CS2048: Introduction to iPhone Development

Lecture 2

Instructor: Daniel Hauagge

Announcements

- Clarification on dates
- Add date: Sep 5th
- Drop date: Sep 19th

Course Description

- Homework after each class
  - Extend app built in class
  - 1 weeks to finish, ~~grade is pass/fail~~ graded
  - ~~Must complete all assignments to pass~~
  - 1 late day, you can use it whenever you want

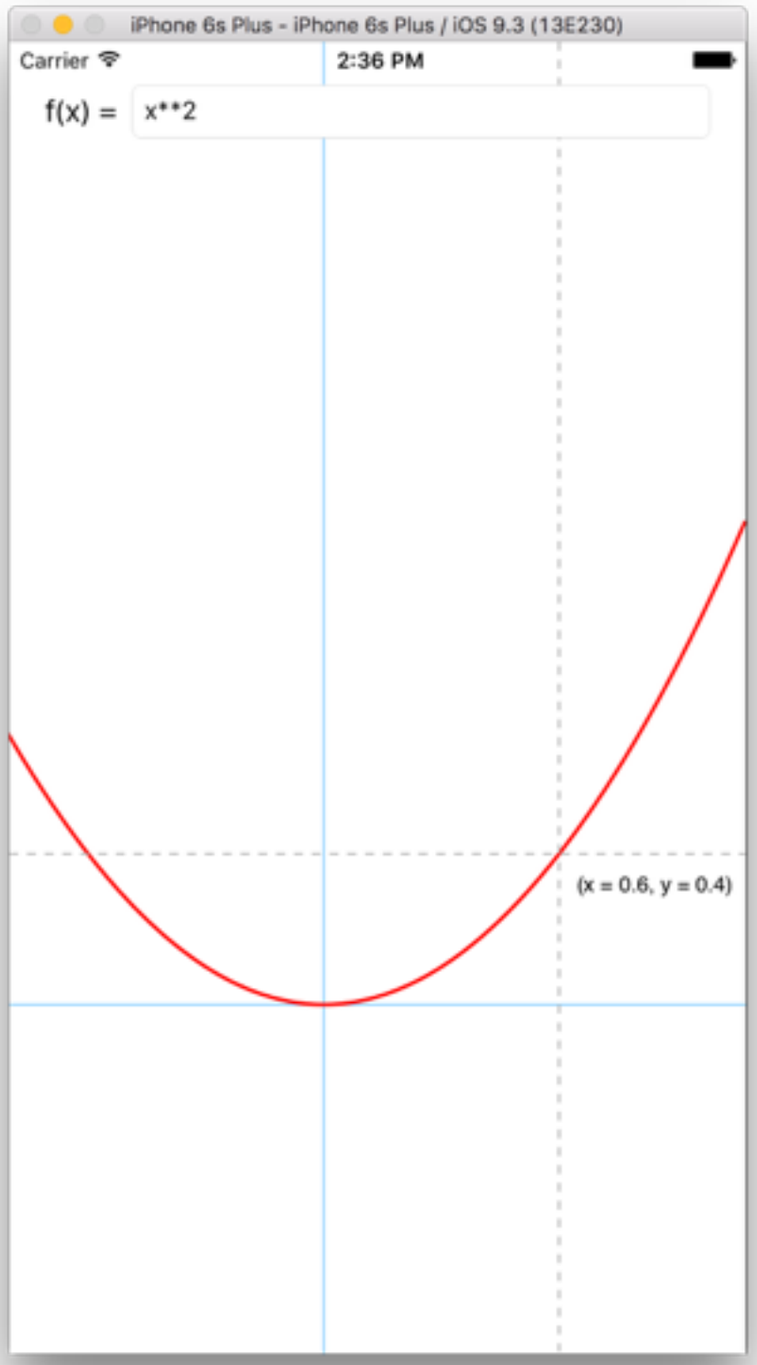
Announcements

- Final grade
  - 3 homework assignments
  - 1 final project: 4 weeks, open ended
- Grade: $\frac{\sum_{i=1}^{3} H_i + 2FP}{5}$
- Passing grade ≥ 50

Today

- Introduction to MVC
- Custom views
- Drawing with UIBezierPath
- Gesture Recognizers

App


iPhone 6s Plus - iPhone 6s Plus / iOS 9.3 (13E230)
Carrier
2:36 PM
f(x) =
x**2
(x = 0.6, y = 0.4)

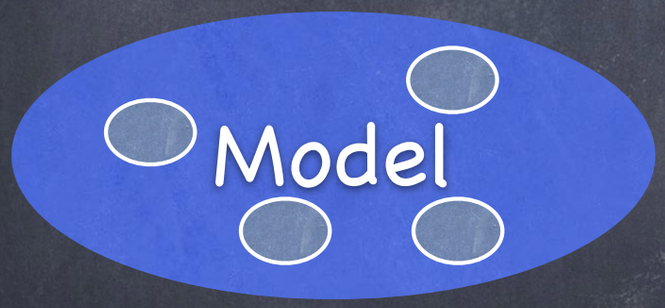
MVC

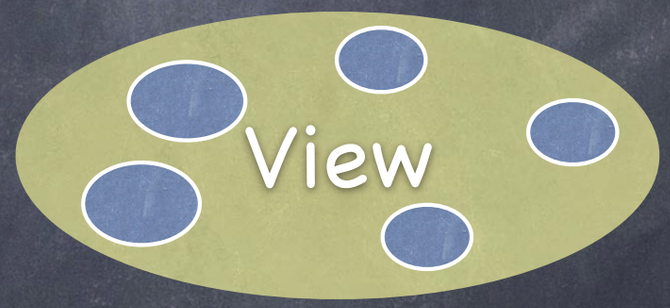
Model-View-Controller

MVC

- Software architectural pattern used throughout iOS
- Reduces coupling between
  - Display logic
  - Model (data)

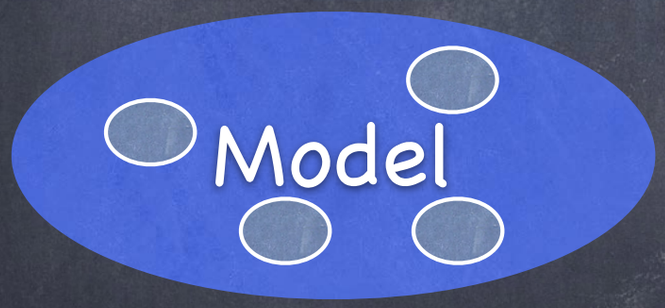
MVC

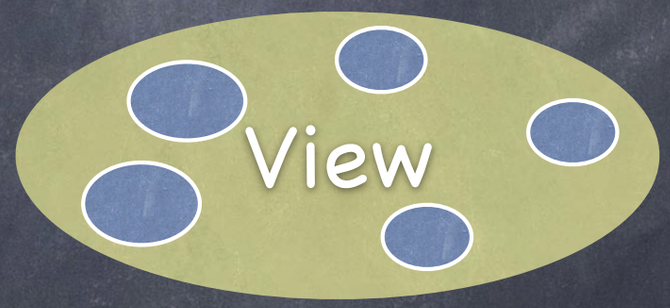




Divide objects in your program into 3 "camps."

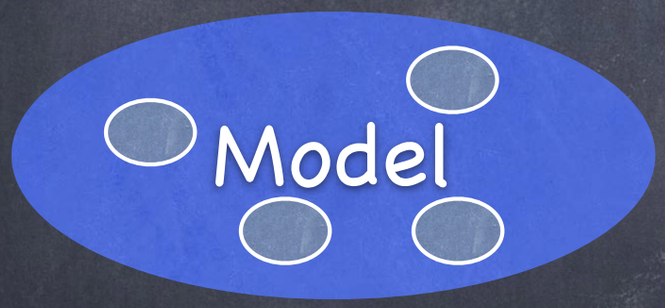
MVC

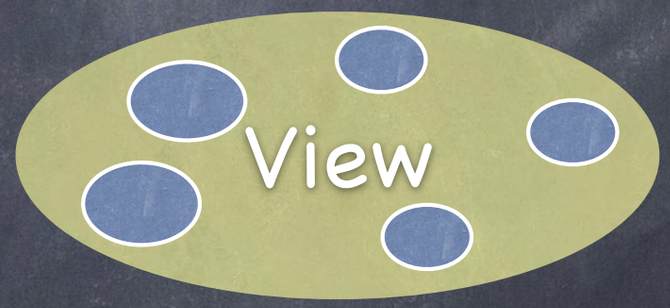




Model = What your application is (but not how it is displayed)

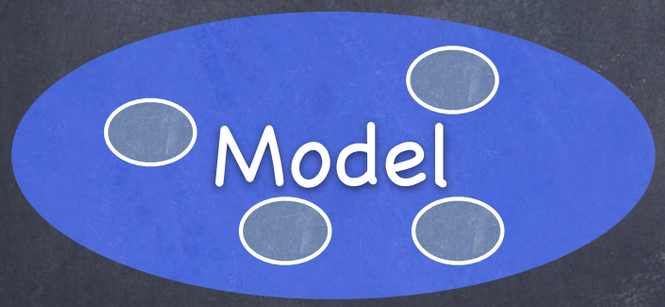
MVC

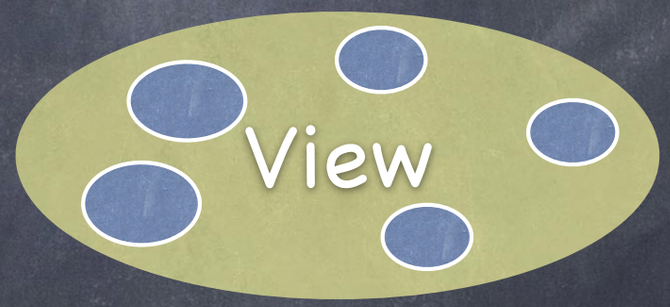




Controller = How your Model is presented to the user (UI logic)

