

# Stanford CS193p

Developing Applications for iOS  
Fall 2011



# Today

- A couple of more things about last Thursday's demo

Why no `weak` or `strong` on the `@property (readonly) id` program?

`[CalculatorBrain ...]` or `[[self class] ...]` when calling a class method from an instance method?

- Autorotation

And how your custom `UIView` will react to bounds changes

And how to initialize a `UIView` (it's more than just overriding `initWithFrame:`)

- Objective C

Protocols

- Gesture Recognizers

Handling Touch Input

- Demo

Happiness

Custom `UIView` with Gestures



# 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  
}
```

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

The frame of all subviews in your Controller's View will be adjusted.

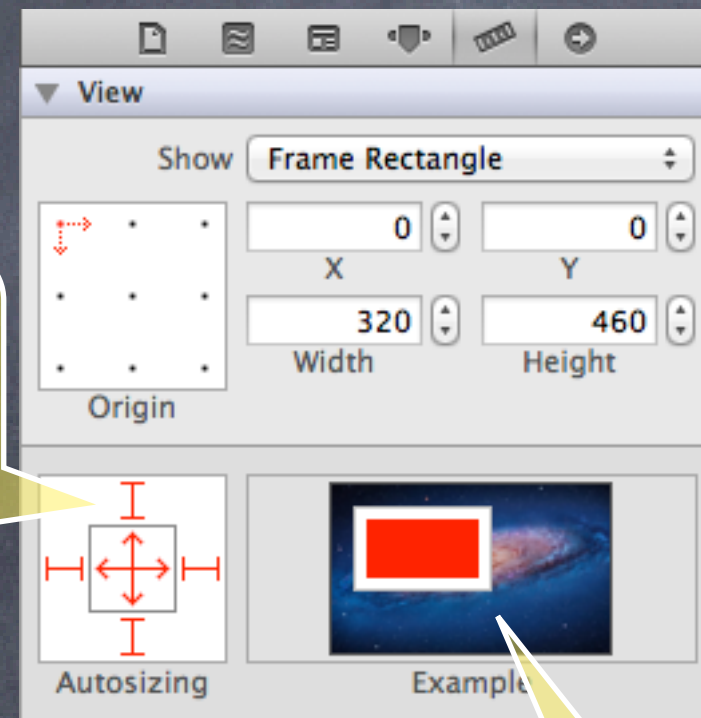
The adjustment is based on their "struts and springs".

You set "struts and springs" in Xcode.

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

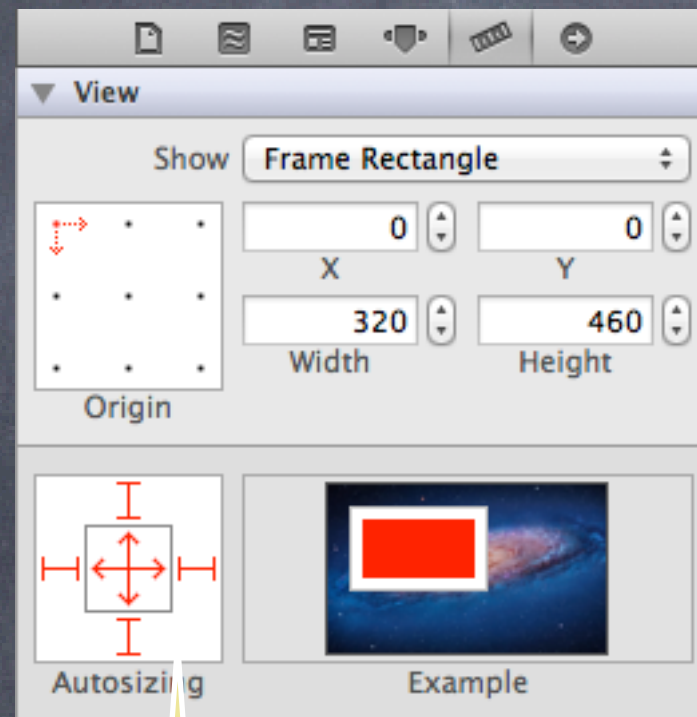


Click to toggle on and off.

Preview

# Struts and Springs

- Set a view's struts and springs in size inspector in Xcode

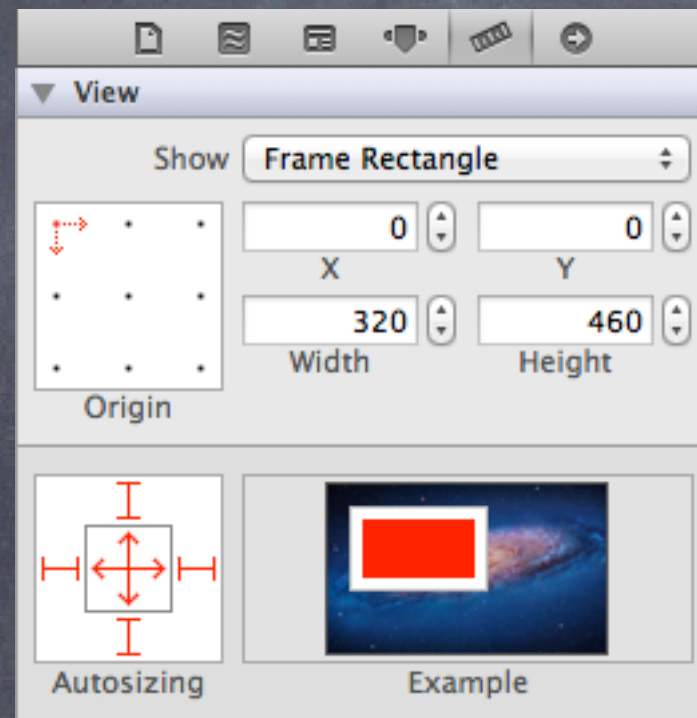


Grows and shrinks as its superview's bounds grow and shrink because struts fixed to all sides and both springs allow expansion.

