University of Regina

CS 455/855
Mobile Computing

Cocoa Touch Programming

Dr. Orland Hoeber
orland.hoeber@uregina.ca
http://www.cs.uregina.ca/~hoeber/cs455/2018F

Readings

- UIKit User Interface Catalog
- Event Handling Guide for iOS

Cocao Touch

- When programming for iOS, you take advantage of a suite of frameworks provided by Apple
  - as a group, these are called Cocoa Touch
  - they are also known as UIKit, since this is the parent class
  - constitute the core iOS API

- Writing an iOS app consists of
  - using framework components as-is
  - customizing framework components by setting parameters
  - customizing framework components through subclassing
  - writing custom code that holds the data (model) and glues everything together (controller)

Cocoa Touch Classes

- Cocoa Touch includes a huge collection of classes that form the basis for your application
  - view classes
    - UIButton, UITextField, etc.
  - controller classes
    - UIViewController, UINavigationController, etc.
  - other support classes and protocols

- For many of the view classes, you will use these as they are, but perhaps with some customization by setting parameters (text, font, etc.)

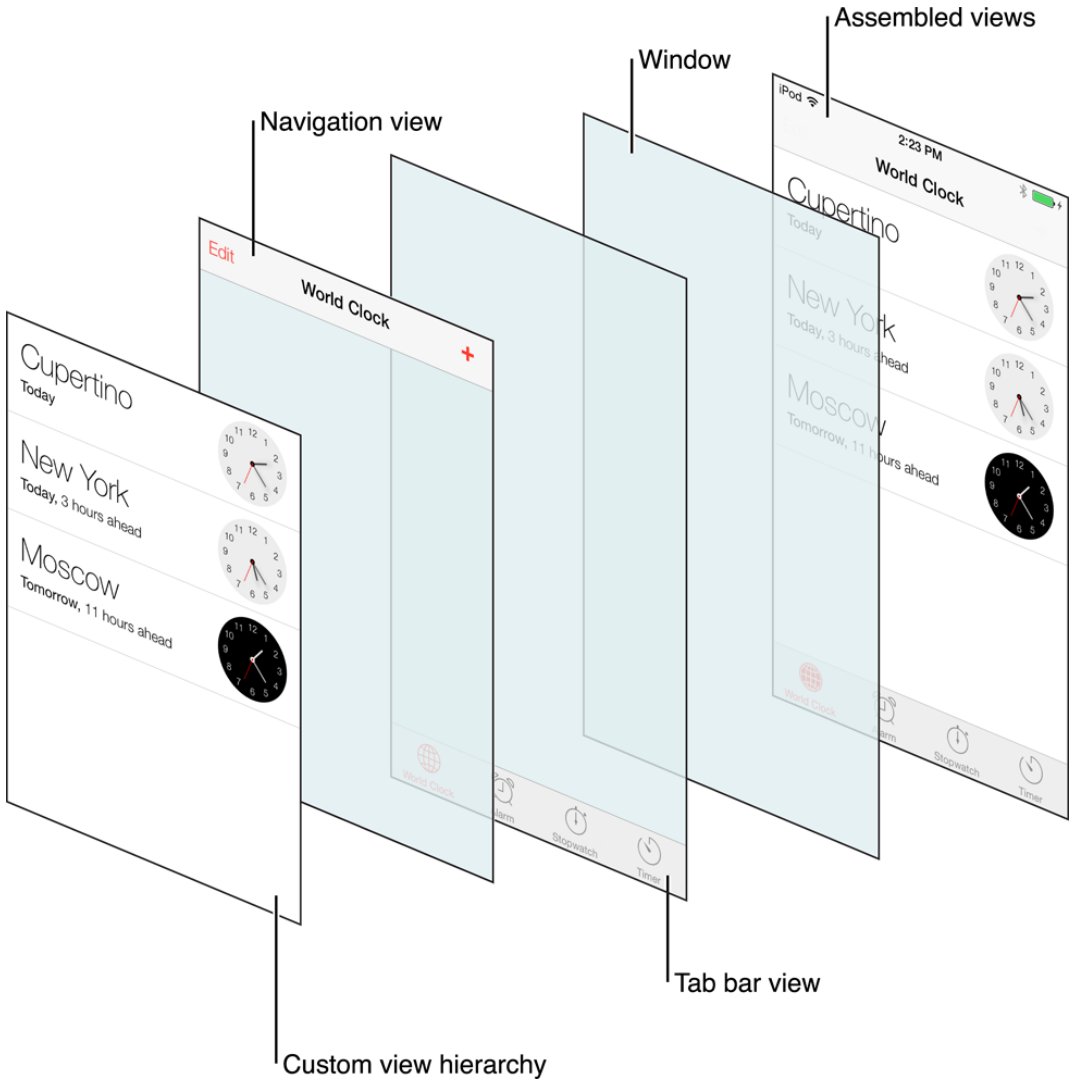
Subclassing Cocoa Touch

- A common approach when building a custom application is to extend the built-in functionality using subclassing
