1. Let’s Get Started Creating an Xcode Project Getting around in Xcode Application Design Model-View-Controller Creating the MainWindowController class Creating the User Interface in Interface Builder Adding view objects Configuring view objects XIB files and NIB files Showing the Window Making Connections Creating an outlet Connecting an outlet Defining an action method Connecting actions Creating the Model Layer Connecting the Model Layer to the Controller Improving Controller Design
2. Swift Types Introducing Swift Types in Swift Using Standard Types Inferring types Specifying types Literals and subscripting Initializers Properties Instance methods Optionals Subscripting dictionaries Loops and String Interpolation Enumerations and the Switch Statement Enumerations and raw values Exploring Apple’s Swift Documentation
3. Structures and Classes Structures Instance methods Operator Overloading Classes Designated and convenience initializers Add an instance method Inheritance Computed Properties Reference and Value Types Implications of reference and value types Choosing between reference and value types Making Types Printable Swift and Objective-C Working with Foundation Types Basic bridging Bridging with collections Runtime Errors More Exploring of Apple’s Swift Documentation Challenge: Safe Landing Challenge: Vector Angle

4. Memory Management

- value type memory is reclaimed when the object goes out of scope
- value types that are members of reference types are reclaimed when the containing reference type is reclaimed
- reference counting is used for reference types to manage object lifetime

Automatic Reference Counting

- addresses these needs:
  - objects need to stay in memory as long as needed
  - objects should be deallocated as soon as they are no longer needed
- mac OS uses ARC (automatic reference counting) to do it

Objects have reference counts

- creation increments reference count to 1
- when assigned or passed elsewhere, reference count is incremented
- when property, outlet, or variable changes or becomes nil, reference count is decremented

Deallocating objects in a hierarchy

- each container that contains a reference to another container maintains the contained references hierarchically
- when the reference type at the top of the hierarchy is deinitialized, all reference types down the hierarchy chain are deinitialized
- actual container types, such as Array, are not reference types but do maintain the lifetime of contained reference types if the types they contain are reference types

Strong and Weak References

- by default, a reference is a strong reference
- an object's reference count only refers to strong references
- a weak reference permits access to a reference type but does not increment the object's reference count
- weak references are always optional types

Strong reference cycles

- strong reference cycle - two objects have references to each other directly or indirectly
- a strong reference cycle causes a portion of the object graph to be leaked because none of the objects in the cycle can be deallocated
- ARC is part of the compiler and is incapable of analyzing the runtime behavior and thus incapable of detecting a strong reference cycle
- commonly use weak references in:
    - a reference to a parent in a hierarchical type relationship
    - outlets to subviews within a window controller -> the window indirectly has strong references to the view hierarchy and the window controller has a strong reference to the window
    - targets from controls -> controller has indirect strong references to its controls, control should have weak references back to the controller where action method is implemented
    - delegates ->

Unowned references

- non-strong, non-optional reference
- does not increment reference count
- not set to nil when deallocated

- used to make Cocoa compatible with ARC
- also sometimes used in closures
- only use when you know the referenced object will not be accessed after the object has been deallocated

What is ARC?

- previous approach was manual retain and release
- Cocoa added garbage collection in 2007 which had drawbacks
- Clang static analyzer facilitated ARC
- promotes optimized, automatic retain and release calls generated within compiler that is less error prone

Advanced Memory Management (Apple Docs)

- process of allocating memory during runtime, using it, and freeing it
- in Objective-C, can be used to distribute limited resources manually
- goal is manage object graphs (i.e. complex interconnections between objects at runtime
- Objective-C provides two methods of application memory management:
  - manual retain-release -> explicitly manage memory using reference counts (managed by NSObject and runtime environment)
  - ARC -> inserts memory management method calls at compile-time

Good Practices Prevent Memory-Related Problems

- problems:
  - premature free -> causes memory corruption, program crash, corrupted user data
  - not freeing data -> causes memory leaks which reduce performance and eventually can cause program termination
- better to think of memory management from the perspective of object ownership and object graphs instead of reference counts

Use Analysis Tools to Debug Memory Problems

- use Clang static analyzer in Xcode
- additionally use NSZombie and other tools (see additional guides)

Memory Management Policy

- uses methods in NSObject protocol

Basic Memory Management Rules

- based on object ownership

- Cocoa's policy:
  - you own any object you create (alloc, new, copy, mutableCopy, newObject)
  - take ownership using retain
    a. in init -> to store an object as a property value
    b. avoid deallocation as a side-effect of another operation
  - release when ownership is no longer needed (use release or autorelease)
  - do not release an unowned object
- retain and release are just Objective-C method calls (messages)
- objects returned by reference are not owned
  - in Objective-C inout parameters with ClassName** or id*
- implement dealloc to release ownership
  - free object's own memory
  - dispose held resources, including object instance variables
- never manually invoke another object's dealloc method
- Core Foundation uses similar but different rules
  - see the related guide

TODO: more here

5. Controls

- most other GUI frameworks work by long chains of subclassing view types
- Cocoa applications prefer creating objects that work with existing classes and connecting them together

Setting up RGBWell

- create Xcode project without Core Data
- select Swift or Objective-C

Creating the MainWindowController class

- cmd-N, select Swift file

```
import Cocoa

class MainWindowController: NSWindowController {}
```

- Cocoa -> Foundation, AppKit, CoreData

Creating an empty XIB file

- cmd-N and select Empty UI file
- use object library to add a window
- refer to document outline for window contents
- use attribute inspector to uncheck "Visible at Launch"
- "Visible at Launch" duplicates some code and masks the relationship between WindowController and Window
- suggested to uncheck as a matter of practice
- generally does not cause problems except when working with Sheets
- remove window from MainMenu.xib

Creating an instance of MainWindowController

- maintains a reference to the window controller to keep the window controller from being deinitialized

```
class AppDelegate: NSObject, NSApplicationDelegate {
    var mainWindowController: MainWindowController?
    func applicationDidFinishLaunching(aNotification: NSNotification) {
        // Create a window controller
        let mainWindowController = MainWindowController()
        // Put the window of the window controller on screen
        mainWindowController.showWindow(self)
        // Set the property to point to the window controller
        self.mainWindowController = mainWindowController
    }
}
```

Connecting a window controller and its window

- MainWindowController inherits a window outlet from NSWindowController
- this outlet must be connected to a window object in a XIB
- NSWindowController's window can be loaded from a NIB or created programmatically
- need to set name of window controller's NIB file in the class
- when the NIB file is loaded, the window object is unarchived
- the NIB is loaded as a side effect of accessing the window property for the first time (applicationDidFinishLaunching -> showWindow -> window is nil so NIB is loaded)
- object exists after window is loaded, showWindow can only complete if the window outlet is connected to the window object
- this process requires the File Owner setting in the XIB file

- XIB is XML repr of objects and connections, both of which must be represented at either end
- can sometimes make connections on each end directly within XIB
- often need to connect to to an object outside of the XIB which requires placeholders and File's Owner
- File's Owner is a placeholder that stands in for the object that will load the NIB file at runtime, generally the controller
- to set, select File's Owner in the document outline, select the icon inspector, find the class attribute and enter the class name of the controller class
- this permits outlet and action connections from the XIB file the file's owner class
- after setting the File's Owner, the window outlet in the file's owner must be connected to the window within the XIB file

```
MainWindowController: NSWindowController
    - windowNibName = "MainWindowController"
    - window

    |
    v

XIB file
    - File's Owner
        - class attribute = MainWindowController
        - Outlet
            |
            v
    - window in XIB file
```

About Controls

- control -> view designed for user interaction that is a subclass of NSControl

```
NSObject
^
|
NSResponder (event handling)
^
|
NSView (window display -> drawRect, etc)
^
|
NSControl (ability to send action messages)
    - target
    - action
    - enabled
    - continuous
    - objectValue
    - stringValue
    - floatValue
```

```
    - integerValue
^
|
NSButton & NSTextField & NSSlider
```

- adds the ability to send action messages to a target
- action -> a selector sent to the target of the action message
- selector -> essentially the name of a method in a string form

Working with Controls

- same class may have a number of different visual styles

A word about NSCell

- each control comes with a subclass of NSCell
- NSCell handles the implementation details of the control
- NSCell has properties that are hooked up to the control's properties
- cell's are being deprecated but still remain

Connecting the slider’s target and action

- must create an IBAction method within the file's owner class
- each IBAction has a single sender parameter
- can implement that sender as the control's class or AnyObject
- use AnyObject for situations where you have multiple types of controls triggering the same action method
- action methods appear in the context menu of the File's Owner within the XIB file and the action sent by a control can be connected using ctrl+drag to the File's Owner Received Actions menu within its context menu
- NSControl defines its target outlet as a weak reference

A continuous control

- the continuous property on a controller is false by default and an action message is only sent at interaction end
- when continuous = true, the action message is sent continuously
- set the continuous attribute in the attributes inspector

Setting the slider's range values

- set min and max in the attributes inspector

Adding two more sliders

NSColorWell and NSColor

- color well by default opens a color picker on click

Disabling a control

- each control has an enabled property
- setting enabled = false means the control does not respond to user interaction
- can set in the attributes inspector

Using the Documentation

- Xcode contains API documentation and guides

Changing the color of the color well

- requires getting the Double value from the slider sender and converting to CGFloat to pass to an NSColor init method

Controls and Outlets

- to access a control outside of an action method, must create an outlet and connect the control to the outlet

```
@IBOutlet weak var rSlider: NSSlider!
```

- good habit to call super.windowDidLoad() within overridden windowDidLoad (and other overridden methods)

Implicitly unwrapped optionals

- optional that never needs to be unwrapped (i.e. forcibly unwrapped)
- outlets should always be implicitly unwrapped

- outlets must be optional because they must be loaded from the NIB
- these will never be nil if the NIB is properly loaded and the connections are properly made
- a missing connection will cause the app to crash

For the More Curious: More on NSColor

- has class methods that return common colors
- also has methods for common systems colors
  - controlTextColor
  - selectedTextBackgroundColor
  - windowBackgroundColor
- see guide Color Management Overview

For the More Curious: Setting the Target Programmatically

- NSControl has an action property which is a Selector
- to set programmatically

```
var playButton: NSButton
func play(sender: NSButton) {}
let playSelector = Selector("play:")
playButton.action = playSelector
```

Challenge: Busy Board Debugging Hints

- hints:
  - can double-click an object in the object library to see the object's class and some other details
  - NSButton has a state property which can be used for checkboxes
  - the Radio Group is based on NSMatrix which is informally deprecated
  - radio groups can be implemented programmatically using their Style property and NSView's tag property

Debugging Hints

- always watch the console
- always use the debug build configuration during development
- common problems:
  - crash when programmatically using a view set up using IB
    - missing connection

- connection was made but app still crashes
    - possible misspelling or other issue invalidating connection
- app crashes
    - accessing a deallocated object
    - can use NSZombie and zombies to debug
- cannot make a connection in IB
    - issue in Swift file syntax

6. Delegation

Setting up SpeakLine

- create Xcode Cocoa project
- remove default window in XIB file
- create MainWindowController class and XIB file, unset show at start, connect File's Owner
- add default code

```
// MainWindowController.swift
class MainWindowController: NSWindowController {
    override var windowNibName: String {
        return "MainWindowController"
    }
    override func windowDidLoad() { super.windowDidLoad() }
}
// AppDelegate.swift
class AppDelegate: NSObject, NSApplicationDelegate {
    var mainWindowController: MainWindowController?
    func applicationDidFinishLaunching(aNotification: NSNotification) {
        // Create a window controller
        let mainWindowController = MainWindowController()
        // Put the window of the window controller on screen
        mainWindowController.showWindow(self)
        // Set the property to point to the window controller
        self.mainWindowController = mainWindowController
    }
}
```

Creating and using an Xcode snippet

- can add code snippets within the snippet library window in Xcode
- can click and drag code snippets to use them

Creating the user interface

- add buttons and text fields and create outlets within controller

Synthesizing Speech

- NSSpeechSynthesizer
  - startSpeakingString(_ text: String!) -> Bool
  - stopSpeaking()

Updating Buttons

- programmatically enable and disable buttons using the enabled property

Delegation

- objects that have specified delegate protocols are defined to have the delegate as a helper object

Being a delegate

- being a delegate means taking on a rol specified by a protocol and conforming (inheriting and implementing specified methods) from that protocol
- passes on behavior while avoiding need to subclass
- optional keyword within a protocol indicates that a conforming class is not required to implement the method
- to conform, inherit from class and implement all required methods
- each class that has a corresponding delegate class also has a delegate property
- to assign

```
override func windowDidLoad() {
    super.windowDidLoad()
    delegatee.delegate = self
}
```

- no delegate properties are strong references

Implementing another delegate

- classes can function as delegates for many properties

Common errors in implementing a delegate

- forgetting to set the delegate property
  - since it is optional, the compiler will not provide a warning if it is not set
- misspelling the name of an optional delegate method

Cocoa classes that have delegates

- AppKit
  - NSAlert
  - NSMatrix
  - NSTabView
  - NSAnimation
  - NSMenu
  - NSTableView
  - NSApplication
  - NSPathControl
  - NSText
  - NSBrowser
  - NSRuleEditor
  - NSTextField
  - NSDatePicker
  - NSSavePanel
  - NSTextStorage
  - NSDrawer
  - NSSound
  - NSTextView
  - NSFontManager
  - NSSpeechRecognizer
  - NSTokenField
  - NSImage
  - NSSpeechSynthesizer
  - NSToolbar
  - NSLayoutManager
  - NSSplitView
  - NSWindow

Delegate protocols and notifications

- many delegate methods take NSNotification as a parameter
- sent via notification center rather than being called by the object on delegate object

NSApplication and NSApplicationDelegate

- each Cocoa app has a single instance of NSApplication and asingle instance of a custom class named AppDelegate
- AppDelegate methods are triggered by application lifecycle events
- @NSApplicationMain -> instantiates NSApplication and AppDelegate and sets NSApplication's delegate property to AppDelegate

The main event loop

- NSApplication starts and maintains main event loop
- when a UI event occurs, the event is forwarded from the window server to the corresponding NSApplication instance and is added to the app's event queue which is then forwarded to the proper event handler when the event is processed

NOTE: a large part of learning Cocoa is learning the proper methods to

implement -> action methods, delegate methods, and notification observers

For the More Curious: How Optional Delegate Methods Work

- only calls implemented optional delegate methods which are note called otherwise
- this is facilitated in Swift using optionals and the short circuits that happen is optionals are nil as well as optional method calls (.didFinishSpeaking?)

