MVC
and
Delegation

Model View Controller Pattern

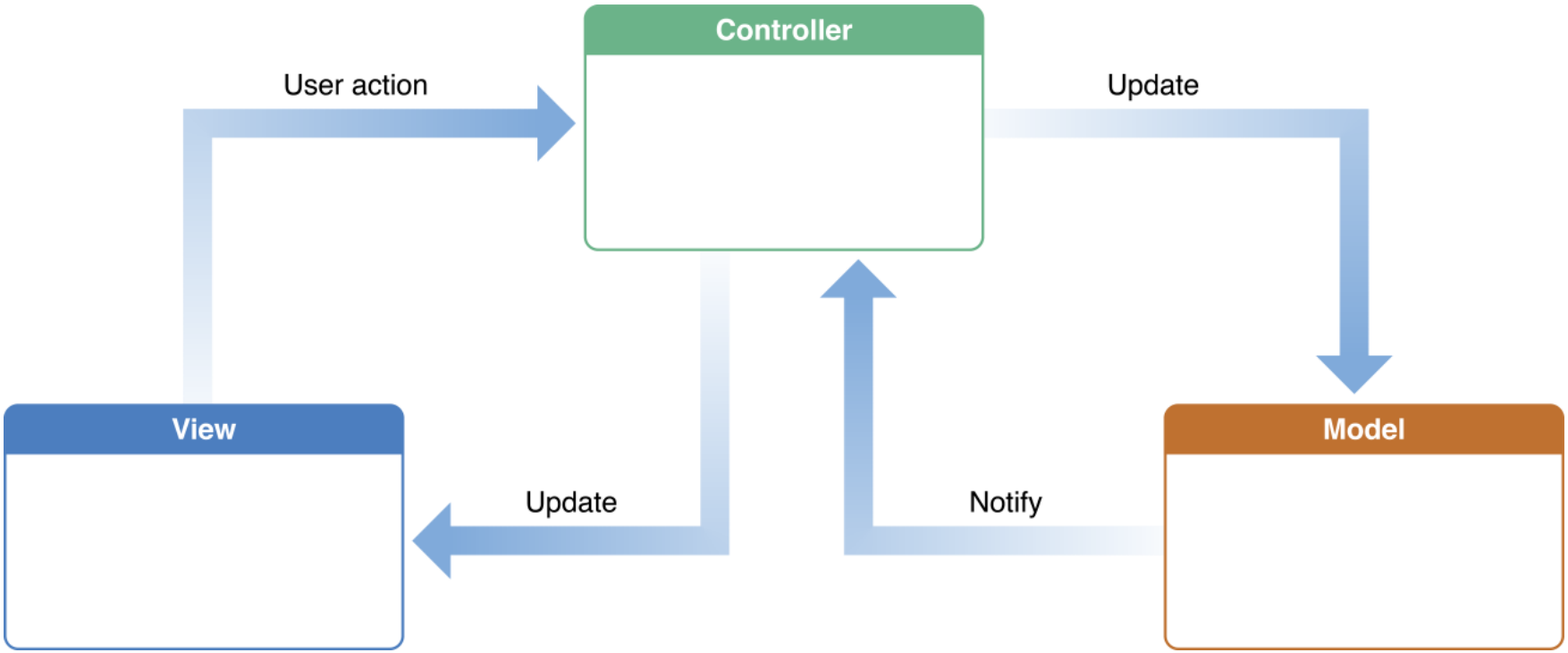
- Commonplace in iOS development
- Made up of three main objects:

    1. The **Model** is where your data resides. Things like persistence, model objects, parsers, managers, and networking code live there

    2. The **View** layer is the face of your app. Its classes are often reusable as they don't contain any domain-specific logic. For example; a `UILabel` is a view that presents text on the screen, and it's reusable and extensible

    3. The **Controller** mediates between the view and the model via the delegation pattern. In an ideal scenario, the controller entity won't know the concrete view it's dealing with. Instead, it will communicate with an abstraction via a protocol. A classic example is the way a `UITableView` communicates with its data source via the `UITableViewDataSource` protocol

Model View Controller Pattern



The Model

- Encompasses your app's data
- Usually includes classes and objects such as:
  - **Network code:** Service clients
  - **Persistence code:** Core DataFlat files
  - **Managers and abstraction layers/classes:**
    - Objects that often act as the glue between other classes or wrappers around robust APIs
    - Often hard to name 😱
  - **Data sources and delegates:**
    - Delegates may also appear in the Controller layer
    - However it is best to keep the role/function of an object as compartmentalized as possible
  - **Constants/ Variables**
  - **Helpers and extensions:**
    - Extensions are commonly kept within their own file (named for the class they extend)
    - However, if the extension will only be used in one location, it can be best to include it "in-line"
- There are more classes and objects you could include, but these seem to be the most common

The View

- The interaction layer
- Should **NOT** contain any business logic
- Typically contains objects such as:
  - `UIView` subclasses/ Classes that are a part of `UIKit`
  - Core Animation
  - Core Graphics

- When deciding whether or not to include a piece of logic in the view, the first question should always be: “Does it try to do anything not related to UI?”