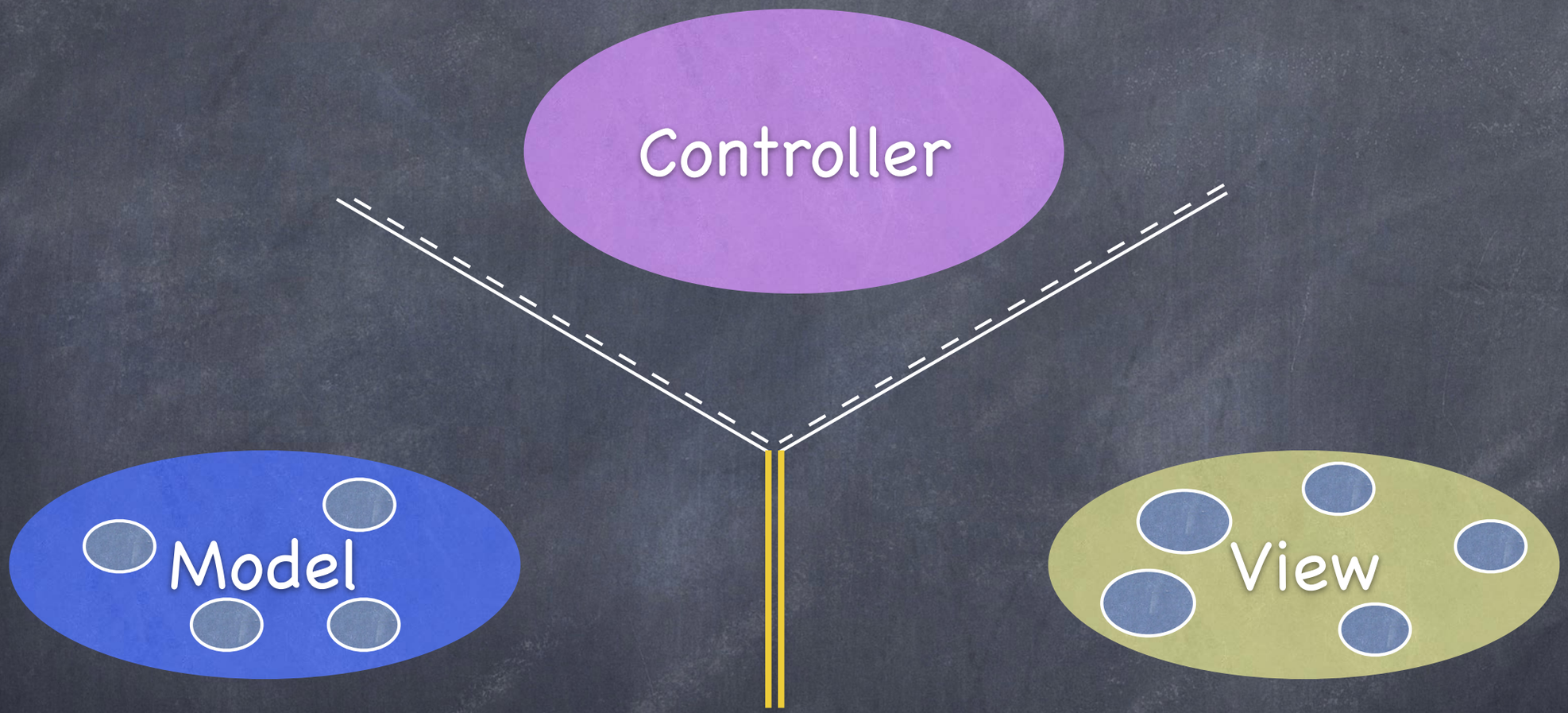
MVC





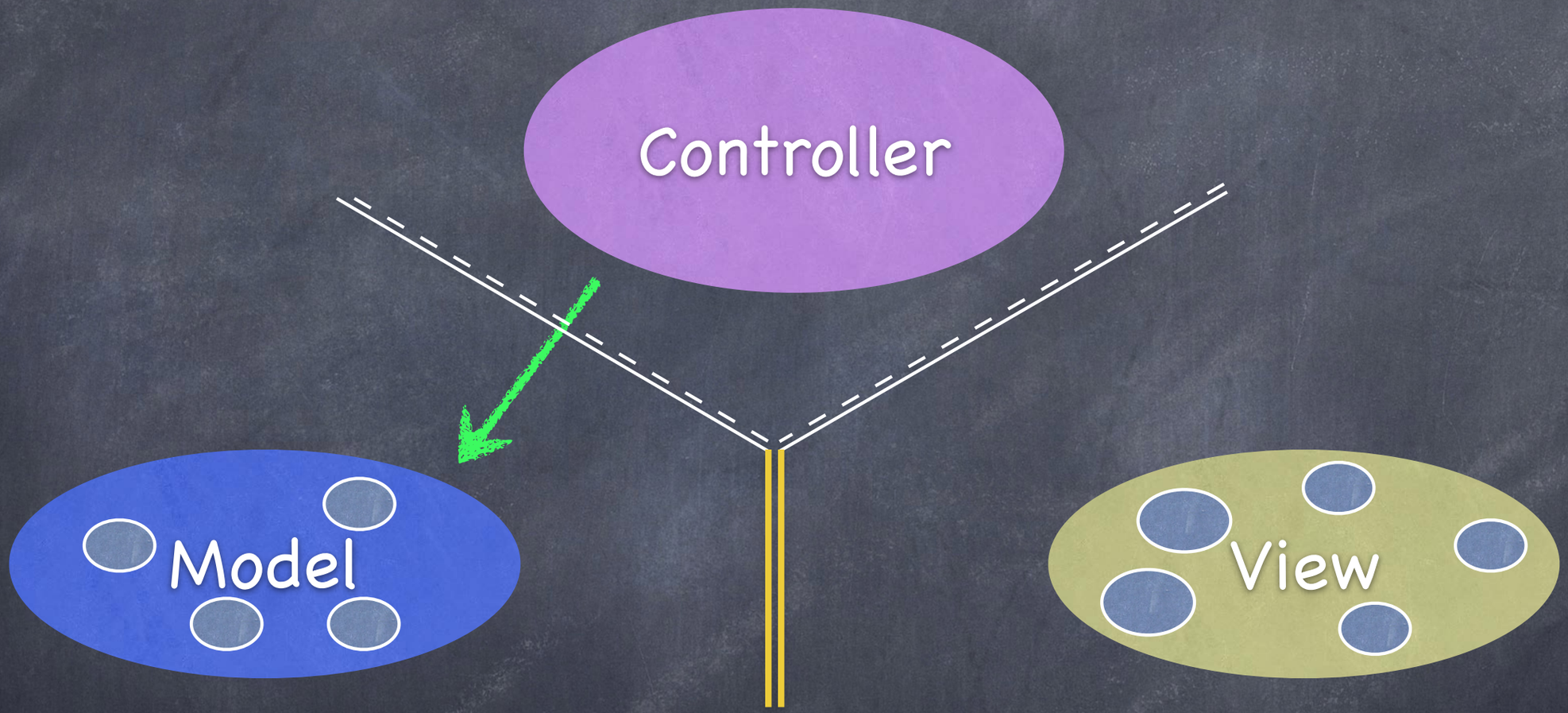
View = Your Controller's minions

MVC



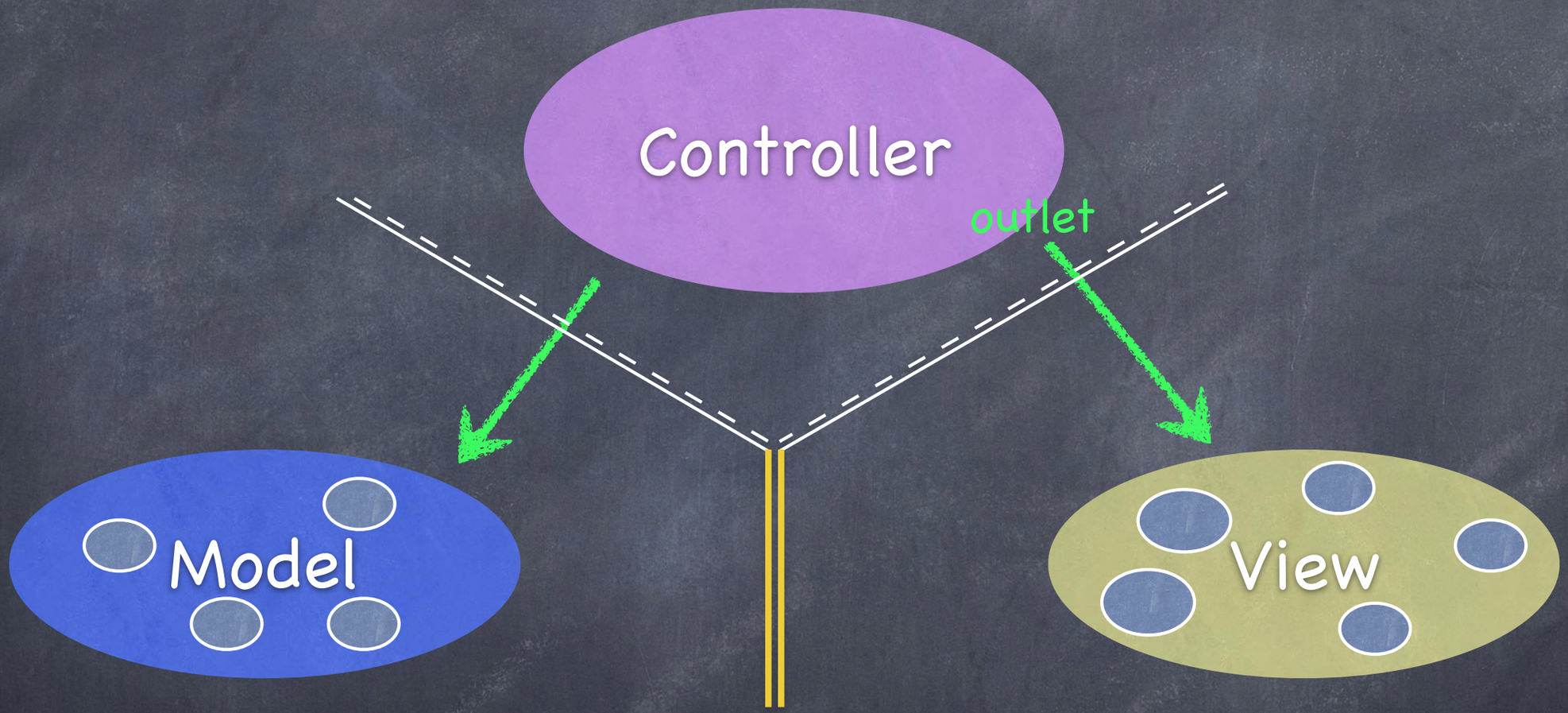
It's all about managing communication between camps

MVC



Controllers can always talk directly to their Model.

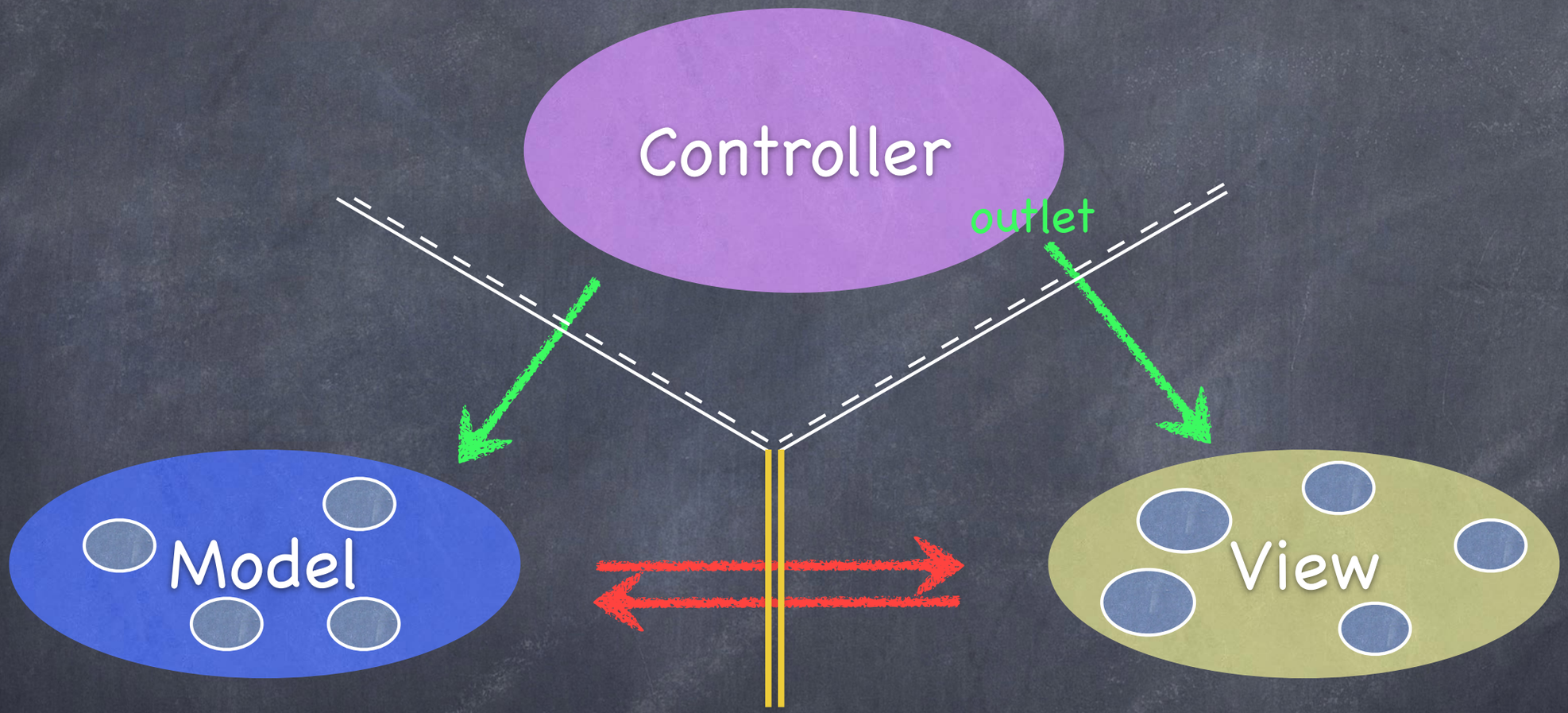
MVC



Controllers can also talk directly to their View.

CS193p
Spring 2016

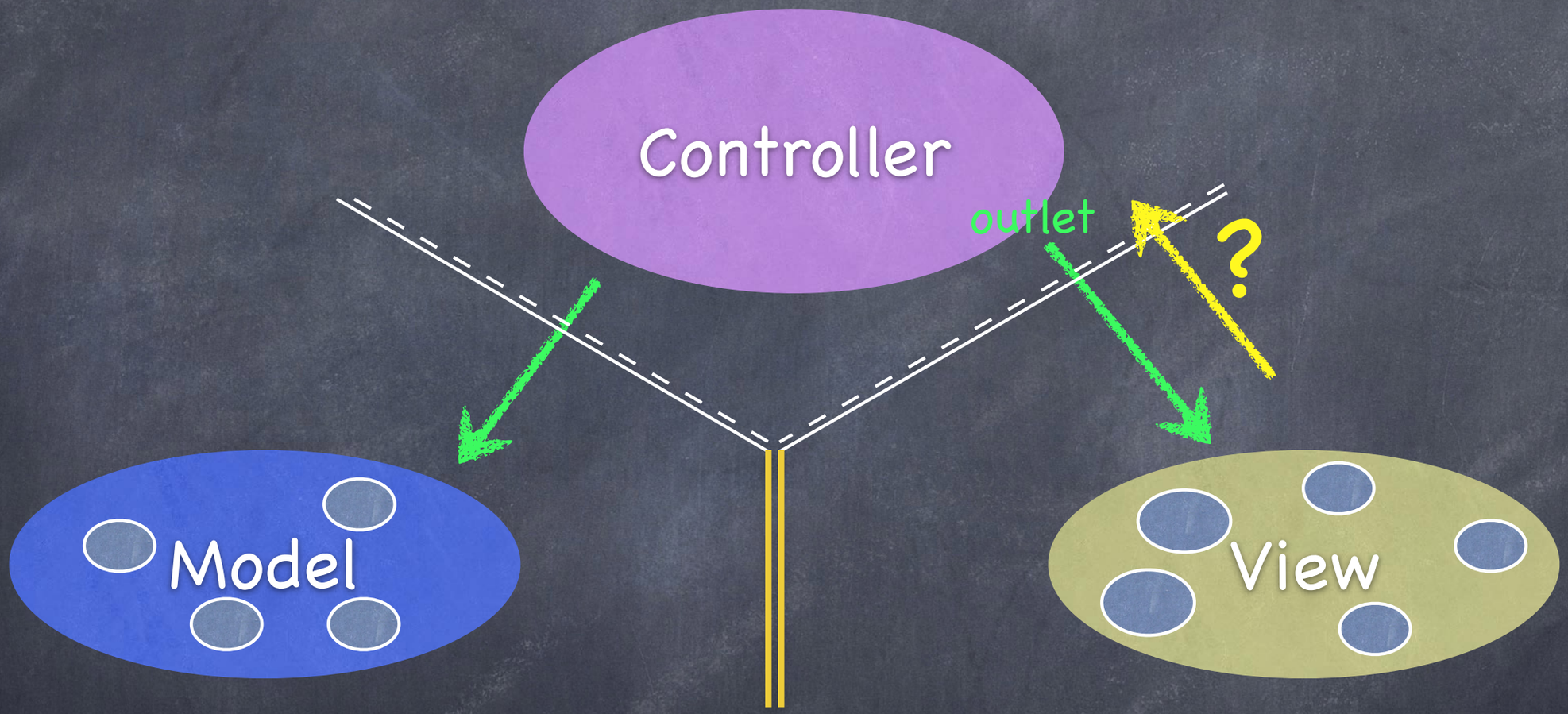
MVC



The Model and View should never speak to each other.

