macOS Development

macOS 10.14 • Xcode 10

STUDENT GUIDE

ABOUTOBJECTS

Contact

About Objects
11911 Freedom Drive
Suite 700
Reston, VA 20190

main: 571–346–7544
email: info@aboutobjects.com
web: www.aboutobjects.com

Course Information

Author: Jonathan Lehr
Revision: 1.0.0
Last Update: 9/14/2019

Classroom materials for a course that provides a rapid introduction to macOS development. Geared to developers interested in learning Cocoa programming on the Macintosh platform.

Copyright Notice

macOS Development

STUDENT GUIDE

About the Author

Jonathan Lehr

Co-Founder and VP, Training
About Objects

jonathan@aboutobjects.com

ABOUTOBJECTS

Section 1: macOS Development

Section 2: Core AppKit Components

NSApplication

- Workspaces and bundles
- The application’s delegate
- Run loops
- Role in event distribution
- Methods to override

NSWindow

- Frame view and content view
- Role in event distribution
- Responder behavior
- The window’s delegate

NSWindowController

- Nib file loading
- Window management
- Working with documents

NSView

- View geometry
- View hierarchy
- Drawing behavior

NSViewController

- View loading (nib file and programmatic)
- Life cycle management
- Controller hierarchy

Nib Files and Storyboards

- Archiving objects via the NSCoding protocol
- How Interface Builder components fit in the design of the AppKit API

Section 3: Windows and Views

Windows and Panels

- An NSWindow instance is a two views:
  - A *frame view*, which is private
  - A *content view*, which is available for you to customize by adding your own subviews
- Windows are owned by NSApp in its windows list
- NSPanel is a subclass of NSWindow whose instances appear as auxiliary windows
  - Panels can be configured to float above ordinary windows.
- NSApp tracks which of its windows currently:
  - Gets keyboard events (`keyWindow`)
  - Is acted on by panels (`mainWindow`)

Instantiating a Window

Instantiating an NSWindow via its designated initializer:

```
let windowRect = NSRect(x: 240, y: 800, width: 0, height: 0)
mainWindow = NSWindow(contentRect: windowRect,
                      styleMask: [.titled, .closable],
                      backing: .buffered,
                      defer: true)
```

Configuring a window programmatically:

```
mainWindow.title = NSLocalizedString("Books", comment: "window title")
mainWindow.isReleasedWhenClosed = false
mainWindow.contentViewController = EditBookController()
```

Setting a window’s `contentView`:

```
guard let frameRect = window.contentView?.frame else {
    return
}
let backgroundView = NSView(frameRect)
window.contentView?.addSubview(backgroundView)
```

The Window's Delegate

- The NSWindowDelegate protocol is declared in **NSWindow.h**
- A window notifies its delegate of changes to its state.
    - In some cases, the delegate can override default window behavior.
    - For example, a delegate implementation of `windowShouldClose(_:)` could return `false` is certain situations to prevent the window from closing.
- Here’s a small sampling of methods declared in the protocol:

```
public protocol NSWindowDelegate : NSObjectProtocol {

    optional func windowShouldClose(_ sender: NSWindow) -> Bool
    optional func windowWillClose(_ notification: Notification)

    optional func windowWillResize(_ sender: NSWindow,
                               to frameSize: NSSize) -> NSSize
    optional func windowDidResize(_ notification: Notification)

    optional func windowWillMove(_ notification: Notification)
    optional func windowDidMove(_ notification: Notification)

    optional func windowDidBecomeKey(_ notification: Notification)
    optional func windowDidResignKey(_ notification: Notification)

    optional func windowDidBecomeMain(_ notification: Notification)
    optional func windowDidResignMain(_ notification: Notification)
```

Window Ordering

NSApplication provides an API for accessing the window list, as well as specific items in the list.

```
open var windows: [NSWindow] { get }

weak open var mainWindow: NSWindow? { get }
weak open var keyWindow: NSWindow? { get }

open var modalWindow: NSWindow? { get }
```

NSWindow provides API for reordering the list dynamically.

```
open func orderFront(_ sender: Any?)
open func orderOut(_ sender: Any?)

open func makeKeyAndOrderFront(_ sender: Any?)

open func orderBack(_ sender: Any?)
open func order(_ place: NSWindow.OrderingMode,
                relativeTo otherWin: Int)
```

Key Window and Main Window

- The *key window* is the window that currently has keyboard focus, if any.
- If no window is currently key, keyboard events go directly to the application object.
- The *main window* is the window that currently would be the default target of panels, if any.

Understanding Window Levels

- Windows, panels, and menus live in different tiers, or levels, in the windowing system.
- By default, panels are in the same level as regular windows
  - However you can set a panels's `isFloatingPanel` property to `true` to move it to a higher window level.
  - This will make the panel float above other windows.
- Similarly, menus are in an even higher level, and therefore float over windows and panels.
- Your code can't directly change these levels, but you can change the ordering of windows within the window level, and floating panels in the panel level.