`// HINT:` – The answer should ideally be “NO!”

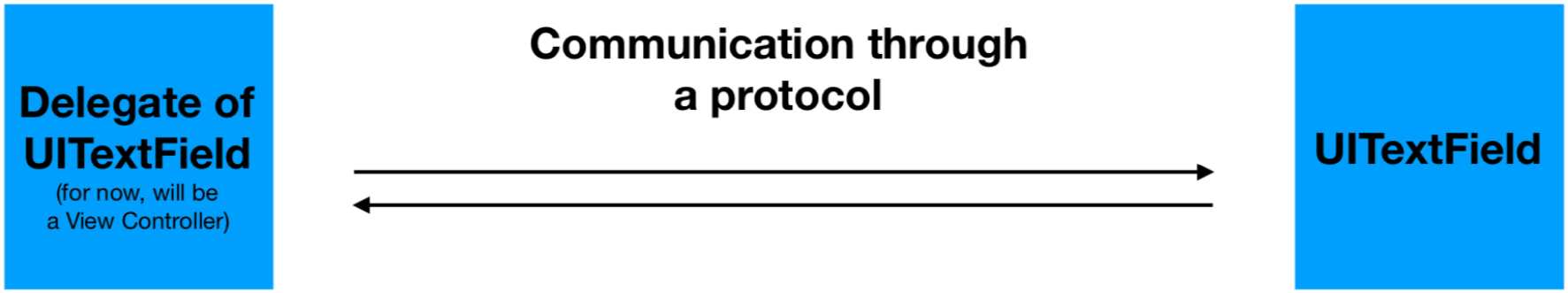
The Controller

- Least reusable layer
- Responsible for presentation logic such as:
  - Instantiating models and components
  - Lifecycle events
  - App flow
- It is typical to see this layer combined/blurred with the **View** layer in traditional iOS development implementations of **MVC**

Delegation

- An **object-oriented approach to *callbacks***: a function supplied in advance of an event and called every time the even occurs
- A design pattern used *everywhere* in iOS
- One object (**the delegate**) acts on behalf of or in coordination with the other object (**the delegating object**)
- Examples: `UITextField`, `UIPickerView`, `UITableView`

An Example of Delegation



Simplified version

```
public protocol UITextFieldDelegate {

    // return NO to disallow editing.
    func textFieldShouldBeginEditing(_ textField: UITextField) -> Bool

    // called when 'return' key pressed. return NO to ignore.
    func textFieldShouldReturn(_ textField: UITextField) -> Bool

    // called when clear button pressed. return NO to ignore
    func textFieldShouldClear(_ textField: UITextField) -> Bool

    // return NO to not change text
    func textField(_ textField: UITextField, shouldChangeCharactersIn range:
        NSRange, replacementString string: String) -> Bool

}
```

Writing a Delegate

Three general steps:

- **Write Protocol**
  - Write a ***class protocol {SomeObject's}*Delegate** with the methods/variables that should be implemented
- **Write Delegate Property**
  - Add it to the object adopting the *delegate*
- **Send Messages to the Delegate**
  - Call the protocol methods on the delegate property to send messages to it

Writing a Delegate

```
protocol GameModelDelegate: class {
    func dataUpdated()
}

class GameModel {
    weak var delegate: GameModelDelegate?

    func addSomethingToModel() {
        // do something, then notify delegate
        delegate?.dataUpdated()
    }
}
```

Using a Delegate

Three general steps:

- **Adopt Protocol**
  - Make the object that *will act as* some object's delegate adopt the ***{SomeObject's}*Delegate** protocol
- **Implement Protocol**
  - Actually write the required methods from the protocol in the class adopting it
- **Assign Delegate**
  - Set the object that has implemented the delegate protocol as the delegate of the object containing the delegate property

Using a Delegate

```
import UIKit
class ViewController: UIViewController, UITableViewDelegate {

    @IBOutlet weak var tableView: UITableView!

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.delegate = self
    }

    func tableView(_ tableView: UITableView,
                   didSelectRowAt indexPath: IndexPath) {
        // implement method
    }
}
```