Protocols and Delegates

Dr. Sarah Abraham

University of Texas at Austin
CS329e
Spring 2019

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

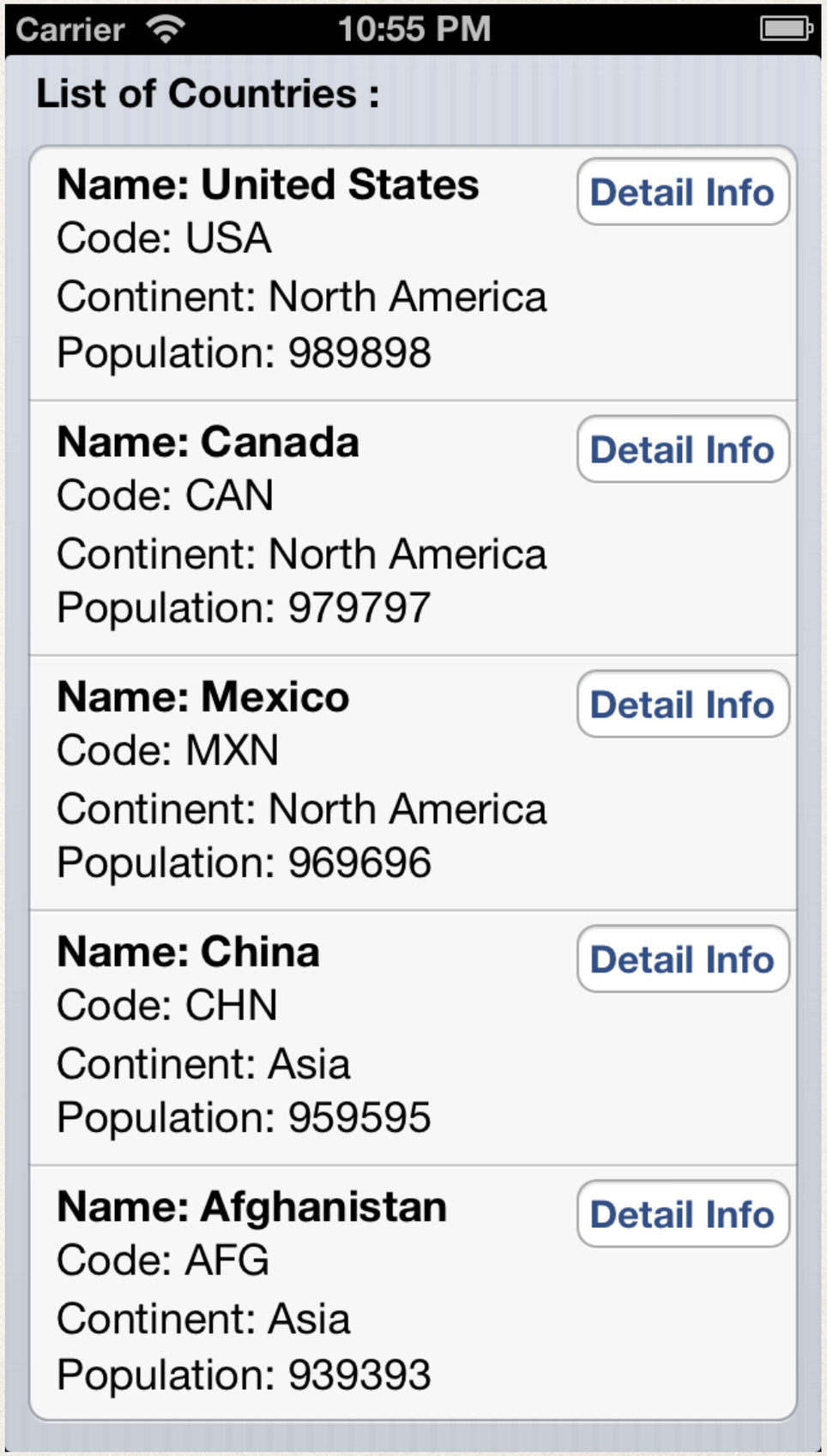
Example

- 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

UITableViewDataSource Protocol

```
protocol UITableViewDataSource : NSObjectProtocol {
. . . . .

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

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

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

    optional func tableView(UITableView, canMoveRowAt: 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