Saving State to User Defaults

- Apps should generally persist certain aspects of UI state on behalf of the user.
- At a minimum, window sizes and positions should be saved.
- Application state is typically saved via the User Defaults system.
- NSUserDefaults provides a simple API for storing and retrieving values in the user's home library directory.
- Here's part of the base API:

```
open class UserDefaults : NSObject {

    // returns a global instance of NSUserDefaults.
    open class var standard: UserDefaults { get }

    // searches for a value stored under the provided key.
    open func object(forKey defaultName: String) -> Any?
    // stores (removes if nil is passed) a value for the provided key.
    open func set(_ value: Any?, forKey defaultName: String)
```

- A window automatically saves its size and position to User Defaults if you set its `frameAutosaveName` property with an arbitrary string value.

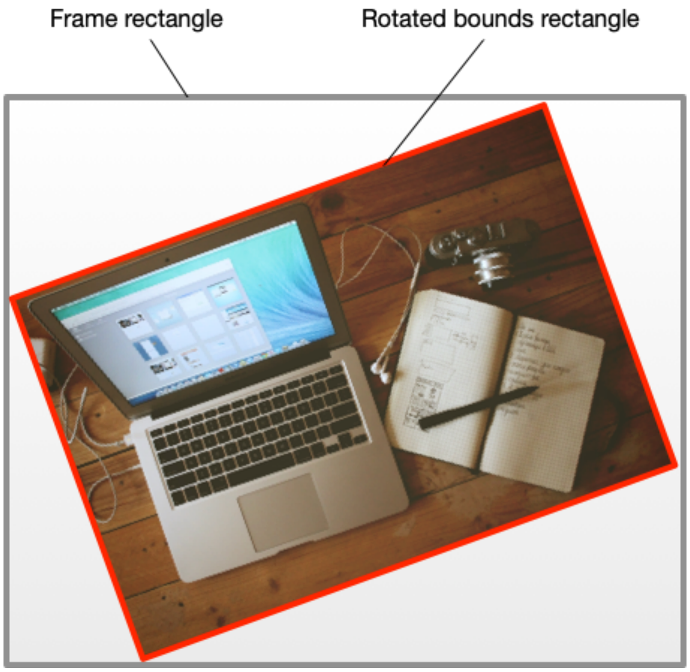
Section 4: Views and Responders

NSView

- NSView inherits its event-handling behavior from NSResponder.
- Additional responsibilities:
  - Geometry
  - Hierarchy
  - Drawing
- View drawing behavior is:
  - CPU-based
  - Lazy — window sends `display()` messages down the view hierarchy; display only calls `drawRect(_:)` if a view's rectangle is marked as dirty.
  - Layer-backed views take advantage of significant, more modern performance optimizations that take better advantage of the graphics hardware (covered later along with Core Animation).

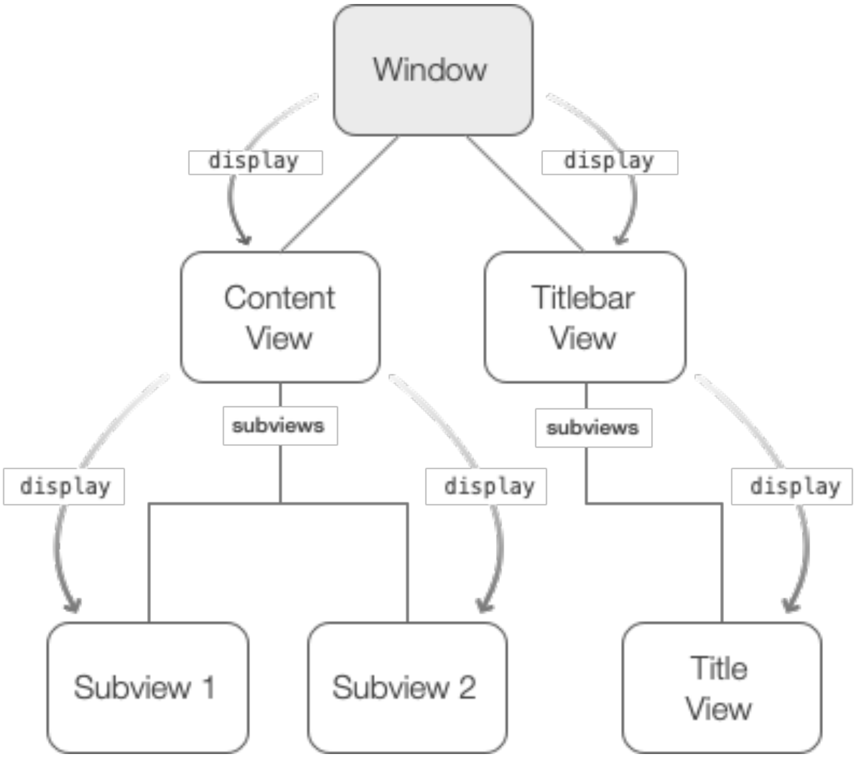
View Geometry

- Views use two different but related coordinate systems.
    - The `bounds` property's origin represents the coordinates a view uses to draw its content. By default, it's **0, 0**.
    - The `frame` property's origin represents where the lower left corner of the view is positioned relative to its superview.
- The *sizes* of a view's bounds and frame are ordinarily the same.
    - They differ only when the view is rotated.
    - When rotated, the frame may temporarily get larger in order to fully fit the rotated bounds.



The View Hierarchy

- Forward and backward chained via NSView's `superview` and `subviews` properties.
- Responsibilities include:
  - Hit testing (determining which view an event is associated with)
  - Forwarding event messages to the next responder
  - Propagating `display` messages down the hierarchy to allow subviews to redraw, if necessary.



