Stanford CS193p

Developing Applications for iOS
Winter 2017

CS193p
Winter 2017

Today

- Error Handling in Swift

  `try`

- Extensions

  A simple, powerful, but easily overused code management syntax

- Protocols

  Last (but certainly not least important) typing mechanism in Swift

- Delegation

  An important use of protocols used throughout the iOS frameworks API

- Scroll View

  Scrolling around in and zooming in on big things on a small screen

Thrown Errors

- In Swift, methods can throw errors

  You will always know these methods because they'll have the keyword `throws` on the end.

```
func save() throws
```

  You must put calls to functions like this in a `do { }` block and use the word `try` to call them.

```
do {
    try context.save()
} catch let error {
    // error will be something that implements the Error protocol, e.g., NSError
    // usually these are enums that have associated values to get error details
    throw error // this would re-throw the error (only ok if the method we are in throws)
}
```

  If you are certain a call will not `throw`, you can force `try` with `try!` …

```
try! context.save() // will crash your program if save() actually throws an error
```

  Or you can conditionally `try`, turning the return into an Optional (which will be `nil` if fail) …

```
let x = try? errorProneFunctionThatReturnsAnInt() // x will be Int?
```

Extensions

- Extending existing data structures

  You can add methods/properties to a class/struct/enum (even if you don't have the source).
  For example, this adds a method contentViewController to UIViewController …

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

  … it can be used to clean up prepare(for segue:, sender:) code …

```
var destination: UIViewController? = segue.destinationViewController
if let navcon = destination as? UINavigationController {
    destination = navcon.visibleViewController
}
if let myvc = destination as? MyVC { … }
```

Extensions

- Extending existing data structures

  You can add methods/properties to a `class`/`struct`/`enum` (even if you don't have the source).

  For example, this adds a method `contentViewController` to `UIViewController` …

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

  … it can be used to clean up `prepare(for segue:, sender:)` code …

```
if let myvc = segue.destinationViewController.contentViewController as? MyVC { … }
```

Extensions

- Extending existing data structures

  You can add methods/properties to a `class`/`struct`/`enum` (even if you don't have the source).
  For example, this adds a method `contentViewController` to `UIViewController` …

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

  Notice that when it refers to `self`, it means the thing it is extending (`UIViewController`).

Extensions

- Extending existing data structures
  - You can add methods/properties to a `class`/`struct`/`enum` (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 (computed only).
- 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

- Protocols are a way to express an API more concisely
  - Instead of forcing the caller of an API to pass a specific `class`, `struct`, or `enum`,
    an API can let callers pass any `class`/`struct`/`enum` that the caller wants
    but can require that they implement certain methods and/or properties that the API wants.
  - To specify which methods and properties the API wants, the API is expressed using a `protocol`.
  - A `protocol` is simply a collection of method and property declarations.
- A `protocol` is a TYPE
  - It can be used almost anywhere any other type is used: `var`s, function parameters, etc.
- The implementation of a Protocol's methods and properties
  - The implementation is provided by an implementing type (any `class`, `struct` or `enum`).
  - Because of this, a protocol can have no storage associated with it
    (any storage required to implement the protocol is provided by an implementing type).
  - It is also possible to add implementation to a `protocol` via an `extension` to that protocol
    (but remember that `extensions` also cannot use any storage)

Protocols

- There are three aspects to a protocol
  1. the `protocol` declaration (which properties and methods are in the `protocol`)
  2. a `class`, `struct` or `enum` declaration that claims to implement the `protocol`
  3. the code in said `class`, `struct` or `enum` that implements the `protocol`
- Optional methods in a protocol

  Normally any `protocol` implementor must implement all the methods/properties in the protocol.
  However, it is possible to mark some methods in a protocol `optional`
  (don't get confused with the type Optional, this is a different thing).
  Any `protocol` that has `optional` methods must be marked `@objc`.
  And any `optional-protocol` implementing `class` must inherit from `NSObject`.
  These sorts of `protocols` are used often in iOS for delegation (more later on this).
  Except for delegation, a `protocol` with `optional` methods is rarely (if ever) used.
  As you can tell from the `@objc` designation, it's mostly for backwards compatibility.