  - if you don't like how a UI component is working, you can create a subclass to redefine how it works
    - e.g., putting a custom border around a UILabel, building a star-rating interface component
  - an extremely common pattern is to subclass the UIViewController
    - by itself, the UIViewController doesn't appear to do much
    - it provides the starting point for you to build the code that manages events

Windows & Views

- Windows and views are the visual components you use to construct the interface of your iPhone application
  - windows
    - background platform for displaying content
    - only one window per application
  - views
    - perform most of the work of drawing and responding to user interaction
    - one or more views that hold the elements of the interface
  - when you need to change what the user sees, you make modifications to the views that are layered overtop of the window

View Hierarchy



UIView

- Views define a rectangular area on the screen
  - instances or subclasses of the UIView class
  - play key roles in both:
    - presenting your interface to the user
    - responding to interaction with the interface
  - each view object has the responsibility of rendering content within its rectangular area, and for responding to touch events in that area

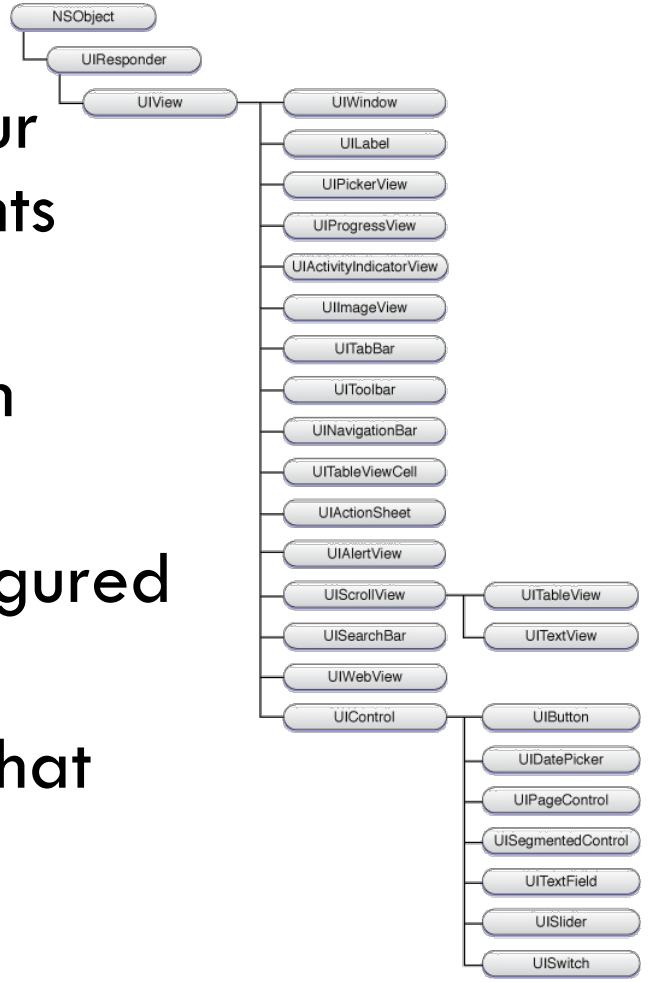
- Because of this dual behaviour, views are the primary mechanism for interacting with the user in your application