CS193p
Spring 2016

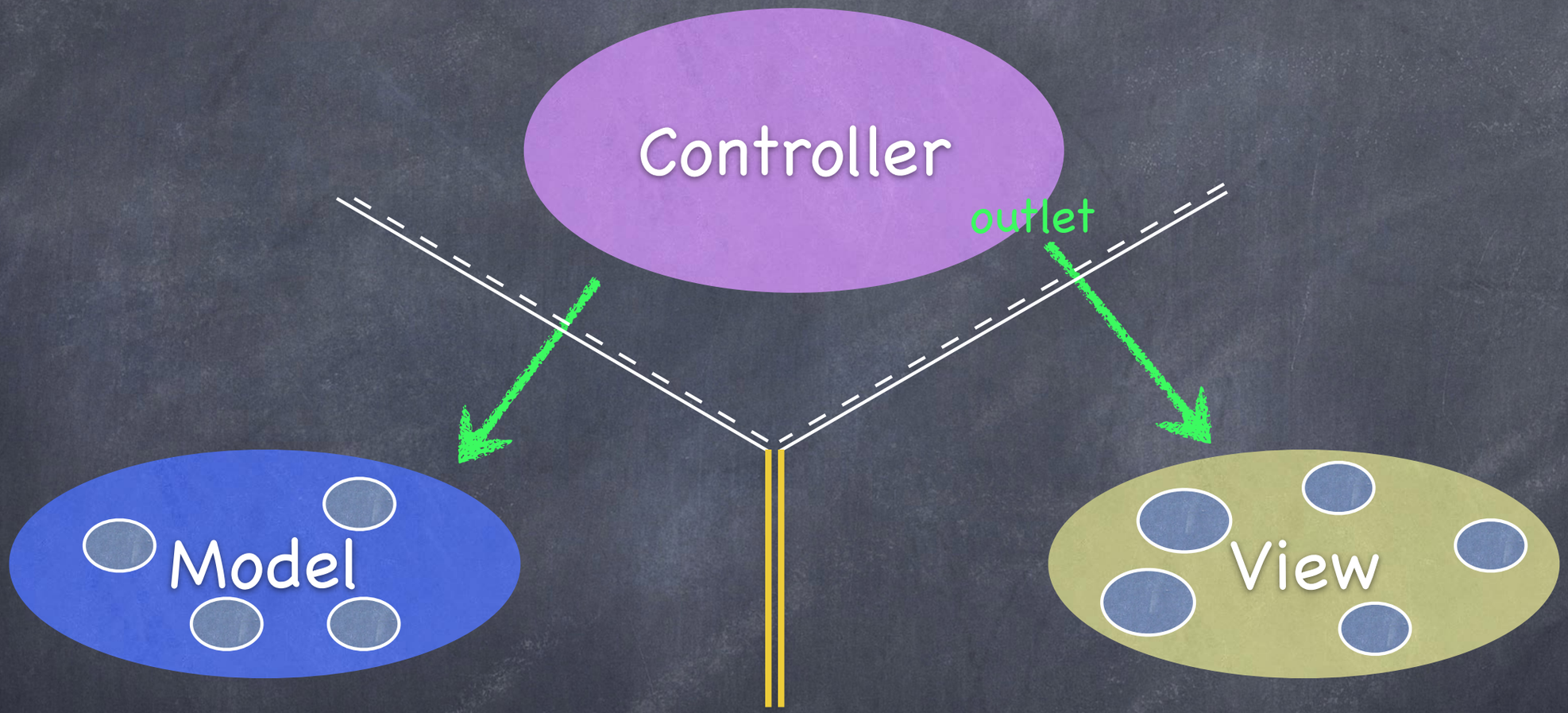
MVC



Can the View speak to its Controller?

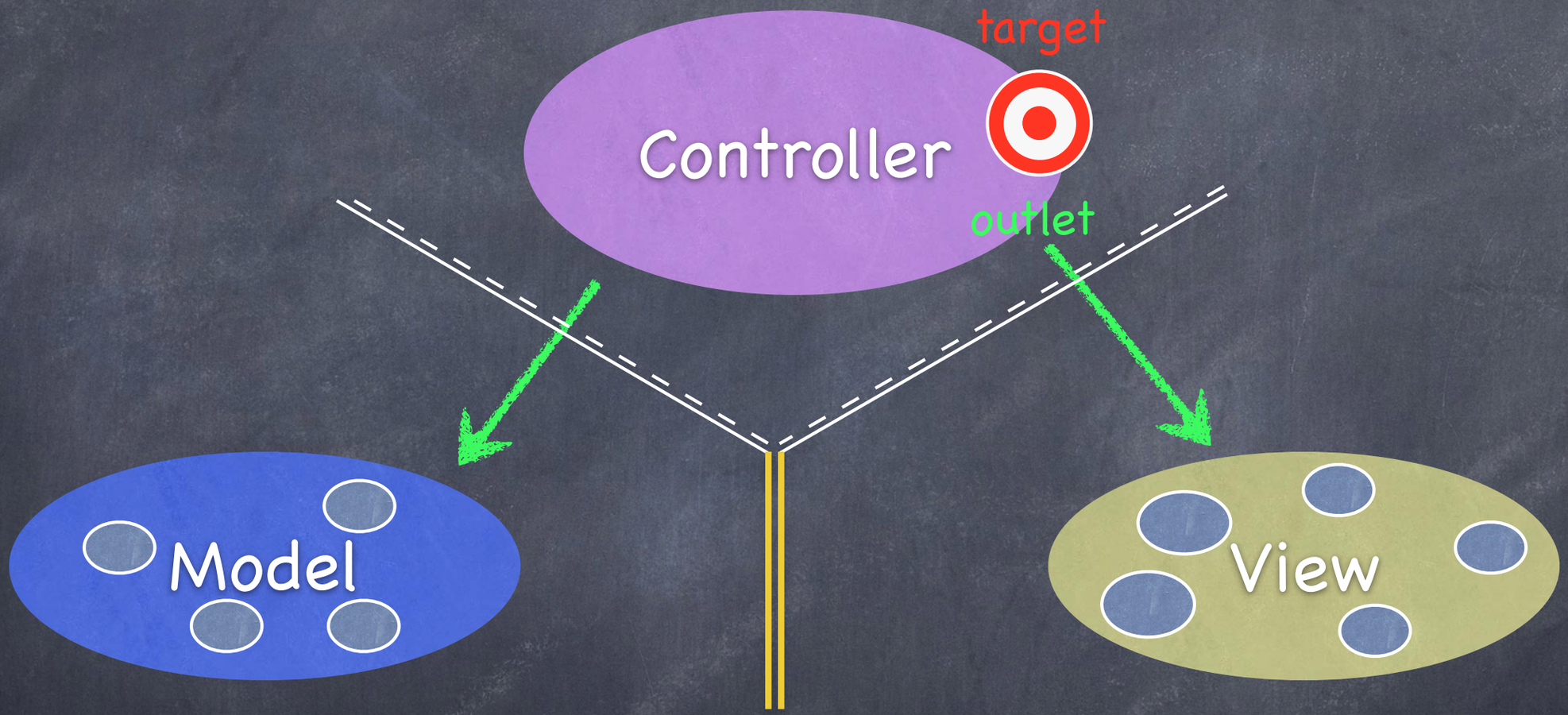
CS193p
Spring 2016

MVC



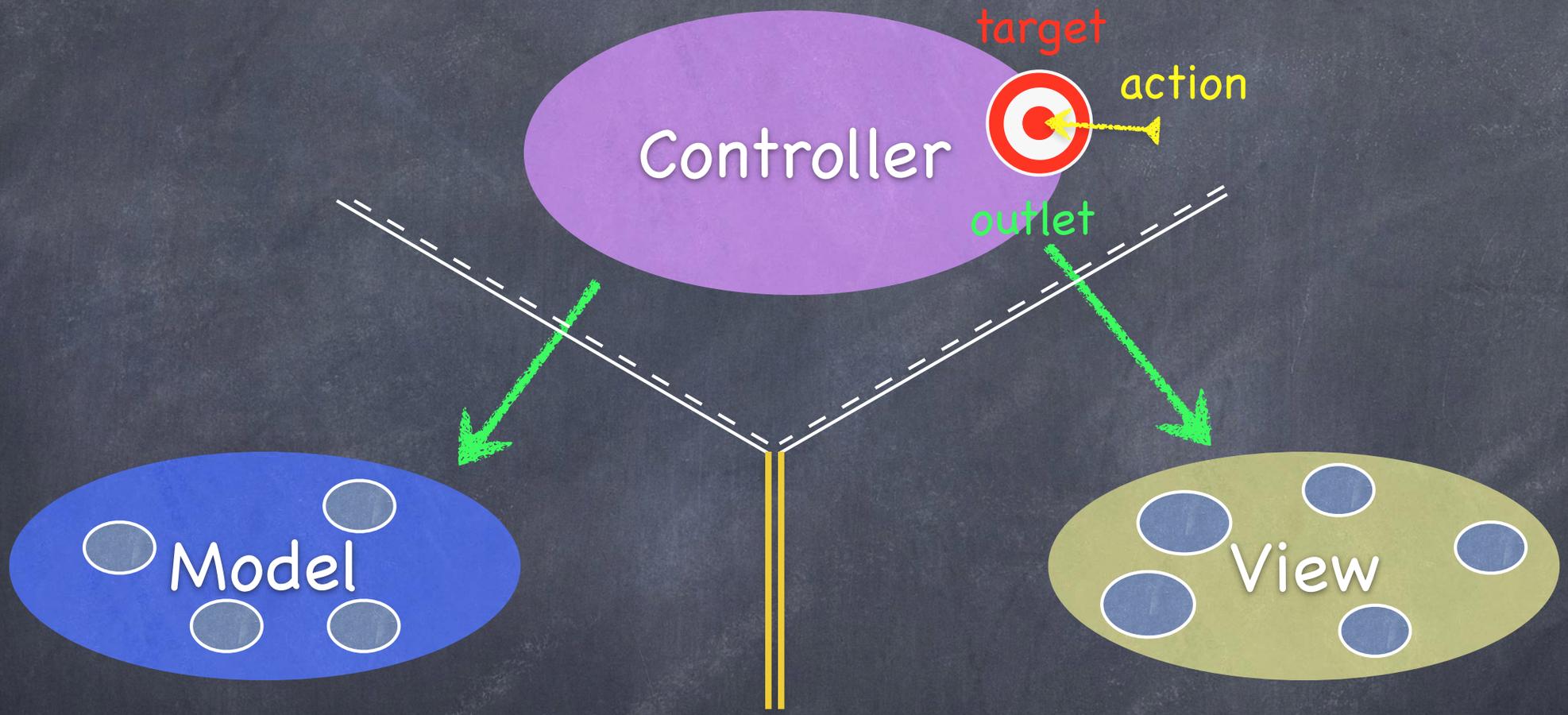
Sort of. Communication is "blind" and structured.

MVC



The Controller can drop a target on itself.

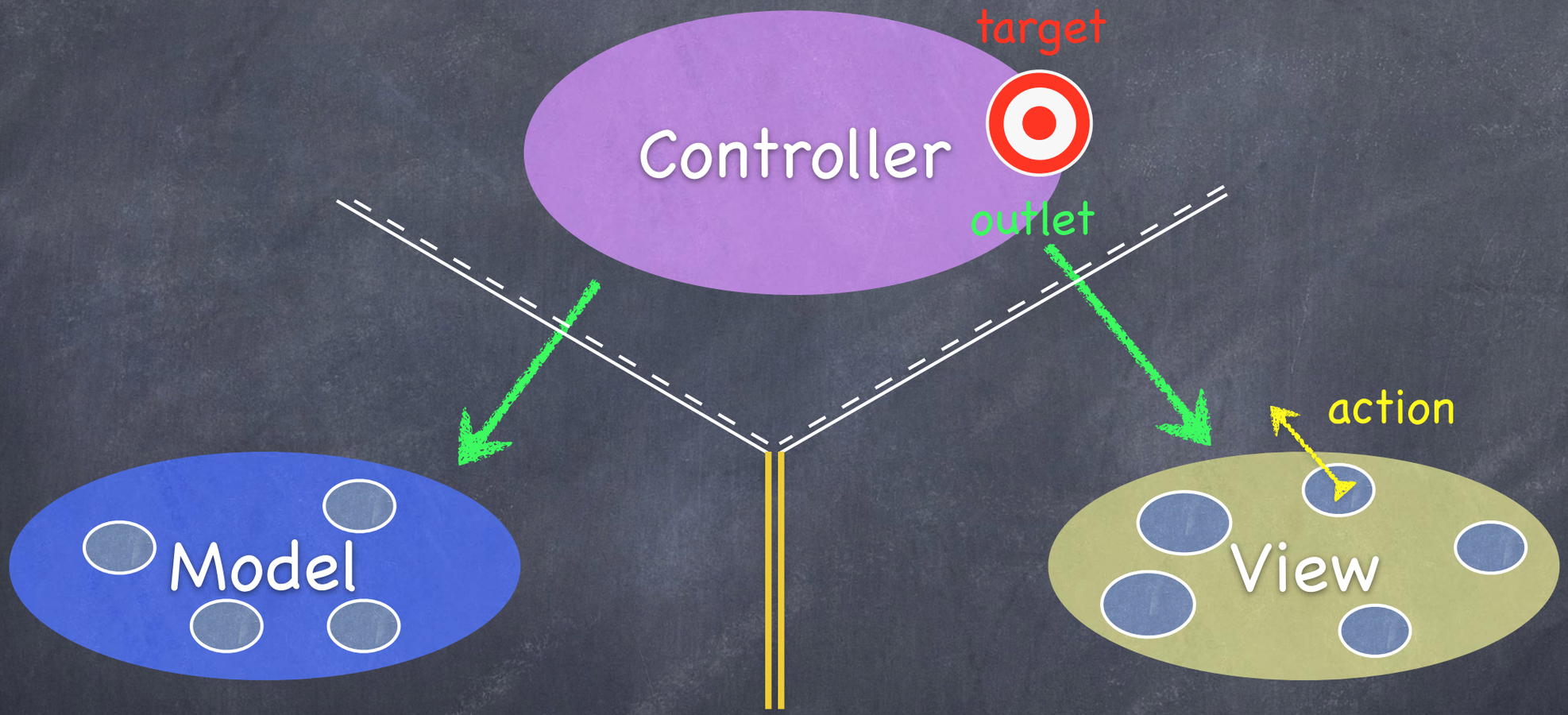
MVC



Then hand out an action to the View.