```
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”)
```

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

UITableViewDataSource Redux

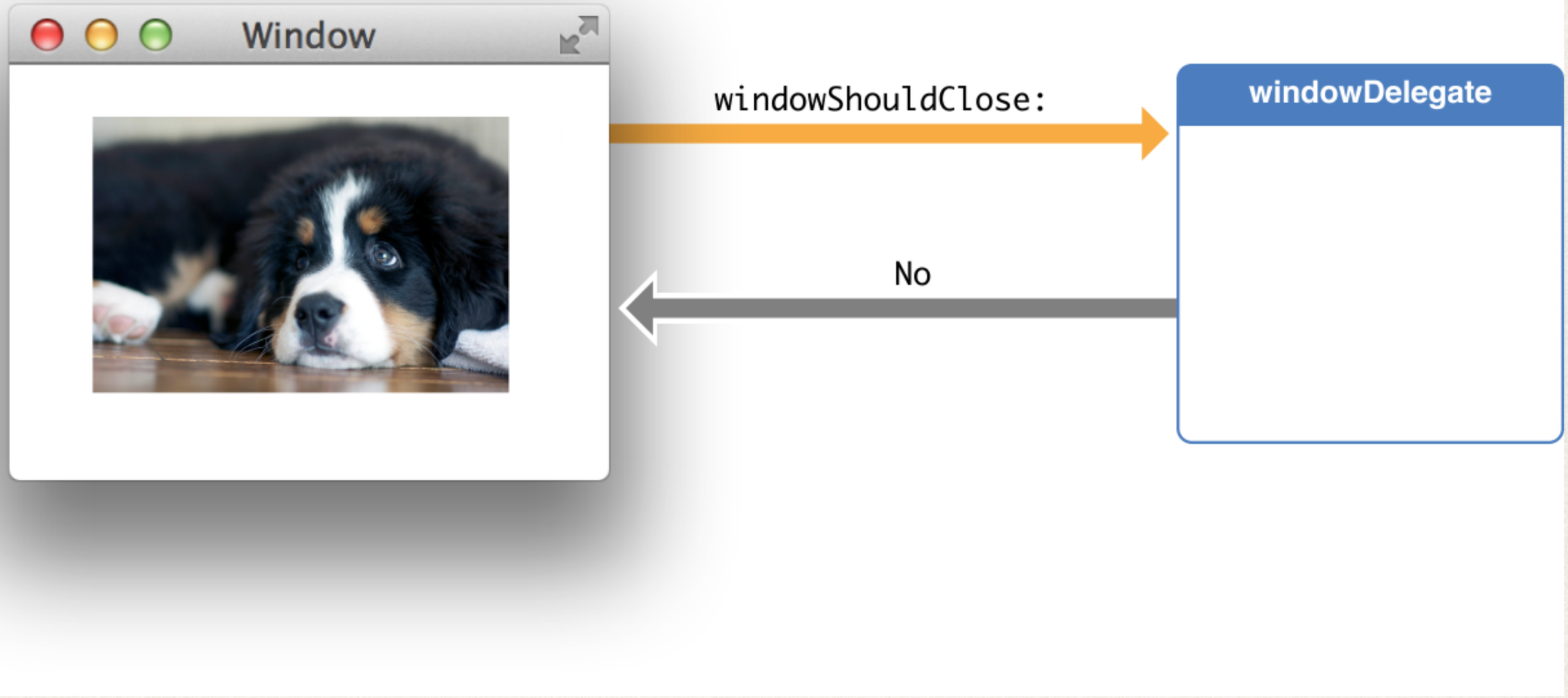
```
protocol UITableViewDataSource : NSObjectProtocol {
. . . . .

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

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

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

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


. . . . .

}
```

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


Window
windowShouldClose:
windowDelegate
No

Delegate Example

```
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(sides: 6)

    var delegate: DiceGameDelegate?