View Responsibilities

- Drawing and animating
  - views draw content in their rectangular areas
  - some view properties can be animated to new values
- Layout and subview management
  - views might manage a list of subviews
  - views define their own resizing behaviour in relation to their parent view
  - views can manually change the size and position of themselves and their subviews as needed
- Event handling
  - views receive touch events
  - views participate in the responder chain

UIKit View Classes

- UIKit defines a number of subclasses of the UIView class to define the specific appearance and behaviour for standard user interface elements
- Most of the views in this hierarchy are designed to be used as-is or in conjunction with a delegate object
- Many of the features can be configured in Xcode/Interface Builder
- You can also define custom views that inherit from the UIView



UIKit View Class Categories

- Controls
  - display specific values and handle the user interaction required to modify the value
  - e.g., UITextField, UIButton, UISwitch
- Displays
  - display information only
  - e.g., UIImageView, UILabel, UIProgressView, UIActivityIndicatorView
- Text and Web views
  - support for multi-line textual content
  - e.g., UITextView, UIWebView

UIKit View Class Categories

- Alert views and action sheets
  - views designed to get the user's attention immediately
  - e.g., UIAlertView, UIActionSheet
- Navigation views
  - support for navigating from one view to another
  - e.g., UITabBar, UINavigationBar
- Containers
  - enhance the functionality of other views
  - provide additional visual separation of the content
  - e.g., UIScrollView, UITableView, UIToolbar

View Architecture

- The standard view classes in the UIKit provide a considerable amount of behaviour to your application for free
  - e.g., animated changes in the UISwitch, keyboard for the UITextField, etc.

- They also provide well-defined integration points where you can customize the behaviour and fulfill the requirements of your application
  - these integration points are normally connected to view controllers following the target-action design pattern

View Interaction Model

- Any time a user interacts with your interface, or your code programmatically changes the data in the view, a complex sequence of events takes place inside UIKit to handle the interaction
- At specific points during this sequence, UIKit calls out to your view classes and gives them a chance to respond on behalf of your application
- Knowing how this interaction model operates is required in order to construct custom views (subclass of UIView)

Integration Points for Custom Views

- The primary integration points (overridden methods from UIView) for custom views are:
  - event-handling methods
    - `touchesBegan(_:with:)`
    - `touchesMoved(_:with:)`
    - `touchesEnded(_:with:)`
    - `touchesCancelled(_:with:)`
  - layout method
    - `layoutSubviews()`
  - drawing method
    - `drawRect()`
- You may find that you do not need to override all of these methods in your subclass of UIView (the default behaviour may be all that is needed)
- When writing the code for these, keep in mind what the MVC deigns pattern tell us (e.g., only do “view” things here; tell the controller about anything else)