Working with Subviews

- Use the `addSubview(_:)` method to add a view to another view’s `subviews` array.
  - Note that `addSubview(_:)` does some important housekeeping we’ll learn about later.
- Here’s an example that adds a new instance of NSView to a window’s content view:

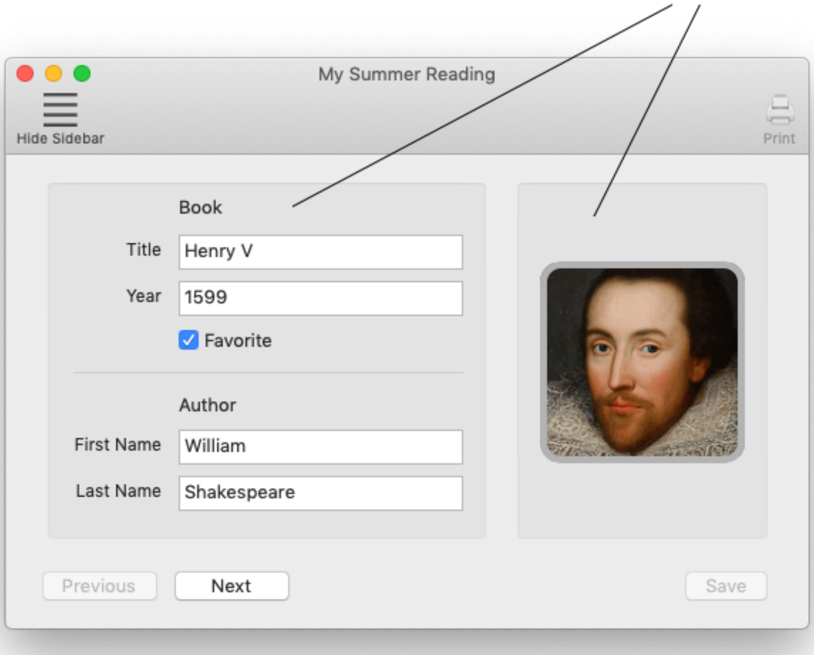
```
let frameRect = window.contentView?.frame ?? CGRect.zero
let backgroundView = NSView(frame: frameRect)


window.contentView?.addSubview(backgroundView)
```

NSBox

- Use NSBox to group related subviews in your UI.
- Once you've created an instance of NSBox, you can add your custom views to its `contentView`.
- NSBox objects have a title property you can configure, if desired.
- You can also customize their corner radius, border width, border color, and fill color.

NSBox instances



Title Henry V

Year 1599

☑ Favorite

First Name William

Last Name Shakespeare

Custom Drawing

- You can override NSView's `draw(_:)` method if you want to make 2D drawing calls directly from your code.
    - Note that this can incur significant overhead, partly because `draw(_:)` does its work in the CPU.
    - Avoid doing any unnecessary work in your `draw(_:)` implementations.
- Views draw themselves lazily.
    - By default `display()` will only call `draw(_:)` when a view's internal state changes (for example, if its bounds rectangle changes).
    - When the view hierarchy propagates `display()` messages, `display()` checks an internal flag to see if the view needs to be redrawn.
    - You can set this flag when necessary (i.e., if the view's bitmap is stale) by calling `setNeedsDisplay(_:)`.

Resizing Behavior

- NSView has an `autoresizingMask` property you can configure to control how an instance lays itself out relative to its superview.
    - Allows you to allow or disallow auto resizing in each axis.
    - Also allows you to pin one or more edges a constant distance from the corresponding edges of the superview.

Layer-backed Views

- Setting the `wantsLayer` property on a view causes the view and all its subviews to be backed by instances of CALayer.
    - A layer-backed view's CALayer instance provides the backing store for the view's rendered content and most of its properties.
    - The bitmap produced by the `draw(_:)` method is automatically stored in the view's layer.
- If the read-only `wantsUpdateLayer` property returns `true`, the parent view will use a different drawing path.
    - `updateLayer()` will be called in addition to or instead of `draw(_:)`.
    - This allows layer instances to share references to a single image instead of copying the image data.
    - Also avoids drawing backgrounds, borders, shadows, etc. in the CPU.

For more information:
Layer-Backed Views: AppKit + Core Animation - WWDC 2012 - Videos

CALayer

- An instance of CALayer can provide a *backing store* for a view's rendered bitmap.
    - The GPU can perform animations with a layer's bitmap directly in graphics hardware by applying transforms such as scaling, rotation, and translation.
    - A layer also provides storage for many of the view's properties.
- CALayer instances have additional properties that define visual state that can be rendered in the GPU for items such as borders, background color, shadows, and corner radius.
- Also provide methods for adding and removing instances of CAAnimation subclasses that define animation effects.
    - Configuring animations this way is referred to as *explicit animation*.
    - Even easier is working with *implicit animations*, which are triggered by simply changing properties of a view such as its frame or transform. (We'll cover that shortly.)

Section 5: Handling Mouse Events

NSResponder Event-Handling

- Mouse-tracking behavior is defined in the **NSResponder** class.
- NSResponder also defines handling for keyboard events, trackpad events, and Touch Bar events.