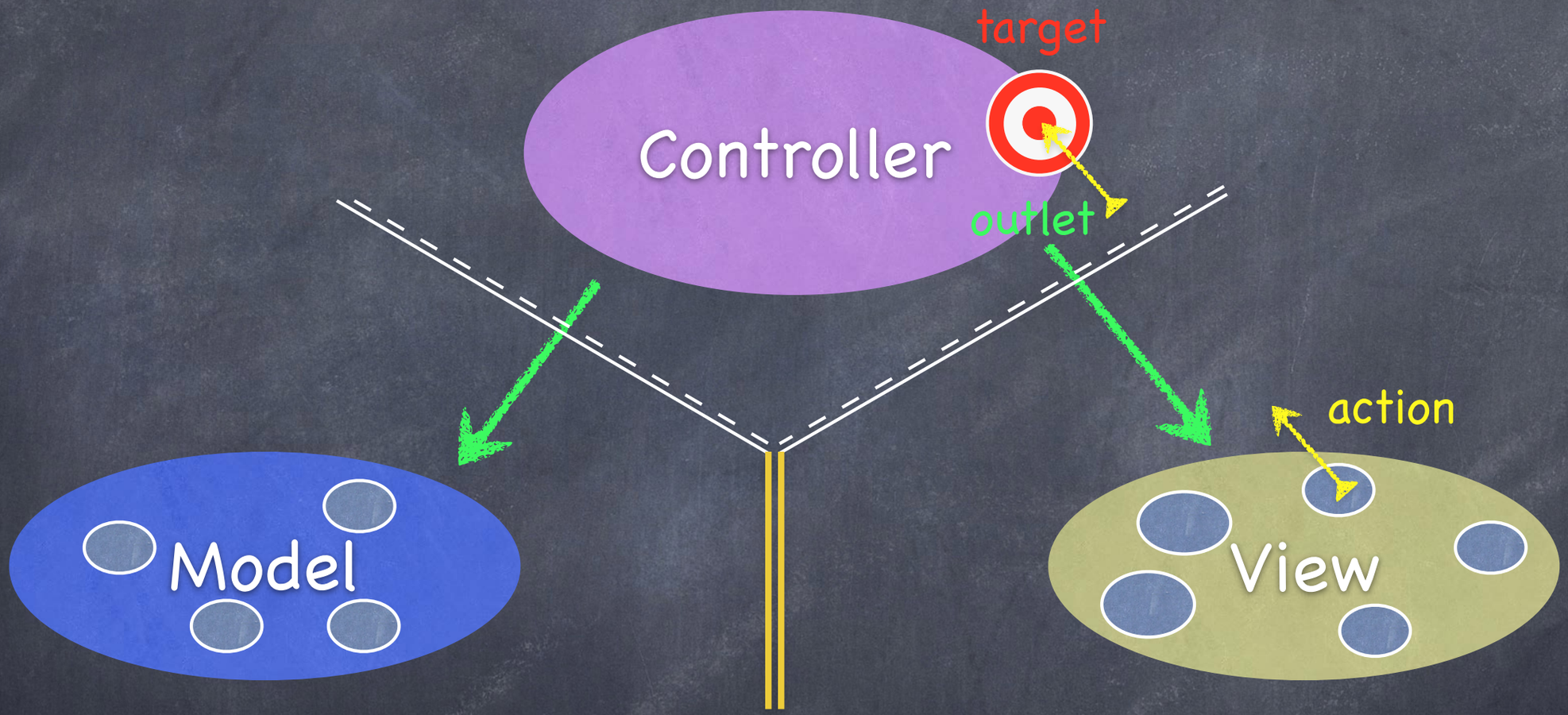
CS193p
Spring 2016

MVC



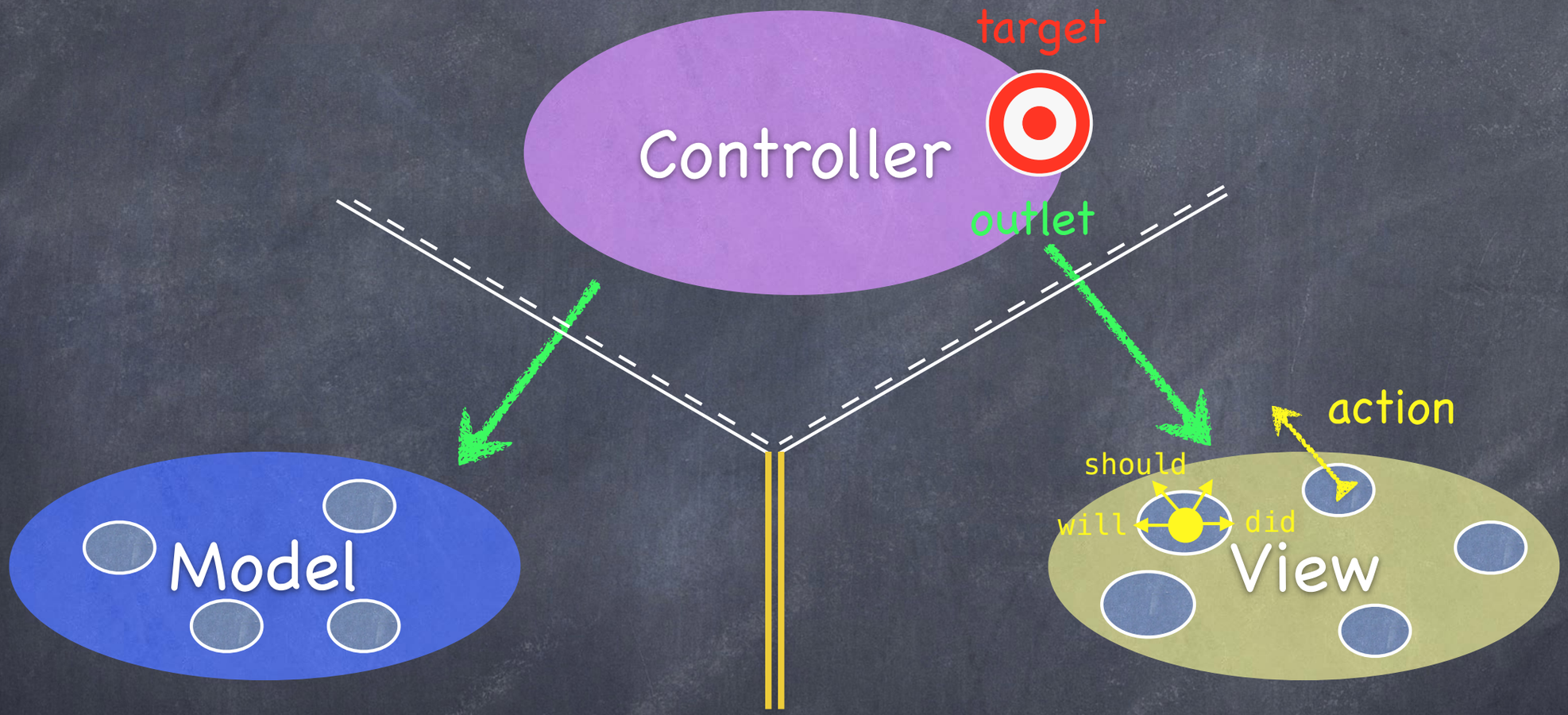
Then hand out an action to the View.

MVC



The View sends the action when things happen in the UI.

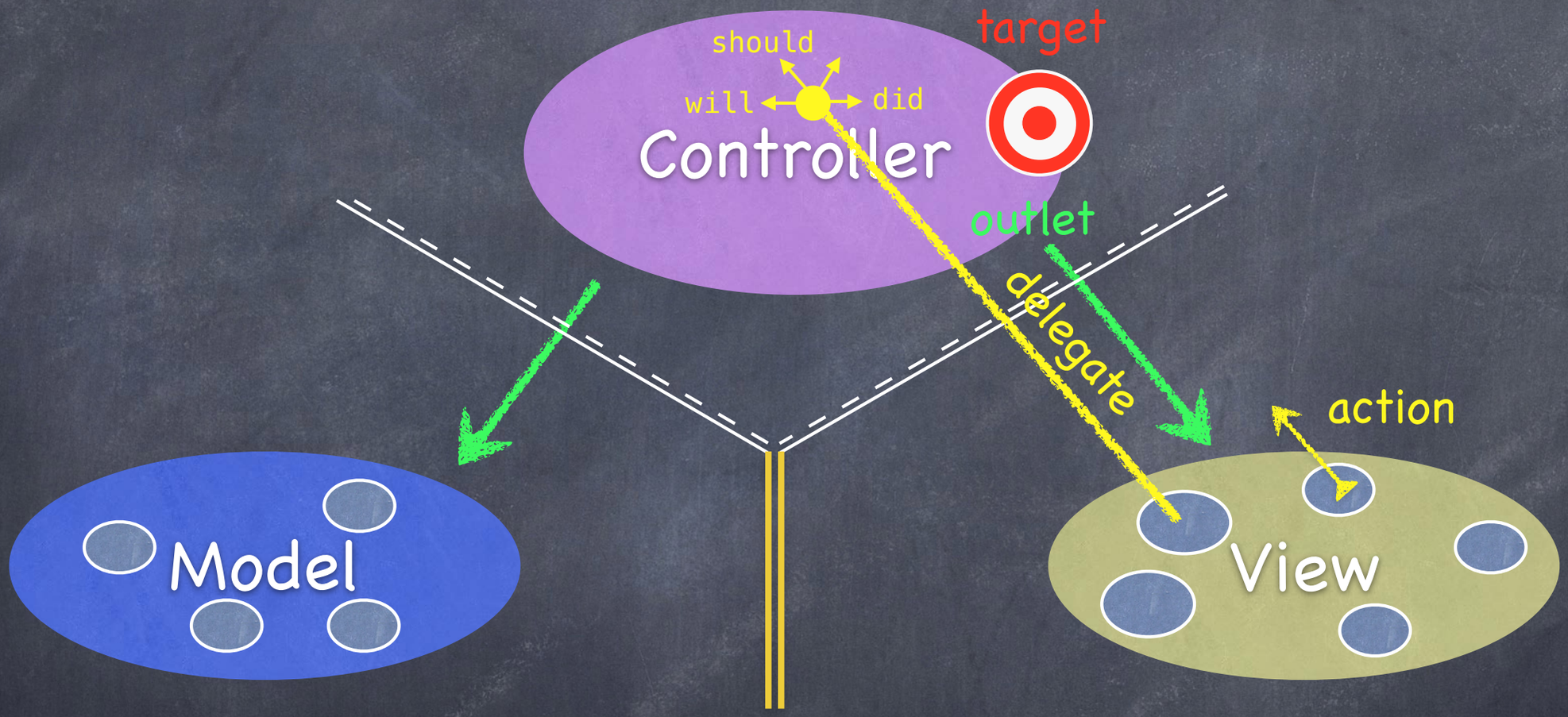
MVC



Sometimes the View needs to synchronize with the Controller.

CS193p
Spring 2016

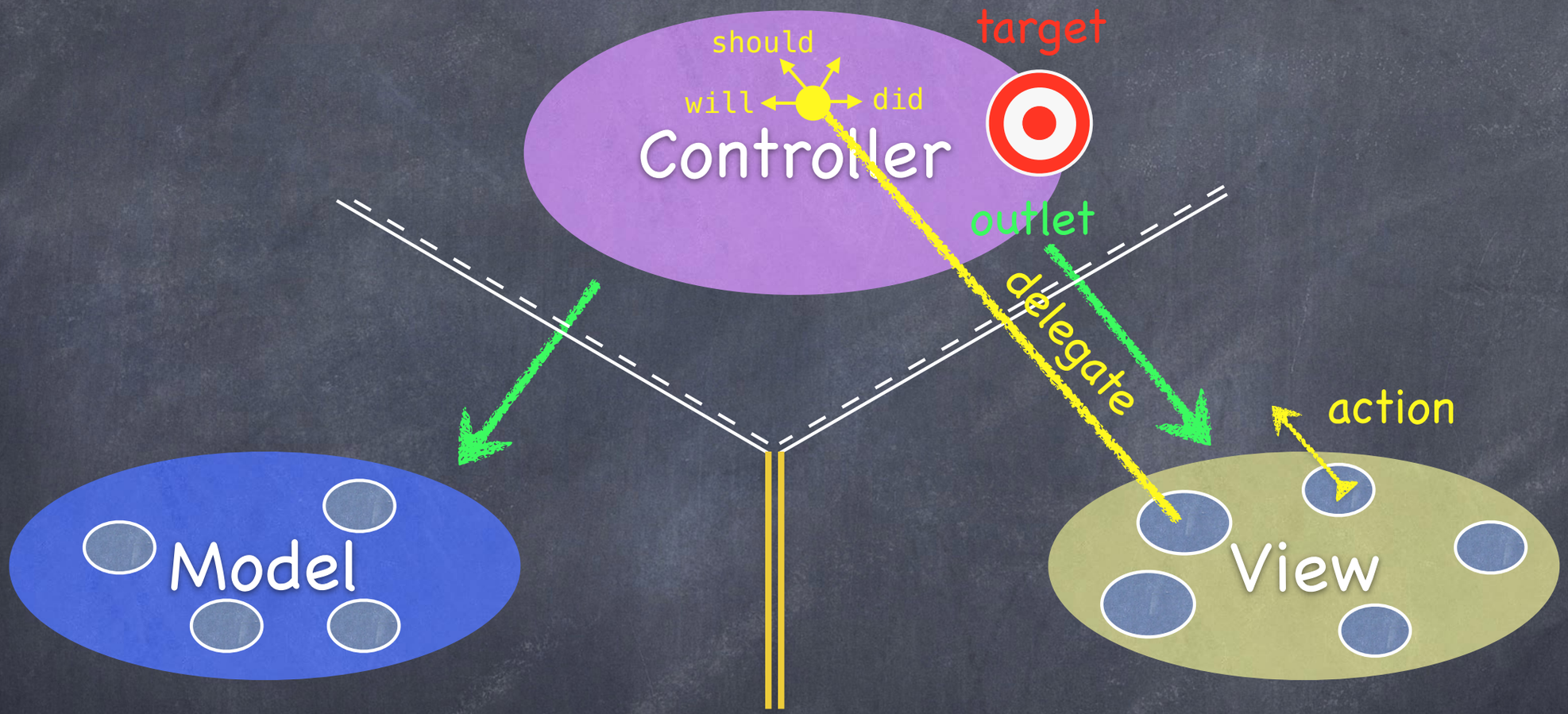
MVC



The Controller sets itself as the View's delegate.