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 `init`ializer

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, `enum`s and `struct`s 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, `enum`s and `struct`s 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, `enum`s and `struct`s 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, `enum`s and `struct`s would not have the superclass part
Any number of protocols can be implemented by a given `class`, `struct` or `enum`
In a `class`, `init`s 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, `enum`s and `struct`s would not have the superclass part
Any number of protocols can be implemented by a given `class`, `struct` or `enum`
In a `class`, `init`s must be marked `required` (or otherwise a subclass might not conform)
You are allowed to add `protocol` conformance via an `extension`

Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

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

Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

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

Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: 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: Slippery & Moveable)
slipAndSlide(prius)
```

Advanced use of Protocols

- Mixing in generics makes protocols even more powerful

  Protocols can be used to restrict a type that a generic can handle

  Consider the type that was "sort of" `Range<T>` … this type is actually …

```
struct Range<Bound: Comparable> {
    let lowerBound: Bound
    let upperBound: Bound
}
```

  `Comparable` is a `protocol` which dictates that the given type must implement greater/less than

  That's how `Range` can know that its `lowerBound` is less than its `upperBound`

  (And it can know this regardless of whether it's a `Range` of `Int`s or `Character`s or `Float`s)

  Making a `protocol` that itself uses generics is also a very leveraged API design approach

  Many, many protocols in Swift's standard library are declared to operate on generic types

Advanced use of Protocols

- "Multiple inheritance" with protocols
  - Consider the `struct` `CountableRange` (i.e. what you get with `3..<5`) …
  - This `struct` implements MANY protocols (here are just a few):
  - `IndexableBase` — `startIndex`, `endIndex`, `index(after:)` and subscripting (e.g. `[]`)
  - `Indexable` — `index(offsetBy:)`
  - `BidirectionalIndexable` — `index(before:)`
  - `Sequence` — `makeIterator` (and thus supports `for in`)
  - `Collection` — basically `Indexable & Sequence`
- Why do it this way?
  - Because `Array`, for example, also implements all of these protocols.
  - So now you can create generic code that operates on a `Collection` and it will work on both!
  - `Dictionary` is also a `Collection`, as is `Set` and `String.UTF16View`.
  - But wait, there's more …

Advanced use of Protocols

- Extensions also contribute to the power of protocols

  An `extension` can be used to add default implementation to a protocol.

  Since there's no storage, said implementation has to be in terms of other API in the protocol (although that other API might well be inherited from another protocol).

  For example, for the `Sequence` protocol, you really only need to implement `makeIterator`.

  (An iterator implements the `IteratorProtocol` which just has the method `next()`.)

  If you do, you will automatically get implementations for all these other methods in `Sequence`: `contains()`, `forEach()`, `joined(separator:)`, `min()`, `max()`, even `filter()` and `map()`, et. al.

  All of these are implemented via an `extension to the Sequence protocol`.

  This `extension` (provided by Apple) uses only `Sequence protocol` methods in its implementation.

- Functional Programming

  By combining `protocols` with generics and `extensions` (default implementations), you can build code that focusses more on the behavior of data structures than storage.

  Again, we don't have time to teach functional programming, but this is a path towards that.

Another Example Protocol

- Converting to a String

  A data structure implementing the `protocol` `CustomStringConvertible` will `print` with `\()`