NSEvent

- An NSEvent is a wrapper for a hardware event received by the app's main event loop (an instance of NSRunLoop).
- NSApp packages raw event info in an NSEvent instance and then dispatches to the appropriate object (usually a window).
- Windows typically then dispatch an event message to one of their views.

Mouse Moved Events

- Not dispatched to windows by default for performance reasons.
- Toggle `acceptsMouseMovedEvents` property to enable, if/when needed.
- Methods to override:

```
open func mouseMoved(with event: NSEvent)
```

Tracks all cursor motion in a view's bounds.

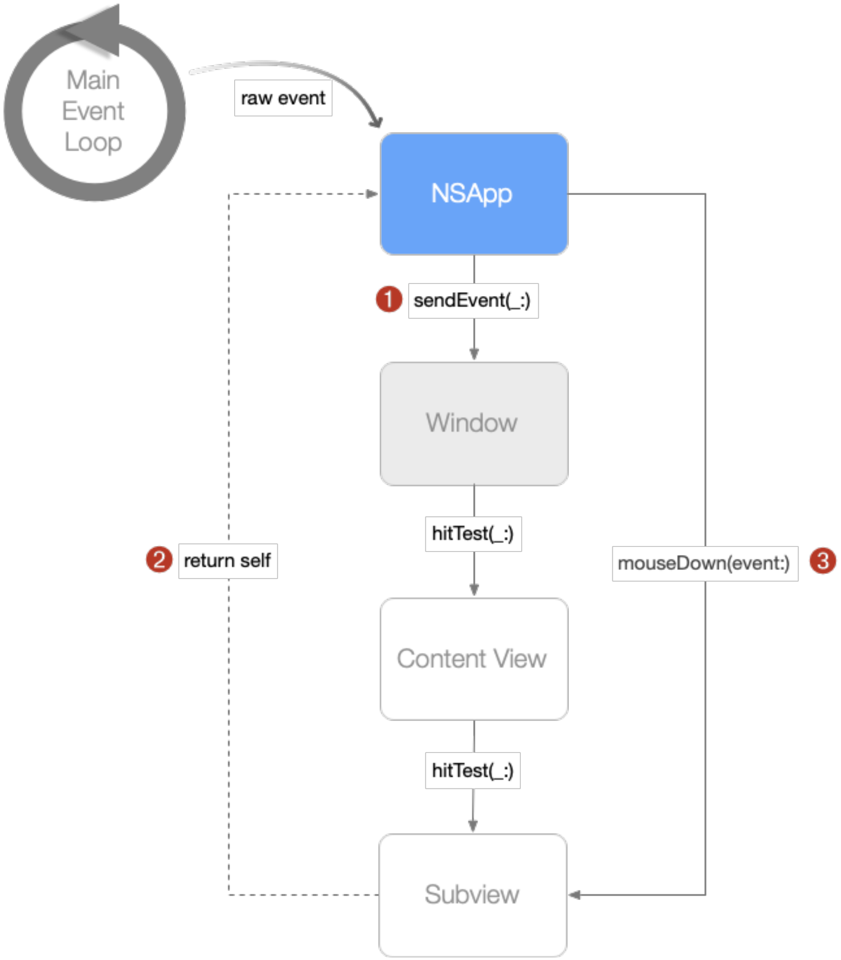
- Potentially significant performance impact.
- By default, windows ignore these events.

```
open func mouseDragged(with event: NSEvent)
```

Tracks all cursor motion in a view's bounds between *left mouse down* and *left mouse up* events.

