Protocols

- Collection of blueprints of methods/properties

  Instead of specifying the full class/struct you can actually specify 'what it is about that class/struct' that you need (some piece of functionality). A Protocol is simply a collection of methods and property declarations.

- A protocol is a TYPE

  It can be used almost everywhere: vars, functions etc.

- No implementation in Protocols

  The implementation of the Protocol's methods/properties happens inside of the class/struct that implements this protocol. However, it is possible to implement methods/properties inside an extension

Protocols

- Optional methods in Protocols

  Normally all of the declarations inside the protocol must be implemented. However, it is possible to mark certain methods/properties to be `optional`.

  The protocol that has an optional methods/properties must be marked as `@objc`

  Any class/struct that uses this protocol must inherit from `NSObject`

- Protocol declaration

```
protocol SomeProtocol: InheritedProtocol, InheritedProtocol2{
    var someProperty{get set}
    func aMethod(a: Double, b: String)-> SomeType
    mutating func change()
    init(arg: Type)
}
```

Protocols

- Example of a protocol

```
protocol Movable{
    mutating func moveTo(p: CGPoint)
}

class Car: Movable{
    func moveTo(p: CGPoint) {}
    func changeOil(){}
}
struct Shape: Movable{
    mutating func moveTo(p: CGPoint) {}
    func draw(){}
}

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

```
var thingToMove: Movable = toyota
thingToMove.moveTo(...)

thingToMove.changeOil()

thingToMove = square
thingToMove.moveTo(...)

let thingsToMove : [Movable] = [
    toyota,
    square
]
```

Delegation

- A very important use of protocols

  It's the way of implementing the 'blind and structured communication' between View and Controller

- How it works?

  1) View declares a delegation protocol(what the view wants the VC to do)
  2) View's API has a `delegate` property whose type of that delegation protocol
  3) View uses the `delegate` property to do things it can't own
  4) The Controller declares that it implements the protocol
  5) The Controller sets `self` as the delegate of the View
  6) The Controller implements the protocols methods

Delegation

- Example

  UIWebView has a delegate property (WebBrowser example)

```
weak var delegate: UIWebViewDelegate?
```

  The protocol of the UIWebView looks like this:

```
public protocol UIWebViewDelegate : NSObjectProtocol {

    @available(iOS 2.0, *)
    optional public func webViewDidStartLoad(_ webView: UIWebView)

    @available(iOS 2.0, *)
    optional public func webViewDidFinishLoad(_ webView: UIWebView)
```

  A Controller with WebView in it's View would be declared like this:

```
class MyViewController: ViewController, UIWebViewDelegate{…}
```

  … and in viewDidLoad() it would do this

```
myWebView.delegate = self
```

  … or control-drag from the WebView to the ViewController