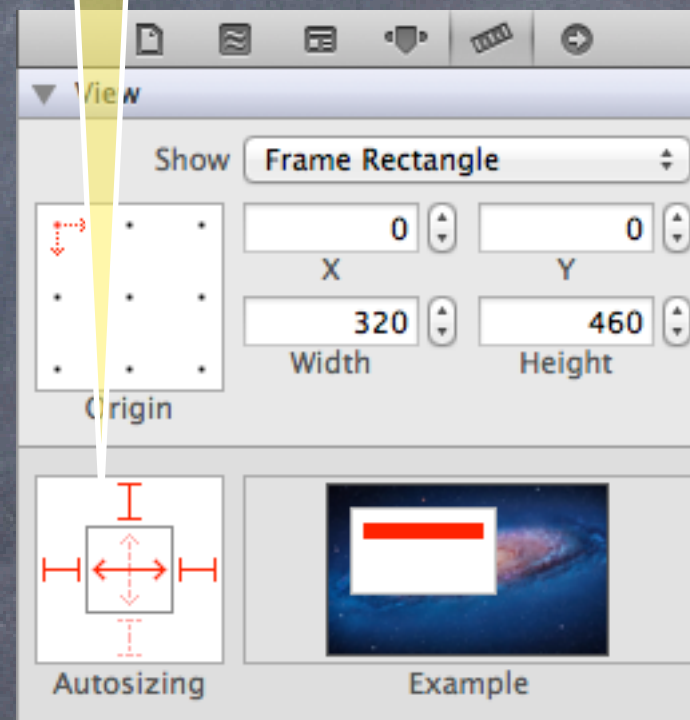


# Struts and Springs

- Set a view's struts and springs in size inspector in Xcode

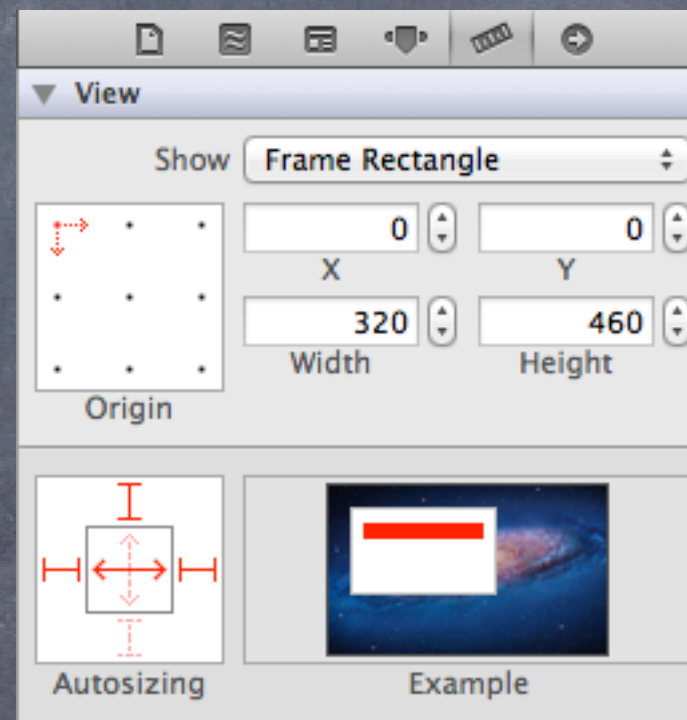
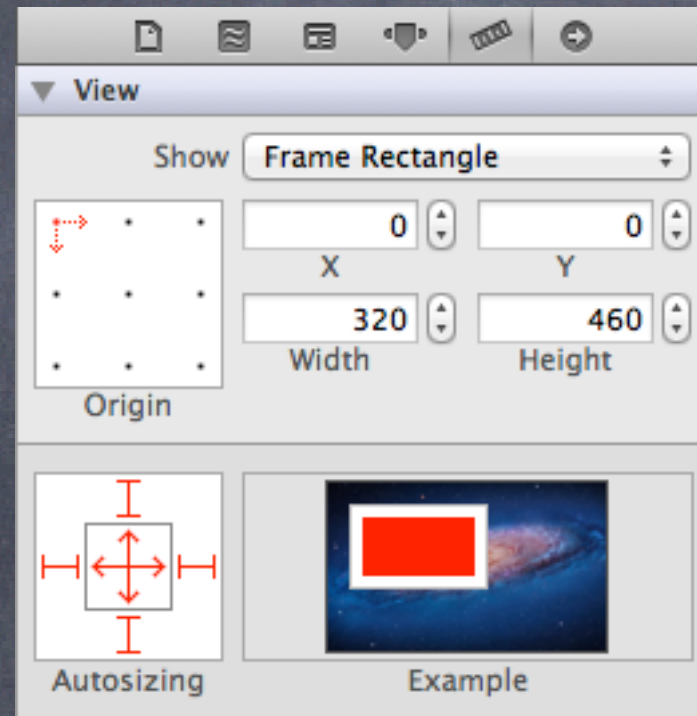


Grows and shrinks only horizontally as its superview's bounds grow and shrink and sticks to the top in its superview.

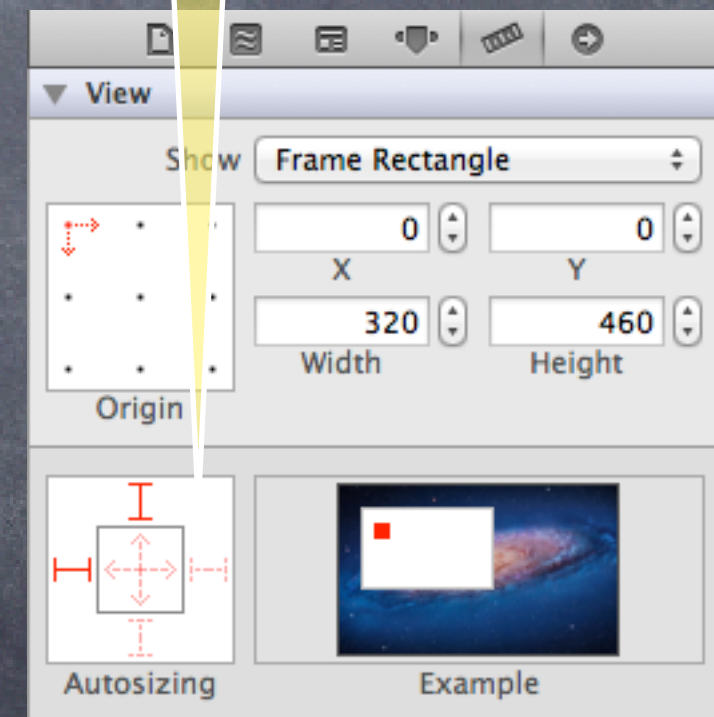


# Struts and Springs

- 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, when your UIView's bounds change, no redraw. 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,Right,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.