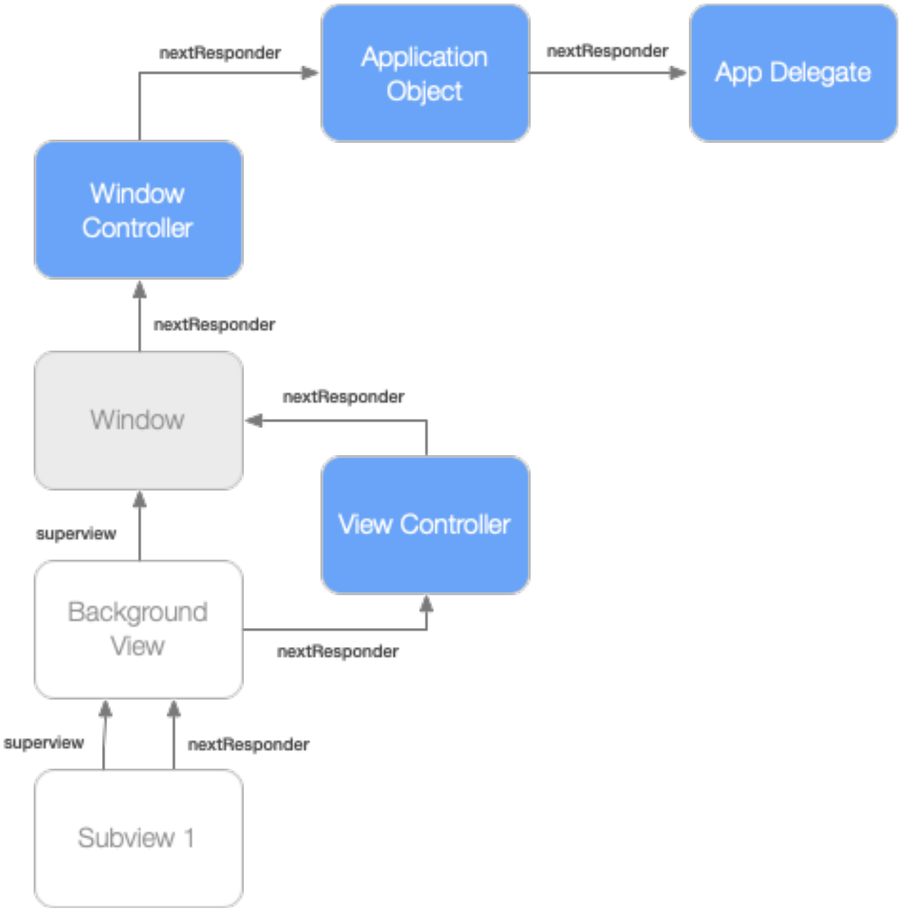
Hit Testing

1. NSApp sends a `sendEvent(_:)` message to the frontmost window whose frame corresponds to a raw event's coordinates.
2. The window performs hit testing, which recursively descends the view hierarchy.
3. The window dispatches an event message to view returned by the hit test message.



The Responder Chain

- NSResponder class provides a `nextResponder` property via which responders are chained together.
- If a responder object receives an NSResponder message such as `mouseDown(event:)` that it doesn't want to handle, it can forward the message to the next responder in the chain. (In fact, this is the default behavior.)



Gesture Recognition

- Subclasses of NSGestureRecognizer automatically recognize complex gestures.
- You configure a gesture recognizer's `target` and `action` properties.
    - When a gesture is recognized, the recognizer will invoke the provided action method on the target object.
- A few commonly used subclasses:
    - NSClickGestureRecognizer
    - NSPanGestureRecognizer
    - NSRotationGestureRecognizer

Section 6: Working with Core Animation

Implicit Animations

- Core Animation automatically animates changes to *animatable* properties of views and layers.
    - However, implicitly animating changes to layer properties of layer-backed views is fraught with peril.
    - You could instead use a layer-hosted view, and live with the limitation that layer-hosted views are leaf nodes in the view hierarchy.
- The following example shows how to take advantage of implicit animation in an NSView subclass:

```
// Enable implicit animation
NSAnimationContext.current.allowsImplicitAnimation = true

// Set animation properties
NSAnimationContext.current.duration = duration

// Modify animatable properties
frame = frame.offsetBy(dx: size.width, dy: -size.height)
frameRotation = -90

// Add a completion handler, if needed
NSAnimationContext.current.completionHandler = { [weak self] in
    self?.doSomethingCoolBecauseTheAnimationHasFinished()
}
```

For More Information:

OS X Development - A Core Animation Manifesto

OS X Development - A short guide to OS X animations

AppKit for UIKit Developers · objc.io

Section 7: Controllers

NSViewController

- NSViewController manages a view that it loads lazily and stores in its `view` property.
    - By default, it loads its view from a nib file.
    - You can change this behavior by overriding a controller’s `loadView()` method.
- After lazily initializing its view, a view controller calls its own `viewDidLoad()` method.
    - Gives your code an opportunity to do further initialization (e.g., populating views with data).
- View controllers have API for interacting with other view controllers.
    - Can have parent-child relationships.
    - Can present another view controller modally.

View Controller Lifecycle Methods

- View controllers automatically receive appearance notifications when their view hierarchy appears or disappears.

```
open func viewWillAppear()

open func viewDidAppear()

open func viewWillDisappear()

open func viewDidDisappear()
```

- View controllers are also notified of layout changes. You can override these methods to add side effects, or to customize layout dynamically.

```
open var preferredContentSize: NSSize

open func viewWillLayout()

open func viewDidLayout()

open func updateViewConstraints()
```

Modal Presentation

- When a view controller is presented modally it blocks interaction with other windows of the current application.
    - Forces user to interact with modally presented item before continuing to interact with other content.
- Built-in support for presenting another VC:
    - In a modal window
    - As a sheet in the current window.
- You can programmatically dismiss the presented view controller in response to user actions.

Presenting and Dismissing Modals

- The following methods present and dismiss a view controller temporarily:

```
        open func present(_ viewController: NSViewController, animator: 
NSViewControllerPresentationAnimator)

        open func dismiss(_ viewController: NSViewController)
```

- The framework provides wrapper methods for presenting a view controller that provide pre-defined animations and that:
    - Set the presenting view controller's `delegate` and `contentViewController` properties to point to the presented view controller.
    - Call `present(_:animator:)` for you, providing an appropriate animator object.

```
    open func presentAsSheet(_ viewController: NSViewController)

    open func presentAsModalWindow(_ viewController: NSViewController)

    open func present(_ viewController: NSViewController,
          asPopoverRelativeTo positioningRect: NSRect,
          of positioningView: NSView,
          preferredEdge: NSRectEdge, behavior: NSPopover.Behavior)
```

Working with NSNotificationCenter

- Use **NSNotificationCenter** to broadcast notifications to any interested observers.
- Each thread has its own instance of **NSNotificationCenter**.
    - **Important:** Make sure that notifications that need to be handled in the main thread are posted to the main thread's notification center.
- Register and unregister for notification callbacks as needed.
    - **Important:** A registered observer must remove itself from any notification centers before being freed.