Challenge: Enforcing a Window’s Aspect Ratio

- use windowWillResize, CGSize typealiased to NSSize, and the initial size in the size inspector of Interface Builder

7. Working with Table Views

About Table Views

- a table view is used to display data arranged in rows and columns

Delegates and data sources

- a table view can have both a delegate and a data source
- typically one object plays both roles for a table view
- for a table view, the delegate must conform to the NSTableViewDelegate protocol
- a data source is like a delegate except it interacts with a model (or other data repository manager)
- for table view, the data source must conform to NSTableViewDataSource
- like for the delegate, the table view has a dataSource property which must be set to the data source instance

The table view-data source conversation

- the table view controls requests for data from the data source
- the table view asks for data like an iterator, one element at a time, e.g. one row x column location at a time
- methods:
  - numberOfRowsInTableView
  - tableView

SpeakLine's table view and helper objects

- uses MainWindowController as both the data source and the delegate

Getting Voice Data

- NSSpeechSynthesizer
  - class func availableVoices which returns `[NSString]` but should cast to Swift String

Retrieving friendly names

- NSSpeechSynthesizer
  - class func attributesForVoice -> optional dictionary

Adding a Table View

- table view is available in the object library

Table view and related objects

- table view is a collection of many objects
    - clip view
        - table view
            - table column
            - table column
            - etc.
        - scroller
        - scroller
        - table header view
- the scroll view is there because a table view can have more data than it can display
- clip view is an instance of NSClipView and manages the currently visible portion of the table and contains the table view
- scrollers control table scroll
- table header is NSTableHeaderView and responds to mouse clicks on the titles
- NSTableColumn
    - change title in attributes inspector
- to disable resize, uncheck resize in attributes inspector

Tables, Cells, and Views

- important to know that cell-based tables, those based on NSCell, have been deprecated
- newer tables are view-based, i.e. work by using NSView

Table cell views

- text table cell view - contains a single text field
    - Table Cell View - NSTableCellView
        - Table View Cell - NSTextField
            - Table View Cell - NSTextFieldCell
- image & text table cell view - text field and image view
- can create custom table cell views
- setting the objectValue property of the table cell view does not affect what the table displays, so must change the text field when objectValue is updated using bindings

The NSTableViewDataSource Protocol

Conforming to the protocol

- inherit from the class

Connecting the dataSource outlet

- click the table view in the XIB file and ctrl-drag to the File's Owner

Implementing data source methods

```
func numberOfRowsInTableView(tableView: NSTableView) -> Int { return voices.count
func tableView(
    tableView: NSTableView,
    objectValueForTableColumn tableColumn: NSTableColumn?,
    row: Int
) -> AnyObject? {
    let voice = voices[row]
    let voiceName = voiceNameForIdentifier(voice)
    return voiceName
}
```

Binding the text field to the table cell view

- use the bindings inspector and select text field in XIB
- under bind to select Table Cell View
- Model Key Path should be objectValue -> property of table cell view

The NSTableViewDelegate Protocol

- drag from table view connections pop-up menu to File's Owner

Making a connection with the assistant editor

- can also use the assistant editor to make connections
- can drag to the related file from a view and create an outlet automatically

Implementing a delegate method

- the table view's selectedRow property contains the necessary information
- the delegate method required is tableViewSelectionDidChange

```
// MARK: - NSTableViewDelegate
func tableViewSelectionDidChange(notification: NSNotification) {
    let row = tableView.selectedRow
    // Set the voice back to the default if the user has deselected all rows
    if row == -1 {
        speechSynth.setVoice(nil)
        return
    }
    let voice = voices[row] speechSynth.setVoice(voice) }
}
```

Pre-selecting the default voice

- within windowDidLoad, get NSSpeechSynthesizer's defaultVoice

```
let defaultVoice = NSSpeechSynthesizer.defaultVoice()
if let defaultRow = find(voices, defaultVoice) {
    let indices = NSIndexSet(index: defaultRow)
    tableView.selectRowIndexes(indices, byExtendingSelection: false)
    tableView.scrollRowToVisible(defaultRow)
}
```

- see the Table View Programming Guide for more info

Challenge: Make a Data Source

- use an add button, an array property in the controller, and the table view's reloadData method

8. KVC, KVO, and Bindings

- Key-value coding is a mechanism that allows the programmer to get and set the value of a property indirectly using a key, i.e. the name of the property as a string
- core data and bindings rely on KVC
- setting:
  - object.setValue("Value", forKey: "keyName")
- getting:
  - object.valueForKey("keyName")

Bindings

- bindings can be used to reduce/eliminate the code to link the view to the model

Setting up Thermostat

- simple setup
- add vertical slider, label, and warmer and cooler buttons

Using bindings

- add a temperature property to MainWindowController
- in bindings inspector for the slider
  - Bind to -> File's Owner
  - Model Key Path -> temperature
- do the same for the label
- this uses KVC and KVO behind the scenes

Key-value observing

- object registers to observe changes to the value of a key
- can add action methods that manually make changes and KVO still happens as long as the changes are made in a KVO compliant manner

Making keys observable

- by default Swift properties are not KVO-compliant
- two options:
  - add the keyword dynamic

```
dynamic var keyName
```

```
– explicitly trigger observers on change
```

```
willChangeValueForKey("keyName")
// do something
didChangeValueForKey("keyName")
```

- more common to see dynamic keyword

- dynamic keyword makes a property behave like an Objective-C property, thus employing the Objective-C runtime

Binding other attributes

- add a switch isOn and bind to power button

KVC and Property Accessors

- KVC will call property accessors (custom getters and setters) if present

```
dynamic var varName: Int {
    set { print() varName = newValue }
    get { print() return otherValue }
}
```

- common to use a computed value (as above) with a private property to control access

KVC and nil

- when a value is set to nil, no assumption is made about how to assign that value
- KVC calls the method setNilValueForKey(_:) (which throws for NSObject)
- nil is sent for an empty text field
- many optional types will work
- optional numeric types can cause a crash, which can be worked around by using NSNumber?

Debugging Bindings

- notoriously difficult to debug
- common problem
    - key name typo
- to reveal all bindings for an object, show the the connections pop-up and look at the bindings section at the bottom

Using the Debugger

Using breakpoints

- can refer to the debug navigator in the navigator area on the left

Stepping through code

- step buttons left to right
  - show/hide debug area
  - toggle all breakpoints
  - continue/pause
  - step over line
  - step into method
  - step out of method
- to set an exception breakpoint, add breakpoint in breakpoint navigator and select exception breakpoint

The LLDB console

Using the debugger to see bindings in action

For the More Curious: Key Paths

- objects are arranged in a network within a program
- key paths extend beyond a single object and can specify a path through a network of objects
- key paths can include operators
  - @avg
  - @count
  - @max
  - @min
  - @sum
- to bind programmatically:

```
textField.bind(
    NSValueBinding, // "value"
    toObject: employeeController,
    withKeyPath: "arrangedObjects.@ avg.expectedRaise",
    options: nil
)
```

- use unbind(_:) to remove the binding

```
textField.unbind(NSValueBinding)
```

For the More Curious: More on Key-Value Observing

- programmatic observer (from NSObject):

```
let mainWindowController = ...
mainWindowController.addObserver(
    self,
    forKeyPath: "temperature",
    options: NSKeyValueObservingOptions.Old,
    context: &MyKVOContext
)
```

- triggered method

```
override func observeValueForKeyPath(
    keyPath: String!,
    ofObject object: AnyObject!,
    change: [NSObject : AnyObject]!,
    context: UnsafeMutablePointer<Void>
) { ... }
```

For the More Curious: Dependent Keys

- for dependent keys, define a class method named keyPathsForValuesAffecting?KeyName? () that returns a set of key paths

```
class func keyPathsForValuesAffectingFullName() -> NSSet { return Set([" firstName
```

Challenge: Convert RGBWell to Use Bindings

- often useful to state what you are binding to determine how to do it
- use default values for properties to avoid defining overriding init
- for NSColor, use the class methods to supply the default property values
- does not require outlets

9. NSArrayController

- create a document based app -> subclass of NSDocument which has a reference to the data which makes up the document
- setup as usual but add document extension in initial options of rsmn

RaiseMan's Model Layer

- uses an Employee class -> is a model so only needs Foundation

```
class Employee: NSObject {
    var name: String? = "New Employee"
    var raise: Float = 0.05
}
```

- Document (view controller in document-based app) contains a collection of employees
- temporarily comment out error lines in dataOfType and readFromData

RaiseMan's View Layer

- table view, two buttons

Introducing NSArrayController

- NSArrayController is another type of view controller that sit between the Document class view controller and its table view
- works by using bindings
- properties:
  - content -> the input to the array controller (rarely used directly, most often used by binding name "Content Array", which gets bound to a model array)
  - arrangedObjects -> output, typically on receiving end of a binding
  - selectionIndexes
- think of array controller as a series of directed components combined
- direct towards an array by binding, array control filters, then runs result of filtering through the sorting (arrangedObjects)
- array controllers also handle selection -> arrangedObjects may not map content so selectionIndexes can map between the two
- NSPopUpButton and NSComboBox can also be bound to an array controller

Adding an Array Controller to the XIB

- can just drag-and-drop to canvas
- under attributes inspector, select RaiseMan.Employee as the Class Name under Object Controller

Binding the Array Controller to the Model

- Bind to -> File's Owner + X
- Controller Key -> ""
- Model Key Path -> employees

Binding the Table View’s Content to the Array Controller

- goto Table Content section for the Table View
- Content -> Array Controller's arrangeObjects
- Model Key Path -> ""

Connecting the Add Employee Button

- ctrl-drag from Add Employee button and set section to add:

Binding the Text Fields to the Table Cell Views

- as before, each cell in a row of a table view is an NSTableCellView with an objectValue
- any control within that must be bound to the value of a property on that object
- select text field in first column's table cell view (NSTextField and not NSTextFieldCell)
- first column
    - Value -> Bind to -> Table Cell View
    - Controller Key -> ""
    - Model Key Path -> objectValue.name
- second column
    - Value -> Bind to -> Table Cell View
    - Controller Key -> ""
    - Model Key Path -> objectValue.raise

Formatting the Raise Text Field

- drag a number formatter onto text field cell of second column
- attributes inspector -> Style -> Percentage & X Lenient

Connecting the Remove Button

- ctrl-drag from the remove button to Array Controller and set the action to remove:
- the remove action does not know how to map which row is selected at this point

Binding the Table View's Selection to the Array Controller

- the selection indexes are the indexes of the arrangedObjects at this point
- selection indexes are not indexes in the array controller's content array
- Bindings ->
    - Selection Indexes ->
        - Bind to -> Array Controller's selectionIndexes
    - Model Key Path -> ""

Configuring RaiseMan's Remove Button

- want to disable remove when no employee is selected
- Bindings ->
    - Availability ->
        - Enabled ->
            - Bind to -> Array Controller
        - Controller Key -> canRemove
- Attributes Inspector -> Key Equivalent -> Delete Key

Sorting in RaiseMan

- clicking on Column Headers changes the sort descriptors (NSSortDescriptor)
- each sort descriptor contains a key (KVC path), ascending vs. descending, comparison selector (comparator)
- in a table view, each column corresponds to a sort descriptor
- select Raise column -> Attributes Inspector -> Sort Key -> raise
- automatically sets Selector to compare: (method selector that is comparator, NSNumber's compare method)
- select Name column -> Attributes Inspector -> Sort Key -> name
    - Selector -> caseInsensitiveCompare: (method on String)
- select Table View -> Bindings Inspector
    - Table Content
        - Sort Descriptors -> Array Controller
        - Controller Key -> sortDescriptors
        - Model Key Path -> ""

How Sorting Works in RaiseMan

- often makes sense to apply multiple sort descriptors at once

For the More Curious: The caseInsensitiveCompare(_:) Method

- returns an NSComparisonResult
  - OrderedAscending - a b
  - OrderedSame - b == a
  - OrderedDescending - b a

For the More Curious: Sorting Without NSArrayController

- can use sortedArrayUsingDescriptors

```
func tableView(
    tableView: NSTableView,
    sortDescriptorsDidChange oldDescriptors: [AnyObject]
) {
    let sortDescriptors = tableView.sortDescriptors
    let objectsArray = objects as NSArray
    let sortedObjects = objectsArray.sortedArrayUsingDescriptors(sortDescriptors)
    objects = sortedObjects tableView.reloadData()
}
```

For the More Curious: Filtering

- NSArrayController does filtering and sorting
  - filterPredicate (NSPredicate) -> see Predicate Programming Guide
  - sortDescriptors (array of NSSortDescriptor objects)
- predicates rely on KVC so for large arrays prefer Array.filter(_:)

```
let matches = employees.filter { employee -> Bool in
    if let name = employee.name {
        let nameMatches = name.caseInsensitiveCompare(someName) == .OrderedSame
        return nameMatches && employee.raise > 0.0
     } else { return false }
}
```

For the More Curious: Using Interface Builder's View Hierarchy Popover

- related to jump bar for hierarchical drill down
- for the popover -> ctrl+shift+click

Challenge: Sorting Names by Length

- can use the .length property on the names

10. Formatters and Validation

- provide validation at the controller layer
- Key-Value Validation provides validation at the model layer

Formatters

- NSNumberFormatter, NSDateFormatter
- for dates, if there is a specific input string format, specify it, otherwise use defaults to take advantage of locale settings

Formatters, programmatically

```
let date = NSDate() // "Feb 28, 2015, 8: 07 PM"
let dateFormatter = NSDateFormatter() //
dateFormatter.dateStyle = .MediumStyle
dateFormatter.timeStyle = .ShortStyle
dateFormatter.stringFromDate(date) // "Feb 28, 2015, 8: 07: 59 PM"

let ransom = 1_000_000 // 1000000
let numberFormatter = NSNumberFormatter()
numberFormatter.numberStyle = .CurrencyStyle
numberFormatter.stringFromNumber(ransom) // "$ 1,000,000.00"
numberFormatter.maximumFractionDigits = 0
numberFormatter.stringFromNumber(ransom) // "$1,000,000"
```

Formatters and a control’s objectValue

- formatters generally straddle the view and controller layer
- user input goes through the formatter when a formatter is added to a text field
- when the objectValue is set on a text field with a formatter, the formatter will convert the object to a string representation, which is available as stringValue

Formatters and localization

- formatters have access to the locale settings
- there are more formatters in Cocoa than mentioned here

Validation with Key-Value Coding

- can use Key-Value Coding Validation at the model and/or controller layer

Adding Key-Value validation to RaiseMan

- as is, the formatter with empty input will throw an unhandled exception
    i. enter empty string
    ii. formatter transforms input to optional NSNumber which is nil
    iii. no validation is done between the binding and the model
    iv. bindings sets Employee.raise to nil
    v. NSNumber does not know how to handle nil and throws an exception
- to enable validation for a binding
    - Table Cell View -> Bindings Inspector
        - Validate Immediately -> X
- the format of a method for validation in this case is validate?KeyName?(_:error:)

```
func validateRaise(
    raiseNumberPointer: AutoreleasingUnsafeMutablePointer<NSNumber?>,
    error outError: NSErrorPointer
) -> Bool
```

- true means that the value is valid
- can change the value and the error within the method since they are pointers

```
func validateRaise(
    raiseNumberPointer: AutoreleasingUnsafeMutablePointer<NSNumber?>,
    error outError: NSErrorPointer
) -> Bool {
    let raiseNumber = raiseNumberPointer.memory
    if raiseNumber == nil {
        let domain = "UserInputValidationErrorDomain"
        let code = 0
        let userInfo = [NSLocalizedDescriptionKey : "An employee's raise must be a
        outError.memory = NSError(domain: domain, code: code, userInfo: userInfo)
        return false
    } else { return true }
}
```

For the More Curious: NSValueTransformer

- NSValueTransformer can be used with bindings to modify the value read by the binding from an object

- Bindings Inspector -> Value Transformer
- one type is NSNegateBoolean
- can subclass NSValueTransformer
- see Value Transformer Programming Guide

11. NSUndoManager

- NSUndoManager maintains two stacks, one for undo and one for redo
- objects on stacks are NSInvocation

Message Passing and NSInvocation

- message passing is a dynamic, runtime polymorphism mechanism that allows the method that is called to be determined when the message is passed
- if an object does not understand a received message it:
  - checks for a forwardInvocation(_:) method
  - throws an exception if it does not have it
- Swift adds its own adaptation of the Objective-C runtime to handle this when necessary

How the NSUndoManager Works

- uses undo and redo stack

Using NSUndoManager

- by default all invocations added to the stack during a single event are grouped together
- each NSDocument already has its own undo manager

Key-Value Coding and To-Many Relationships

- RaiseMan requires redoing and undoing adding employees
- Document is not changing employees, NSArrayController is
- employees property represents and ordered to-many relationship on Document
- KVC recognizes the following relationships:
  - simple attributes -> typically strings, numbers, dates, etc
  - To-one relationships -> reference type, i.e. class/custom class
  - ordered to-many -> collection with ordering (inherent to relationship, not display)
  - unordered to-many -> collection without inherent ordering (i.e. NSMutableSet)
- KVC expects a number of conventional methods for these relationship types

Adding Undo to RaiseMan

- this adds a layer between adding or removing employees which prepares the undo and redo stacks by modifying the NSUndoManager on the Document

```
// Document

func insertObject(employee: Employee, inEmployeesAtIndex index: Int) {
    print(" adding \( employee) to the employees array") // Add the inverse of thi
    let undo: NSUndoManager = undoManager!
    undo.prepareWithInvocationTarget(self).removeObjectFromEmployeesAtIndex(employ
    if !undo.undoing { undo.setActionName("Add Person") }
    employees.append(employee)
}

func removeObjectFromEmployeesAtIndex(index: Int) {
    let employee: Employee = employees[index]
    print("removing \( employee) from the employees array")
    // Add the inverse of this operation to the undo stack
    let undo: NSUndoManager = undoManager!
    undo.prepareWithInvocationTarget(self).insertObject(employee, inEmployeesAtInd
    if !undo.undoing { undo.setActionName("Remove Person") } // Remove the Employe
    employees.removeAtIndex(index)
}
```

Key-Value Observing

- method for registration on NSObject

```
func addObserver(
    observer: NSObject,
    forKeyPath keyPath: String,
    options: NSKeyValueObservingOptions,
    context: UnsafeMutablePointer<Void>
)
```

- specify key path, not just key
- options specifies information that should be passed to observer
- context indicates what level of the class hierarchy is observing the notification
- notification calls the following method on the observer

```
func observeValueForKeyPath(
    keyPath: String,
    ofObject object: AnyObject,
    change: [NSObject : AnyObject],
    context: UnsafeMutablePointer<Void>
)
```

