Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- Interface Builder

  Demo: Viewing and Editing your custom UIViews in your storyboard (FaceView)

- The Happiness MVC's Model

  It's happiness, of course! (which is different from smiliness)

- Protocols and Delegation

  How can generic UIViews not "own their data" and still draw that data?

  Demo: Showing the Happiness MVC's Model using generic FaceView in its View

- Gestures

  Demo: Happiness pinch and pan

- Multiple MVCs (time permitting)

  Split View Controllers & Navigation Controllers & Tab Bar Controllers

  Segues

  Popovers

Extensions

- Miscellaneous topic!
  - You can add methods and properties to a class (even if you don't have the source).
- There are some restrictions
  - You can't re-implement methods or properties that are already there (only add new ones).
  - The properties you add can have no storage associated with them.
- This feature is easily abused
  - It should be used to add clarity to readability not obfuscation!
  - Don't use it as a substitute for good object-oriented design technique.
  - Best used (at least for beginners) for very small, well-contained helper functions.
  - Can actually be used well to organize code but requires architectural commitment.
  - When in doubt (for now), don't do it.

Protocols

- A way to express an API minimally

  Instead of forcing the caller to pass a `class`/`struct`, we can ask for specifically what we want

  We just specify the properties and methods needed

- A protocol is a TYPE just like any other type, except …

  It has no storage or implementation associated with it

  Any storage or implementation required to implement the protocol is in an implementing type

  An implementing type can be any `class`, `struct` or `enum`

  Otherwise, a `protocol` can be used as a type to declare variables, as a function parameter, etc.

- There are three aspects to a protocol

  1. the `protocol` declaration (what properties and methods are in the `protocol`)
  2. the declaration where a `class`, `struct` or `enum` says that it implements a `protocol`
  3. the actual implementation of the `protocol` in said `class`, `struct` or `enum`

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2

You must specify whether a property is get only or both get and set

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2

You must specify whether a property is get only or both get and set

Any functions that are expected to mutate the receiver should be marked mutating

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : class, InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2

You must specify whether a property is get only or both get and set

Any functions that are expected to mutate the receiver should be marked mutating

(unless you are going to restrict your protocol to class implementers only with class keyword)

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2
You must specify whether a property is get only or both get and set
Any functions that are expected to mutate the receiver should be marked mutating
(unless you are going to restrict your protocol to class implementers only with class keyword)
You can even specify that implementers must implement a given initializer

Protocols

- How an implementer says "I implement that protocol"

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
     // implementation of SomeClass here
     // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class

Protocols

- How an implementer says "I implement that protocol"

```
enum SomeEnum : SomeProtocol, AnotherProtocol {
    // implementation of SomeEnum here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part

Protocols

- How an implementer says "I implement that protocol"

```
struct SomeStruct : SomeProtocol, AnotherProtocol {
    // implementation of SomeStruct here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part

Protocols

- How an implementer says "I implement that protocol"

```
struct SomeStruct : SomeProtocol, AnotherProtocol {
     // implementation of SomeStruct here
     // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part
Any number of protocols can be implemented by a given class, struct or enum

Protocols

- How an implementer says "I implement that protocol"

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here, including …
    required init(…)
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part
Any number of protocols can be implemented by a given class, struct or enum
In a class, inits must be marked required (or otherwise a subclass might not conform)

Protocols

- How an implementer says "I implement that protocol"

```
extension Something : SomeProtocol {
    // implementation of SomeProtocol here
    // no stored properties though
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part
Any number of protocols can be implemented by a given class, struct or enum
In a class, inits must be marked required (or otherwise a subclass might not conform)
You <u>are</u> allowed to add protocol conformance via an extension

Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

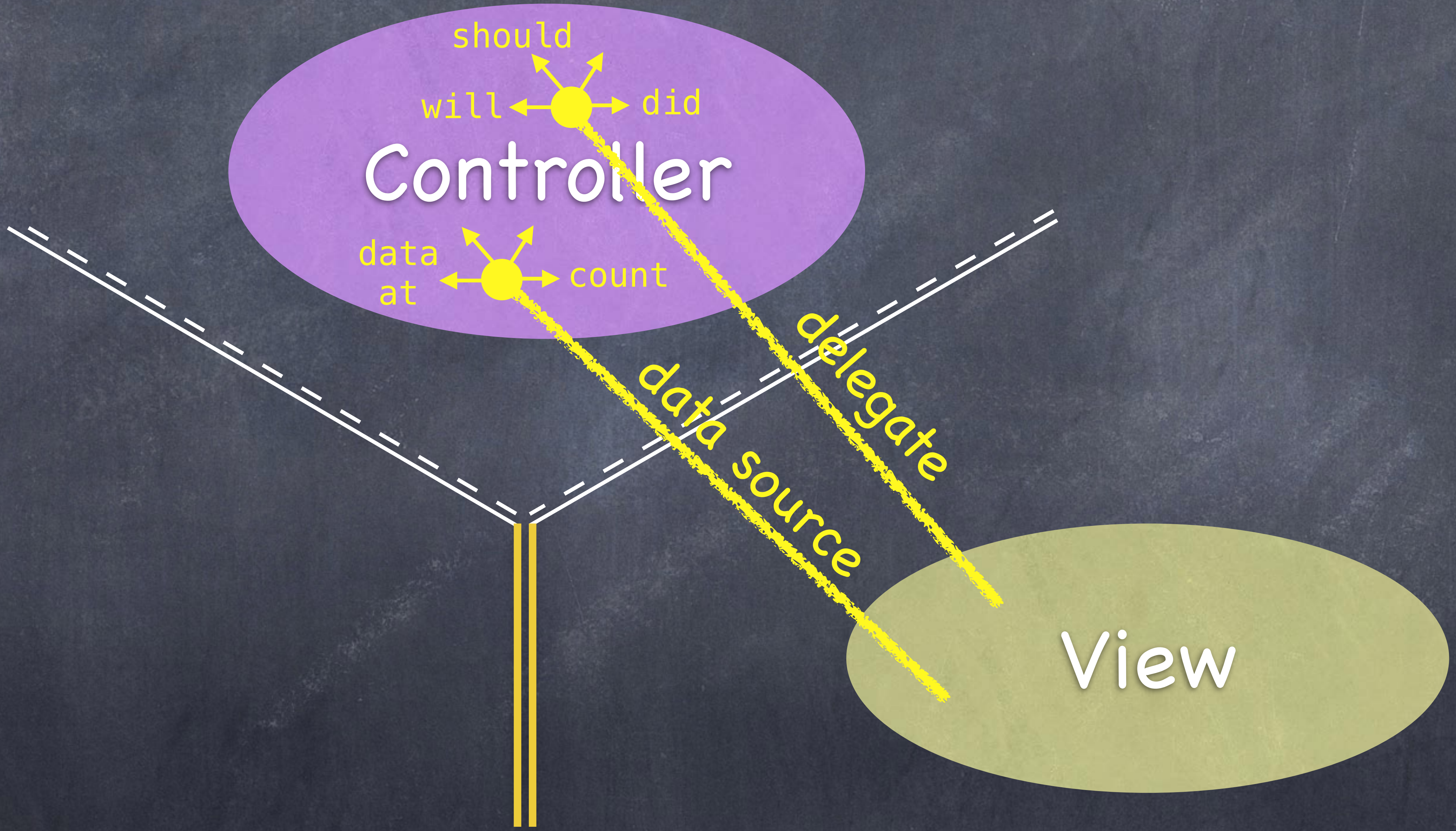
```
var thingToMove: Moveable = prius
thingToMove.moveTo(…)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]

func slide(slider: Moveable) {
    let positionToSlideTo = …
    slider.moveTo(positionToSlideTo)
}
slide(prius)
slide(square)
func slipAndSlide(x: protocol<Slippery,Moveable>)
slipAndSlide(prius)
```

Delegation

- A very important use of protocols

  It's how we can implement "blind communication" between a View and its Controller



Delegation

- A very important use of protocols

  It's how we can implement "blind communication" between a View and its Controller

- How it plays out …

  1. Create a delegation protocol (defines what the View wants the Controller to take care of)
  2. Create a delegate property in the View whose type is that delegation protocol
  3. Use the delegate property in the View to get/do things it can't own or control
  4. Controller declares that it implements the protocol
  5. Controller sets self as the delegate of the View by setting the property in #2 above
  6. Implement the protocol in the Controller

- Now the View is hooked up to the Controller

  But the View still has no idea what the Controller is, so the View remains generic/reusable

Demo

- Let's see FaceView delegate its "data"

  That way FaceView can stay generic

  It won't be tied to HappinessViewController, so it can be used by other Controllers

  Since it's doing this to get its data (its smiliness), we'll call our delegate property dataSource

Gestures

- We've seen how to draw in a `UIView`, how do we get touches?
  - We can get notified of the raw touch events (touch down, moved, up, etc.)
  - Or we can react to certain, predefined "gestures." The latter is the way to go!
- Gestures are recognized by instances of `UIGestureRecognizer`
  - The base class is "abstract." We only actually use concrete subclasses to recognize.
- There are two sides to using a gesture recognizer
  1. Adding a gesture recognizer to a `UIView` (asking the `UIView` to "recognize" that gesture)
  2. Providing a method to "handle" that gesture (not necessarily handled by the `UIView`)
- Usually the first is done by a Controller
  - Though occasionally a `UIView` will do this itself if the gesture is integral to its existence
- The second is provided either by the `UIView` or a `Controller`
  - Depending on the situation. We'll see an example of both in our demo.

Gestures

- Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let recognizer = UIPanGestureRecognizer(target: self, action: "pan:")
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

This is just a normal outlet to the UIView we want to recognize the gesture

Gestures

- Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let recognizer = UIPanGestureRecognizer(target: self, action: "pan:")
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

This is just a normal outlet to the UIView we want to recognize the gesture
We use its property observer to get involved when the outlet gets hooked up by iOS

Gestures

- Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let recognizer = UIPanGestureRecognizer(target: self, action: "pan:")
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

This is just a normal outlet to the UIView we want to recognize the gesture
We use its property observer to get involved when the outlet gets hooked up by iOS
Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)

Gestures

- Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller’s View to recognize a “pan” gesture ...