NSWindowController

- An NSWindowController manages a single window.
    - Typically, window controllers load their window from a nib file.
- Responsibilities include:
    - Loading and displaying the window
    - Closing the window
    - Managing the window’s title
    - Persisting the window’s frame in User Defaults.

Section 8: Controls and Cells

NSControl

- Abstract subclass of NSView.

- AppKit provides numerous concrete subclasses, such as NSButton, NSSlider, NSTextField, etc.

- A control works by sending an *action message* to its *target* object on activation (e.g., when a button is clicked).

```
open class NSControl : NSView {

    weak open var target: AnyObject?

    open var action: Selector?
```

The Target-Action Paradigm

- Surprisingly, controls don’t dispatch action messages themselves.
- Instead they call the following method on `self`:

```
open func sendAction(_ action: Selector?, to target: Any?) -> Bool
```

- The above method, in turn, calls a similar method on NSApplication, which performs the actual dispatch.
- This allows NSApp to handle nil-targeted action messages by forwarding them to the key window’s first responder.

```
open func sendAction(_ action: Selector, to target: Any?,
                     from sender: Any?) -> Bool
```

- This capability is important because menu items and other UI elements rely heavily on it.

The Objective-C Runtime System

- Objective-C provides a powerful set of runtime introspection capabilities.
    - Implemented in `libobjc`, a C library.
- The library exposes a public API you can call to many, powerful things at runtime, such as:
    - Changing a method's implementation (method swizzling)
    - Changing the class of an object (isa swizzling)
    - Modifying a class's dispatch table
    - Dynamically creating new classes in memory
    - Using introspection to discover properties and invoke methods
- Foundation, AppKit, Interface Builder, and other Cocoa frameworks and tools rely heavily on these capabilities.

Invoking Methods via Introspection

- The NSObject class provides wrapper methods that call into the Objective-C runtime on your behalf.
- Objective-C objects inherit these introspection capabilities, including the ability to:
  - Discover properties and methods dynamically
  - Invoke methods by name
  - Dynamically access property values by property name
- This is an enabling technology for target-action and a number of other fundamental Cocoa mechanisms.

NSCell

- Performance optimization designed for the original NeXT implementation of AppKit.
- Used in complex views, such as controls and table views:
    - Provide cheaper storage for view state.
    - Can be reused.
- NSMatrix is a control designed to contain a grid of NSCell instances, but is now deprecated.
- Apple has been gradually deprecating NSCell.

Handling Value Changes

Observing Value Changes

- Instances of NSControl post `valueDidChange` notifications whenever their values change.

- Add your object as observer in NotificationCenter to receive notification messages, for example:

```
override func viewDidLoad() {
    super.viewDidLoad()
    NotificationCenter.default.addObserver(self,
                                           selector: #selector(myMethod(_:)),
                                           name: NSControl.textDidChangeNotification,
                                           object: nil)
}
```

Managing UI State

- Update NSWindow’s `setDocumentEdited` flag to reflect whether the window currently has unsaved changes.

- Enable/disable controls accordingly.

Section 9: Handling Keyboard Events

Keyboard Interface Control

- Cycling through the key view loop
- Selecting controls

Key	Effect
Tab	Move to next key view.
Shift-Tab	Move to previous key view.
Space	Select, as with mouse click in a check box (for example), or toggle state. In selection lists, selects or deselects highlighted item.
Arrow keys	Move within compound view, such as NSForm objects.
Control-Tab (Control-Shift-Tab)	Go to next (previous) key view from views where tab characters have other significance (for example, NSTextView objects).
Option or Shift	Extend the selection, not affecting other selected items.

Manipulating the key view loop

- Cycling through the key view loop
- Selecting controls

TODO: Swift version:

```
- (void)textDidEndEditing:(NSNotification *)notification {
    NSTextView *text = [notification object];
    unsigned whyEnd = [[[notification userInfo] objectForKey:@"NSTextMovement"]
unsignedIntValue];
    NSTextView *newKeyView = text;

    // Unscroll the previous text.
    [text scrollRangeToVisible:NSMakeRange(0, 0)];

    if (whyEnd == NSTabTextMovement) {
        newKeyView = (NSTextView *)[text nextKeyView];
    } else if (whyEnd == NSBacktabTextMovement) {
        newKeyView = (NSTextView *)[text previousKeyView];
    }

    // Set the new key view and select its whole contents.
    [[text window] makeFirstResponder:newKeyView];
    [newKeyView setSelectedRange:NSMakeRange(0, [[newKeyView textStorage] length])];
}
```

Section 10: Interface Builder

Nib Files and Storyboards

- A nib (a file with a **.nib** extension) is an archive file containing graphs of serialized objects.
    - Xcode maintains an XML serialization format in files with a **.xib** extension to make it easier to merge files in a repository.
    - Xcode automatically compiles .xib files into .nib files during builds.
- Archives are read and written by NSKeyedArchiver and NSKeyedUnarchiver.
- Objects that conform to the NSCoding protocol implement two methods that define how they are encoded and decoded: `init(coder:)` and `encode(coder:)`.
- Storyboards are simply collections of nib files.
    - Because they're really groups of nibs, storyboards can define *segues* from one portion of the UI (nib) to another.