Using the Context Pointer Defensively

- possible to mistakenly intercept messages intended for a superclass
- use a class specific pointer as the context
- if the notification is not intended for the current subclass, it should be passed up the class hierarchy to the correct superclass

Undo for Edits

```
// Document
// outside of class definition
private var KVOContext: Int = 0
// method
// MARK: - Key Value Observing
func startObservingEmployee(employee: Employee) {
    employee.addObserver(self, forKeyPath: "name", options: .Old, context: &KVOCon
    employee.addObserver(self, forKeyPath: "raise", options: .Old, context: &KVOCo
}
func stopObservingEmployee(employee: Employee) {
    employee.removeObserver(self, forKeyPath: "name", context: &KVOContext)
    employee.removeObserver(self, forKeyPath: "raise", context: &KVOContext)
}

// ...
var employees: [Employee] = [] {
    willSet { for employee in employees { stopObservingEmployee(employee) } }
    didSet { for employee in employees { startObservingEmployee(employee) } }
}

// ...
override func observeValueForKeyPath(
    keyPath: String,
    ofObject object: AnyObject,
    change: [NSObject : AnyObject],
    context: UnsafeMutablePointer<Void>
) {
    if context != &KVOContext {
        // If the context does not match, this message
        // must be intended for our superclass.
        super.observeValueForKeyPath(keyPath, ofObject: object, change: change, co
        return
    }
    var oldValue: AnyObject? = change[NSKeyValueChangeOldKey]
    if oldValue is NSNull { oldValue = nil }
    let undo: NSUndoManager = undoManager!
    print(" oldValue =\( oldValue)")
    undo.prepareWithInvocationTarget(object).setValue(oldValue, forKeyPath: keyPat
}
```

- need to also make Document stop observing employees on window close

- Document is defaulted to being the delegate for the Window in IB (File's Owner) but need to inherit from the NSWindowDelegate protocol explicitly in code

```
// MARK: - NSWindowDelegate
func windowWillClose(notification: NSNotification) { employees = [] }
```

- above implementation triggers willSet on employees property

Begin Editing on Insert

- add table view and array controller outlets to document
- add an addEmployee action

```
@IBAction func addEmployee(sender: NSButton) {
    let windowController = windowControllers[0] as! NSWindowController
    let window = windowController.window!
    let endedEditing = window.makeFirstResponder(window)
    if !endedEditing {
        print(" Unable to end editing.")
        return
    }
    let undo: NSUndoManager = undoManager!
    // Has an edit occurred already in this event?
    if undo.groupingLevel > 0 {
        // Close the last group
        undo.endUndoGrouping()
        // Open a new group
        undo.beginUndoGrouping()
    }
    // Create the object
    let employee = arrayController.newObject() as! Employee
    // Add it to the array controller's contentArray
    arrayController.addObject(employee)
    // Re-sort (in case the user has sorted a column)
    arrayController.rearrangeObjects()
    // Get the sorted array
    let sortedEmployees = arrayController.arrangedObjects as! [Employee]
    // Find the object just added
    let row = find(sortedEmployees, employee)!
    // Begin the edit in the first column
    print("starting edit of \(employee) in row \(row)")
    tableView.editColumn(0, row: row, withEvent: nil, select: true)
}
```

- connect add button to addEmployee action and connect File's Owner outlets for table view and array controller

For the More Curious: Windows and the Undo Manager

- a view can add inverse actions for edits to the undo manager
- view queries delegate for the undo manager to use

```
optional func undoManagerForTextView( view: NSTextView!) -> NSUndoManager?
```

- then queries window

```
// property on NSWindow inherited from NSResponder
var undoManager: NSUndoManager? { get }
```

- windows delegate can supply the following method

```
optional func windowWillReturnUndoManager( window: NSWindow) -> NSUndoManager?
```

- key window: active window (Cocoa term because it gets keyboard events)

12. Archiving

- object-oriented program is composed of complex graph of interconnected objects
- archiving is a mechanism to represent this graph as a stream of bytes
- aka serialization
- only need properties, property values, and name of class, not methods in archive
- XIB files use their own archiving and unarchiving (via XML)
- two steps -> define how to archive and indicate when to archive

NSCoder and NSCoding

- NSCoding protocol
  - init(coder aDecoder: NSCoder)
  - encodeWithCoder(aCoder: NSCoder)
- add to Employee
- NSCoder is an abstract class
- will create NSKeyedArchiver and NSKeyedUnarchiver as concrete classes

Encoding

- commonly use encodeObject(anObject: AnyObject?, forKey aKey: String)
    - triggers encodeWithCoder on anObject
- NSCoder has the following for common value types

```
func encodeBool(boolv: Bool, forKey key: String)
func encodeDouble(realv: Double, forKey key: String)
func encodeFloat(realv: Float, forKey key: String)
func encodeInt(intv: Int32, forKey key: String)
```

- inherit from NSCoding in Employee

```
func encodeWithCoder(aCoder: NSCoder) {
    if let name = name { aCoder.encodeObject(name, forKey: "name") }
    aCoder.encodeFloat( raise, forKey: "raise")
}
// or if superclass also implements
override func encodeWithCoder(aCoder: NSCoder) {
    super.encodeWithCoder(aCoder)
    if let name = name { aCoder.encodeObject( name, forKey: "name") }
    aCoder.encodeFloat( raise, forKey: "raise")
}
```

Decoding

- methods for decoding common value types

```
func decodeObjectForKey(key: String) -> AnyObject?
func decodeBoolForKey(key: String) -> Bool
func decodeDoubleForKey(key: String) -> Double
func decodeFloatForKey(key: String) -> Float
func decodeIntForKey(key: String) -> Int32
```

- add following methods

```
// call super.init(coder:) if it has it
required init(coder aDecoder: NSCoder) {
    name = aDecoder.decodeObjectForKey("name") as! String?
    raise = aDecoder.decodeFloatForKey("raise")
    super.init()
}
// also add init()
```

- do not always need to inherit or add init methods in Swift

The Document Architecture

- composed of 3 classes
  - NSDocumentController
  - NSDocument
  - NSWindowController
- handles MVC and directly or via array controller
  - save model data to file
  - load data from file
  - give view data to display
  - handle user input and update model

Info.plist and NSDocumentController

- Info.plist contains application info
- indicates whether app is document based (and creates NSDocumentController on load if so)
- rarely interact with doc controller

```
let documentController = NSDocumentController.sharedDocumentController()
```

NSDocument

- generally can just subclass and not interact with the rest of the doc architecture
- may need to add NSWindowController subclasses for view components
- for save, save all, and save as

```
func dataOfType(typeName: String, error outError: NSErrorPointer) -> NSData?
func fileWrapperOfType(typeName: String!, error outError: NSErrorPointer) -> NSFil
func writeToURL(url: NSURL, ofType typeName: String, error outError: NSErrorPointe
```

- NSErrorPointer -> errors are stored in NSErrorPointer's memory field -> must create error if method fails
- for open, open recent, revert to saved

```
func readFromData(data: NSData, ofType typeName: String, error outError: NSErrorPo
func readFromFileWrapper(fileWrapper: NSFileWrapper, ofType typeName: String, erro
func readFromURL(url: NSURL, ofType typeName: String, error outError: NSErrorPoint
```

- need to call the following to set up the UI after loading

```
func windowControllerDidLoadNib(windowController: NSWindowController!)
// or for revert to saved update UI in load method by checking outlets for nil
```

-

NSWindowController

- one instance for each window displayed (in windowControllers property)
- generally can rely on the default behavior of the single window controller
- cases for customizing
  - more than one window on same document i.e. design, CAD, DAW
  - separate UI controller and model controller logic

Saving and NSKeyedArchiver

- saving requires an NSData object -> returning this will allow writing to file
- NSKeyedArchiver has the following method

```
class func archivedDataWithRootObject(rootObject: AnyObject) -> NSData
```

- encoding is performed through a propagating process of encoding
- to be able to save

```
override func dataOfType(typeName: String, error outError: NSErrorPointer) -> NSDa
    // End editing
    tableView.window?.endEditingFor(nil)
    // Create an NSData object from the employees array
    return NSKeyedArchiver.archivedDataWithRootObject(employees)
}
```

- will not throw

Loading and NSKeyedUnarchiver

- use NSKeyedUnarchiver's method

```
class func unarchiveObjectWithData( data: NSData) -> AnyObject?
```

- modify Document

```
override func readFromData(
    data: NSData,
    ofType typeName: String,
```

```
    error outError: NSErrorPointer
) -> Bool {
    print("About to read data of type \(typeName).")
    employees = NSKeyedUnarchiver.unarchiveObjectWithData(data) as! [Employee]
    return true
}
```

- NSArrayController handles UI updates but will later modify windowControllerDidLoadNib
- document specifies nib through windowNibName property

Setting the Extension and Icon for the File Type

- for icon, add .icns file to project by dragging into Supporting Files and copying if necessary
- for doc type info, select RaiseMan in project navigator
    - Info
        - set name, Icon and identifier, and UTI under target info
        - set UTI Description, Identifier, and Extensions to settings above
        - Conforms To -> public.data
- UTI - describes the data contained in the file
- changes are saved to Info.plist

Application Data and URLs

- shoebox app (collections of documents of type) should store data in ~/Library/Application Support
- do not build many of these default URLs manually
- for common directories, use NSFileManager's URLsForDirectory(_:inDomains:)
    - ApplicationSupportDirectory
    - CachesDirectory
    - DocumentDirectory
- for the first time, create directory using NSFileManager's createDirectoryAtURL (_:withIntermediateDirectories:attributes:error:)

For the More Curious: Preventing Infinite Loops

- for archiving object graphs with cycles, NSKeyedArchiver creates objects with tokens
- if an object is encountered again, only its token is added to the stream
- when unarchiving, if a token is found with no data, the NSKeyedUnarchiver knows to look up the token in the table
- the following NSCoder method is often confusing to developers

```
func encodeConditionalObject(objv: AnyObject?, forKey key: String)
```

- used for when object A has a ref to B but B does not need to be archived
- if another object has archived B, A adds B's token to stream, otherwise treat as nil

For the More Curious: Creating a Protocol

- define interface using protocol keyword
- tagging with @objc adds the ability to add optional methods

For the More Curious: Automatic Document Saving

- 10.7 and after added auto saving which is triggered by a change count and allows browsing of versions via a Time Machine-like interface
- opt in

```
override class func autosavesInPlace() -> Bool { return true }
```

- see Mac App Programming Guide for more info

For the More Curious: Document-Based Applications Without Undo

- NSDocument keeps track of the number of changes

```
func updateChangeCount(change: NSDocumentChangeType)
```

- has following types
  - ChangeDone - increments
  - ChangeUndone - decrements
  - ChangeCleared - zeroes
- window marked dirty when change count is not 0

For the More Curious: Universal Type Identifiers

- answering question what a file represents is answered in a number of ways
  - file extensions
  - creator codes
  - MIME types

- Apple's long term solution is UTI
- hierarchically organized
- documents can read a set of UTIs (including custom) specified in Info.plist
- pasteboard also uses UTIs to identify data types
- see Uniform Type Identifiers Reference guide for more info

13. Basic Core Data

- core data handles writing data objects (i.e. generates the object definition from a schema), saving, loading, and undo

Defining the Object Model

- must define the managed object model
- entities: similar to object, has attributes (value types) and relationships (refs to other objects)
- stored in an .xcdatamodeld file and instance of NSManagedObjectModel class
- documents subclass NSPersistentDocument to manage reading and saving
- NSManagedObjectContext manages all of the entities (create, get, delete) (managedObjectContext property on NSPersistentDocument)
- NSManagedObject is the base class for models
- create Xcode project
    - document based
    - ignore extension
    - use core data
- open Document.xcdatamodeld
    - Add Entity -> Car
        - condition -> Integer 16
        - datePurchased -> Date
        - makeModel -> String
        - onSpecial -> Boolean
        - photo -> Transformable
        - price -> Decimal

Configure the Array Controller

- add array controller
- Bindings Inspector
    - Bind to -> File's Owner
    - Model Key Path -> managedObjectContext

- Attributes Inspector
    - Object Controller
        - Mode -> Entity Name
        - Entity Name -> Car
        - Prepares Content -> On
- Prepares Content to on makes the controller fetch immediately after creation
- to add a label to an object in IB
    - Identity Inspector
        - Label -> value
- set NSArrayController's label to Cars

Add the Views

- add a table view
- Attributes Inspector
    - Columns -> 3
- Column names -> Make/Model, Price, Special
- Delete the table cell view in the first column and drop and drop an Image & Text Table Cell View onto it
- add a number formatter to second column
    - Style -> Currency
- remove table cell view in third column and replace with a checkbox (Check Box Cell)
- Attributes Inspector for text fields in Make/Model column and Price column
    - Editable -> X
- add add and remove buttons, NSDatePicker, NSImage (Image Wells), NSLevelIndicator, and 2 labels
- NSImageView
    - Attributes Inspector
        - Editable -> X
- NSLevelIndicator
    - Attributes Inspector
        - Level Indicator
            - Style -> Rating
            - State -> Editable (X)
        - Value
            - Minimum -> 0
            - Maximum -> 5

- Select date picker, image view, 2 labels, and level indicator
    - top menu
        - Editor
            - Embed In
                - Box

Connections and Bindings

- table view
    - Content -> arrangedObjects of Car
    - Model Key Path -> ""
    - Selection Indexes -> selectionIndexes of Car
- each column's table cell view
    - First
        - Image View
            - Binding -> Value
            - Bind to -> Table Cell View
            - Controller Key -> ""
            - Key Path -> objectValue
        - Text Field
            - Binding -> Value
            - Bind to -> Table Cell View
            - Controller Key -> ""
            - Key Path -> objectValue
    - Second
        - Binding -> Value
        - Bind to -> Table Cell View
        - Controller Key -> ""
        - Key Path -> objectValue
    - Third
        - Binding -> Value
        - Bind to -> Table Cell View
        - Controller Key -> ""
        - Key Path -> objectValue
- add button -> action s/b add:
- remove button -> action s/b remove:
    - Bindings Inspector
        - Enabled -> Cars array controller
        - Controller Key -> canRemove
        - Model Key Path -> ""

- detail controls bindings
    - textual date picker with stepper
        - Binding -> Value
        - Bind to -> Cars
        - Controller Key -> selection
        - Key Path -> datePurchased
    - rating level indicator
        - Binding -> Value
        - Bind to -> Cars
        - Controller Key -> selection
        - Key Path -> condition
    - image well
        - Binding -> Value
        - Bind to -> Cars
        - Controller Key -> selection
        - Key Path -> photo
        - Conditionally Set Editable -> X
- NSBox
    - Title With Pattern
        - Display Pattern Title1 -> Cars
    - Controller Key -> selection
    - Model Key Path -> makeModel
    - Display Pattern -> Details for %{title}@
    - No Selection Placeholder ->
    - Null Placeholder -> <no Make/Model>
- first two column text fields
    - Font Bold
        - Value -> Table Cell View
        - Controller Key -> ""
        - Model Key Path -> objectValue.onSpecial

How Core Data Works

- NSPersistentDocument reads in the model and creates NSManagedObjectModel
- object model has 1 NSEntityDescription (describes Car)
    - has several instances of NSAttributeDescription
- doc creates instance of NSPersistentStoreCoordinator and NSManagedObjectContext
- NSManagedObjectContext fetches instances of NSManagedObject from the object store
- while objects are in memory, context observes them
    - when changed, registers undo with NSUndoManager

    - knows when objects need to be saved on change
- NSManagedObjectContext is the object most interacted with by programmers
    - fetch objects and save change to object graph

Fetching Objects from the NSManagedObjectContext

- create NSFetchRequest (typically only used once)
    - use name of entity
    - add NSPredicate and NSSortDescriptor to manage filtering and sorting

```
// Build the fetch request:
let fetchRequest = NSFetchRequest(entityName: "Car")
fetchRequest.predicate = NSPredicate(format: "price > = 1000")
fetchRequest.sortDescriptors = [NSSortDescriptor( key: "makeModel", ascending: tru
// Execute the fetch request:
var error: NSError?
let fetchResult: [AnyObject]? = managedObjectContext.executeFetchRequest(fetchRequ

// usage
if let cars = fetchResult as? [NSManagedObject] {
    for car in cars {
        if let makeModel = car.valueForKey("makeModel") as? String {
            print(" Fetched car: \( makeModel)")
        }
    }
}
```