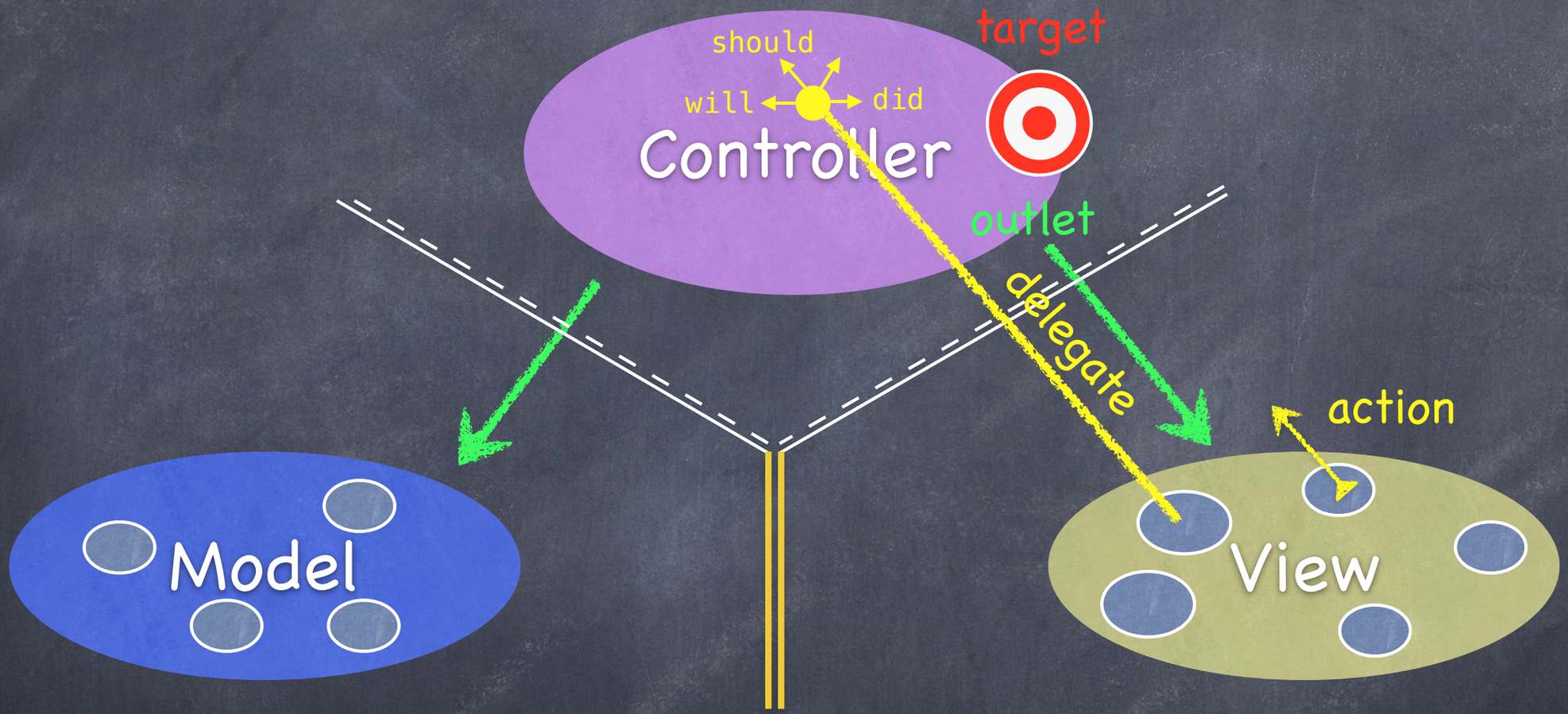
CS193p
Spring 2016

MVC



The delegate is set via a protocol (i.e. it's "blind" to class).

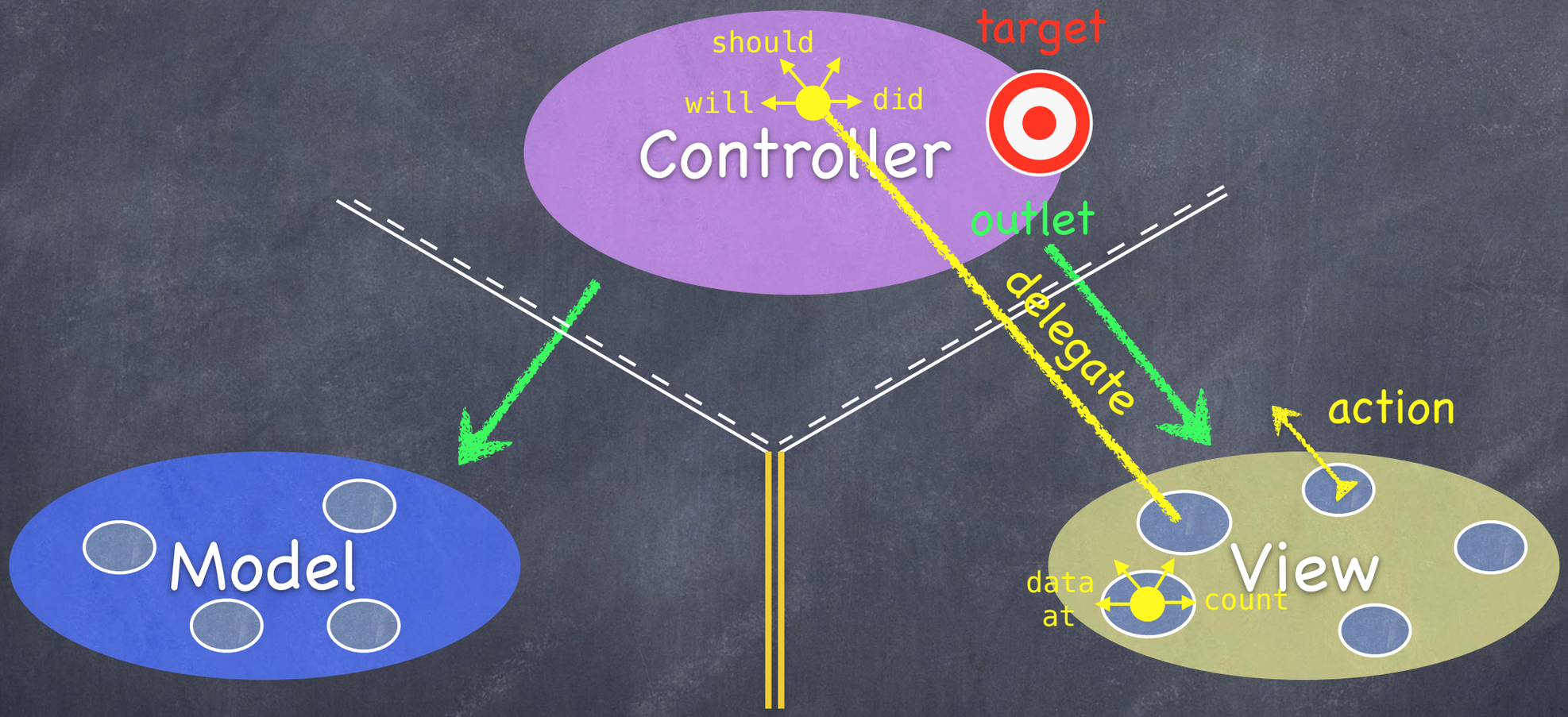
MVC



Views do not own the data they display.

CS193p
Spring 2016

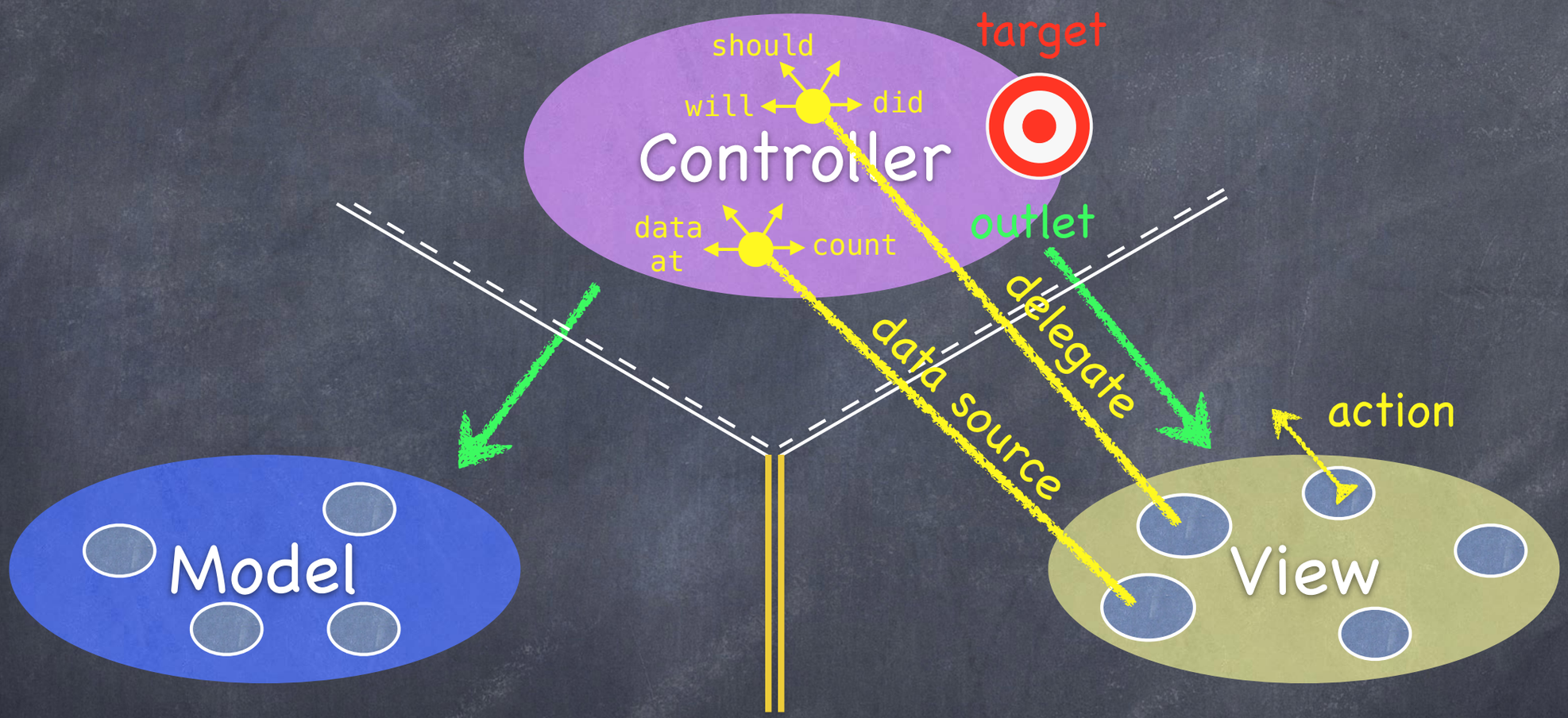
MVC



So, if needed, they have a protocol to acquire it.

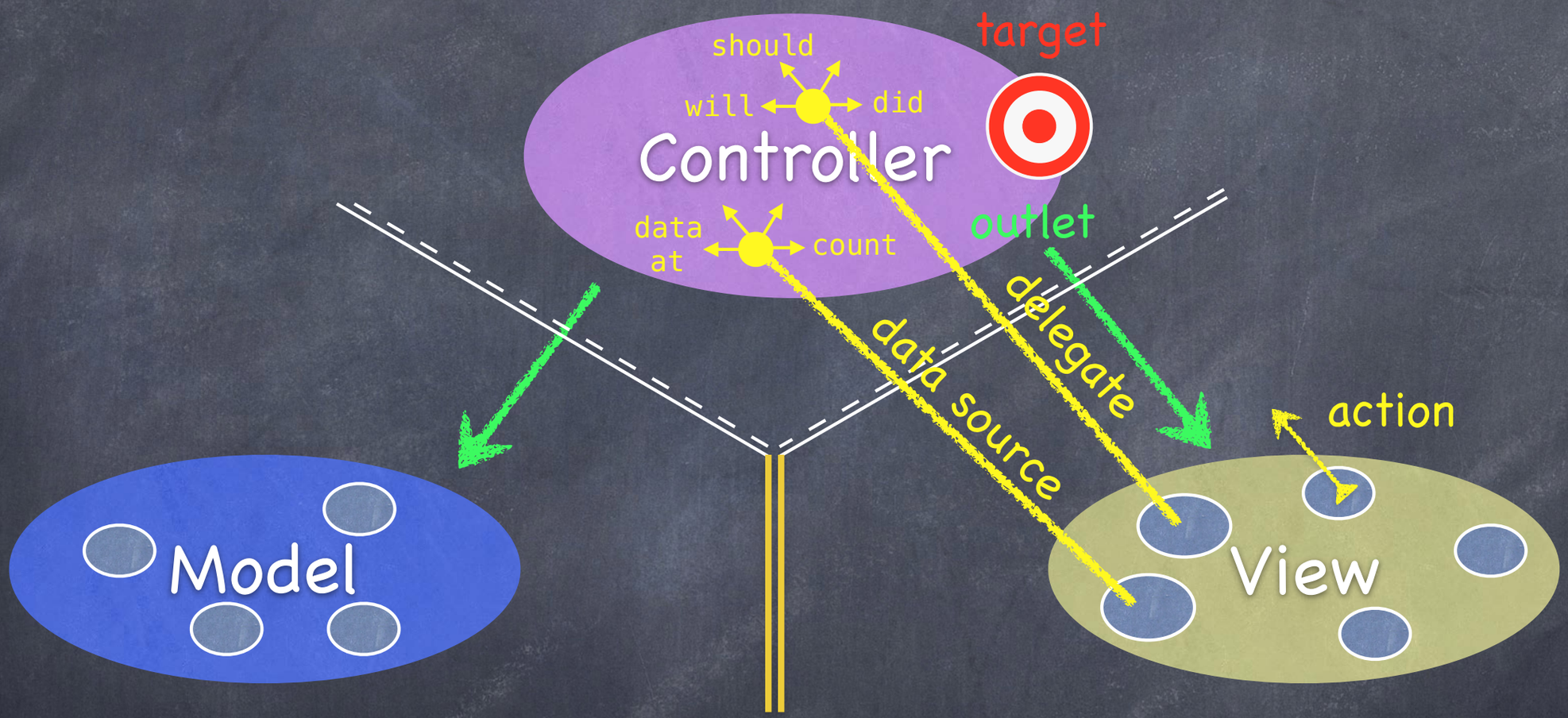
CS193p
Spring 2016

MVC



Controllers are almost always that data source (not Model!).