```
protocol CustomStringConvertible {
    var description: String { get }
}
```

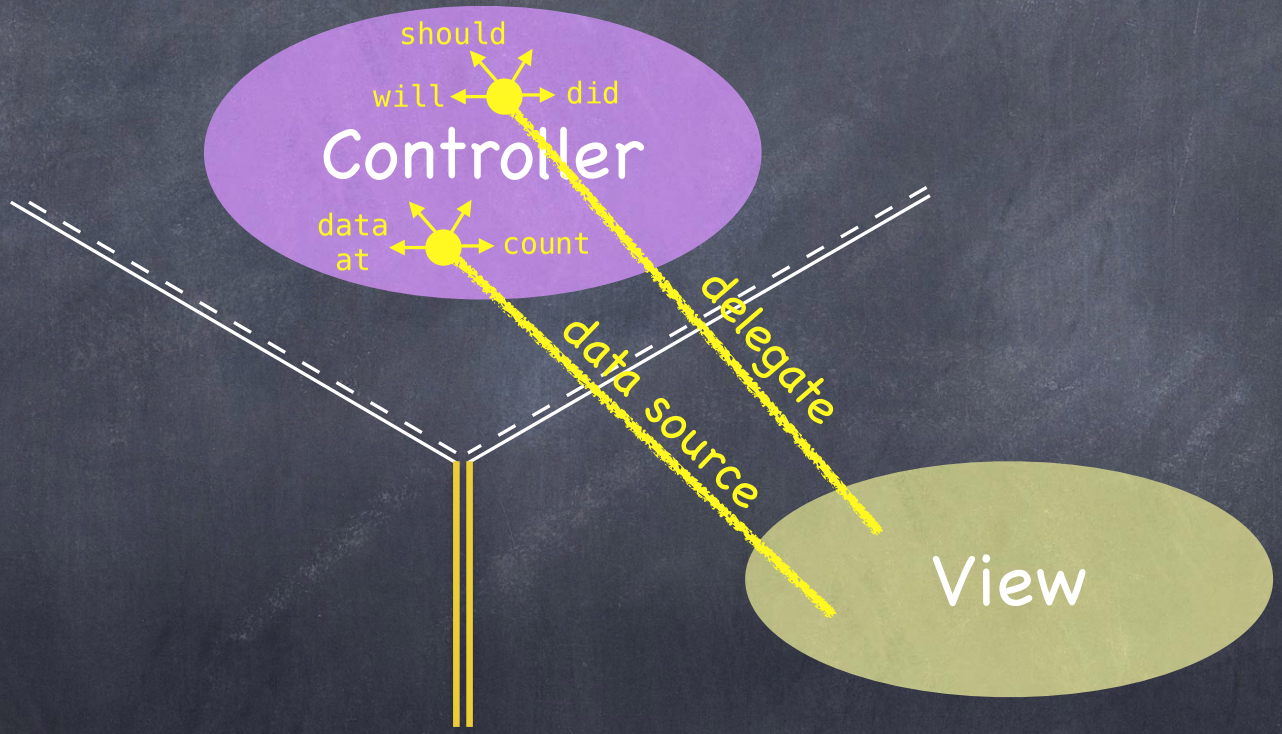
  You could make a `CalculatorBrain` `print` with `\()` just by adding this to its declaration:

```
struct CalculatorBrain: CustomStringConvertible
```

  This works because `CalculatorBrain` already implements that `description` `var`.

Delegation

- A very important (simple) use of `protocols`
  - It's a way to implement "blind communication" between a View and its Controller



Delegation

- A very important (simple) use of `protocols`
  - It's a way to implement "blind communication" between a View and its Controller
- How it plays out ...
  1. A View declares a delegation `protocol` (i.e. what the View wants the Controller to do for it)
  2. The View's API has a `weak delegate` property whose type is that delegation `protocol`
  3. The View uses the `delegate` property to get/do things it can't own or control on its own
  4. The Controller declares that it implements the `protocol`
  5. The Controller sets `self` as the `delegate` of the View by setting the property in #2 above
  6. The Controller implements the `protocol` (probably it has lots of `optional` methods in it)
- 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
- This mechanism is found throughout iOS
  - However, it was designed pre-closures in Swift. Closures are often a better option.

Delegation

- Example

  UIScrollView (which we'll talk about in a moment) has a delegate property …