Persistent Store Types

- supported persistent store types
    - for saving and loading
        - SQlite
        - XML
        - Binary
    - caching
        - In-Memory
- prefer SQLite for most applications
- XML is slower
- Core Data is not a database adapter so you cannot use your own databases with it
- see Core Data Programming Guide

Choosing a Cocoa Persistence Technology

- complex topic

- uses faulting to only load fetched objects
- handles tracking changes, helping to manage schema versions and more
- downside is that it is a black box and can be difficult to debug
- remember NSManagedObject instances are closely tied to the NSManagedObjectContext from which they were fetched
- archiving and core data both handle saving and loading, core data has speed over the need to save and load the entire object graph
- prefer core data for medium level persistence needs

Customizing Objects Created by NSArrayController

- create CarArrayController.swift and add

```
override func newObject() -> AnyObject {
    let newObj = super.newObject() as! NSObject
    let now = NSDate() newObj.setValue(now, forKey:" datePurchased")
    return newObj
}
```

- set class of array controller to CarArrayController in Identity Inspector

Challenge: Begin Editing on Add

- look at RaiseMan Document.addEmployee(_:)
- requires an outlet on CarArrayController

Challenge: Implement RaiseMan Using Core Data

14. User Defaults

- store preferences for behavior or appearance in user's defaults database

NSUserDefaults

- NSUserDefaults class
- can use factory defaults
- at an application level, the user modifies preferences
- NSUserDefaults works with defaults
    - register, write, and read defaults to reflect user's preferences
- uses strings for settings like keys in a dictionary

- shared defaults

```
class func standardUserDefaults() -> NSUserDefaults
// returns the shared defaults object
```

- registering defaults

```
func registerDefaults(registrationDictionary: [NSObject : AnyObject])
// registers the factory defaults for the application
```

- setting defaults

```
func setBool(value: Bool, forKey defaultName: String)
func setFloat(value: Float, forKey defaultName: String)
func setInteger(value: Int, forKey defaultName: String)
func setObject(value: AnyObject?, forKey defaultName: String)
// set the value for the defaultName default
// use the method that is correct for the type of value that is being stored
```

- removing defaults

```
func removeObjectForKey(defaultName: String)
// removes the value for the defaultName default
// the application will return to using the value given by the factory defaults or
// elsewhere
```

- reading defaults

```
func boolForKey(defaultName: String) -> Bool
func floatForKey(defaultName: String) -> Float
func integerForKey(defaultName: String) -> Int
func objectForKey(defaultName: String) -> AnyObject?
// read the value for the defaultName default
// usually this will be the user's previously stored default
// value, or, if it has not been set, the factory default
```

Adding User Defaults to SpeakLine

- use a PreferenceManager wrapper class

```
class PreferenceManager {
    private let userDefaults = NSUserDefaults.standardUserDefaults()
}
```

Create Names for the Defaults

```
private let activeVoiceKey = "activeVoice"
private let activeTextKey = "activeText"
```

- use global variables to allow compiler to check typos

Register Factory Defaults for the Preferences

- import Cocoa for NSSpeechSynthesizer

```
func registerDefaultPreferences() {
    let defaults = [
        activeVoiceKey : NSSpeechSynthesizer.defaultVoice(),
        activeTextKey : "Able was I ere I saw Elba."
    ]
    userDefaults.registerDefaults(defaults)
}

init() { registerDefaultPreferences() }
```

- add init to prevent any code from registering the value for a default before the factory default has been registered
- add to MainWindowController

```
let preferenceManager = PreferenceManager()
```

Reading the Preferences

- add computed properties to preference class
    - activeVoice
    - activeText

```
var activeVoice: String? {
    get { return userDefaults.objectForKey(activeVoiceKey) as? String }
}
var activeText: String? {
    get { return userDefaults.objectForKey(activeTextKey) as? String }
}
```

Reflecting the Preferences in the UI

- read properties from from PreferenceManager in MainWindowController on window load

```
func windowDidLoad() {
    super.windowDidLoad()
    updateButtons()
    speechSynth.delegate = self
    for voice in voices {
        print(voiceNameForIdentifier(voice)!)
    }
    let defaultVoice = preferenceManager.activeVoice! // **
    if let defaultRow = find(voices, defaultVoice) {
        let indices = NSIndexSet(index: defaultRow)
        tableView.selectRowIndexes(indices, byExtendingSelection: false)
        tableView.scrollRowToVisible(defaultRow)
    }
    textField.stringValue = preferenceManager.activeText! // **
}
```

Writing the Preferences to User Defaults

```
var activeVoice: String? {
    set (newActiveVoice) {
        userDefaults.setObject(newActiveVoice, forKey: activeVoiceKey)
    }
    get { return userDefaults.objectForKey( activeVoiceKey) as? String }
}
var activeText: String? {
    set (newActiveText) {
        userDefaults.setObject(newActiveText, forKey: activeTextKey)
    }
    get { return userDefaults.objectForKey( activeTextKey) as? String }
}
```

Storing the User Defaults

- in tableViewSelectionDidChange save user's selection of voice into preferences

```
preferenceManager.activeVoice = voice
```

- persist spoken text across runs
    - conform to NSTextFieldDelegate
    - connect IB text field connection menu
        - delegate -> File's Owner

- implement controlTextDidChange(_:)

```
override func controlTextDidChange(notification: NSNotification) {
    preferenceManager.activeText = textField.stringValue
}
```

What Can Be Stored in NSUserDefaults?

- property list objects: instances of Objective-C classes
  - NSDictionary
  - NSArray
  - NSString
  - NSNumber
- plist data structure must be composed entirely of these objects
- can store any data structure in NSUserDefaults that can be represented as a plist
- Swift basic types are bridged to these types so they can be stored without issues

Precedence of Types of Defaults

- highest to lowest priority
  - CLI args
    - rarely used on mac OS in production app
  - application
    - settings in the user's defaults database.
  - global defaults
    - system-wide defaults
  - language defaults
    - based on the user's preferred language
  - registered defaults
    - factory defaults registered at launch

What is the User's Defaults Database?

- ~/Library/Preferences/*.plist (can be viewed with Xcode, binary plist files)
- uses bundle identifier as the name
- could edit on older OSes but managed by daemon in Mavericks and later
- use `defaults` command-line tool for Mavericks and later

For the More Curious: Reading/ Writing Defaults from the Command Line

- read

```
defaults read com.apple.dt.Xcode
```

- write

```
defaults write com.apple.Xcode NSNavLastRootDirectoryForOpen /Users
```

- read global defaults

```
defaults read NSGlobalDomain
```

For the More Curious: NSUserDefaultsController

- to bind to a value from the NSUserDefaults object, use NSUserDefaultsController
- SpeakLine's text field Bindings Inspector
  - Bind to -> Shared User Defaults Controller
  - Controller Key -> values
  - Model Key Path -> activeTextKey

Challenge: Reset Preferences

- must update window to reflect changes after reset

15. Alerts and Closures

- use NSAlert to warn user

NSAlert

- for simple alert

```
let alert = NSAlert()
alert.messageText = "Alert message."
alert.informativeText = "A more detailed description of the situation."
alert.addButtonWithTitle("Default")
```

```
alert.addButtonWithTitle("Alternative")
alert.addButtonWithTitle("Other")
let result = alert.runModal()
```

- NSAlert properties
  - configure the text that the alert will display
    - var messageText: String?
    - var informativeText: String?
  - configure the images that the alert will display (you can supply an icon but by default the alert will use the application's icon)
    - var icon: NSImage!
    - var alertStyle: NSAlertStyle
- NSAlert methods
  - add the buttons on alert
    - func addButtonWithTitle(title: String) -> NSButton
  - present
    - func runModal
- check result

```
let result = alert.runModal()
switch result {
    case NSAlertFirstButtonReturn: // The user clicked "Default"
        break
    case NSAlertSecondButtonReturn: // The user clicked "Alternative"
        break
    case NSAlertThirdButtonReturn: // The user clicked "Other"
        break
    default:
        break
}
```

- automatically adds an OK button if no buttons are added

Modals and Sheets

- shows modal and blocks until user dismisses the alert
- can also run as a sheet on top of other windows
  - stops events for related window
  - other windows in app will receive events

Completion Handlers and Closures

- completion handler = callback
  - wait for user

    - network
    - calculations on a separate thread
- uses closures in Swift

```
let say = { (text: String) -> Void in print(" The closure says: \( text)"") }
say(" Hello")
```

- for alerts as sheets, configure alert and then use beginSheetModalForWindow (_:completionHandler:)

```
let alert = NSAlert() ...
let completionHandler = { (response: NSModalResponse) -> Void in print("The user s
alert.beginSheetModalForWindow(window, completionHandler: completionHandler)
```

Closures and capturing

- can capture constants and variables in enclosing scope
- captures reference objects with strong reference by default
- this can cause strong reference cycle
- solution is a capture list

```
chauffeur.completionHandler = { [unowned chauffeur] in chauffeur.sendMessage(" I h
```

- use keywords unowned and weak
- unowned -> captured reference should never be deallocated before the closure (similar to implicitly unwrapped optional)
- weak -> treats capture as optional and may be deallocated before closure
- to capture properties or methods, must use the self reference and can include in capture list

Make the User Confirm the Deletion

- in RaiseMan open Document.xib
    - select table view Attributes Inspector
        - Selection
            - Multiple -> X
- Document -> display alert as sheet and call remove depending on selection

```
@IBAction func removeEmployees(sender: NSButton) {
    let selectedPeople: [Employee] = arrayController.selectedObjects as! [Employee
    let alert = NSAlert()
    alert.messageText = "Do you really want to remove these people?"
    alert.informativeText = "\(selectedPeople.count) people will be removed."
```

```
    alert.addButtonWithTitle("Remove")
    alert.addButtonWithTitle("Cancel")
    let window = sender.window!
    alert.beginSheetModalForWindow(
        window,
        completionHandler: { (response) -> Void in
            // If the user chose "Remove", tell the array controller to delete the
            switch response {
                // The array controller will delete the selected objects
                // The argument to remove() is ignored
                case NSAlertFirstButtonReturn:
                    self.arrayController.remove(nil)
                default: break
            }
        }
    )
}
```

- need to change target action on remove button
  - File's Owner -> removeEmployees:

For the More Curious: Functional Methods and Minimizing Closure Syntax

- map, reduce, filter
- if closure is last parameter it can sit outside of parens
- if closure is only parameter, no parens are needed
- if closure is single statement, can leave off return keyword from closure
- for simple closures, can use $0, $1, etc for parameters

Challenge: Don’t Fire Them Quite Yet

- add "Keep, but no raise" button which does not delete employee and sets raise to 0

Challenge: Different Messages for Different Situations

- modify informativeText to handle various plural cases and display the name of the employee for one person
- modify messageText to handle various plural cases

16. Using Notifications

What Notifications Are

- each application has a single instance of NSNotificationCenter
- objects register as observers and posters
- when a poster posts a notification that is forwarded to all related observers

What Notifications Are Not

- not IPC, limited to same application
- look at NSDistributedNotification center for passing notifications between applications

NSNotification

- properties
  - name: String
  - object: AnyObject?
- object is almost always the object that posted the notification

NSNotificationCenter

- 3 main pieces of functionality:
  - register observers
  - post notifications
  - unregister observers
- commonly used methods

```
class func defaultCenter() -> NSNotificationCenter
func addObserver(observer: AnyObject, selector aSelector: Selector, name aName: St
func postNotification(notification: NSNotification)
func postNotificationName(aName: String, object anObject: AnyObject?)
func removeObserver(observer: AnyObject)
```

Starting the Chatter Application

- create Xcode project as usual (uncheck Use Storyboards, Create Document-Based Application, and Use Core Data)
- remove MainMenu.xib window and outlet to window in AppDelegate.swift

- create ChatWindowController.swift and .xib
- xib Attributes Inspector
    - Visible at Launch -> `[]` (will not display until showWindow is called)
- add windowNibName

```
class ChatWindowController: NSWindowController {
    // MARK: - Lifecycle override
    var windowNibName: String { return "ChatWindowController" }
    ...
```

- in AppDelegate

```
var windowControllers: [ChatWindowController] = []

func addWindowController() {
    let windowController = ChatWindowController()
    windowController.showWindow(self)
    windowControllers.append(windowController)
}

func applicationDidFinishLaunching(aNotification: NSNotification) { addWindowContr

@IBAction func displayNewWindow(sender: NSMenuItem) { addWindowController() }
```

- in .xib ctrl-drag from menu
    - File -> New -> App Delegate
        - action -> displayNewWindow:
- in ChatWindowController.swift

```
dynamic var log: NSAttributedString = NSAttributedString(string: "")
dynamic var message: String? // NSTextView does not support weak references.
@IBOutlet var textView: NSTextView!

@IBAction func send(sender: AnyObject) { }
```

- add text view, text field and button to .xib
- text view Bindings Inspector
    - Attributed String -> File's Owner log property
- text view Attributes Inspector
    - Editable -> `[]`
- File's Owner's textView -> text view
- text field Bindings Inspector
    - Value -> File's Owner message property
- button connections pop-up
    - target -> File's Owner

    - action -> send:
- window connections pop-up
    - initialFirstResponder -> text field (makes text field active on window open)

Using Notifications in Chatter

- each window controller posts a notification when a user sends a message
- each window observes and when they observe a message they update text views to display message text
- in ChatWindowController.swift
    i. name of notification to post
    ii. key in the notification's user info dictionary to store the message

```
// global constants
private let ChatWindowControllerDidSendMessageNotification = "com.bignerdranch.cha
private let ChatWindowControllerMessageKey = "com.bignerdranch.chatter.ChatWindowC

... in class

override func windowDidLoad() {
    super.windowDidLoad()

    let notificationCenter = NSNotificationCenter.defaultCenter()
    notificationCenter.addObserver(self, selector: Selector("receiveDidSendMessage
}

@IBAction func send(sender: AnyObject) {
    sender.window?.endEditingFor(nil)
    if let message = message {
        let userInfo = [ChatWindowControllerMessageKey : message]
        let notificationCenter = NSNotificationCenter.defaultCenter()
        notificationCenter.postNotificationName(ChatWindowControllerDidSendMessage
    }
    message = ""
}

// ChatWindowControllerDidSendMessageNotification
func receiveDidSendMessageNotification(note: NSNotification) {
    let mutableLog = log.mutableCopy() as! NSMutableAttributedString
    if log.length > 0 {
        mutableLog.appendAttributedString(NSAttributedString( string: "\ n"))
    }
    let userInfo = note.userInfo! as! [String : String]
    let message = userInfo[ChatWindowControllerMessageKey]!
    let logLine = NSAttributedString(string: message)
    mutableLog.appendAttributedString(logLine)
    log = mutableLog.copy() as! NSAttributedString
    textView.scrollRangeToVisible(NSRange( location: log.length, length: 0))
}

deinit {
```

```
    let notificationCenter = NSNotificationCenter.defaultCenter()
    notificationCenter.removeObserver(self)
}
```

- add ChatWindowController to the default notification center as an observer of ChatWindowControllerDidSendMessageNotification whenever any object posts to it inside windowDidLoad
- send
  - check if message is non-nil
  - if not, add to user info dictionary
  - post to notification center
- handle a notification by posting text to text view and scroll to ensure display
- windows observe notifications from themselves as well
- add cleanup in deinit

For the More Curious: Delegates and Notifications

- a delegate for an object should (most-likely) receive notifications for that object
- for standard Cocoa objects, the delegate is automatically registered as an observer for the methods it implements
- method naming convention
  - name of notification -> NS?some notification name?Notification
  - downcase first letter ?some notification name? (removing NS and Notification)
  - argument is (notification: NSNotification)
- refer to docs for each delegate type

Challenge: Beep-beep!