```
UIViewContentModeScale{ToFill,AspectFill,AspectFit} // bit stretching/shrinking
```

```
UIViewContentModeRedraw // call drawRect: (this is many times what you want)
```

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.



# Initializing a UIView

- Yes, you can override `initWithFrame:`

Use previously explained `self = [super initWithFrame:aRect]` syntax.

- 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 { ... }  
- (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;           // implementors do not have to implement this
- (void)doSomethingOptionalWithArgument:(NSString *)argument; // also optional
@required
- (NSArray *)getManySomethings:(int)howMany; // back to being "must implement"
@property (nonatomic, strong) NSString *fooProp; // note that you must specify strength
// (unless it's readonly, of course)
@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;           // implementors do not have to implement this
- (void)doSomethingOptionalWithArgument:(NSString *)argument; // also optional
@required
- (NSArray *)getManySomethings:(int)howMany; // back to being "must implement"
@property (nonatomic, strong) NSString *fooProp; // note that you must specify strength
// (unless it's readonly, of course)
@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
```

- You must implement all non-@optional methods

Or face the wrath of the compiler if you do not.

- 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)giveMeFooObject:(id <Foo>)anObjectImplementingFoo;
@property (nonatomic, weak) id <Foo> myFooProperty; // properties too!
```

If you call these and pass an object which does not implement Foo ... compiler warning!



# Protocols

- 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

- #1 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 <UISomeObjectDelegate> 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`)

& the delegate is that View's Controller.

Controllers always create and clean up their View objects (because they are their "minions").

Thus the Controller will always outlive its View objects.

`dataSource` is just like a delegate, but, as the name implies, we're 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

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

@implementation MyViewController
- (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, 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.
- 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 *pangr =  
        [[UIPanGestureRecognizer alloc] initWithTarget:pannableView action:@selector(pan:)];  
    [pannableView addGestureRecognizer:pangr];  
}
```

This is a concrete subclass of **UIGestureRecognizer** that recognizes “panning” (moving something around with your finger).

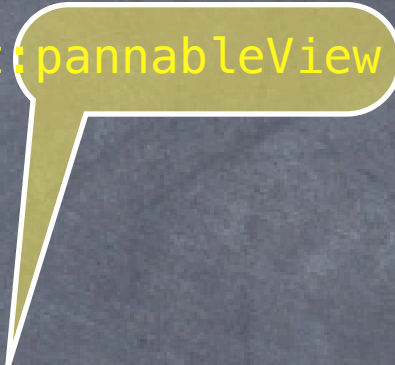
There are, of course, other concrete subclasses (for swipe, pinch, tap, etc.).



# UIGestureRecognizer

## 👁 Adding a gesture recognizer to a **UIView** from a Controller

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




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 modified the Model.

# UIGestureRecognizer

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

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



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'd prefer.

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 *pangr =  
        [[UIPanGestureRecognizer alloc] initWithTarget:pannableView action:@selector(pan:)];  
    [pannableView addGestureRecognizer:pangr];  
}
```

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 *pangr =
        [[UIPanGestureRecognizer alloc] initWithTarget:pannableView action:@selector(pan:)];
    [pannableView addGestureRecognizer:pangr];
}
```

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;

- Also, 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'll move on to the **Changed** and eventually the **Ended** state

Continuous 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're 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);
    }
}
```

# 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

- UISwipeGestureRecognizer

This one you “set up” (w/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? **more?**

- UITapGestureRecognizer

Set up (w/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?

# Demo

- Happiness

Shows a level of happiness graphically using a smiley/frowny face

- Model

```
int happiness; // very simple Model!
```

- View

Custom view called FaceView

- Controller

HappinessViewController

- Watch for ...

**drawRect:** (including a drawing “subroutine” (push/pop context))

How FaceView delegates its data ownership to the Controller with a protocol

One gesture handled by Controller (Model change), the other by View (View change)



# Coming Up

## 👁 Thursday

View Controller Lifecycle  
Controllers of Controllers  
Storyboarding  
Universal Applications  
Demo

## 👁 Friday

Getting your project running on a device