```
weak var delegate: UIScrollViewDelegate?
```

  The UIScrollViewDelegate protocol looks like this …

```
@objc protocol UIScrollViewDelegate {
    optional func scrollViewDidScroll(scrollView: UIScrollView)
    optional func viewForZooming(in scrollView: UIScrollView) -> UIView
   … and many more …
}
```

  A Controller with a UIScrollView in its View would be declared like this …

```
class MyViewController : UIViewController, UIScrollViewDelegate { … }
```

  … and in its viewDidLoad() or in the scroll view outlet setter, it would do …

```
scrollView.delegate = self
```

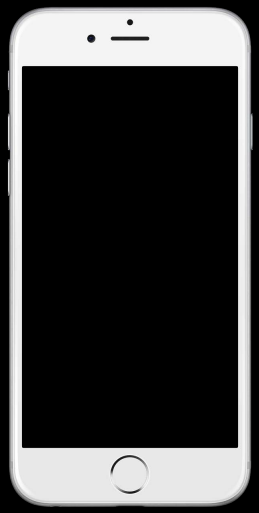
  … and it then would implement any of the protocol's methods it is interested in.

Adding subviews to a normal UIView …

```
logo.frame = CGRect(x: 300, y: 50, width: 120, height: 180)
view.addSubview(logo)
```

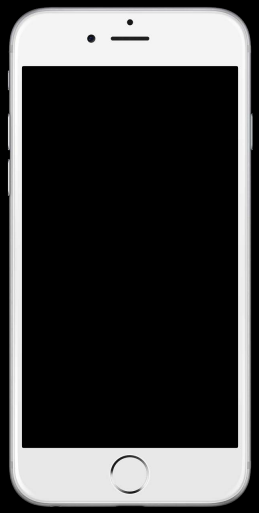
Adding subviews to a UIScrollView …

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
```



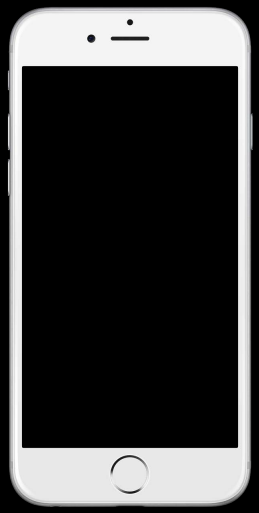
Adding subviews to a UIScrollView …

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
logo.frame = CGRect(x: 2700, y: 50, width: 120, height: 180)
scrollView.addSubview(logo)
```



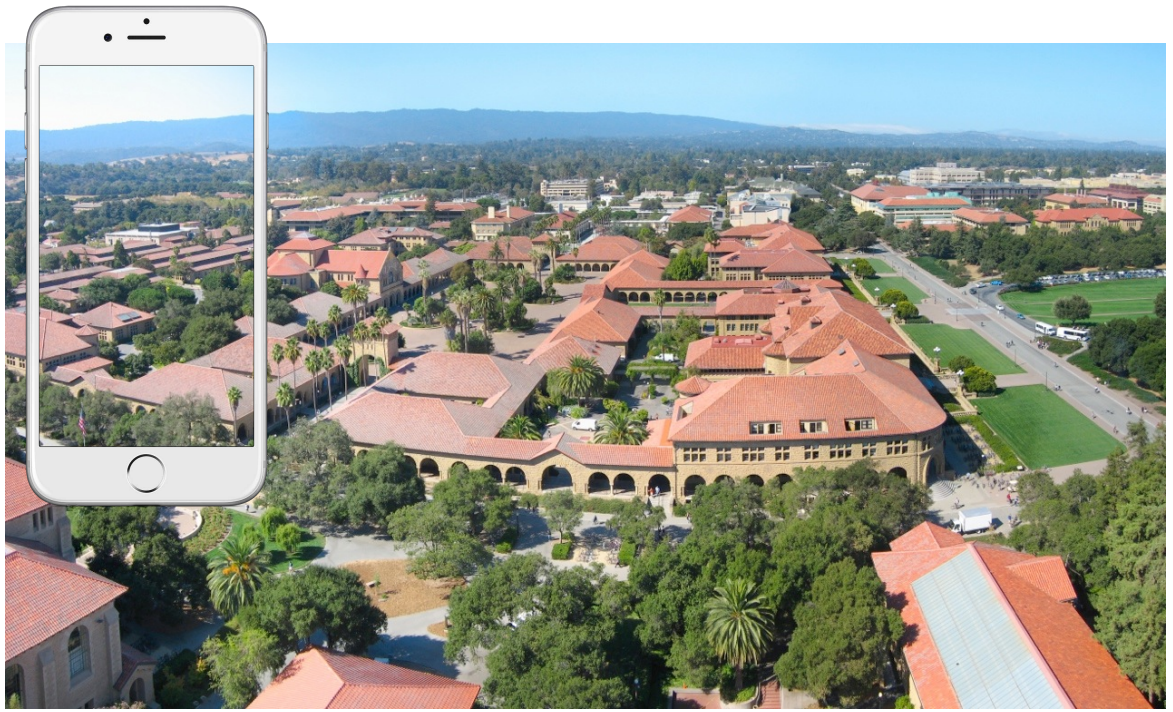
Adding subviews to a UIScrollView …

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
aerial.frame = CGRect(x: 150, y: 200, width: 2500, height: 1600)
scrollView.addSubview(aerial)
```