- to beep when relinquishing an application's active status, use NSApplication's NSApplicationDidResignActiveNotification notification
- AppDelegate handles this
- NSBeep() causes a system beep

Challenge: Add Usernames

- add text field to the top of ChatWindowController to enter a username
- add username or a good default to notifications
- display username in text window

Challenge: Colored Text

- to distinguish messages from current window vs other windows
    - in receiveDidSendMessageNotification(_:) handle posting by self
    - add attribute to logLine to display other color with NSForegroundColorAttributeName

Challenge: Disabling the Send Button

- should disable send button when no text has been entered
- bind send button's Enabled -> File's Owner message property
- to handle String -> Bool conversion, use subclass of NSValueTransformer (IsNoteEmptyTransformer)
- update button state as user types
    - text field's Bindings Inspector
        - Value -> Continuously Updates Value -> X

17. NSView and Drawing

- visible objects are windows or views (NSWindow or NSView)
- window has collection of views
    - view draws inside rectangles and handles events
- subclasses of NSView so far
    - NSButton
    - NSTextField
    - NSTableView
    - NSColorWell
- to expand functionality, subclass NSView

Setting Up the Dice Application

- setup Xcode project (unchecking usual)
- add AppDelegate snippet

```
class AppDelegate: NSObject, NSApplicationDelegate {
    var mainWindowController: MainWindowController?
    func applicationDidFinishLaunching(aNotification: NSNotification) {
        let mainWindowController = MainWindowController()
        mainWindowController.showWindow(self)
        self.mainWindowController = mainWindowController
```

```
    }
}
```

- create MainWindowController.swift and .xib
- override windowNibName

```
class MainWindowController: NSWindowController {
    override var windowNibName: String { return "MainWindowController" }
    override func windowDidLoad() { super.windowDidLoad() }
}
```

Creating a view subclass

- create DieView class

```
import Cocoa
class DieView: NSView { }
```

- add Custom View to .xib
- custom view Identity Inspector
    - Custom Class
        - Class -> DieView

Views, Rectangles, and Coordinate Systems frame bounds

- to display a rectangle of a view, Cocoa uses
    - uses origin (lower left corner in relation to origin of superview)
    - size
- frame
    - NSRect
        - origin: NSPoint (lower left corner as offset from superview)
        - size: NSSize
- NSPoint
    - x: CGFloat
    - y: CGFloat
- NSSize:
    - width: CGFloat
    - height: CGFloat
- all are structs for performance and to avoid bugs as value types
- NSPoint and NSSize are measured in points (not related to NSPoint)
- point is 1/72 of an inch (from typography) because original mac used 72 pixels per inch

- points are resolution independent (facilitates supporting both standard and retina displays easily)
- NSRect, NSPoint, and NSSize are type aliases for the Core Graphics types CGRect, CGPoint, and CGSize
- bounds
    - same type as frame
    - frame is used to position subviews of the view
    - bounds are used to draw the view itself

Custom Drawing

- three steps
    - set color (using NSColor with RGB, HSV, CMYK, or grayscale)
    - construct path (NSBezierPath, can store as property)
    - draw path in color (NSBezierPath -> stroke(), shapes -> fill())
- handle steps in NSView drawRect(_:)

drawRect(_:)

```
class DieView: NSView {
    override func drawRect(dirtyRect: NSRect) {
        let backgroundColor = NSColor.lightGrayColor()
        backgroundColor.set()
        NSBezierPath.fillRect(bounds)

        NSColor.greenColor().set()
        let path = NSBezierPath()
        path.moveToPoint(NSPoint(x: 0, y: 0))
        path.lineToPoint(NSPoint(x: bounds.width, y: bounds.height))
        path.stroke()
    }
}
```

- notice use of bounds.width and bounds.height

When is my view drawn?

- drawRect called automatically
- don't redraw with drawRect directly, set view's needsDisplay to true
- sets view to dirty and system will call drawRect on all dirty views
- do not need to mark Cocoa's standard views as dirty because they handle that themselves
- mark custom views as dirty on change

Graphics contexts and states

- Cocoa configures graphics context for a view before calling drawRect (held in thread local storage)
- tracks current color, font and transform
- drawing commands read and write to that state

Drawing a die face

- in DieView

```
// for dots
var intValue: Int? = 1 { didSet { needsDisplay = true } }

func metricsForSize(size: CGSize) -> (edgeLength: CGFloat, dieFrame: CGRect) {
    let edgeLength = min(size.width, size.height)
    let padding = edgeLength / 10.0
    let drawingBounds = CGRect(x: 0, y: 0, width: edgeLength, height: edgeLength)
    let dieFrame = drawingBounds.rectByInsetting(dx: padding, dy: padding)
    return (edgeLength, dieFrame)
}

func drawDieWithSize(size: CGSize) {
    if let intValue = intValue {
        let (edgeLength, dieFrame) = metricsForSize(size)
        let cornerRadius:CGFloat = edgeLength / 5.0

        // Dot positioning
        let dotRadius = edgeLength / 12.0
        let dotFrame = dieFrame.rectByInsetting(dx: dotRadius * 2.5, dy: dotRadius

        // Draw the rounded shape of the die profile:
        NSColor.whiteColor().set()
        NSBezierPath(roundedRect: dieFrame, xRadius: cornerRadius, yRadius: corner

        // Dot drawing
        NSColor.blackColor().set()
        // Nested function to make drawing dots cleaner:
        func drawDot(u: CGFloat, v: CGFloat) {
            let dotOrigin = CGPoint(x: dotFrame.minX + dotFrame.width * u, y: dotF
            let dotRect = CGRect( origin: dotOrigin, size: CGSizeZero).rectByInset
            NSBezierPath(ovalInRect: dotRect).fill()
        }

        // If intValue is in range...
        if find(1...6, intValue) != nil {
            // Draw the dots:
            if find([ 1, 3, 5], intValue) != nil {
                drawDot(0.5, 0.5) // Center dot
            }
```

```
            if find(2...6, intValue) != nil {
                drawDot( 0, 1) // Upper left
                drawDot( 1, 0) // Lower right
            }
            if find(4...6, intValue) != nil {
                drawDot( 1, 1) // Upper right
                drawDot( 0, 0) // Lower left
            }
            if intValue == 6 {
                drawDot( 0, 0.5) // Mid left/right
                drawDot( 1, 0.5)
            }
        }
    }
}

override func drawRect(dirtyRect: NSRect) {
    let backgroundColor = NSColor.lightGrayColor()
    backgroundColor.set()
    NSBezierPath.fillRect(bounds)
    drawDieWithSize(bounds.size)
}
```

Saving and Restoring the Graphics State

- sometimes useful to treat graphics state like a stack, saving and restoring at points
- adding to drawDieWithSize causes all subsequent drawing commands to add shadow

```
let shadow = NSShadow()
shadow.shadowOffset = NSSize(width: 0, height: -1)
shadow.shadowBlurRadius = edgeLength/ 20 shadow.set()
```

- use NSGraphicsContext: saveGraphicsState() and restoreGraphicsState()

```
NSGraphicsContext.saveGraphicsState()
let shadow = NSShadow()
shadow.shadowOffset = NSSize(width: 0, height: -1)
shadow.shadowBlurRadius = edgeLength/ 20 shadow.set()
// Draw the rounded shape of the die profile:
NSColor.whiteColor().set()
NSBezierPath(roundedRect: dieFrame, xRadius: cornerRadius, yRadius: cornerRadius).
NSGraphicsContext.restoreGraphicsState()
// Shadow will not apply to subsequent drawing commands
```

Cleaning up with Auto Layout

- view instance does not currently resize

- select DieView in .xib
    - four Auto Layout buttons in bottom right of canvas
    - click pin button
        - in popover
            - Add New Constraints -> click all four struts
            - set all four text fields to 20
- if a mistake is made, click Resolve Auto Layout Issues
- a view's intrinsic content size is the natural size of its content
- Auto Layout will not reduce a view's size below its intrinsic content size
- in DieView

```
override var intrinsicContentSize: NSSize { return NSSize( width: 20, height: 20)
```

Drawing Images

- create ImageTiling Xcode project
- create TiledImageView.swift and .xib

```
import Cocoa class TiledImageView: NSView {
    var image: NSImage?
    let columnCount = 5
    let rowCount = 5

    override func drawRect(dirtyRect: NSRect) {
        super.drawRect(dirtyRect)
        if let image = image {
            for x in 0..<columnCount {
                for y in 0..<rowCount {
                    let frame = frameForImageAtLogicalX(x, y: y)
                    image.drawInRect(frame)
                }
            }
        }
    }

    func frameForImageAtLogicalX(logicalX: Int, y logicalY: Int) -> CGRect {
        let spacing = 10
        let width = 100
        let height = 100
        let x = (spacing + width) * logicalX
        let y = (spacing + height) * logicalY
        return CGRect( x: x, y: y, width: width, height: height)
    }
}
```

- add Custom View to .xib

- need to set image (could add outlet to image view and set image property in windowDidLoad)

Inspectable properties and designable views

- in TiledImageView class

```
@IBInspectable var image: NSImage?
```

- in .xib, select image view
  - Attributes Inspector
    - Image -> image name (NSComputer)
- to see from Xcode add @IBDesignable to class

Drawing images with finer control

- use the following method

```
func drawInRect(
    rect: NSRect,
    fromRect: NSRect,
    operation op: NSCompositingOperation,
    fraction delta: CGFloat
)
```

Scroll Views

- NSScrollView is used for cases where view overflows displayable area
- overflowing view is known as document view
- displayed portion is visible rect
- in TiledImageView .xib
  - select image view
    - menu Editor -> Embed In -> Scroll View
- give view an intrinsic content size

```
override var intrinsicContentSize: NSSize {
    let furthestFrame = frameForImageAtLogicalX(columnCount-1, y: rowCount-1)
    return NSSize(width: furthestFrame.maxX, height: furthestFrame.maxY)
}
```

Creating Views Programmatically

- create a button

```
let superview = window.contentView
let frame = NSRect(x: 10, y: 10, width: 200, height: 100)
let button = NSButton(frame: frame)
button.title = "Click me!"
superview.addSubview( button)
```

- to recreate the style of an existing view, add view in IB and go through properties in Attributes Inspector
- use playground for rapid iteration

For the More Curious: Core Graphics and Quartz

- AppKit drawing APIs use Core Graphics (generally referred to as Quartz)
- generally same concepts within a C API and uses CGContextRef
- lower-level and sometimes finer control of drawing

For the More Curious: Dirty Rects

- for time-consuming drawing, best to only redraw dirty rects
- setting needsDisplay to true redraws entire rectangle
- can use

```
let dirtyRect = NSRect(x: 0, y: 0, width: 50, height: 50)
myView.setNeedsDisplayInRect(dirtyRect)
```

For the More Curious: Flipped Views

- AppKit views follow Cartesian coordinates by default
- flipped views
    - origin upper left
    - y increases as coordinates go down rectangle
- to use flipped views

```
override var flipped: Bool { return true }
```

Challenge: Gradients

- pass NSBezierPath to a method on NSGradient

Challenge: Stroke

- use a stroke to add border to NSBezierPath

Challenge: Make DieView Configurable from Interface Builder

- use @IBDesignable to display DieView in IB

18. Mouse Events

NSResponder

- all event handling methods are declared in NSResponder
- mouse events

```
func mouseDown(theEvent: NSEvent)
func rightMouseDown(theEvent: NSEvent)
func otherMouseDown(theEvent: NSEvent)
func mouseUp( theEvent: NSEvent)
func rightMouseUp(theEvent: NSEvent)
func otherMouseUp(theEvent: NSEvent)
func mouseDragged(theEvent: NSEvent)
func scrollWheel(theEvent: NSEvent)
func rightMouseDragged(theEvent: NSEvent)
func otherMouseDragged(theEvent: NSEvent)
```

NSEvent

- interesting properties for mouse events

```
var locationInWindow: NSPoint { get }
var modifierFlags: NSEventModifierFlags { get }
var timestamp: NSTimeInterval { get }
unowned(unsafe) var window: NSWindow! { get }
var clickCount: Int { get }
var pressure: Float { get }
var deltaX: CGFloat { get }
var deltaY: CGFloat { get }
var deltaZ: CGFloat { get }
```

Getting Mouse Events

- in DieView

```
override func mouseDown(theEvent: NSEvent) {
    print("mouseDown")
}
override func mouseDragged(theEvent: NSEvent) {
    print("mouseDragged location: \(theEvent.locationInWindow)")
}
override func mouseUp(theEvent: NSEvent) {
    print("mouseUp clickCount: \(theEvent.clickCount)")
}
```

Click to Roll

- randomize side to show

```
func randomize() { intValue = Int( arc4random_uniform( 5)) + 1 }
```

- check for double-click

```
override func mouseUp(theEvent: NSEvent) {
    print("mouseUp clickCount: \(theEvent.clickCount)")
    if theEvent.clickCount = = 2 { randomize() }
}
```

- add interaction for mouseDown and mouseUp

```
var pressed: Bool = false { didSet { needsDisplay = true } }
// in mouseDown set pressed to true
// in mouseUp set pressed to false

// in metrics for size change dieFrame
var dieFrame = drawingBounds.rectByInsetting(dx: padding, dy: padding)
if pressed { dieFrame = dieFrame.rectByOffsetting(dx: 0, dy: -edgeLength/ 40) }

// in drawDieWithSize
shadow.shadowBlurRadius = (pressed ? edgeLength/ 100 : edgeLength/ 20)
```

Improving Hit Detection

- can check hit detection with CGRect contains(_:) and metricsForSize(:) to get rectangle
- NSEvent provides locationInWindow

- to convert locationInWindow, NSView has

```
func convertPoint(aPoint: NSPoint, fromView aView: NSView?) -> NSPoint
func convertPoint(aPoint: NSPoint, toView aView: NSView?) -> NSPoint
```

- can call with nil
- update mouseDown

```
let dieFrame = metricsForSize(bounds.size).dieFrame
let pointInView = convertPoint(theEvent.locationInWindow, fromView: nil)
pressed = dieFrame.contains(pointInView)
```

Gesture Recognizers

- similar to UIKit's gesture recognizer on iOS
- main gesture recognizers
  - NSClickGestureRecognizer
  - NSPanGestureRecognizer
  - NSMagnificationGestureRecognizer
  - NSPressGestureRecognizer
  - NSRotationGestureRecognizer
- provide much more fine grained and responsive control over gestures and mouse events
- see section for code examples

Challenge: NSBezierPath-based Hit Testing

- NSBezierPath has a containsPoint method that will make the hit testing more accurate

Challenge: A Drawing App

- use init(ovalInRect: NSRect) -> NSBezierPath to allow user to draw ovals in arbitrary locations
- add save and read
- add undo

19. Keyboard Events

- typing event -> window manager -> active application -> key window -> active view
- active view is determined by firstResponder property on a view

- when user changes active view, views check whether they can accept firstResponder status
  - NO means it will not accept first responder status
  - current first responder then checked to resign first responder status

NSResponder

```
var acceptsFirstResponder: Bool { get }
func resignFirstResponder() -> Bool
func becomeFirstResponder() -> Bool
func keyDown(theEvent: NSEvent)
func keyUp(theEvent: NSEvent)
func flagsChanged(theEvent: NSEvent)
```

NSEvent

- common keyboard event properties

```
var characters: String! { get }
var ARepeat: Bool { get }
var keyCode: UInt16 { get }
var modifierFlags: NSEventModifierFlags { get }
```

Adding Keyboard Input to DieView

Accept first responder

- set accepts first responder

```
override var acceptsFirstResponder: Bool { return true }
override func becomeFirstResponder() -> Bool { return true }
override func resignFirstResponder() -> Bool { return true }
```

- cannot receive keyboard events without accepting first responder

Receive keyboard events

- do not need to implement keyDown and interpret characters manually
- use interpretKeyEvents which calls insertText

```
override func keyDown(theEvent: NSEvent) { interpretKeyEvents([theEvent]) }
override func insertText(insertString: AnyObject) {
    let text = insertString as! String
```

```
    if let number = text.toInt() { intValue = number }
}
```

Putting the dice in Dice

- remove constraints
  - Editor -> Resolve Auto Layout Issues -> Clear Constraints
- resize die view to 104x104 in Size Inspector
- duplicate die view twice
- change title of window to Dice

Focus Rings

- focus = first responder
- generally indicated with blue ring
- for table views -> selection highlight may change from gray to blue

```
override func drawFocusRingMask() { NSBezierPath.fillRect(bounds) }
override var focusRingMaskBounds: NSRect { return bounds }
```

- can draw any path with drawFocusRingMask

The Key View Loop

- sequence of first responder-accepting views in each window
- Tab or Shift-Tab cycles them
- can manually modify with nextKeyView and initialFirstResponder
- to handle tab events add insertTab and inserBacktab

```
override func insertTab(sender: AnyObject?) { window?.selectNextKeyView(sender) }
override func insertBacktab(sender: AnyObject?) { window?.selectPreviousKeyView(se
```

- cmd-W should close the window

For the More Curious: Rollovers

- use mouseMoved, mouseEntered, mouseExited
- set acceptsMouseMovedEvents to true
- for rollovers, generally use mouseEntered and mouseExited

