called because we would have never added this gesture recognizer to the view's list of gestures that it recognizes. Think of this as "turning the handling of this gesture on."

UIGestureRecognizer

Adding a gesture recognizer to a UIView from a Controller

```
- (void)setPannableView:(UIView *)pannableView
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *panGRec =
        [[UIPanGestureRecognizer alloc]
         initWithTarget:pannableView
         action:@selector(pan:)];
    [pannableView addGestureRecognizer:panGRec];
}
```

- Only UIView instances can recognize a gesture (because UIViews handle all touch input).
- But any object can tell a UIView to recognize a gesture (by adding a recognizer to the UIView).
- And any object can handle the recognition of a gesture (by being the target of the gesture's action).

UIGestureRecognizer

- How do we implement the target of a gesture recognizer?
Each concrete class provides some methods to help you do that.
- For example, UIPanGestureRecognizer provides 3 methods:
 - `(CGPoint)translationInView:(UIView *)aView;`
 - `(CGPoint)velocityInView:(UIView *)aView;`
 - `(void)setTranslation:(CGPoint)translation
inView:(UIView *)aView;`

UIGestureRecognizer

- The base class `UIGestureRecognizer` provides this @property:

```
@property (readonly) UIGestureRecognizerState state;
```

- Gesture Recognizers sit around in the state `Possible` until they start to be recognized.
- Then they either go to `Recognized` (for discrete gestures like a tap).
- Or they go to `Began` (for continuous gestures like a pan).
- At any time, the state can change to `Failed` (so watch out for that).
- If the gesture is continuous, it will move on to the `Changed` and eventually the `Ended` state.
- Continuous gestures can also go to `Cancelled` state (if the recognizer realizes it's not this gesture after all).

UIGestureRecognizer

So, given these methods, what would `pan:` look like?


```
- (void)pan:(UIPanGestureRecognizer *)recognizer  
{
```

```
}
```

UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
- (void)pan:(UIPanGestureRecognizer *)recognizer  
{  
  if (recognizer.state == UIGestureRecognizerStateChanged ||  
      recognizer.state == UIGestureRecognizerStateEnded)  
  {
```



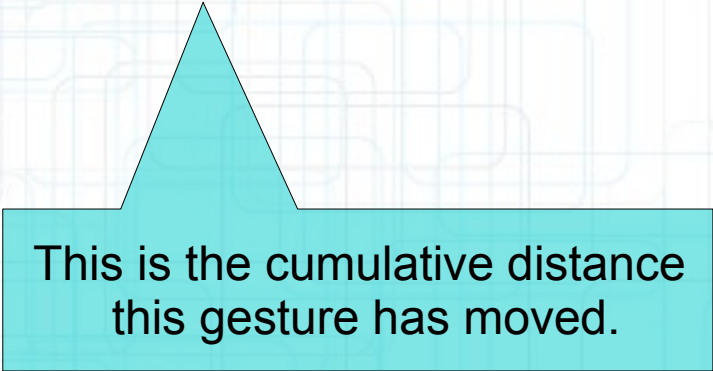
We are going to update our view every time the touch moves (and when the touch ends). This is “smooth panning”.

```
}
```

UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
- (void)pan:(UIPanGestureRecognizer *)recognizer
{
    if (recognizer.state == UIGestureRecognizerStateChanged ||
        recognizer.state == UIGestureRecognizerStateEnded)
    {
        CGPoint translation = [recognizer translationInView:self];
```



This is the cumulative distance
this gesture has moved.

```
}
```


UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
- (void)pan:(UIPanGestureRecognizer *)recognizer
{
if (recognizer.state == UIGestureRecognizerStateChanged ||
    recognizer.state == UIGestureRecognizerStateEnded)
{
CGPoint translation = [recognizer translationInView:self];

/* Move something in myself (I'm a UIView)
 * by translation.x and translation.y.
 * For example, if I were a graph and my origin
 * was set by an @property called origin. */
self.origin = CGPointMake(self.origin.x + translation.x,
                          self.origin.y + translation.y);
[recognizer setTranslation:CGPointZero inView:self];
}
}
```

Here we are resetting the cumulative distance to zero. Now each time this is called, we'll get the "incremental" movement of the gesture (which is what we want). If we wanted the "cumulative" movement of the gesture, we would not include this line of code.

Other Concrete Gestures

UIPinchGestureRecognizer

```
@property CGFloat scale;  
//note that this is not readonly (can reset each movement)  
  
@property (readonly) CGFloat velocity;  
//note that this is readonly; scale factor per second
```

UIRotationGestureRecognizer

```
@property CGFloat rotation;  
//note that this is not readonly; in radians  
  
@property (readonly) CGFloat velocity;  
//note that this is readonly; radians per second
```

Other Concrete Gestures

UISwipeGestureRecognizer

- This one you “set up” (with the following) to find certain swipe types, then look for Recognized state:

```
@property UISwipeGestureRecognizerDirection direction;  
// what direction swipes you want  
  
@property NSUInteger numberOfTouchesRequired;  
// two finger swipes? or just one finger? or more?
```

UITapGestureRecognizer

- Set up (with the following) then look for Recognized state:

```
@property NSUInteger numberOfTapsRequired;  
// single tap or double tap or triple tap, etc.  
  
@property NSUInteger numberOfTouchesRequired;  
// e.g., require two finger tap?
```

Next Time

View Controllers and Storyboarding:

- MVCs Working Together
- Segues
- Navigation Controllers
- View Controllers
- Tab Bar Controller