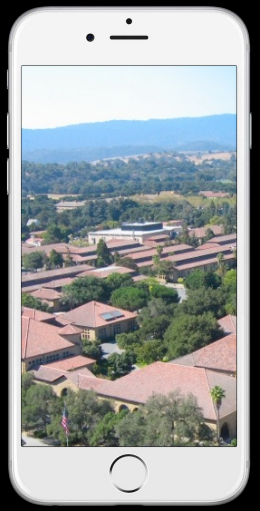


Adding subviews to a UIScrollView …

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
aerial.frame = CGRect(x: 150, y: 200, width: 2500, height: 1600)
scrollView.addSubview(aerial)
```



Scrolling in a UIScrollView …



Scrolling in a UIScrollView …

Scrolling in a UIScrollView …

Positioning subviews in a UIScrollView …

Positioning subviews in a UIScrollView …

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
```

Positioning subviews in a UIScrollView …

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)
```

Positioning subviews in a UIScrollView …

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)
scrollView.contentSize = CGSize(width: 2500, height: 1600)
```

That's it!

That's it!

CS193p
Winter 2017

That's it!

That's it!

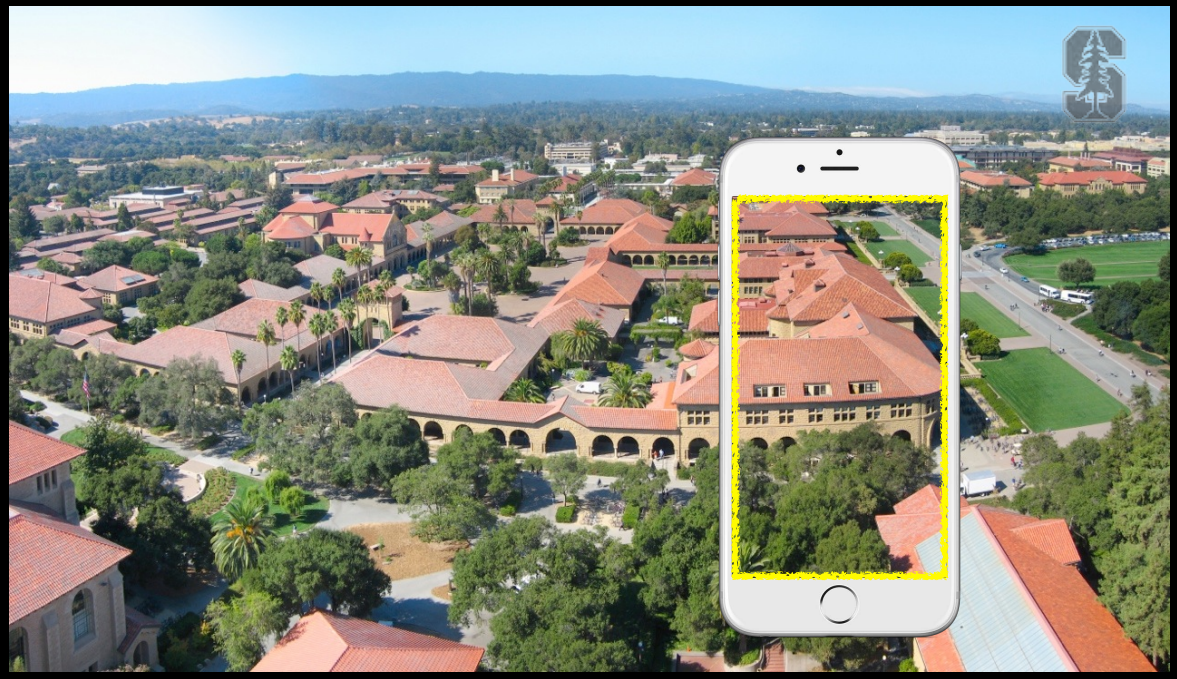
Where in the content is the scroll view currently positioned?

```
let upperLeftOfVisible: CGPoint = scrollView.contentOffset
```

In the content area's coordinate system.

What area in a subview is currently visible?

```
let visibleRect: CGRect = aerial.convert(scrollView.bounds, from: scrollView)
```



Why the `convertRect`? Because the `scrollView`'s `bounds` are in the `scrollView`'s coordinate system. And there might be zooming going on inside the `scrollView` too ...

UIScrollView

- How do you create one?

  Just like any other UIView. Drag out in a storyboard or use UIScrollView(frame:).

  Or select a UIView in your storyboard and choose "Embed In -> Scroll View" from Editor menu.

- To add your "too big" UIView in code using addSubview ...

```
if let image = UIImage(named: "bigimage.jpg") {