- when a view is added to a view's hierarchy, viewDidMoveToWindow is called
  - MouseEnteredAndExited -> mouseEntered(_:) and mouseExited(:) will be called on the owner of the tracking area
  - ActiveAlways -> the area will be active regardless of first-responder status
  - InVisibleRect -> tells the tracking area to ignore the rect parameter and automatically match the view's visible rect as the tracking area bounds
- see section for code examples

20. Drawing Text with Attributes

NSFont

- two types of methods
  - class methods for getting the font you want
  - methods for getting metrics on the font, such as letter height

```
class func systemFontOfSize(fontSize: CGFloat) -> NSFont!
class func boldSystemFontOfSize(fontSize: CGFloat) -> NSFont!
class func labelFontOfSize(fontSize: CGFloat) -> NSFont!
class func userFontOfSize(fontSize: CGFloat) -> NSFont!
class func userFixedPitchFontOfSize(fontSize: fontSize: CGFloat) -> NSFont!
class func titleBarFontOfSize(fontSize: CGFloat) -> NSFont!
class func messageFontOfSize(fontSize: CGFloat) -> NSFont!
class func toolTipsFontOfSize(fontSize: CGFloat) -> NSFont!

init(name fontName: String!, size fontSize: CGFloat) -> NSFont
```

- recommended to use previous methods over init to maintain consistency with the system

NSAttributedString

- used to set attributes for a range of a string
- NSRange
  - location: Int
  - length: Int
- can use NSMakeRange() but faster to use range expression initializer NSRange(0...4)
- Cocoa has NSAttributedString and NSMutableAttributedString

```
let attrStr = NSMutableAttributedString(string: "Big Nerd Ranch")
attrStr.addAttribute(
    NSFontAttributeName,
    value: NSFont.boldSystemFontOfSize(22),
    range: NSRange(0...2)
)
```

```
attrStr.addAttribute(
    NSForegroundColorAttributeName,
    value: NSColor.orangeColor(),
    range: NSRange(4...7)
)
attrStr.addAttribute(
    NSUnderlineStyleAttributeName,
    value: 1,
    range: NSRange(9...13)
)
```

- global vars for most commonly used attributes
    - NSFontAttributeName - font object - 12-point Helvetica
    - ForegroundColorAttributeName - color - Black
    - ParagraphStyleAttributeName - NSParagraphStyle object - Standard paragraph style
    - UnderlineColorAttributeName - color - same as the foreground
    - UnderlineStyleAttributeName - number - 0
    - SuperscriptAttributeName - number - 0 (which means no superscripting or subscripting)
    - ShadowAttributeName - NSShadow object
- easiest way to create is from a file
- can read
    - string: typically from a plain-text file
    - RTF: Rich Text Format is a standard for text with multiple fonts and colors read and set the contents of the attributed string with an instance of NSData
    - RTFD: RTF with attachments and images
    - HTML: attributed string can do basic HTML layout use WebView for best quality
    - Word: attributed string can read and write simple .doc files
    - OpenOffice: attributed string can read and write simple .odt files
- dictionary can contain extra data for attributed string or supply nil

```
let rtfUrl = ... // Obtain an NSURL
var error: NSError?
let rtfData = NSData.dataWithContentsOfURL(
    rtfUrl,
    options: NSDataReadingOptions.DataReadingMappedIfSafe, error: &error) // -> NS
)
if rtfData != nil {
    var docAttrs: NSDictionary?
    let rtfAttrStr = NSAttributedString(
        data: rtfData,
        options: nil,
        documentAttributes: &docAttrs,
        error: &error
    )
    button.attributedTitle = rtfAttrStr
}
```

Drawing Strings and Attributed Strings

- drawing NSAttributedString onto views

```
func drawAtPoint(point: NSPoint)
func drawInRect(rect: NSRect)
var size: NSSize { get }
```

- drawing NSString onto views

```
func drawAtPoint(point: NSPoint, withAttributes attrs: [NSObject : AnyObject]!)
func drawInRect(rect: NSRect, withAttributes attrs: [NSObject : AnyObject]!)
func sizeWithAttributes(attrs: [NSObject : AnyObject]!) -> NSSize
```

Drawing Text Die Faces

- in DieView drawDieWithSize

```
...
    if intValue == 6 {
        drawDot(0, 0.5) // Mid left/right
        drawDot(1, 0.5)
    }
} else {
    var paraStyle = NSParagraphStyle.defaultParagraphStyle().mutableCopy() as! NSM
    paraStyle.alignment = .CenterTextAlignment
    let font = NSFont.systemFontOfSize(edgeLength * 0.5)
    let attrs = [NSForegroundColorAttributeName: NSColor.blackColor(), NSFontAttri
    let string = "\(intValue)" as NSString string.drawInRect(dieFrame, withAttribu
}
```

- options to center
    i. inline in the method where it is needed -> works but clutters code
    ii. add a method to the class to find the correct rect -> enables encapsulation, options for reuse are limited to instances of this class
    iii. create new method on NSString that draws the centered string -> best choice... enables code reuse (use extensions)

Extensions

- cannot add stored properties on extensions
- can add computed properties but cannot add new storage to a class using them
- in NSString+Drawing.swift

```
import Cocoa

extension NSString {
    func drawCenteredInRect(rect: NSRect, attributes: [NSString: AnyObject]!) {
        let stringSize = sizeWithAttributes(attributes)
        let point = NSPoint(x: rect.origin.x + (rect.width - stringSize.width) / 2
        drawAtPoint(point, withAttributes: attributes)
    }
}
```

- use in DieView

```
let string = "\(intValue)" as NSString
string.drawCenteredInRect(dieFrame, attributes: attrs)
```

- common naming convention for extension files is ?ClassName?+?descr of extension?.swift

Getting Your View to Generate PDF Data

- AppKit provides methods for drawing views to PDFs

```
func dataWithPDFInsideRect( rect: NSRect) -> NSData
```

- in DieView

```
@IBAction func savePDF(sender: AnyObject!) {
    let savePanel = NSSavePanel()
    savePanel.allowedFileTypes = ["pdf"]
    savePanel.beginSheetModalForWindow(
        window!,
        completionHandler: { [unowned savePanel] (result) in
            if result == NSModalResponseOK {
                let data = self.dataWithPDFInsideRect(self.bounds)
                var error: NSError?
                let ok = data.writeToURL(
                    savePanel.URL!,
                    options: NSDataWritingOptions.DataWritingAtomic,
                    error: &error
                )
                if !ok {
                    let alert = NSAlert(error: error!)
                    alert.runModal()
                }
            }
        }
    )
}
```

- in .xib, add new menu item from object library to the File menu

- relabel to Save To PDF...
- ctrl-drag from new menu item to First Responder
    - connect to savePDF: method

For the More Curious: NSFontManager

- NSFontManager can be used to customize fonts

```
let fontManager = NSFontManager.sharedFontManager()
let boldFont = fontManager.convertFont(someFont, toHaveTrait: NSFontTraitMask.Bold
```

Challenge: Color Text as SpeakLine Speaks It

- from NSSpeechSynthesizerDelegate, add

```
optional func speechSynthesizer(sender: NSSpeechSynthesizer, willSpeakWord charact
```

- use the range and NSAttributedString to change colors of words

```
let range: NSRange = ...
let theString: String = ...
let attributedString = NSMutableAttributedString(string: theString)
attributedString.addAttribute(NSForegroundColorAttributeName, value:NSColor.greenC
```

- read and write text field's attributedStringValue property (from NSControl)
- enable and disable text field during speaking

```
func updateTextField() {
    if speechSynth.speaking { textField.enabled = false }
    else { textField.enabled = true }
}
```

21. Pasteboards and Nil-Targeted Actions

- pasteboard server process (/usr/sbin/pboard) -> daemon that manages pasteboard
- the same data can be copied onto the pasteboard in many formats and the application that reads the data can choose the format to read it as
- UTIs are used for the various types of data on the pasteboard
- application typically clears before writing to the pasteboard
- there is a pasteboard writing protocol
- pasteboard asks for an array of classes

- pasteboard can also work with data lazily
- there are also APIs for working with the pasteboard with a finer-grained control
- multiple pasteboards
    - general
    - drag-and-drop
    - copy and paste
    - last searched string
    - ruler copying
    - font copying

NSPasteboard

- NSPasteboard methods

```
class func generalPasteboard() -> NSPasteboard!
init( name: String!) -> NSPasteboard
func clearContents() -> Int
func writeObjects(objects: [AnyObject]!) -> Bool
func readObjectsForClasses(classArray: [AnyObject]!, options: [NSObject : AnyObjec
```

- UTIs are not used directly
- NSPasteboardItem conforms to NSPasteboardReading and NSPasteboardWriting

```
func setDataProvider(dataProvider: NSPasteboardItemDataProvider!, forTypes types:
func setData(data: NSData!, forType type: String!) -> Bool
func setString( string: String!, forType type: String!) -> Bool
func setPropertyList(propertyList: AnyObject!, forType type: String!) -> Bool
var types: [AnyObject]! { get }
func availableTypeFromArray(types: [AnyObject]!) -> String!
func dataForType(type: String!) -> NSData!
func stringForType(type: String!) -> String!
func propertyListForType(type: String!) -> AnyObject!
```

Add Cut, Copy, and Paste to Dice

- in DieView

```
func writeToPasteboard(pasteboard: NSPasteboard) {
    if let intValue = intValue {
        pasteboard.clearContents()
        pasteboard.writeObjects(["\(intValue)"])
    }
}

func readFromPasteboard(pasteboard: NSPasteboard) -> Bool {
```

```
    let objects = pasteboard.readObjectsForClasses([NSString.self], options: [:])
    if let str = objects.first {
        intValue = str.toInt()
        return true
    }
    return false
}

@IBAction func cut(sender: AnyObject?) {
    writeToPasteboard(NSPasteboard.generalPasteboard())
    intValue = nil
}

@IBAction func copy(sender: AnyObject?) {
    writeToPasteboard(NSPasteboard.generalPasteboard())
 }

@IBAction func paste(sender: AnyObject?) {
    readFromPasteboard(NSPasteboard.generalPasteboard())
}
```

Nil-Targeted Actions

- if target of a control is nil, the application tries first responder of the key window
- this means the active view of the active window gets the cut and paste messages
- NSView, NSApplication, NSWindow inherit from NSResponder
- NSResponder has nextResponder
- if object does not respond to nil-targeted action, its nextResponder gets a chance (usually superview) -> passes through responder chain
- order of responder chain before nil-targeted action gets discarded (i.e. responder chain)
    i. firstResponder of the keyWindow and its responder chain responder chain would typically include the superviews the key window last
    ii. delegate of the key window
    iii. for document-based application, the NSWindowController and then the NSDocument object for the key window.
    iv. if the main window is different from the key window
        - 1, 2, 3 for main window
    v. instance of NSApplication
    vi. delegate of the NSApplication
    vii. NSDocumentController
- Document based application also gets Save, Save As..., Revert to Saved, Print..., and Page Setup

Looking at the XIB file

- cut, copy, and paste are connected to First Responder which represents nil (target of drag)

Menu Item Validation

- Cocoa automatically disables menu items when it cannot find the item's action in the responder chain
- validation of existence is performed by validateMenuItem
- in DieView

```
override func validateMenuItem(menuItem: NSMenuItem) -> Bool {
    switch menuItem.action {
    case Selector(" copy:"):
        return intValue == nil
    default:
        return super.validateMenuItem(menuItem)
    }
}
```

- can set the state property to NSOnState or NSOffState to check or uncheck the menu item

For the More Curious: Which Object Sends the Action Message?

- target on Cut, Copy, Paste items is nil
- sending message to nil does not crash application (doesn't do anything)
- target-action messages are handled by NSApplication which knows to try to send messages to objects in responder chain

```
func sendAction(_: Selector, to: AnyObject!, from: AnyObject!) -> Bool
```

For the More Curious: UTIs and the Pasteboard Custom UTIs

- NSString uses NSPasteboardTypeString when reading from pasteboard
- UTIs enable broad or specific requests from pasteboard
- for custom data, use reversed, DNS-style domain names
    - implement NSPasteboardWriting and NSPasteboardReading
    - or use NSPasteboardItem as an abstraction layer
- only export custom UTIs for use with other applications

For the More Curious: Lazy Copying

- for copying large data or many options for types of data can just declare types it can copy as and only copy when another application requests it

```
let pboard = NSPasteboard.generalPasteboard()
pboard.clearContents()
let item = NSPasteboardItem()
item.setDataProvider(self, forTypes:...)
pboard.writeObjects([item])

...
func pasteboard(_ pasteboard: NSPasteboard!, item item: NSPasteboardItem!, provide
    item.setData(..., forType:type)
}
```

- when terminating, an application will be requested for the data
- when lazy copying, must use a snapshot to paste state of data when copied
- copies later will negate the need to keep this snapshot
- the following method is called to indicate previous snapshot is no longer necessary

```
optional func pasteboardFinishedWithDataProvider(_ pasteboard: NSPasteboard!)
```

Challenge: Write Multiple Representations

- put both string and PDF onto pasteboard (using NSPasteboardItem)

Challenge: Menu Item

- add menu item that triggers removeEmployee in Document

22. Drag-and-Drop

- much like copy and paste
- what makes it more difficult is user feedback
- NSDragOperation type

```
struct NSDragOperation : RawOptionSetType {
    init(_ value: UInt)
    static var None: NSDragOperation { get }
    static var Copy: NSDragOperation { get }
    static var Link: NSDragOperation { get }
```

```
    static var Generic: NSDragOperation { get }
    static var Private: NSDragOperation { get }
    static var Move: NSDragOperation { get }
    static var Delete: NSDragOperation { get }
    static var Every: NSDragOperation { get }
}
```

- both source and destination must agree on operation
- for the exchange
    - drag source
    - drag destination

Make DieView a Drag Source

- conform to NSDraggingSource

```
func draggingSession(
    session: NSDraggingSession,
    sourceOperationMaskForDraggingContext,
    context: NSDraggingContext
) -> NSDragOperation { return .Copy }
```

- for a drag method is called twice
    - context = .WithinApplication (operations possible in application)
    - context = .OutsideApplication (operations possible outside application)

Starting a drag

- to start a drag operation

```
func beginDraggingSessionWithItems(
    items: [AnyObject],
    event: NSEvent,
    source: NSDraggingSource
) -> NSDraggingSession
```

- takes array of NSDraggingItem
- dragging item knows how to write itself to pasteboard
- event is mouseDown event
- source is usually view itself
- in DieView

```
var mouseDownEvent: NSEvent?
...
override func mouseDown(theEvent: NSEvent) {
```

```
    print("mouseDown")
    mouseDownEvent = theEvent
    let dieFrame = metricsForSize(bounds.size).dieFrame
    let pointInView = convertPoint(theEvent.locationInWindow, fromView: nil)
    pressed = dieFrame.contains(pointInView)
}
```

- need to draw into image -> can use NSImage e.g.

```
let imageSize = ...
let greenImage = NSImage(size: imageSize, flipped: false) {
    (imageBounds) in NSColor.greenColor.set()
    NSBezierPath.fillRect(imageBounds)
}
```

- initiate dragging session in mouseDragged

```
override func mouseDragged(theEvent: NSEvent) {
    print("mouseDragged location: \(theEvent.locationInWindow)")
    let downPoint = mouseDownEvent!.locationInWindow
    let dragPoint = theEvent.locationInWindow
    let distanceDragged = hypot(downPoint.x - dragPoint.x, downPoint.y - dragPoint
    if distanceDragged < 3 { return }
    pressed = false
    if let intValue = intValue {
        let imageSize = bounds.size
        let image = NSImage(size: imageSize, flipped: false) {
            (imageBounds) in self.drawDieWithSize(imageBounds.size)
            return true
        }
        let draggingFrameOrigin = convertPoint(downPoint, fromView: nil)
        let draggingFrame = NSRect(origin: draggingFrameOrigin, size: imageSize).r
        let item = NSDraggingItem(pasteboardWriter: "\(intValue)")
        item.draggingFrame = draggingFrame
        item.imageComponentsProvider = {
            let component = NSDraggingImageComponent(key: NSDraggingImageComponent
            component.contents = image
            component.frame = NSRect(origin: NSPoint(), size: imageSize)
            return [component]
        }
        beginDraggingSessionWithItems([item], event: mouseDownEvent!, source: self
    }
}
```

After the drop

- drag source will be notified on drag end by implementing draggingSession (and draggedImage)

```
func draggingSession( session: NSDraggingSession, endedAtPoint screenPoint: NSPoin

// e.g.
func draggingSession(session: NSDraggingSession, sourceOperationMaskForDraggingCon
    return .Copy | .Delete
}

func draggedImage(session: NSDraggingSession, endedAtPoint screenPoint: NSPoint, o
    if operation = = .Delete { intValue = nil }
}
```