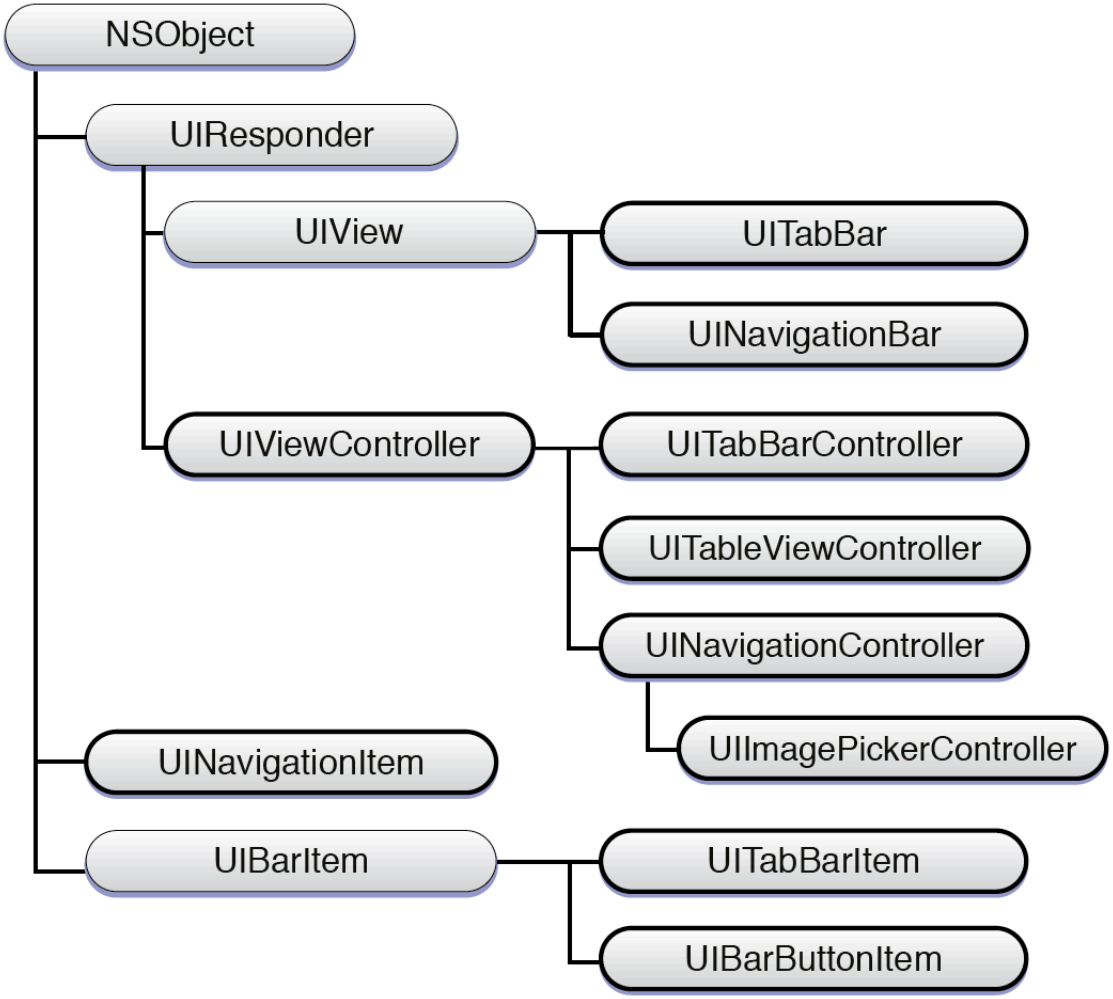
Simple Example

```
import UIKit
import os.log

@IBDesignable class TouchLogButton: UIButton {


    //MARK: Override Touch Events
    override func touchesBegan(_ touches: Set<UITouch>, with event:
UIEvent?) {
        os_log("touches began", type:.debug)
    }


    override func touchesMoved(_ touches: Set<UITouch>, with event:
UIEvent?) {
        let location = touches.first?.location(in: self)
        os_log("touch at location (%f, %f)", type:.debug, location!.x,
location!.y)


    }


}
```