    let iv = UIImageView(image: image)  // iv.frame.size will = image.size
    scrollView.addSubview(iv)
}
```

  Add more subviews if you want.

  All of the subviews' frames will be in the UIScrollView's content area's coordinate system (that is, (0,0) in the upper left & width and height of contentSize.width & .height).

- Now don't forget to set the contentSize

  Common bug is to do the above lines of code (or embed in Xcode) and forget to say:

  `scrollView.contentSize = imageView.frame.size` (for example)

UIScrollView

- Scrolling programmatically

  `func scrollRectToVisible(CGRect, animated: Bool)`

- Other things you can control in a scroll view

  Whether scrolling is enabled.

  Locking scroll direction to user's first "move".

  The style of the scroll indicators (call `flashScrollIndicators` when your scroll view appears).

  Whether the actual content is "inset" from the content area (`contentInset` property).

UIScrollView

- Zooming

  All UIView's have a property (transform) which is an affine transform (translate, scale, rotate).
  Scroll view simply modifies this transform when you zoom.
  Zooming is also going to affect the scroll view's contentSize and contentOffset.

- Will not work without minimum/maximum zoom scale being set

```
scrollView.minimumZoomScale = 0.5   // 0.5 means half its normal size
scrollView.maximumZoomScale = 2.0   // 2.0 means twice its normal size
```

- Will not work without delegate method to specify view to zoom

```
func viewForZooming(in scrollView: UIScrollView) -> UIView
```

  If your scroll view only has one subview, you return it here. More than one? Up to you.