Make DieView a Drag Destination

- first declare view a drag destination for certain types

```
func registerForDraggedTypes(newTypes: [AnyObject])
```

- implement 6 methods from NSDraggingInfo
  - as the image is dragged into the destination, the destination is sent a draggingEntered (_:) message (destination view may update its appearance)
  - while the image remains within the destination, a series of draggingUpdated(*:) messages are sent (implementing draggingUpdated(* :) is optional)
  - if the image is dragged outside the destination, draggingExited(_:) is sent
  - if the image is released on the destination, either it slides back to its source (and breaks the sequence) or a prepareForDragOperation(*:) message is sent to the destination depending on the value returned by the most recent invocation of draggingEntered(* :) (or draggingUpdated(_:) if the view implemented it)
  - if the prepareForDragOperation(*:) message returns true, then a performDragOperation(* :) message is sent (typically where the application reads data off the pasteboard)
  - if performDragOperation(*:) returned true, concludeDragOperation(* :) is sent (appearance may change or other UX effects)

registerForDraggedTypes(_:)

- in DieView

```
override init(frame frameRect: NSRect) {
    super.init(frame: frameRect) commonInit()
}
required init?(coder: NSCoder) {
    super.init(coder: coder)
    commonInit()
}
func commonInit() {
```

```
    self.registerForDraggedTypes([NSPasteboardTypeString])
}
```

Add highlighting

```
var highlightForDragging: Bool = false { didSet { needsDisplay = true } }

override func drawRect(dirtyRect: NSRect) {
    let backgroundColor = NSColor.lightGrayColor()
    backgroundColor.set()
    NSBezierPath.fillRect(bounds)
    if highlightForDragging {
        let gradient = NSGradient(
            startingColor: NSColor.whiteColor(),
            endingColor: backgroundColor
        )
        gradient.drawInRect(bounds, relativeCenterPosition: NSZeroPoint)
    } else { drawDieWithSize( bounds.size) }
}
```

Implement the dragging destination methods

```
override func draggingEntered(sender: NSDraggingInfo) -> NSDragOperation {
    if sender.draggingSource() === self { return .None }
    highlightForDragging = true
    return sender.draggingSourceOperationMask()
}

override func draggingExited(sender: NSDraggingInfo?) { highlightForDragging = fal

override func prepareForDragOperation(sender: NSDraggingInfo) -> Bool { return tru

override func performDragOperation(sender: NSDraggingInfo) -> Bool {
    let ok = readFromPasteboard(sender.draggingPasteboard())
    return ok
}

override func concludeDragOperation(sender: NSDraggingInfo?) { highlightForDraggin
```

- draggingEntered(_:) -> disallows drag if dragging source is same as the destination

For the More Curious: Operation Mask

- negotiation of event on drop may be complex
- can modify action with ctrl, opt, or cmd
- drag info object will do most of the work by getting source's op mask and filter depending on modifier keys

- to observe use

```
override func draggingUpdated(sender: NSDraggingInfo) -> NSDragOperation {
    print("operation mask = \(sender.draggingSourceOperationMask().rawValue)")
    if sender.draggingSource() === self { return .None }
    return .Copy | .Delete
}
```

23. NSTimer

- object with target, selector, and interval in seconds

NSTimer-based Animation

- use new method roll in mouseUp to create NSTimer based animation
- API to create timer

```
class func scheduledTimerWithTimeInterval( ti: NSTimeInterval, target aTarget: Any
```

- in DieView

```
var rollsRemaining: Int = 0

...

func roll() {
    rollsRemaining = 10
    NSTimer.scheduledTimerWithTimeInterval(0.15, target: self, selector: Selector(
    window?.makeFirstResponder(nil)
}

func rollTick(sender: NSTimer) {
    let lastIntValue = intValue
    while intValue == lastIntValue { randomize() }
    rollsRemaining--
    if rollsRemaining == 0 {
        sender.invalidate()
        window?.makeFirstResponder(self)
    }
}

override func mouseUp(theEvent: NSEvent) {
    print("mouseUp clickCount: \(theEvent.clickCount)")
    if theEvent.clickCount == 2 {
        randomize()
        roll()
    }
```

```
    pressed = false
}
```

How Timers Work

- timers use the run loop to fire
- calculates next scheduled fire and requests that run loop to notify when that time arrives
- works for basics but not good for high resolution/high performance
- documented to have a resolution 50-100ms
- supports timer coalescing
- can use tolerance property to make system massage timer fires to occur at once which improves energy usage

NSTimer and Strong/Weak References

- run loop keeps strong reference to timer (i.e. owns it) so do not keep strong reference in your own code
- may not be necessary to keep any reference at all except to invalidate or to check if running
- when a ref is needed use a weak optional

```
weak var timer: NSTimer?
```

- weak references must be optional in Swift
- like controls, but keeps a strong reference to its target
- means that while a timer is scheduled its target will stay in memory
- invalidate timers at appropriate times to invalidate

For the More Curious: NSRunLoop

- object that waits for events to arrive and forwards them to NSApplication
- when timer events arrive they are forwarded to NSTimer
- can attach a network socket to the run loop and wait for data on the socket

24. Sheets

- common pattern for collecting info from user and associating it with a window (doc save panel, i.e. window presented from another window)
- NSWindow methods to present sheets

```
func beginSheet(sheetWindow: NSWindow, completionHandler: (( NSModalResponse) -> V
func beginCriticalSheet(sheetWindow: NSWindow, completionHandler: (( NSModalRespon
func endSheet(sheetWindow: NSWindow, returnCode: NSModalResponse)
```

Adding a Sheet

- subclass NSWindowController with .xib

Create the Window Controller

- ConfigurationWindowController

```
import Cocoa class ConfigurationWindowController: NSWindowController {
    private dynamic var color: NSColor = NSColor.whiteColor()
    private dynamic var rolls: Int = 10

    override var windowNibName: String { return "ConfigurationWindowController" }

    override func windowDidLoad() {
        super.windowDidLoad()
        // Implement this method to handle post-nib-load initialization.
    }

    @IBAction func okayButtonClicked(button: NSButton) { print("OK clicked") }

    @IBAction func cancelButtonClicked(button: NSButton) { print("Cancel clicked")
}
```

- on presentation, need to provide color and number of rolls for sheet and read values out on "OK"

```
struct DieConfiguration {
    let color: NSColor
    let rolls: Int

    init(color: NSColor, rolls: Int) {
        self.color = color
        self.rolls = max(rolls, 1)
    }
}

// in ConfigurationWindowController
var configuration: DieConfiguration {
    set {
        color = newValue.color
        rolls = newValue.rolls
    }
```

```
    get { return DieConfiguration(color: color, rolls: rolls) }
}
```

- generally do not need init in struct but does more work here

Set Up the Menu Item

- in MainWindowController

```
@IBAction func showDieConfiguration(sender: AnyObject?) { print("Configuration men
```

- add a menu item to .xib with title "Configure Die"
- connect menu item to First Responder and set action to showDieConfiguration

Lay Out the Interface

- in ConfigurationWindowController.xib
- window Attributes Inspector
  - Resize -> `[]`
  - Visible At Launch -> `[]`
- add 2 labels, 2 buttons, a color well, a text field and a stepper
- color well Bindings Inspector
  - Value -> File's Owner color
- text field Bindings Inspector
  - Value -> File's Owner rolls
- stepper Bindings Inspector
  - Value -> File's Owner rolls
- ctrl-drag from OK button to File's Owner
  - action -> okayButtonClicked
- ctrl-drag from cancel button to File's Owner
  - action -> cancelButtonClicked

Configuring the Die Views

- add configurable properties to DieView

```
var color: NSColor = NSColor.whiteColor() { didSet { needsDisplay = true } }
var numberOfTimesToRoll: Int = 10
```

- replace uses of color with color property in drawRect and drawDieWithSize

```
// drawRect
let gradient = NSGradient(startingColor: color, endingColor: backgroundColor)

// drawDieWithSize
color.set()
```

- change roll to use numberOfTimesToRolle

```
rollsRemaining = numberOfTimesToRoll
```

Present the Sheet

- on presentation, sheet should be configuring the currently selected DieView (use first responder)
- in DieView

```
var configurationWindowController: ConfigurationWindowController?

@IBAction func showDieConfiguration(sender: AnyObject?) {
    print("Configuration menu item clicked")
    if let window = window, let dieView = window.firstResponder as? DieView {
        // Create and configure the window controller to present as a sheet:
        let windowController = ConfigurationWindowController()
        windowController.configuration = DieConfiguration(color: dieView.color, ro
        window.beginSheet(
            windowController.window!,
            completionHandler: { response in
                // The sheet has finished. Did the user click 'OK'?
                if response = = NSModalResponseOK {
                    let configuration = self.configurationWindowController!.config
                    dieView.color = configuration.color
                    dieView.numberOfTimesToRoll = configuration.rolls
                }
                // All done with the window controller.
                self.configurationWindowController = nil
            }
        )
        configurationWindowController = windowController
    }
}
```

- sheet should end when buttons are clicked

```
@IBAction func okayButtonClicked(button: NSButton) {
    print("OK clicked")
    window?.endEditingFor(nil)
    dismissWithModalResponse(NSModalResponseOK)
}
```

```
@IBAction func cancelButtonClicked(button: NSButton) {
    print("Cancel clicked")
    dismissWithModalResponse(NSModalResponseCancel)
}

func dismissWithModalResponse(response: NSModalResponse) {
    window!.sheetParent!.endSheet(window!, returnCode: response)
}
```

- okayButtonClicked calls endEditingFor

Modal Windows

- similar to sheets but presented independently from other windows and block user input for the rest of the application
- use runModalForWindow and stopModalWithCode on NSApplication

```
func showModalWindow() {
    let windowController = CriticalWindowController()
    let app = NSApplication.sharedApplication()
    let returnCode = app.runModalForWindow(windowController.window!)
    if returnCode == CriticalWindowController.returnAccept { ... }
}
```

Encapsulating Presentation APIs

- sheet and modal APIs allow for use of custom Int as error code

```
class CriticalWindowController: NSWindowController {
    enum ModalResult: Int {
        case Accept
        case Cancel
    }

    func runModal() -> ModalResult {
        let app = NSApplication.sharedApplication()
        let returnCode = app.runModalForWindow(window!)
        if let result = ModalResult(rawValue: returnCode) { return result }
        else { fatalError("Failed to map \(returnCode) to ModalResult") }
    }

    @IBAction func dismiss(sender: NSButton) {
        let app = NSApplication.sharedApplication()
        app.stopModalWithCode(ModalResult.Accept.rawValue)
    }
}

...
func showModalWindow() {
    let windowController = CriticalWindowController()
```

```
    switch windowController.runModal() {
    case .Accept:
        ...
    case .Cancel:
        break
    }
}
```

- clarifies API usage through use of enum with clear cases

Challenge: Encapsulate Sheet Presentation

- add a method to ConfigurationWindowController: presentAsSheetOnWindow(_: completionHandler:).
- completion handler is type (DieConfiguration?)->(Void)

Challenge: Add Menu Item Validation

- disable Configure Die menu item if no DieView is presently first responder
- the is operator returns true if the left operand is the same type as the right operand

25. Auto Layout

- views are positioned by their frame (views position within superview)
- by default, a view created in IB is static
- can observe resizing programmatically but prefer Auto Layout

What is Auto Layout?

- uses constraints to determine where views within a window should be positioned (just equations)

Adding Constraints to RaiseMan

Constraints from subview to superview

- select scroll view
  - click the Add New Constraints Button
    - bottom, left and top position constraint -> Use Standard Value
    - click "Add 3 Constraints" button
- blue I-beam = good, orange = ambiguous

- select vertical I-beam below the scroll view -> Attributes Inspector
    - Constant -> Standard
- Attributes Inspector for a constraint
    - type of constraint
    - info about constraint relationships
    - priority (priority of 10000 = required)
    - Remove at build time (allows for runtime constraints)
    - Constant -> can be modified
    - can drag constraint and observe changes
- Add Employee button -> Add New Constraints
    - top and right position constraint -> Use Standard Value
    - click "Add 2 Constraints"
- text fields in Name and Raise column -> Add New Constraints
    - select all four I-beams
    - top, bottom -> 0
    - left, right -> 2
    - click "Add 4 Constraints"
- remaining components not working as expected
    - scroll view width does not change
    - Add Employee slides beneath the scroll view when the width is too narrow
    - Remove button does ot move

Constraints between siblings

- Remove button -> Add New Constraints
    - top, left -> Use Standard Value
    - click "Add 2 Constraints"
- can add constraints by ctrl-dragging between views
- can set views to have equal width or height by ctrl-dragging
- ctrl-drag Remove button -> Add Employee button
    - Equal Widths in constraint type panel
- can also select two views and select Add New Constraints to set equal widths
- Add Employee button -> Add New Constraints
    - left -> Use Standard Value
    - click "Add 1 Constraint"
- this should have made a warning indicator disappear (IB has a bug with Auto Layout)

Size constraints

- scroll view -> Add New Constraints
  - Width -> X (> 200)
  - Height -> X (> 100)
- select Width constraint in document outline -> Attributes Inspector
  - Relation -> Greater Than or Equal
- select Width constraint in document outline -> Attributes Inspector
  - Relation -> Greater Than or Equal

Intrinsic Content Size

- certain views have intrinsic content size (often determined by contained text)
- Auto Layout handles resizing based on intrinsic content sizes

Creating Layout Constraints Programmatically

- can use NSLayoutConstraint

```
let scrollView: NSScrollView = ...
let superview = scrollView.superview!
let constraint = NSLayoutConstraint(
    item: scrollView,
    attribute: .Leading,
    relatedBy: .Equal,
    toItem: superview,
    attribute: .Leading,
    multiplier: 1.0,
    constant: 20.0
)
superview.addConstraint( constraint)
```

- leading and trailing edges will use left as the leading edge for languages that read left-to-right and right for right-to-left languages
- can add and remove constraints at runtime
- can animate layout changes to smooth transitions
- see Auto Layout Guide

Visual Format Language

- new Cocoa project AutoLabelOut
- in AppDelegate

```
func applicationDidFinishLaunching(aNotification: NSNotification) {
    // Insert code here to initialize your application
    // Create the controls for the window:
    // Create a label:
    let label = NSTextField(frame: NSRect.zeroRect)
    label.translatesAutoresizingMaskIntoConstraints = false
    label.stringValue = "Label"
    // Give this NSTextField the styling of a label:
    label.backgroundColor = NSColor.clearColor()
    label.editable = false
    label.selectable = false
    label.bezeled = false
    // Create a text field:
    let textField = NSTextField(frame: NSRect.zeroRect)
    textField.translatesAutoresizingMaskIntoConstraints = false
    // Make the text field update the label's text
    // when the text field's text changes:
    textField.action = Selector("takeStringValueFrom:")
    textField.target = label
    let superview = window.contentView as! NSView
    superview.addSubview(label)
    superview.addSubview(textField)
    // Create the constraints between the controls:
    let horizontalConstraints = NSLayoutConstraint.constraintsWithVisualFormat(
        "|-[label]-[textField(>= 100)]-|",
        options:. AlignAllBaseline,
        metrics:nil,
        views: ["label" : label, "textField" : textField]
    )
    NSLayoutConstraint.activateConstraints(horizontalConstraints)
    let verticalConstraints = NSLayoutConstraint.constraintsWithVisualFormat(
        "V: |-[textField]-|",
        options:. allZeros,
        metrics: nil,
        views: ["textField" : textField]
    )
    NSLayoutConstraint.activateConstraints(verticalConstraints)
}
```

- allows you to specify constraints using an ASCII art style language
- pipes - edges of superview
- `[view name]`
- value in points

Does Not Compute, Part 1: Unsatisfiable Constraints

- conflicts occur generally when there are too many or too few constraints
- aka unsatisfiable (too many constraints)
- Auto Layout will show conflicting constraints in debug window and auto adjusts at runtime

Does Not Compute, Part 2: Ambiguous Layout

- ambiguous when too few constraints to determine the position of all of the views
- can use NSWindow's visualizeConstraints method
- this will display an Exercise Ambiguity button that can be used to demonstrate the constraint ambiguity
- applicationDidFinishLaunching

```
window.visualizeConstraints( superview.constraints)

// or
superview.updateConstraintsForSubtreeIfNeeded()
if superview.hasAmbiguousLayout { superview.exerciseAmbiguityInLayout() }
```