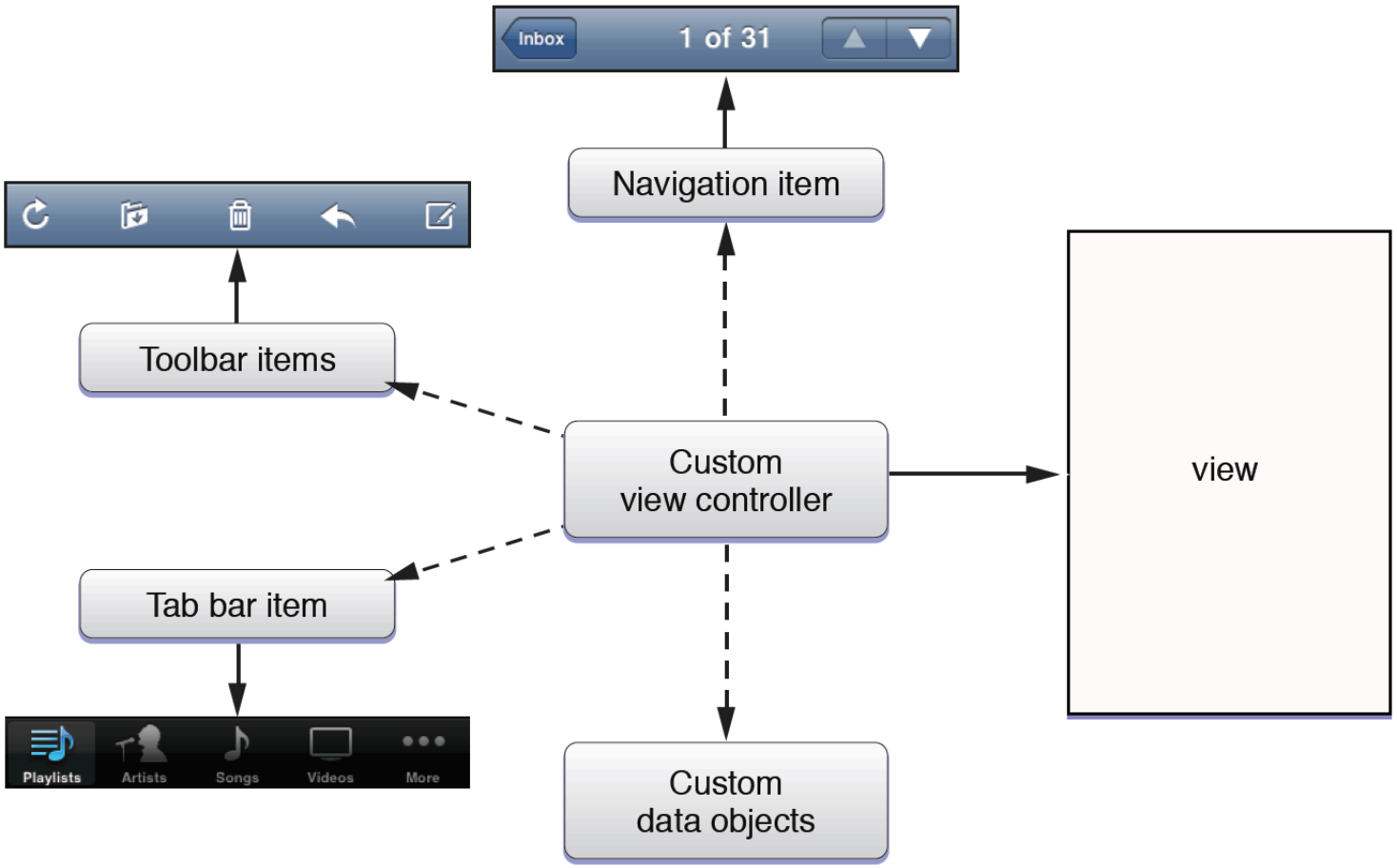
View Controllers

- View controllers are Swift classes that link the user interface controls (UIView subclasses) to the custom code that you develop to make them do interesting things
- A view controller must be defined to inherit from UIViewController
- Responsibilities:
  - create the set of views it is responsible for managing
  - flushes these views from memory in low-memory situations
  - respond to system behaviour, e.g. device orientation change
  - respond to user interaction (links for UIView classes)
- The view controller is also able to display new views modally on top of the view it controls