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let recognizer = UIPanGestureRecognizer(target: self, action: “pan:”)
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

This is just a normal outlet to the UIView we want to recognize the gesture
We use its property observer to get involved when the outlet gets hooked up by iOS
Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
The target gets notified when the gesture is recognized (in this case, the Controller itself)

Gestures

- Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture ...

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let recognizer = UIPanGestureRecognizer(target: self, action: "pan:")
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

This is just a normal outlet to the UIView we want to recognize the gesture
We use its property observer to get involved when the outlet gets hooked up by iOS
Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
The target gets notified when the gesture is recognized (in this case, the Controller itself)
The action is the method invoked on recognition (the : means it has an argument)

Gestures

- Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture ...

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let recognizer = UIPanGestureRecognizer(target: self, action: “pan:”)
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

This is just a normal outlet to the UIView we want to recognize the gesture
We use its property observer to get involved when the outlet gets hooked up by iOS
Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
The target gets notified when the gesture is recognized (in this case, the Controller itself)
The action is the method invoked on recognition (the : means it has an argument)
Here we ask the UIView to actually start trying to recognize this gesture in its bounds

Gestures

Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller’s View to recognize a “pan” gesture …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let recognizer = UIPanGestureRecognizer(target: self, action: “pan:”)
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

This is just a normal outlet to the UIView we want to recognize the gesture
We use its property observer to get involved when the outlet gets hooked up by iOS
Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
The target gets notified when the gesture is recognized (in this case, the Controller itself)
The action is the method invoked on recognition (the : means it has an argument)
Here we ask the UIView to actually start trying to recognize this gesture in its bounds
Let’s talk about how we implement the handler …

Gestures

- A handler for a gesture needs gesture-specific information

  So each concrete subclass provides special methods for handling that type of gesture

- For example, `UIPanGestureRecognizer` provides 3 methods

  `func translationInView(view: UIView) -> CGPoint` // cumulative since start of recognition

  `func velocityInView(view: UIView) -> CGPoint` // how fast the finger is moving (points/s)

  `func setTranslation(translation: CGPoint, inView: UIView)`

  This last one is interesting because it allows you to reset the translation so far

  By resetting the translation to zero all the time, you end up getting "incremental" translation

- The abstract superclass also provides state information

  `var state: UIGestureRecognizerState { get }`

  This sits around in `.Possible` until recognition starts

  For a discrete gesture (e.g. a Swipe), it changes to `.Recognized` (Tap is not a normal discrete)

  For a continues gesture (e.g. a Pan), it moves from `.Began` thru repeated `.Changed` to `.Ended`

  It can go to `.Failed` or `.Cancelled` too, so watch out for those!

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was "pan:" (if no colon, we would not get the gesture argument)

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was "pan:" (if no colon, we would not get the gesture argument)
We are only going to do anything when the finger moves or lifts up off the device's surface

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was “pan:” (if no colon, we would not get the gesture argument)
We are only going to do anything when the finger moves or lifts up off the device’s surface
fallthrough means “execute the code for the next case down”

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was "pan:" (if no colon, we would not get the gesture argument)
We are only going to do anything when the finger moves or lifts up off the device's surface
fallthrough means "execute the code for the next case down"
Here we get the location of the pan in the pannableView's coordinate system

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was "pan:" (if no colon, we would not get the gesture argument)
We are only going to do anything when the finger moves or lifts up off the device's surface
`fallthrough` means "execute the code for the next case down"
Here we get the location of the pan in the `pannableView`'s coordinate system
Now we do whatever we want with that information

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was "pan:" (if no colon, we would not get the gesture argument)
We are only going to do anything when the finger moves or lifts up off the device's surface
`fallthrough` means "execute the code for the next case down"
Here we get the location of the pan in the `pannableView`'s coordinate system
Now we do whatever we want with that information
By resetting the translation, the next one we get will be how much it moved since this one

Gestures

- UIPinchGestureRecognizer

```
var scale: CGFloat            // not read-only (can reset)
var velocity: CGFloat { get } // scale factor per second
```

- UIRotationGestureRecognizer

```
var rotation: CGFloat         // not read-only (can reset); in radians
var velocity: CGFloat { get } // radians per second
```

- UISwipeGestureRecognizer

  Set up the direction and number of fingers you want, then look for .Recognized

```
var direction: UISwipeGestureRecoginzerDirection // which swipes you want
var numberOfTouchesRequired: Int                 // finger count
```

- UITapGestureRecognizer

  Set up the number of taps and fingers you want, then look for .Ended

```
var numberOfTapsRequired: Int    // single tap, double tap, etc.
var numberOfTouchesRequired: Int // finger count
```

Demo

- FaceView Gestures

  Add a gesture recognizer (pinch) to the FaceView to zoom in and out (control its own scale)

  Add a gesture recognizer (pan) to the FaceView to control happiness (Model) in the Controller