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

    func play() {

        while !gameEnded {

            let diceRoll = dice.roll()

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

        }

        delegate?.gameDidEnd(self)

    }

}
```

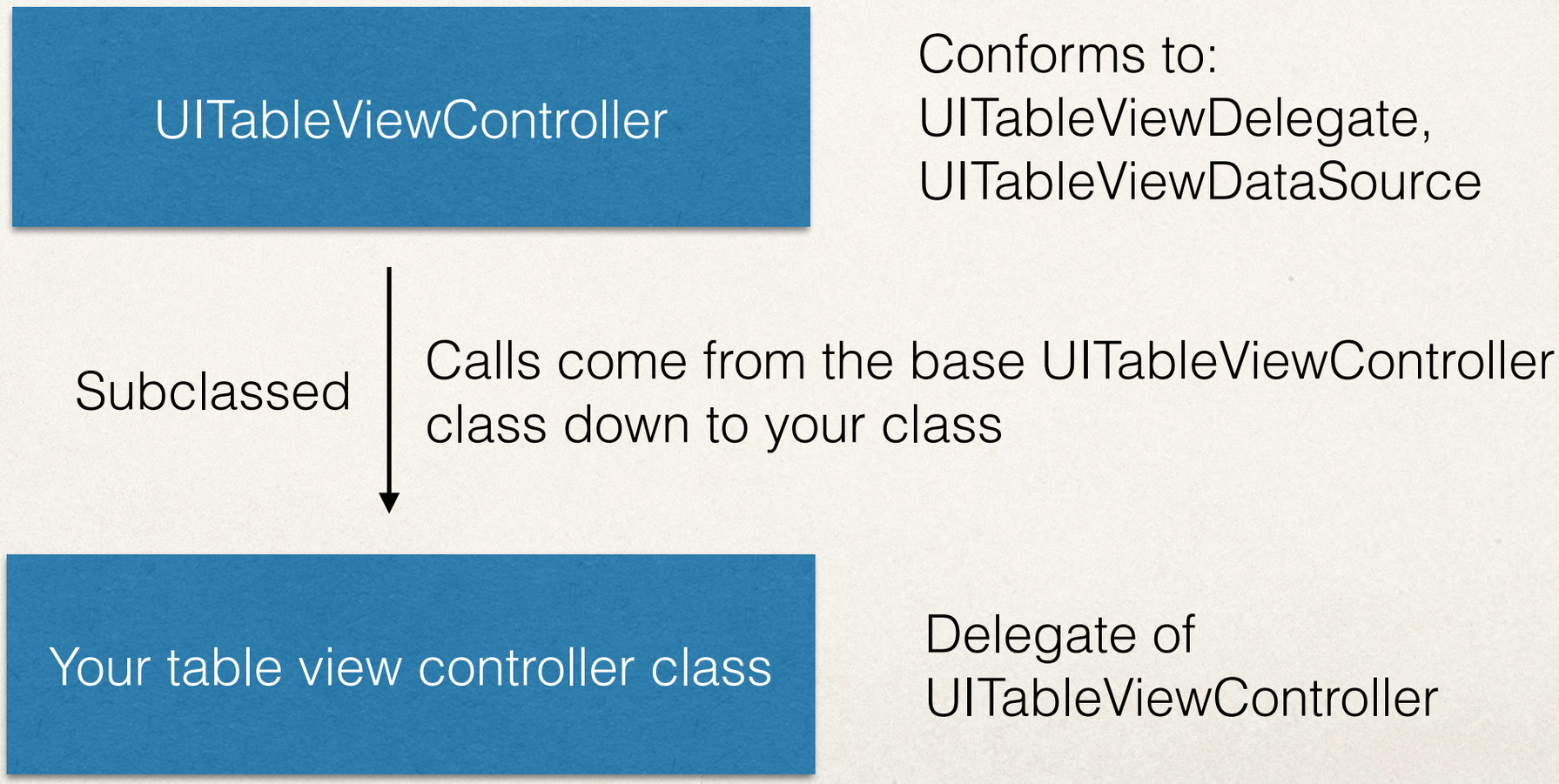
Creating a Delegate

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

```
let tracker = DiceGameTracker()

let game = SnakesAndLadders()

game.delegate = tracker

game.beginPlay()

game.play()
```

Protocols and Delegates

- Classes can use both protocols and delegates



Delegates in iOS

- Used everywhere!
- Allow UIViews to communicate with UIViewControllers
- Allow asynchronous handling of network data
- Manages lifecycle of app (UIApplicationDelegate)
  - Handles opening and closing of app
  - Performs memory management within app