View Controllers Classes in UIKit



Anatomy of a Custom View Controller



Template View Controllers

- There are a number of template view controllers that are available in the Interface Builder tool:
  - Navigation Controller
    - manages navigation through a hierarchy of views
  - Tab Bar Controller
    - manages a set of view controllers that represent tab bar items
  - Table View Controller
    - manages a table view of data
  - Image Picker Controller
    - manages views for choosing and taking pictures

Actions

- An action is a message sent by an instance of a UIView subclass reporting that a significant user event has occurred within that view
  - ultimately, we will want the View Controller to do something about this
    - requires that we know which action we want to respond to and that we link the UIView subclass to the View Controller for this action
    - control-drag from the view to the controller, and select the appropriate action
  - the type of action that a view will emit will depend on the specific view (i.e., a UIButton will not send any editing-based message because it can't be edited)

Delegate Protocols

- A number of UIView subclasses defined in UIKit have somewhat complex interactions that need to be managed to work properly
  - protocols are provided to give you guidance for which methods need to be specified to support the UIView
  - whenever a delegate protocol exists for a UIView class, it is wise to make the view controller class conform to the protocol

Delegate Protocols

- Protocols for delegates of UIView classes exist for the following UIView subclasses:
  - UITabBar
  - UINavigationBar
  - UIActionSheet
  - UIAlertView
  - UIScrollView
  - UITableView
  - UITextView
  - UIWebView
  - UISearchBar
  - UITextField

More on Touch Events

- Touch events in iOS are based on a Multi-Touch Model
  - users touch the screen of the device to manipulate objects, enter data, or otherwise convey their intentions
  - iOS recognizes one or more fingers touching the screen as part of a multi-touch sequence
    - the sequence begins when the first finger touches down on the screen
    - the sequence ends when the last finger is lifted from the screen
  - for each finger touching the screen, information such as the location and the time the touch occurred is recorded

Fat Fingers? No Problem!

- When a user touches the screen, the area of contact is actually elliptical, and tends to be offset below the point where the user thinks he or she touched
- This "contact patch" also varies in size and shape based on which finger is touching the screen, the size of the finger, the pressure of the finger on the screen, the orientation of the finger, etc.
- The underlying multi-touch system analyzes all of this information for you and computes a *single* touch point
- As such, you don't need to worry about the different physical characteristics of your users – you can assume that the point you get is a reasonable location for your user's fat fingers

Built-in UI Objects

- The built-in UI objects handle the touch events in a manner that is appropriate for the object
  - buttons can be pressed
  - sliders and switches can be dragged
  - scroll views can be scrolled
  - pages can be flicked

- In most applications, there is no need for you to do anything special in terms of handling events – the default event behaviour of the interface objects is sufficient
- However, if you create your own custom UI elements (which inherit from UIView), you will need to implement methods to handle the touch events

Events & Touches

- A touch (UITouch) is the presence or movement of a finger on the screen that is part of a unique multi-touch sequence
  - e.g., a pinch-close gesture has two touches: two fingers on the screen moving toward each other
  - includes both temporal and spatial aspects
    - the temporal aspect (phase) indicates when a touch has just begun, whether it is moving or stationary, and when it ends
    - the spatial aspect indicates the location of the touch in the view or window, and the previous location (if any)

- An event (UIEvent) is an object that the system continually sends to an application as fingers touch the screen and move across its surface
  - provides a snapshot of all touches during a multi-touch sequence