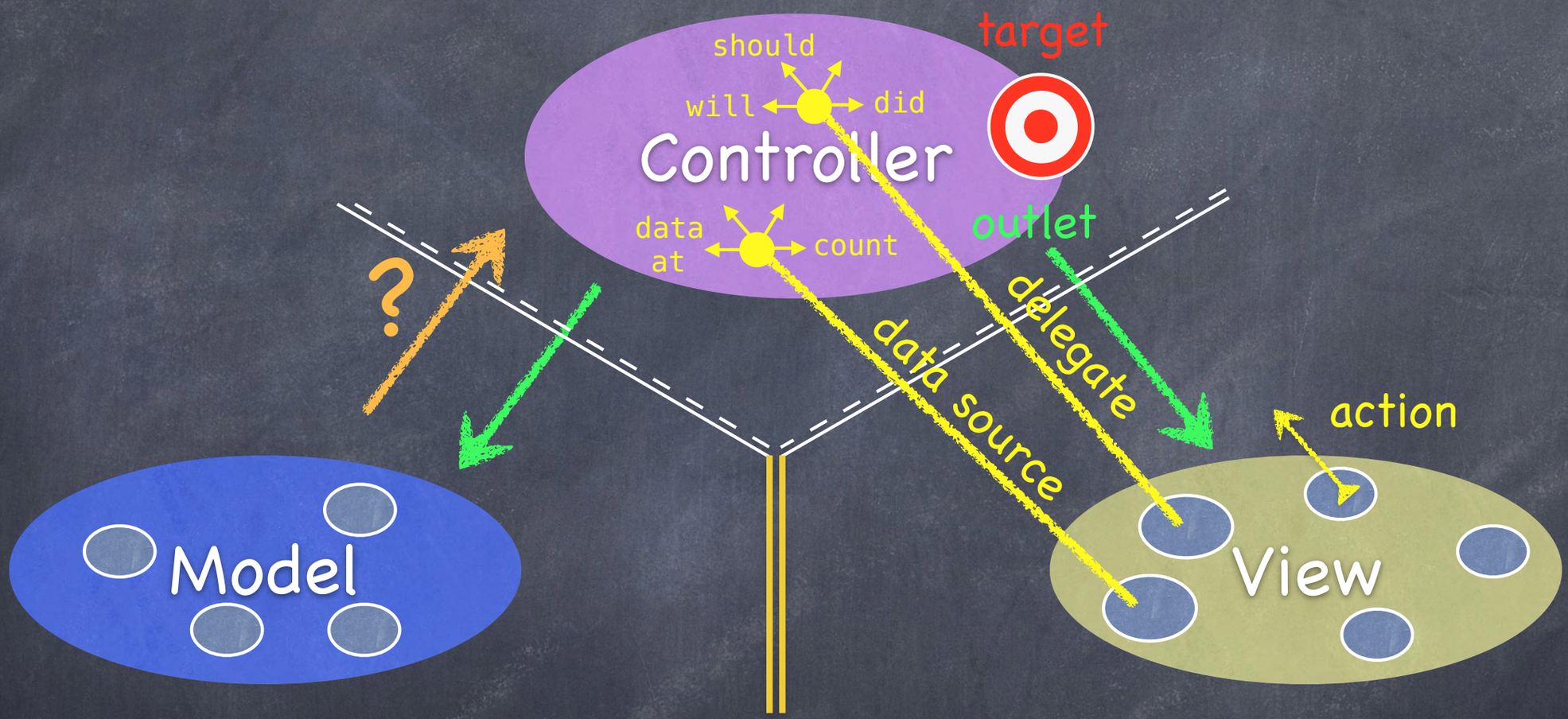
MVC



Controllers interpret/format Model information for the View.

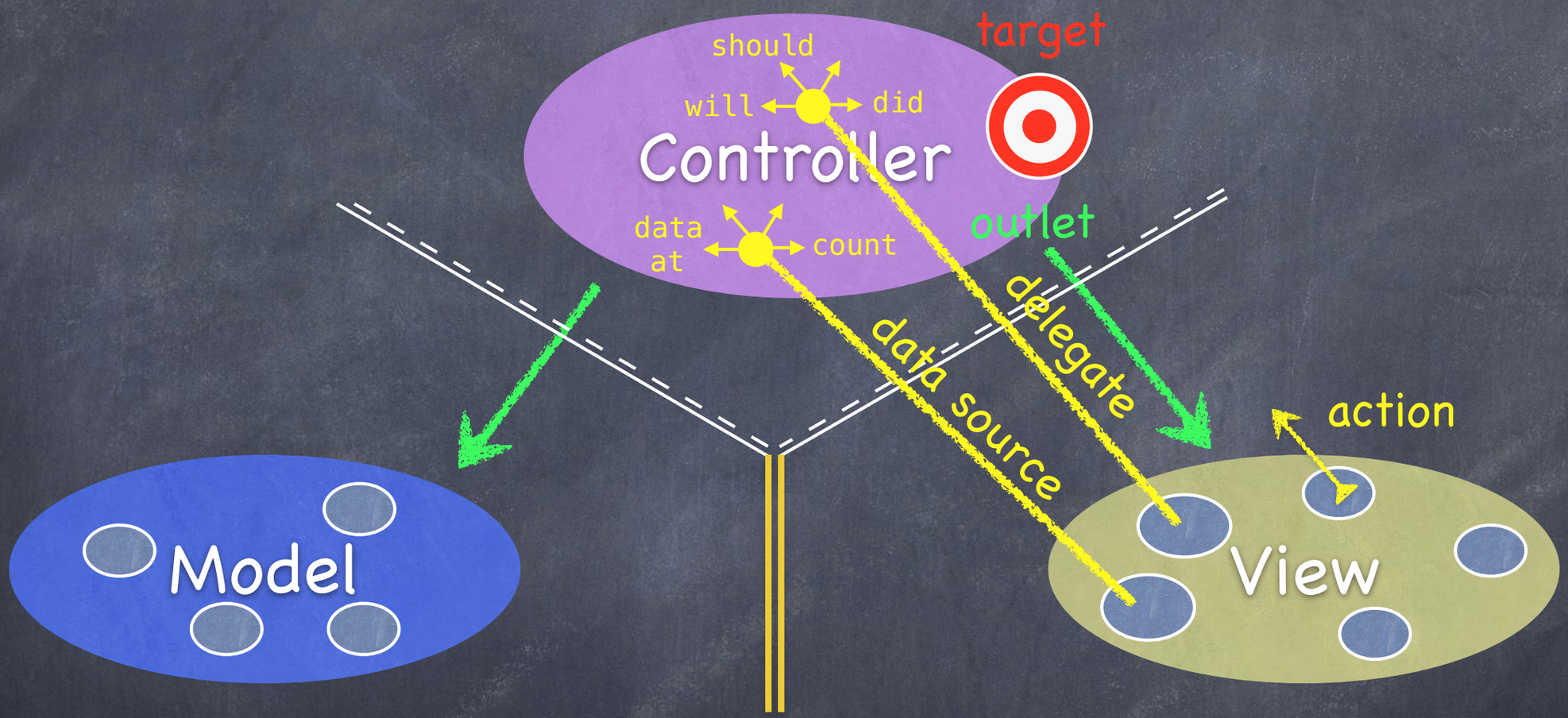
CS193p
Spring 2016

MVC



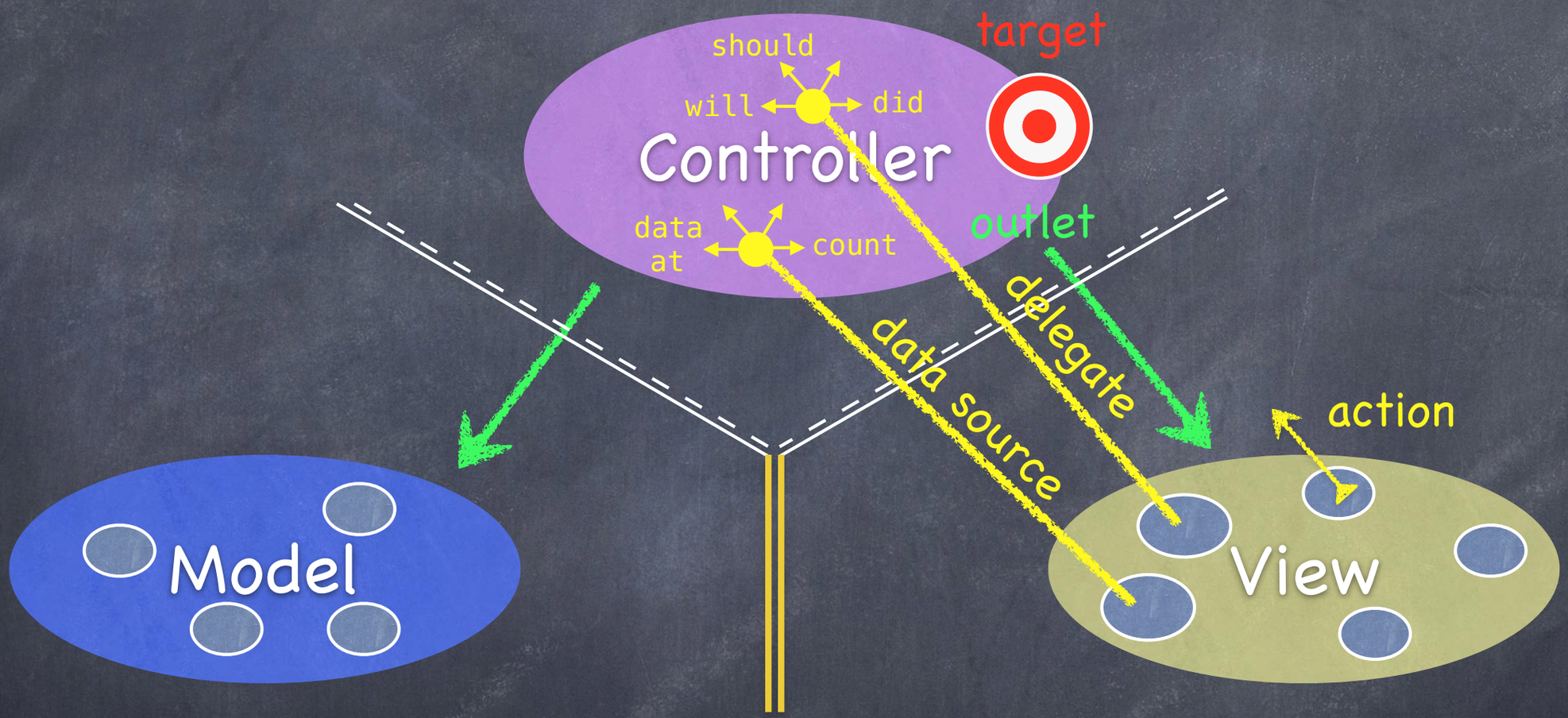
Can the Model talk directly to the Controller?

MVC



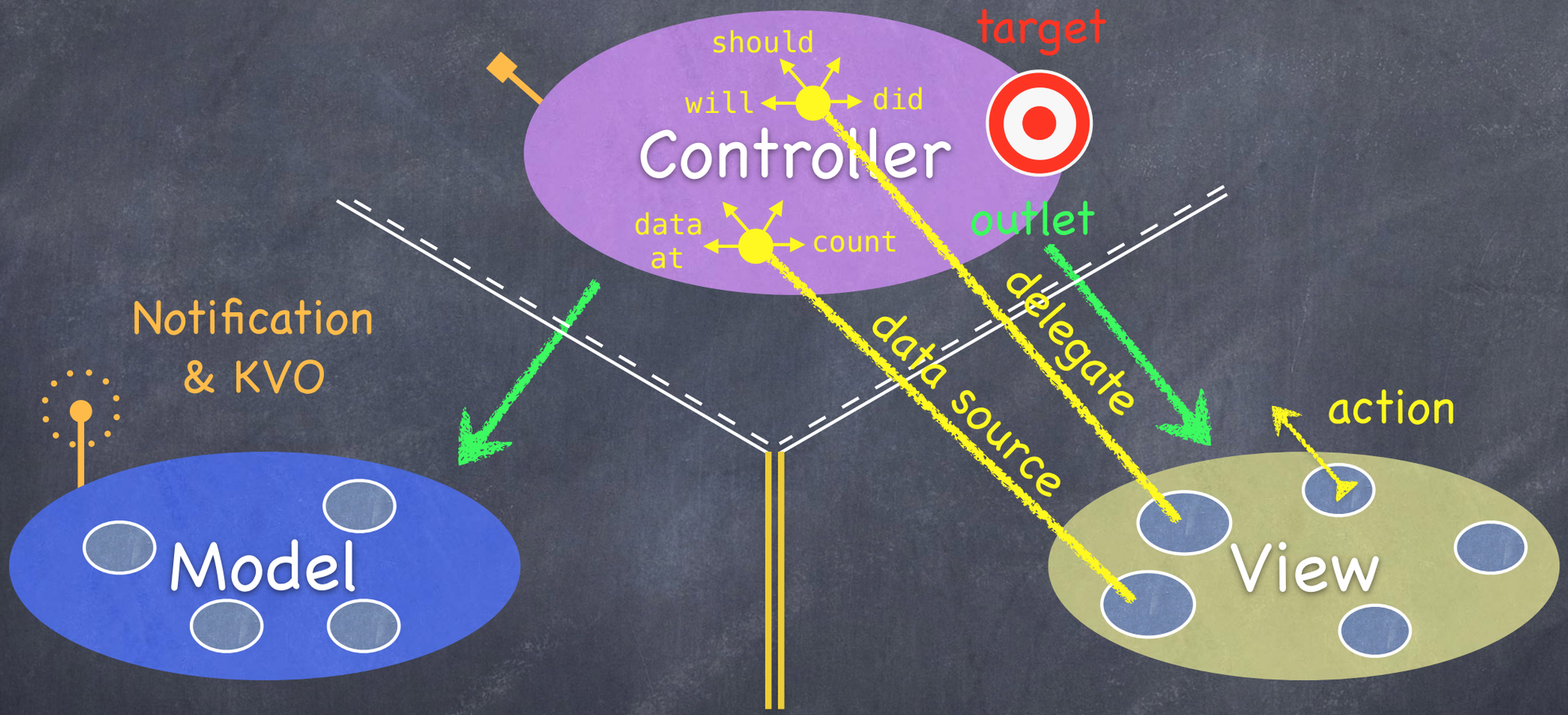
No. The Model is (should be) UI independent.

