Protocols and Delegates
Dr. Sarah Abraham
Protocols

- Group of related properties and methods that can be implemented by any class
- Independent of any class
  - Known in other languages as interfaces
  - Abstract with no default implementation
- Used throughout iOS development
A table view displays tabular information

Must talk to a data source to know what to display

Data source must respond to table view’s messages

Protocol declares methods expected (or required) for this situation
protocol UITableViewDataSource : NSObjectProtocol {
  . . . . .

  func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
  /* Data source returns the number of rows in a given section of table view */

  func tableView(_ tableView: UITableView, cellForItemAt indexPath: IndexPath) -> UITableViewCell
  /* Data source returns cell to insert in particular location of table view */

  optional func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool
  /* Optional: Data source determines whether row is editable */

  optional func tableView(_ tableView: UITableView, canMoveRowAt indexPath: IndexPath) -> Bool
  /* Optional: Data source determines whether row is movable */

  . . . . .
}

• UITableViewDataSource provides information to TableView about:

• Number of cells
• Content of the cells
• Optional editing functionality
Using a Protocol

- Class must conform to protocol in order to implement it
- Classes that conform to protocol must implement all required methods
- Setup of protocol determines what is required versus optional
- Protocols can inherit from other protocols
Creating a Protocol

```swift
protocol FullyNamed {

    // requires conforming type to provide full name

    var fullName: String { get }
}
```
Implementing a Protocol

struct Person: FullyNamed {

    //Person struct conforms to FullyNamed protocol

    var fullName: String
}

let player1 = Person(fullName: "Johnny Appleseed")
Optionals in Protocols

- Protocol must be flagged with `@objc` attribute to implement optional methods or variables
- All protocol methods and variables must then be flagged with `@objc`
- Only classes can implement protocols flagged with `@objc`
Optionals in Protocols Example

```swift
import Foundation

@objc protocol FullyNamed {
    // requires conforming type to provide full name
    var fullName: String { get }

    optional func getFirstName() -> String
}

class Person: FullyNamed {
    // Person struct conforms to FullyNamed protocol
    @objc var fullName: String

    init(fullName: String) {
        self.fullName = fullName
    }

    @objc func getFirstName() -> String {
        let nameList = fullName.componentsSeparatedByString(" ")
        return nameList[0]
    }
}
```
Why Use Protocols?

- More flexible than normal class interface
- Reuse single API declaration in unrelated classes
- Clear design and purpose in code structure
- Specify an object’s role across application
Delegates

- Pattern where one object acts on behalf of (or in coordination with) another object
- Allows for customization of several objects’ behavior via one central object
- Simplifies communication between objects
- Used extensively in iOS development
- Closely associated with protocols
How Delegates Work

- Delegating object keeps reference to delegate
- Sends message to delegate at appropriate time
- Delegate returns with message at appropriate time
- Multiple delegates allowed per delegating object
Delegate Example

```swift
protocol DiceGame {
    // Protocol to add dice functionality into a game
    var dice: Dice { get } // Dice class rolls a sided die at random
    func beginPlay()
    func play()
}

protocol DiceGameDelegate {
    // Protocol (delegate) to track the game’s progress
    func gameDidStart(game: DiceGame)
    func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(game: DiceGame)
}
```
Implementing a Delegate

class DiceGameTracker: DiceGameDelegate {
    var turns = 0

    func gameDidStart(game: DiceGame) {
        turns = 0
    }

    func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        turns += 1
    }

    func gameDidEnd(game: DiceGame) {
    }
}
Using a Delegate

class SnakesAndLadders: DiceGame {
    let dice = Dice()

    var delegate: DiceGameDelegate?

    func beginPlay() { delegate?.gameDidStart(self) }

    func play() {
        while !gameEnded {
            let diceRoll = dice.roll()

            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
        }

        delegate?.gameDidEnd(self)
    }
}

Creating a Delegate

- Must instantiate the delegate and protocol classes
- Protocol class’ delegate must point to valid delegate

```swift
let tracker = DiceGameTracker()

let game = SnakesAndLadders()

game.delegate = tracker

game.play()
```
Protocols and Delegates

- Classes can use both protocols and delegates

![Diagram]

UITableViewCell

Subclassed

Conforms to:
UITableViewDelegate, UITableViewDataSource

Calls come from the base UITableViewController
class down to your class

Delegate of
UITableViewViewController

Your table view controller class