Handling Touch Events

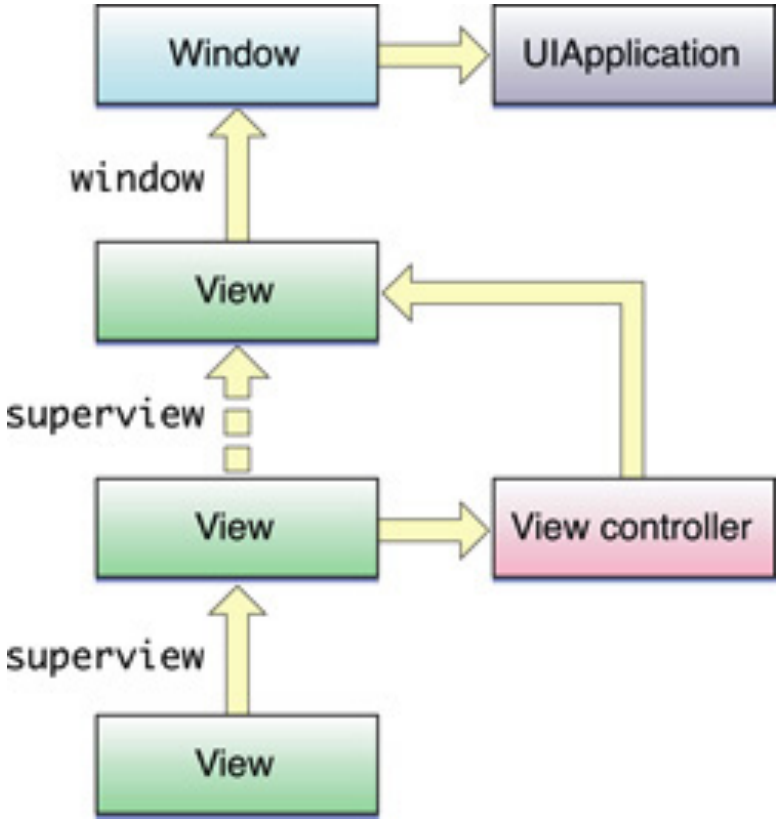
- There are generally two methods in iOS to handle touch events:
  - manually by overriding methods of UIView
    - prior to iOS 4, this was the only option
    - easy to program basic interactions (see previous example)
    - can get complex as more types of interactions are added
  - automatically by assigning a gesture recognizer to the view that is to receive the event
    - pre-defined gestures exist for basic interaction mechanisms
    - processing the gesture then becomes the responsibility of the view controller
    - new gesture recognizers can be created (and re-used)

Responder Chain

- A responder object is an object that respond to events
  - UIResponder is the base class for all responder objects
  - UIView includes UIResponder in its class hierarchy; therefore, any object that inherits from UIView can be a responder object
- The responder chain is a linked series of responder objects
  - from subview to superview (foreground to background)
  - it defines the order in which events are processed

Handling a Touch Event

- The receiving view is given the first chance to respond to a touch event
- If it chooses to not respond, the event is then passed through the responder chain, seeking a responder that is capable of handling the touch event
- If no view (or UIWinow or UIApplication) is willing to handle the event, it is discarded



Touch Event Handling Code

- In order for a UIView subclass to handle a touch event, it must override one or more of the following methods:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {}
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {}
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {}
override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?) {}
```

  - each method receives two parameters:
    - a set of UITouch objects (one for each finger on the object)
    - a UIEvent object that represents the entire multitouch sequence

  - as the touch events proceed, it is these same UITouch and UIEvent objects that are mutated and passed back into these methods

UITouch Object

- When overriding the touch methods, we will make extensive use of the properties and methods of the UITouch objects
- Some of the useful ones are:
  - location(in:)
  - previousLocation(in:)
  - tapCount
  - timestamp
  - phase

Example: moving an object

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {

    // get the current and previous locations
    let location = touches.first?.location(in: self.superview)
    let oldLocation = touches.first?.previousLocation(in: self.superview)

    if (oldLocation != nil) {
        // find the distances moved
        let deltaX = location!.x - oldLocation!.x
        let deltaY = location!.y - oldLocation!.y

        // get the center of self and update it
        var c = self.center
        c.x += deltaX
        c.y += deltaY
        self.center = c
    }
}
```

Example 2: responding to a tap

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    os_log("touches began", type:.debug)

    // save the start time of the tap
    touchStart = touches.first!.timestamp
}

override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    // find out how long the tap lasted
    let lengthOfTouch = touches.first!.timestamp - touchStart