- Zooming programatically

```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoom(to rect: CGRect, animated: Bool)
```

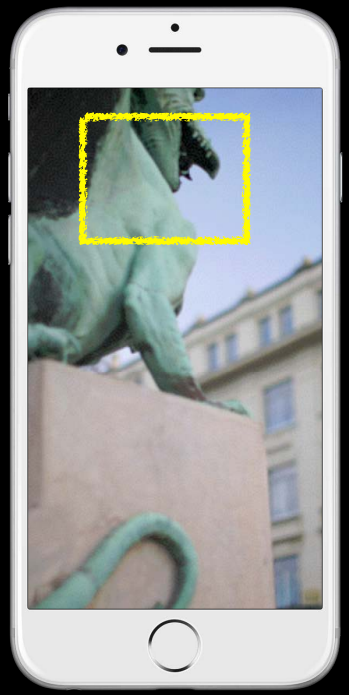
```
scrollView.zoomScale = 1.2
```

CS193p
Winter 2017

```
scrollView.zoomScale = 1.0
```

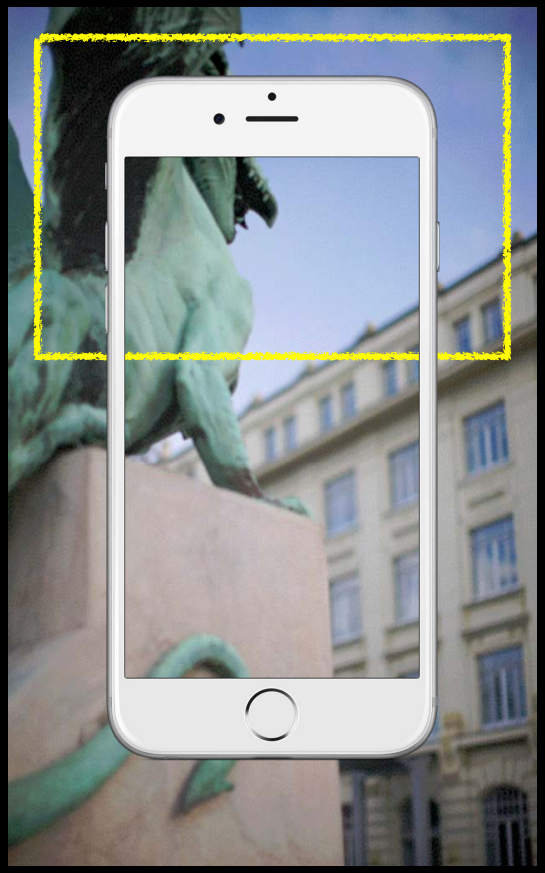
```
scrollView.zoomScale = 1.2
```

CS193p
Winter 2017

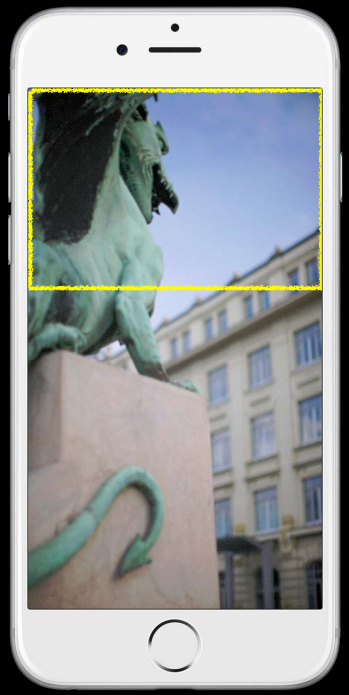


```
zoom(to rect: CGRect, animated: Bool)
```

```
zoom(to rect: CGRect, animated: Bool)
```



zoom(to rect: CGRect, animated: Bool)



zoom(to rect: CGRect, animated: Bool)

UIScrollView

- Lots and lots of `delegate` methods!

  The scroll view will keep you up to date with what's going on.

- Example: `delegate` method will notify you when zooming ends

```
func scrollViewDidEndZooming(UIScrollView,
                             with view: UIView,  // from delegate method above
                             atScale: CGFloat)
```

  If you redraw your view at the new scale, be sure to reset the transform back to identity.