MVC



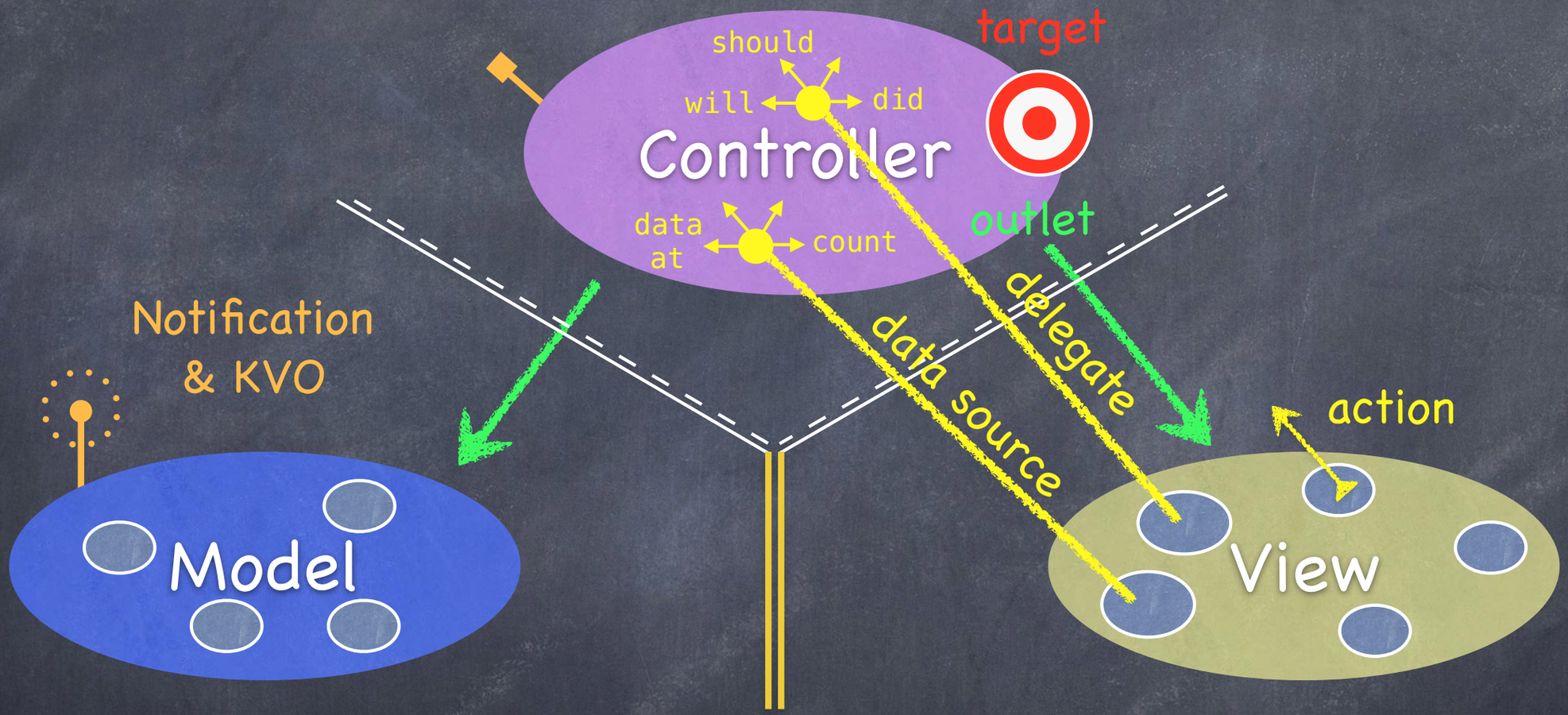
So what if the Model has information to update or something?

MVC



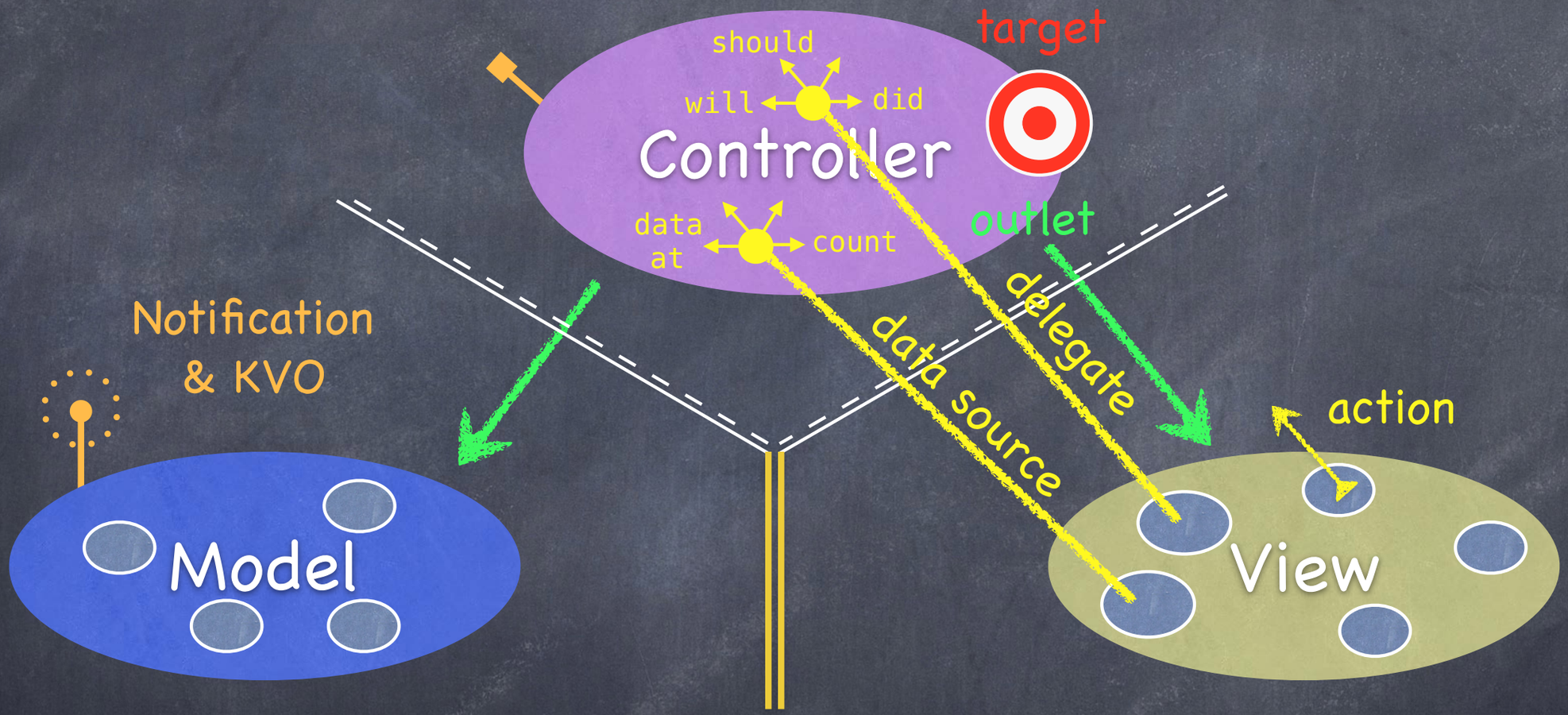
It uses a "radio station"-like broadcast mechanism.

MVC



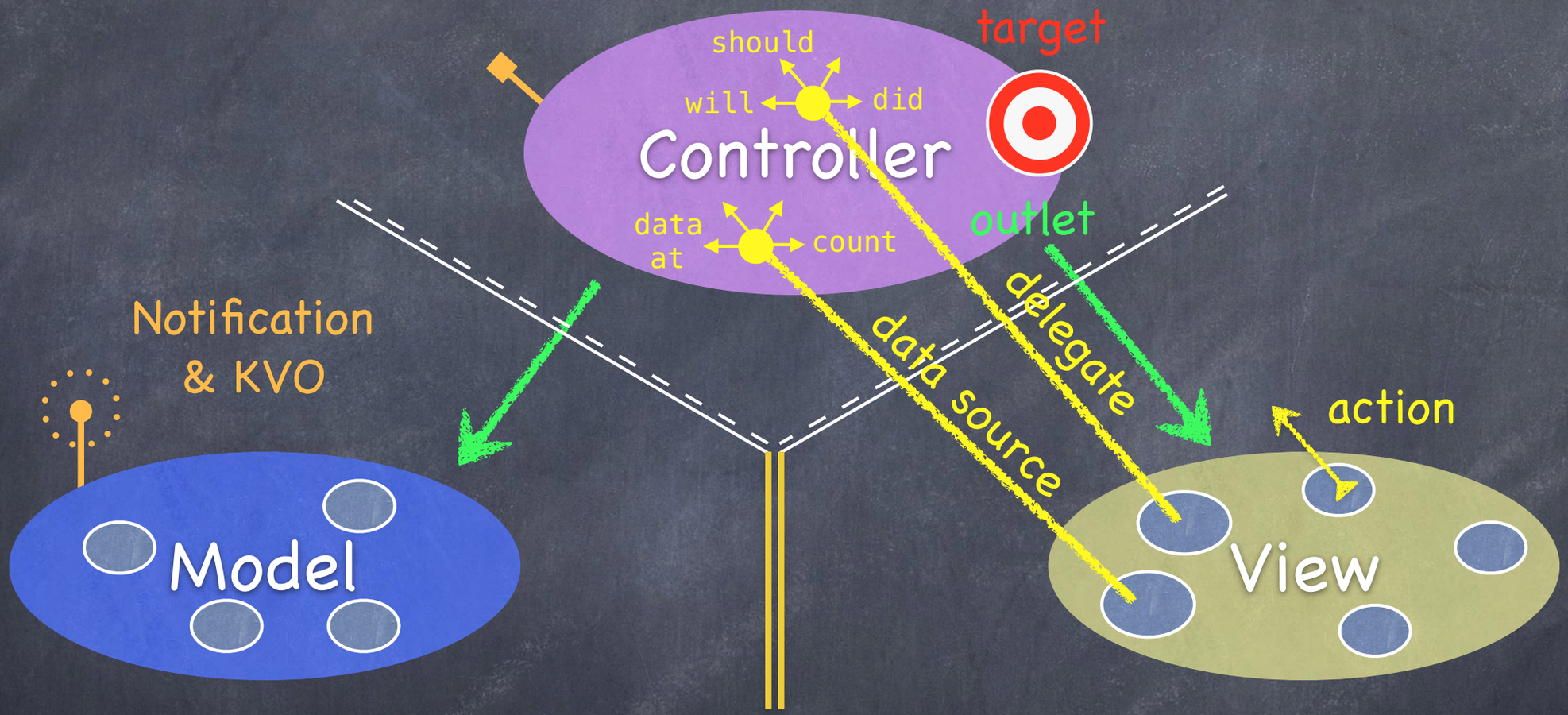
Controllers (or other Model) "tune in" to interesting stuff.

MVC



A View might "tune in," but probably not to a Model's "station."

MVC



Now combine MVC groups to make complicated programs ...

{ Coding Session }

UITextView & Delegation

UITextField

- Delegate
  - Determining whether the user should be allowed to edit the text field’s contents.
  - Validating the text entered by the user.
  - Responding to taps in the keyboard’s return button.
  - Forwarding the user-entered text to other parts of your app.

Views & Drawing

Views

- A view (i.e. `UIView` subclass) represents a rectangular area
  - Defines a coordinate space
  - For drawing
  - And for handling touch events
- Hierarchical
  - A view has only one superview … `var superview: UIView?`
  - But it can have many (or zero) subviews … `var subviews: [UIView]`
  - The order in the `subviews` array matters: those later in the array are on top of those earlier
  - A view can clip its `subviews` to its own `bounds` or not (the default is not to)
- `UIWindow`
  - The `UIView` at the very, very top of the view hierarchy (even includes status bar)
  - Usually only one `UIWindow` in an entire iOS application … it's all about views, not windows

Views

- The hierarchy is most often constructed in Xcode graphically
  Even custom views are usually added to the view hierarchy using Xcode
- But it can be done in code as well

```
addSubview(aView: UIView) // sent to aView's (soon to be) superview
removeFromSuperview()     // this is sent to the view you want to remove (not its superview)
```

- Where does the view hierarchy start?
  The top of the (useable) view hierarchy is the Controller's `var view: UIView`.
  This simple property is a very important thing to understand!
  This view is the one whose `bounds` will change on rotation, for example.
  This view is likely the one you will programmatically add `subviews` to (if you ever do that).
  All of your MVC's View's `UIViews` will have this view as an ancestor.
  It's automatically hooked up for you when you create an MVC in Xcode.

Initializing a UIView

- As always, try to avoid an initializer if possible

  But having one in `UIView` is slightly more common than having a `UIViewController` initializer

- A `UIView`'s initializer is different if it comes out of a storyboard

```
init(frame: CGRect)  // initializer if the UIView is created in code
init(coder: NSCoder) // initializer if the UIView comes out of a storyboard
```

- If you need an initializer, implement them both …

```
func setup() { … }

override init(frame: CGRect) {                  // a designed initializer
    super.init(frame: frame)
    setup()
}
required init(coder aDecoder: NSCoder) {  // a required initializer
    super.init(coder: aDecoder)
    setup()
}
```

CS193p
Spring 2016

Initializing a UIView

- Another alternative to initializers in `UIView` ...

  `awakeFromNib()` // this is only called if the `UIView` came out of a storyboard

  This is not an initializer (it's called immediately after initialization is complete)

  All objects that inherit from `NSObject` in a storyboard are sent this

  Order is not guaranteed, so you cannot message any other objects in the storyboard here

Coordinate System Data Structures

- `CGFloat`

  Always use this instead of `Double` or `Float` for anything to do with a `UIView`'s coordinate system

  You can convert to/from a `Double` or `Float` using initializers, e.g., `let cfg = CGFloat(aDouble)`

- `CGPoint`

  Simply a `struct` with two `CGFloats` in it: `x` and `y`.

```
var point = CGPoint(x: 37.0, y: 55.2)
point.y -= 30
point.x += 20.0
```

- `CGSize`

  Also a `struct` with two `CGFloats` in it: `width` and `height`.

```
var size = CGSize(width: 100.0, height: 50.0)
size.width += 42.5
size.height += 75
```

CS193p
Spring 2016

Coordinate System Data Structures

- CGRect

  A struct with a CGPoint and a CGSize in it …

```
struct CGRect {
    var origin: CGPoint
    var size: CGSize
}
let rect = CGRect(origin: aCGPoint, size: aCGSize) // there are other inits as well
```

  Lots of convenient properties and functions on CGRect like …

```
var minX: CGFloat            // left edge
var midY: CGFloat            // midpoint vertically
intersects(CGRect) -> Bool // does this CGRect intersect this other one?
intersect(CGRect)            // clip the CGRect to the intersection with the other one
contains(CGPoint) -> Bool  // does the CGRect contain the given CGPoint?
```

  … and many more (make yourself a CGRect and type . after it to see more)

(0,0)

increasing x

View Coordinate System

(500, 35)

- Origin is upper left
- Units are points, not pixels

A point is a standard length equivalent to 1x1 pixels on a non-retina device, and 2x2 pixels on a retina device