The File's Owner

- The Identity Inspector allows you to set the type of the currently selected object.
- File’s Owner is a proxy for the object that will load the nib at runtime.
- Set the identity of the File’s Owner to allow IB to introspect the File’s Owner’s API.

Section 11: Menus

Automatic Validation

- If menu item's target is non-nil, checks if target implements the configured action.
- If not, menu item disables itself.
- Otherwise, calls `validateMenuItem:` or `validateUserInterfaceItem:`, if implemented.
- If target is nil, NSMenu searches up the responder chain, and enables/disables based on whether the action method is found.
- If menu is contextual, starts with the view that triggered the menu and searches up the responder chain.
- If not found, then checks the window, the window's delegate, NSApp, and finally the application delegate.

Validating Menu Items

- Generally best to implement `validateUserInterfaceItem:` because it also works with other objects, such as toolbar items.
- Avoid directly calling `setEnabled:` on menu items

TODO: Swift example

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)anItem
{
    SEL theAction = [anItem action];

    if (theAction == @selector(copy:)) {
        if ( /* there is a current selection and it is copyable */ )
        {
            return YES;
        }
        return NO;
    }
    else {
        if (theAction == @selector(paste:)) {
            if ( /* there is a something on the pasteboard we can use and
                the user interface is in a configuration in which it makes sense to
paste */ ) {
                return YES;
            }
            return NO;
        }
        else {
        /* check for other relevant actions ... */
        }
    }
    // Subclass of NSDocument, so invoke super's implementation
    return [super validateUserInterfaceItem:anItem];
}
```

Managing the Windows Menu

- By default, NSMenu automatically adds an item to the **Windows** menu when a window is moved on-screen and removes it when the window is moved off-screen.

- NSMenu also automatically hides a menu item directly connected to a specific window instance when that instance is moved off-screen.

- Set the `isExcludedFromWindowsMenu` property to **false** to override these behaviors.

Section 12: Table Views

Section 13: Concurrency

Run Loops

- **NSRunLoop** objects manage input sources and timers.
- Example input sources:
    - Touch, motion, and keyboard events
    - **performSelector:onThread:...** messages
- A run loop must be present and running in order for the following to work:
    - **performSelector…** methods
    - **NSTimer** instances
    - Keeping a thread alive to perform work periodically
    - Using Mach ports or custom input sources to communicate with other threads

Run Loop Modes

- Run loops can run in several different *modes*.
  - Modes allow events from a given set of input sources to be funneled to a specific observer.

- Predefined modes are as follows:

  Default: **NSDefaultRunLoopMode**

  Modal: **NSModalPanelRunLoopMode**

  Event Tracking: **NSEventTrackingRunLoopMode**

  Common Modes: **NSRunLoopCommonModes**

- You can also define custom run loop modes.

Multithreading

- In iOS, every thread must have its own:
    - autorelease pool;
    - run loop.
    - If you create your own threads, you’re responsible for creating and managing these yourself.

- Threading code is hard to write and debug, and threads are resource intensive. Avoid creating your own threads by hand.

- Instead use **Grand Central Dispatch** or higher-level APIs that are layered on top of GCD (e.g. **NSOperation**).

Grand Central Dispatch

- Block-based, C API
- Creates and manages:
  - highly-optimized thread pool
  - global concurrent queue
  - main (serial) queue
- GCD tasks can be
  - grouped
  - dispatched based on timer intervals
  - dispatched in a loop
- To use GCD, import **dispatch.h**.

```
#import <dispatch/dispatch.h>
```

NSOperation

- **NSOperation** is an abstract class used to encapsulate state and behavior for a given task.
    - Concrete subclasses are **NSInvocationOperation** and **NSBlockOperation**.
    - You can easily create your own custom subclasses if desired.
- Operations can be executed in either of two ways:
    - Directly (by sending them a **run** message)
    - By adding them to an **NSOperationQueue**.
- An operation can wrap one or more dependent operations.
    - Will not execute until all its dependent operations have completed.

NSOperationQueue

- Operations start executing as soon as they're added to a queue.
  - Automatically removed from queue when finished.
  - Queue holds strong references to its operations.

- To add operations to a queue:

```
- (void)addOperation:(NSOperation *)op;
- (void)addOperations:(NSArray *)ops waitUntilFinished:(BOOL)wait
- (void)addOperationWithBlock:(void (^)(void))block
```

- Managing a queue's operations:

```
- (void)cancelAllOperations;
- (void)waitUntilAllOperationsAreFinished;
```

- Supporting concurrent operations:

```
- (void)setMaxConcurrentOperationCount:(NSInteger)count;
```

Section 14: User Defaults

Section 15: Core Data

Overview

- Core Data is an object-relational mapping (ORM) framework for Cocoa and Cocoa touch.

- Primarily a mechanism for fetching and storing object graphs in database tables.

- However, object graph management features can be highly useful even when data is not persisted.

- Supports the following persistent storage types on iOS:
    - SQLite relational database (incremental)
    - binary (atomic)
    - in-memory
    - custom (atomic or incremental)

Core Data Features

- Automated object persistence
- Object version tracking and optimistic locking for automatic conflict resolution
- Objects lazily loaded by default to optimize performance
- Automatic maintenance of relational integrity
- Automatic value change observation, and built-in undo/redo management
- Automatic validation of property values
- Automatic data migration to accommodate schema changes
- Controller integration to help automate UI synchronization
- Grouping, filtering, and ordering of data
- Sophisticated query compilation with NSPredicate instead of hand-written SQL

SQLite Overview

- SQLite is a high performance, ACID-compliant database.

- It's a C library linked directly into your app, rather than a server running as a separate process.

- SQLite store files are typically created and managed in the **Library/Application Support** directory in the app sandbox container directory.

- Note: you can locate the **Library** directory by printing the value of the following expression:

```
FileManager.default.urls(for: .libraryDirectory, in: .userDomainMask)
```

Section 16: Web Services

The URL Loading System

- Provides on-disk and in-memory cache on a per-application basis.
  - Individual URL requests can specify their desired cache policy.
  - Can also control caching policy by implementing a delegate callback.
- Supports authentication and credentials, including configurable credential persistence in memory while the app is running, or for greater durations via the user's keychain.
- Provides object-oriented access to cookies.
- Built-in support for **http**, **https**, **file**, and **ftp** protocols, as well as custom protocols.

Apple's *URL Loading System Programming Guide* is an excellent resource.

URL Connections

- **NSURLConnection** has three related delegate protocols:
    - **NSURLConnectionDelegate** – asynchronous callbacks sent to the delegate during authentication challenges, as well as to notify the delegate of connection failure.
    - **NSURLConnectionDataDelegate** – asynchronous callbacks sent to the delegate as the connection is downloading data.
    - **NSURLConnectionDownloadDelegate** – asynchronous sent to notify the delegate about download progress.
- Asynchronous loading:
    - Implement delegate callbacks; or
    - Use block-based API

```
+ (void)sendAsynchronousRequest:(NSURLRequest *) request
                          queue:(NSOperationQueue *) queue
              completionHandler:(void (^)(NSURLResponse *response,
                                          NSData *data,
                                          NSError *connectionError)) handler;
```

- Synchronous loading methods also available.
    - *Avoid using on the main thread*.

URL Sessions

- **NSURLSession** creates and manages instances of **NSURLSessionTask**.
  - Session task manages an **NSURLConnection**.
- Session tasks are created in suspended state.
  - Send a **resume** message to execute.
- Subclasses of **NSURLSessionTask**:

  NSURLSessionDataTask — Loads URL resource in memory as instance of **NSData**.

  NSURLSessionUploadTask — Initialized with a file, data object, or stream to upload.

  NSURLSessionDownloadTask — Downloads response data directly to a file. Notifies its delegate when download complete.

Working with URL Sessions

- `NSURLSession` provides a global session via its `+sharedSession` method.
- You can also create your own sessions.
- Custom sessions can be configured as background sessions.
  - Managed by macOS.
  - Can continue download when app is not running.
  - macOS will relaunch app when download completes.

URL Protocols (Optional)

Reachability

- Add input sources to main run loop to receive notifications about changes to reachability of network endpoints.
- File SCNetworkReachability.h in SystemConfiguration framework declares C functions for:
  - Configuring callbacks
  - Scheduling and unscheduling in run loop
- Callback structure includes flags with detailed network status.

Testing Tips

- Apple provides **Network Link Conditioner**
  - System Preferences plugin
  - Simulates various network conditions, including performance degradation.
- **Charles Proxy** and similar tools provide richer network conditioning and monitoring.
- Consider providing a way to use dummy data in development when web services are unavailable.

Section 17: Cocoa Bindings

Inter-Object Communication

- Many techniques available, including
    - Target-action
    - Delegation
    - Closures
    - NotificationCenter
- KVO (Key-Value Observing) offers yet another mechanism
    - Establishes one-to-one communication between an arbitrary pair of objects
    - Allows a given object to observe value changes in the other object in a pair.

NSKeyValueBinding

Primitive methods

Declared in an extension on NSObject

```
open func bind(_ binding: NSBindingName,
           to observable: Any,
     withKeyPath keyPath: String,
                 options: [NSBindingOption : Any]? = nil)


open func unbind(_ binding: NSBindingName)
```

Object Controllers

- NSController
- NSArrayController

Section 18: XPC Services

Overview

- XPC Services is an API defined in `libSystem`.
- Lightweight mechanism for interprocess communication.
- Benefits
    - Increased stability (can make apps more crash-resistant)
    - Enhanced security via privilege separation
-

Working with launchd

- launchd can provide XPC services with the following:
    - Launch on demand
    - Restart on crash
    - Terminate when idle

The NSSecureCoding Protocol

- Classes that adopt NSSecureCoding and override `initWithCoder(_:)` must override `supportsSecureCoding` and return `true`.

```
public protocol NSSecureCoding : NSCoding {
    static var supportsSecureCoding: Bool { get }
}
```

- The decoding methods in the NSSecureCoding API require passing the expected type as an additional parameter, for example:

```
public func decodeObject<DecodedObjectType>(of cls: DecodedObjectType.Type,
                                           forKey key: String) -> DecodedObjectType?
                                           where DecodedObjectType:
                                           NSObject, DecodedObjectType : NSCoding
```

- This prevents attacks that substitute a different class than the expected one

Communicating Across XPC

- To do…

For More Information:

XPC · objc.io