    // if the tap is long, do something about it
    if (lengthOfTouch > 2) {
        self.backgroundColor = UIColor.red
    } else {
        self.backgroundColor = UIColor.blue
    }
    self.setNeedsDisplay()
}
```

Merging Multiple Interactions

- Things start to get rather tricky when we need to manage multiple different types of touch events within the same object
  - everything needs to be in the four touches… methods
  - your code needs to keep track of the various states
  - even things like detecting a single-tap from a double-tap becomes surprisingly complex
    - need to delay the activity of the single tap until we know for sure that there is no other tap coming in
    - must hard-code in the time interval limit for a double-tap
- The solution to this complexity problem is the use of gesture recognizers

Gesture Recognizers

- A gesture recognizer (subclass of UIGestureRecognizer) is an object that is attached to a UIView for the purposes of managing the touch events
  - if a new event is to be delivered to a view, it is also delivered to the attached gesture recognizers
  - each gesture recognizer maintains its own state in order to determine the type of gesture that is occurring
    - no need to integrate state-maintenance code within the view itself
    - allows the gesture to be easily added to other views

Continuous vs. Discrete Gestures

- Some gestures need to tell the class that something is happening continuously
  - pinch
  - rotate
  - pan/drag
- Others detect discrete events
  - tap
  - swipe
  - long press

- When a gesture is recognized, it either sends a series of messages (continuous) or a single message (discrete)

Pre-built Gesture Recognizers

- There are a number of pre-built gesture recognizers in iOS (all subclasses of UIGestureRecognizer)
  - UITapGestureRecognizer
  - UIPinchGestureRecognizer
  - UIRotationGestureRecognizer
  - UISwipeGestureRecognizer
  - UIPanGestureRecognizer
  - UILongPressGestureRecognizer

Adding a Gesture Recognizer

- There are two ways to add a gesture recognizer to an UIView object
  - programmatically
    - create a new instance of the gesture recognizer during the initialization of the object
    - use the self.addAddGestureRecognizer method
    - can only be done if you sub-class the interface object
  - using Interface Builder
    - drag the gesture recognizer onto the object
    - configure the properties of the gesture recognizer
    - control-drag from the gesture recognizer to the view controller to create an @IBAction
    - put the code of what to do into the method

Example (using IB)

- Drag the Pan Gesture Recognizer onto a Button
- Link it to the View Controller Class
- Write the method to tell it what to do
- within the UIView subclass initializer

```
@IBAction func moveButton(_ sender: UIPanGestureRecognizer) {
    let delta = sender.translation(in: self.view)
    var c = moveableButton.center
    c.x += delta.x
    c.y += delta.y

    moveableButton.center = c

    sender.setTranslation(CGPoint.zero, in: self.view)
}
```

Handling Multiple Gestures

- As we can see from this example, handling a single gesture is simple
- Handling multiple gestures follows the same pattern:
  - add another gesture recognizers to the object
  - configure the settings of the gesture recognizer (e.g., one detecting a single tap, another detecting a double-tap)
  - let the gesture recognizers detect the difference between each gesture
    - once a gesture recognizer succeeds in recognizing its gesture, any other gesture recognizers associated with its touches are forced into a failed state

Custom Gesture Recognizers

- Creating custom gesture recognizers can be a little complex
  - use the default gesture recognizers whenever possible
  - if you need custom functionality, you can create subclasses of UIGestureRecognizer
    - add in your own state instance variables/properties
    - override the core methods that manage the gesture detection
      - these are the same touches… methods we used when managing the touch events directly within a UIView subclass
    - override other methods as necessary to implement the desired functionality

Homework

- Next topic: Sensor Programming

- Short Paper # 1 (CS 855)
  - due Friday October 12
- Project Milestone 2: Project Design
  - due Wednesday October 17
- Assignment #2
  - due Friday October 26