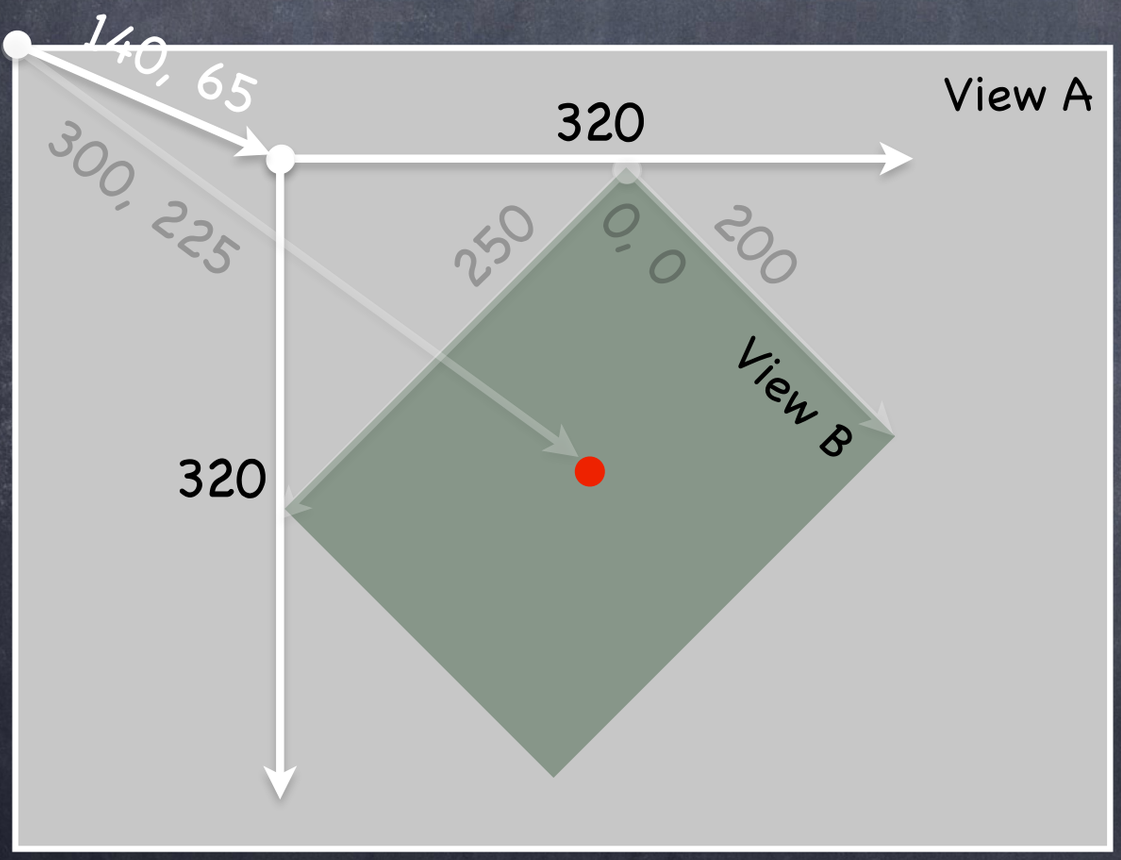
  - Pixels are the minimum-sized unit of drawing your device is capable of
  - Points are the units in the coordinate system
  - Most of the time there are 2 pixels per point, but it could be only 1 or something else
  - How many pixels per point are there? `UIView`'s `var contentScaleFactor: CGFloat`

- The boundaries of where drawing happens
  - `var bounds: CGRect` // a view's internal drawing space's origin and size
  - This is the rectangle containing the drawing space in its own coordinate system
  - It is up to your view's implementation to interpret what `bounds.origin` means (often nothing)
- Where is the `UIView`?
  - `var center: CGPoint` // the center of a `UIView` in its superview's coordinate system
  - `var frame: CGRect` // the rect containing a `UIView` in its superview's coordinate system

increasing y

bounds vs frame

- Use `frame` and/or `center` to position a `UIView`
  - These are never used to draw inside a view's coordinate system
  - You might think `frame.size` is always equal to `bounds.size`, but you'd be wrong …



Views can be rotated (and scaled and translated)

View B's `bounds = ((0,0),(200,250))`
View B's `frame = ((140,65),(320,320))`
View B's `center = (300,225)`

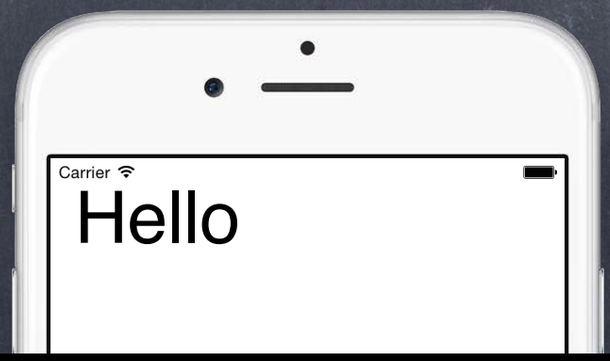
View B's middle in its own coordinates is …
`(bounds.midX, bounds.midY) = (100, 125)`

Views are rarely rotated, but don't misuse `frame` or `center` anyway by assuming that.

Creating Views

- Most often your views are created via your storyboard
  - Xcode's Object Palette has a generic UIView you can drag out
  - After you do that, you must use the Identity Inspector to changes its class to your subclass
- On rare occasion, you will create a UIView via code
  - You can use the frame initializer … `let newView = UIView(frame: myViewFrame)`
  - Or you can just use `let newView = UIView()` (frame will be CGRectZero)
- Example

```
// assuming this code is in a UIViewController
let labelRect = CGRect(x: 20, y: 20, width: 100, height: 50)
let label = UILabel(frame: labelRect) // UILabel is a subclass of UIView
label.text = “Hello”
view.addSubview(label)
```



Custom Views

- When would I 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'll talk about handling touch events in a bit. First we'll focus on drawing.
- To draw, just create a `UIView` subclass and override `drawRect:`

```
override func drawRect(regionThatNeedsToBeDrawn: CGRect)
```

  - You can draw outside the `regionThatNeedsToBeDrawn`, but it's never required to do so
  - The `regionThatNeedsToBeDrawn` is purely an optimization
  - It is our `UIView`'s `bounds` that describe the entire drawing area (the region is a subarea).
- NEVER call `drawRect`!! EVER! Or else!
  - Instead, if you view needs to be redrawn, let the system know that by calling ...

```
setNeedsDisplay()
setNeedsDisplayInRect(regionThatNeedsToBeRedrawn: CGRect)
```

  - iOS will then call your `drawRect` at an appropriate time

Custom Views

- So how do I implement my `drawRect`?
  - You can use a C-like (non object-oriented) API called Core Graphics
  - Or you can use the object-oriented `UIBezierPath` class (which is how we'll do it)
- Core Graphics Concepts
  1. You get a context to draw into (other contexts include printing, off-screen buffer, etc.)
     The function `UIGraphicsGetCurrentContext()` gives a context you can use in `drawRect`
  2. Create paths (out of lines, arcs, etc.)
  3. Set drawing attributes like colors, fonts, textures, linewidths, linecaps, etc.
  4. Stroke or fill the above-created paths with the given attributes

Custom Views

- So how do I implement my `drawRect`?
  - You can use a C-like (non object-oriented) API called Core Graphics
  - Or you can use the object-oriented `UIBezierPath` class (which is how we'll do it)
- Core Graphics Concepts
  1. You get a context to draw into (other contexts include printing, off-screen buffer, etc.)
     The function `UIGraphicsGetCurrentContext()` gives a context you can use in `drawRect`
  2. Create paths (out of lines, arcs, etc.)
  3. Set drawing attributes like colors, fonts, textures, linewidths, linecaps, etc.
  4. Stroke or fill the above-created paths with the given attributes
- `UIBezierPath`
  - Same as above, but captures all the drawing with a `UIBezierPath` instance
  - `UIBezierPath` automatically draws in the "current" context (`drawRect` sets this up for you)
  - `UIBezierPath` has methods to draw (lineto, arcs, etc.) and set attributes (linewidth, etc.)
  - Use `UIColor` to set stroke and fill colors
  - `UIBezierPath` has methods to `stroke` and/or `fill`

{ Coding Session }

Custom UIView

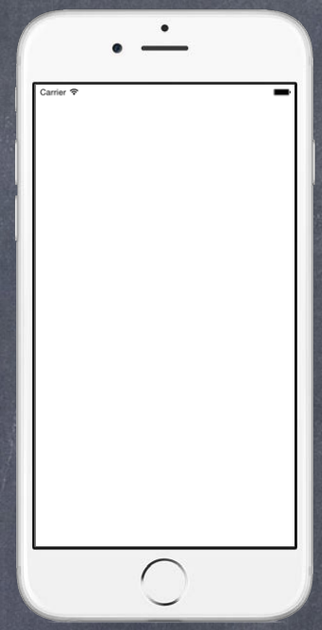
Defining a Path

- Create a UIBezierPath

```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50))
```



CS193p
Spring 2016

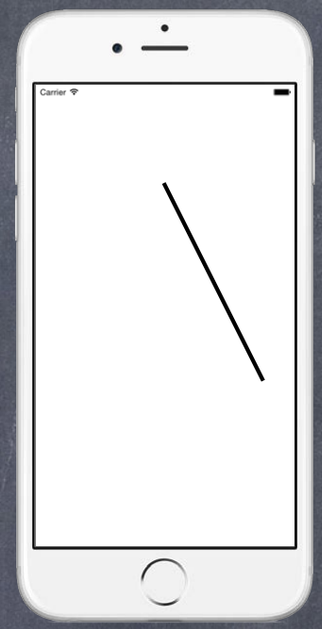
Defining a Path

- Create a UIBezierPath

```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50))
path.addLineToPoint(CGPoint(140, 150))
```



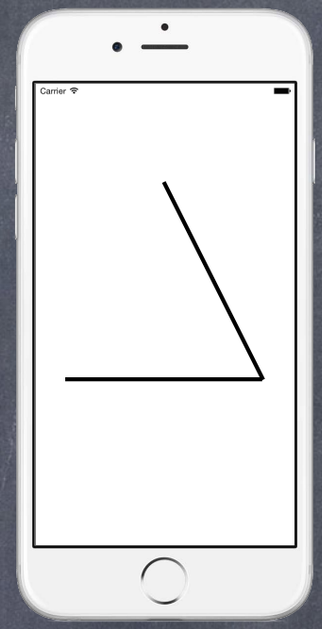
Defining a Path

- Create a UIBezierPath

```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50))
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```



Defining a Path

- Create a UIBezierPath

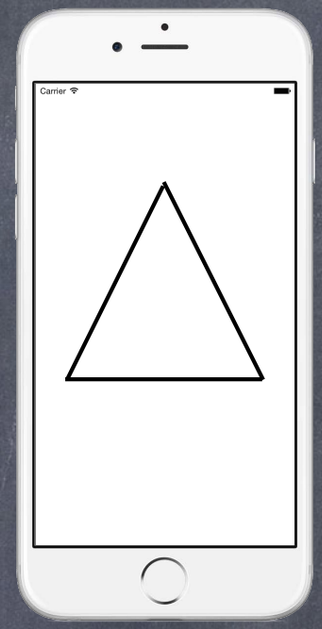
```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50))
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```



Defining a Path

- Create a UIBezierPath

```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50))
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```

- Now that you have a path, set attributes and stroke/fill

```
UIColor.greenColor().setFill() // note this is a method in UIColor, not UIBezierPath
UIColor.redColor().setStroke() // note this is a method in UIColor, not UIBezierPath
path.linewidth = 3.0           // note this is a property in UIBezierPath, not UIColor
```

Defining a Path

- Create a UIBezierPath

```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

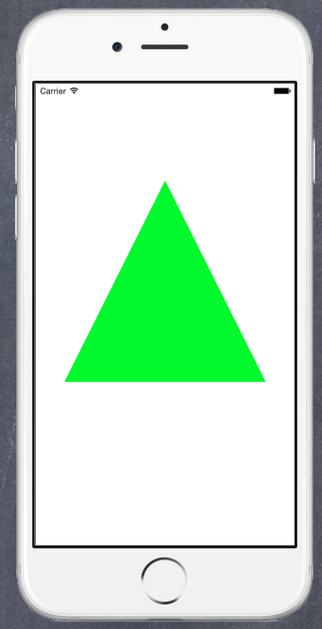
```
path.moveToPoint(CGPoint(80, 50))
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```

- Now that you have a path, set attributes and stroke/fill

```
UIColor.greenColor().setFill() // note this is a method in UIColor, not UIBezierPath
UIColor.redColor().setStroke() // note this is a method in UIColor, not UIBezierPath
path.linewidth = 3.0           // note this is a property in UIBezierPath, not UIColor
path.fill()                    // method in UIBezierPath
```



Defining a Path

- Create a UIBezierPath

```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

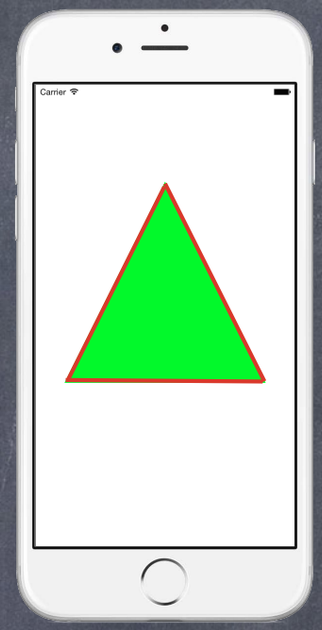
```
path.moveToPoint(CGPoint(80, 50))
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```

- Now that you have a path, set attributes and stroke/fill

```
UIColor.greenColor().setFill() // note this is a method in UIColor, not UIBezierPath
UIColor.redColor().setStroke() // note this is a method in UIColor, not UIBezierPath
path.linewidth = 3.0           // note this is a property in UIBezierPath, not UIColor
path.fill()                    // method in UIBezierPath
path.stroke()                  // method in UIBezierPath
```



Drawing Text

- Usually we use a `UILabel` to put text on screen
  - But there are certainly occasions where we want to draw text in our `drawRect`
- To draw in `drawRect`, use `NSAttributedString`

```
let text = NSAttributedString("hello")
text.drawAtPoint(aCGPoint)
let textSize: CGSize = text.size // how much space the string will take up
```

- Mutability is done with `NSMutableAttributedString`
  - It is not like `String` (i.e. where `let` means immutable and `var` means mutable)
  - You use a different class if you want to make a mutable attributed string …

```
let mutableText = NSMutableAttributedString("some string")
```

- `NSAttributedString` is not a `String`, nor an `NSString`
  - You can get its contents as an `String`/`NSString` with its `string` or `mutableString` property

CS193p
Spring 2016

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

- 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(UIView) -> CGPoint // cumulative since start of recognition
  func velocityInView(UIView) -> CGPoint // how fast the finger is moving (points/s)
  func setTranslation(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 `_:`, then no gesture argument)

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
```