For the More Curious: Autoresizing Masks

- used to specify how a subview resizes in response to its superview resizing
- no means to specify layout in relation to sibling views
- spring -> resize in vertical or horizontal direction
- strut -> preserve distance between sides of view and superview
- for legacy code use translatesAutoresizingMaskIntoConstraints = false
- to use autoresizing masks in IB, must turn off Auto Layout on a per .xib file basis
- prefer Auto Layout

Challenge: Add Vertical Constraints

- add a constraint to make sure that the bottom of the Remove button is always at least the standard distance from the bottom of the window

Challenge: Add Constraints Programmatically

- add same constraints to RaiseMan two different ways
  - init(item:attribute:relatedBy:toItem:attribute:multiplier:constant:)
  - constraintsWithVisualFormat(_: options:metrics:views:)
- keep all existing constraints but select Remove at build time in Attributes Inspector Placeholder section

26. Localization and Bundles

- main languages to plan on supporting
  - English
  - French
  - Spanish
  - German
  - Dutch
  - Italian
  - Mandarin
  - Japanese
- create strings files with additional .lproj files for locales

Different Mechanisms for Localization

- User-facing string literals
  - labels, alerts, etc
  - always localized with strings files
- General resources
  - all application resource files (except XIBs) exposed to user
  - need to generate a version with translated text for each locale
- XIB files
  - localized XIB (need a copy for each locale)
  - strings file (Xcode generate strings file for locale but programmer needs to edit)
- XIB files allow for greater customization but is more resource intensive and difficult to maintain
- strings files are easier to maintain but less customizable
- can mix and match approaches for locales

Localizing a XIB File

- project navigator -> RaiseMan project
  - Info tab -> Localizations section
    -
      - button -> French
    - select Document.xib
    - in File Types pop-up -> keep Localizable Strings (rather than Interface Builder Cocoa XIB)
    - click Finish

- creates file with pre-populated info from an existing file (from Reference Language)
- provides details of French strings translations
- uses object IDs and strings
- to test change System Preferences preferred language

Localizing String Literals

- uses a file Localizable.strings
- change uses of string literals to NSLocalizedString

```
func NSLocalizedString(key: String, #comment: String) -> String
```

- details string literal usage in app here
- RaiseMan group -> Project Navigator
    - utility area -> file inspector
        - Identity and Type -> Full Path -> copy to clipboard
        - cd to path in terminal
        - `genstrings Document.swift_and_mac -o Base.lproj`
        - drag Localizable.strings file into Xcode under Document/xib hierarchy
            - utility area -> file inspector
                - check English and French
- details updating English and French Localizable.strings files here

Demystifying NSLocalizedString and genstrings

- strings file is just sets of key-value pairs
- can programmatically use NSBundle to get the localized strings when your app is running

```
let mainBundle: NSBundle = NSBundle.mainBundle()
let string: String = mainBundle.localizedStringForKey("NO_RESULTS", value: "No res
```

- searches in Find.strings and returns "No results found" on failure
- most simple applications just use Localizable.strings
- NSLocalizedString just calls the method on NSBundle.mainBundle()

Explicit Ordering of Tokens in Format Strings

- may need to reconsider ordering of tokens for format strings

NSBundle

- bundle -> directory of resources that may be used by an application
    - any kind of file
        - images
        - sounds
        - NIB files
        - compiled code
- application is a bundle, and app image is a directory that is called the main bundle

```
let bundle = NSBundle.mainBundle()

// to access resources in another bundle
let bundleURL: NSURL = ...
if let bundle = NSBundle( URL: bundleURL) { ... }

// ask for general resources
let url: NSURL? = bundle.URLForResource(" Chapter", withExtension: "xml")

// ask for images
if let appIconImage = NSImage(named: NSImageNameApplicationIcon), let badgeImage =
```

NSBundle's role in localization

- NSBundle manages localization at runtime
- resources can be any file (not source code) and will be packaged with app if it has the resource checked
- does more work for strings file
    - global resource
        - not localized
        - top-level by default
    - region-specific localized resource
        - user can specify their region in Language & Region preference pane
        - if a localization is found in current language and user specified region, localization if preferred over generic localization
    - language-specific localized resource
    - development language resource
    - base localization resource
- returns first resource found

Loading code from bundles

- load principal class from the bundle

```
if let principalClass = pluginBundle.principalClass as? NSObject.Type {
    if let rover = principalClass() as? Rover { rover.fetch() }
}
```

- configured in Info.plist
- see Code Loading Programming Topics

For the More Curious: Localization and Plurality

- for correct plurality across all locales use stringdict
- create file named Localizable.stringdict along Localizable.stringsdict
- creates a full dictionary
- two fields
    - a localized format string
    - a dictionary showing how to expand the string
- each local for a stringsdict will have the proper number of plural forms for that language
- see Internationalization and Localization guide

Challenge: Localizing the Default Name for a Newly Added Employee

- localize New Employee string

Challenge: Localizing the Undo Action Names

- localize the Edit menu strings

27. Printing

- difficult due to
    - pagination
    - margins
    - page orientation
- for Document based app and a view that knows how to draw itself implement printOperationWithSettings(_: error:)

```
override func printOperationWithSettings(
    printSettings: [NSObject : AnyObject],
    error outError: NSErrorPointer
) -> NSPrintOperation? {
    let printableView: NSView = ...
    // View that draws document's data for printing.
    let printInfo: NSPrintInfo = self.printInfo
    let printOperation = NSPrintOperation(view: printableView, printInfo: printInf
    return printOperation
}
```

- this is all that is needed if document can be drawn on a single page

Dealing with Pagination

- two methods

```
// How many pages?
func knowsPageRange(aRange: NSRangePointer) -> Bool
// Where is each page?
func rectForPage( pageNumber: Int) -> NSRect
```

- use page number in rect for page and drawRect for page number

Adding Printing to RaiseMan

- create class/file EmployeesPrintingView

```
import Cocoa

private let font: NSFont = NSFont.userFixedPitchFont(ofSize: 12.0)!
private let textAttributes: [String: AnyObject] = [NSFontAttributeName : font]
private let lineHeight: CGFloat = font.capHeight * 2.0

class EmployeesPrintingView: NSView {

    let employees: [Employee]

    var pageRect = NSRect()
    var linesPerPage: Int = 0
    var currentPage: Int = 0

    // MARK: - Lifecycle

    init(employees: [Employee]) {
        self.employees = employees
        super.init(frame: NSRect())
    }
```

```
required init?(coder: NSCoder) {
    fatalError("unimplemented: instantiate programmatically instead")
}

// MARK: - Pagination

override func knowsPageRange(_ range: NSRangePointer) -> Bool {
    let printOperation = NSPrintOperation.current()!
    let printInfo: NSPrintInfo = printOperation.printInfo

    // Where can I draw?
    pageRect = printInfo.imageablePageBounds
    let newFrame = NSRect(origin: CGPoint(), size: printInfo.paperSize)
    frame = newFrame

    // How many lines per page?
    linesPerPage = Int(pageRect.height / lineHeight)

    // Construct the range to return
    var rangeOut = NSRange(location: 0, length: 0)

    // Pages are 1-based. That is, the first page is 1.
    rangeOut.location = 1

    // How many pages will it take?
    rangeOut.length = employees.count / linesPerPage
    if employees.count % linesPerPage > 0 {
        rangeOut.length += 1
    }

    // Return the newly constructed range, rangeOut, via the range pointer
    range.pointee = rangeOut

    return true
}

override func rectForPage(_ page: Int) -> NSRect {
    // Note the current page
    // Although Cocoa uses 1=based indexing for the page number
    //  it's easier not to do that here.
    currentPage = page - 1

    // Return the same page every time
    return pageRect
}

// MARK: - Drawing

// The origin of the view is at the upper left corner
override var isFlipped: Bool {
    return true
}

override func draw(_ dirtyRect: NSRect) {
    var nameRect = NSRect(x: pageRect.minX,
                          y: 0,
```

```
                        width: 200.0,
                       height: lineHeight)
        var raiseRect = NSRect(x: nameRect.maxX,
                               y: 0,
                        width: 100.0,
                       height: lineHeight)

        for indexOnPage in 0..<linesPerPage {
            let indexInEmployees = currentPage * linesPerPage + indexOnPage
            if indexInEmployees >= employees.count {
                break
            }

            let employee = employees[indexInEmployees]

            // Draw index and name
            nameRect.origin.y = pageRect.minY + CGFloat(indexOnPage) * lineHeight
            let employeeName = (employee.name ?? "")
            let indexAndName = "\(indexInEmployees) \(employeeName)"
            indexAndName.draw(in: nameRect, withAttributes: textAttributes)
            // Draw raise
            raiseRect.origin.y = nameRect.minY
            let raise = String(format: "%4.1f%%", employee.raise * 100)
            let raiseString = raise
            raiseString.draw(in: raiseRect, withAttributes: textAttributes)
        }
    }
}
```

- in Document.swift

```
override func printOperation(withSettings printSettings: [String : Any]) throws ->
    let employeesPrintingView = EmployeesPrintingView(employees: employees)
    let printInfo: NSPrintInfo = self.printInfo
    let printOperation = NSPrintOperation(view: employeesPrintingView, printInfo:
    return printOperation
}
```

- MainMenu.xib -> Print... menu item is nil-targeted (action -> printDocument: -> printOperationWithSettings(_: error:))

For the More Curious: Are You Drawing to the Screen?

- difference between drawing on screen and drawing in printing
- can query in drawRect(_:)

```
if NSGraphicsContext.currentContextDrawingToScreen() { // ... draw the grid ... }
```

Challenge: Add Page Numbers

Challenge: Persist Page Setup

- NSPrintInfo stored in document's printInfo property conforms to NSCoding
- modify Document to achive and unarchive printInfo and employees
- change rsmn file to use dictionary as top-level object
- when unarchiving, set printInfo but disable undo registration while setting property

```
override func readFromData(
    data: NSData,
    ofType typeName: String,
    error outError: NSErrorPointer
) -> Bool {
    ...
        ...
            if let unarchivedPrintInfo = dictionary["printInfo"] as? NSPrintInfo {
                undoManager.disableUndoRegistration()
                printInfo = unarchivedPrintInfo
                undoManager.enableUndoRegistration()
            }
```

28. Web Services

- requires HTTP or HTTPS generally
- general pattern
    - format a request as required by the web service
    - send the request over HTTP/HTTPS
    - receive the response
    - parse the response and use that data in your application

Web Services APIs

- many involved classes
    - NSURL
    - NSURLRequest
    - NSURLSession
    - NSURLSessionTask
    - NSURLSessionConfiguration
- NSURL used for files and web services

- wrap in NSURLRequest (for modification use NSMutableURLRequest)
- use NSURLSession to manage a number of related connections
- NSURLRequest -> NSURLSession -> NSURLSessionTask
- task completes on server response
- Cocoa has classes for working with XML and JSON

RanchForecast Project

- new Xcode project RanchForecast
  - uncheck Create Document-Based Application, Use Core Data, and Use Storyboards
  - single window app
  - AppDelegate and MainWindowController changes

```
View                    NSTableView
                             ^
                             |
                             V
Controller              NSArrayController
                             ^
                             |
                             V
                        MainWindowController
                             ^
                             |
                             V
Model       ScheduleFetcher <-> Course (s)
```

- Course

```
class Course: NSObject {
    let title: String
    let url: NSURL
    let nextStartDate: NSDate

    init(title: String, url: NSURL, nextStartDate: NSDate) {
        self.title = title
        self.url = url
        self.nextStartDate = nextStartDate super.init()
    }
}
```

NSURLSession and asynchronous API design

- NSURLSessionTask
  - download to a file on disk
  - upload from a file on disk

    - download to memory

```
func dataTaskWithRequest(request: NSURLRequest, completionHandler: (( NSData!, NSU
```

- use the returned task to manage connection and get progress info
- calls completion handler when response is received (async)
- do not want to use blocking APIs here
- should put querying class in model layer
- ScheduleFetcher

```
import Foundation

class ScheduleFetcher {
    let session: NSURLSession init() {
        let config = NSURLSessionConfiguration.defaultSessionConfiguration()
        session = NSURLSession(configuration: config)
    }
}
```

- have a number of options to start fetch
    - delegate (ScheduleFetcherDelegate)
    - NSNotificationCenter
        - best for broadcasting notifications to many observers over time
        - no type safety
    - completion handler
        - good for one shot callbacks
- use completion handler
- takes 3 params -> NSData, NSURLResponse, NSError

```
func fetchCoursesUsingCompletionHandler(completionHandler: (FetchCoursesResult) ->
    let url = NSURL(string: "http://bookapi.bignerdranch.com/courses.json")!
    let request = NSURLRequest(URL: url)
    let task = session.dataTaskWithRequest(
        request,
        completionHandler: { (data, response, error) -> Void in
            var result: FetchCoursesResult
            if data == nil { result = .Failure( error) }
            else {
                print("Received \(data.length) bytes.")
                result = .Success([])
                // Empty array until parsing is added
            }
        NSOperationQueue.mainQueue().addOperationWithBlock({ completionHandler(res
        }
    )
```

```
    task.resume()
}
```

- in MainWindowController

```
import Cocoa

class MainWindowController: NSWindowController {

    @IBOutlet var tableView: NSTableView!
    @IBOutlet var arrayController: NSArrayController!

    let fetcher = ScheduleFetcher()
    dynamic var courses: [Course] = []


    override var windowNibName: String! {
        return "MainWindowController"
    }


    override func windowDidLoad() {
        super.windowDidLoad()

        tableView.target = self
        tableView.doubleAction = #selector(MainWindowController.openClass(_:))

        fetcher.fetchCoursesUsingCompletionHandler { result in
            switch result {
            case .success(let courses):
                print("Got courses: \(courses)")
                self.courses = courses
            case .failure(let error):
                print("Got error: \(error)")
                NSAlert(error: error).runModal()
                self.courses = []
            }
        }
    }
}
```

NSURLSession, HTTP status codes, and errors

- HTTP response code is 100 - 599
- NSURLSession only reports errors at the network layer for failed requests
- provides completion handler with reference to NSURLResponse (NSHTTPURLResponse)
- in ScheduleFetcher

```
func errorWithCode(code: Int, localizedDescription: String) -> NSError {
    return NSError(
```

```
        domain: "ScheduleFetcher",
        code: code,
        userInfo: [NSLocalizedDescriptionKey: localizedDescription]
    )
}

func fetchCoursesUsingCompletionHandler(completionHandler: (FetchCoursesResult) ->
    let url = NSURL(string: "http:// bookapi.bignerdranch.com/ courses.json")!
    let req = NSURLRequest(URL: url)
    let task = session.dataTaskWithRequest(
        req,
        completionHandler: { (data, response, error) -> Void in
            var result: FetchCoursesResult
            if data == nil {
                result = .Failure(error)
            } else if let response = response as? NSHTTPURLResponse {
                print("\(data.length) bytes, HTTP \(response.statusCode).")
                if response.statusCode == 200 {
                    result = .Success([])
                    // Empty array until parsing is added
                } else {
                    let error = self.errorWithCode(1, localizedDescription: "Bad s
                    result = .Failure(error)
                }
            } else {
                let error = self.errorWithCode(1, localizedDescription: "Unexpecte
                result = .Failure(error)
            }
            NSOperationQueue.mainQueue().addOperationWithBlock({completionHandler(
        }
    )
    task.resume()
}
```

Add JSON parsing to ScheduleFetcher

- uses NSJSONSerialization class
- produces property list tree of NSDictionary, NSArray, NSString, NSNumber
- can also serialize to an NSData object
- ScheduleFetcher

```
func courseFromDictionary(_ courseDict: NSDictionary) -> Course? {
    let title = courseDict["title"] as! String
    let urlString = courseDict["url"] as! String
    let upcomingArray = courseDict["upcoming"] as! [NSDictionary]
    let nextUpcomingDict = upcomingArray.first!
    let nextStartDateString = nextUpcomingDict["start_date"] as! String

    let url = URL(string: urlString)!

    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "yyyy-MM-dd"
```

```
    let nextStartDate = dateFormatter.date(from: nextStartDateString)!

    return Course(title: title, url: url, nextStartDate: nextStartDate)
}
```

- does not check for errors in individual fields

```
func resultFromData(data: NSData) -> FetchCoursesResult {
    var error: NSError?
    let topLevelDict = NSJSONSerialization.JSONObjectWithData(data, options: NSJSO
    if let topLevelDict = topLevelDict {
        let courseDicts = topLevelDict["courses"] as! [NSDictionary]
        var courses: [Course] = []
        for courseDict in courseDicts {
            if let course = courseFromDictionary(courseDict) {
                courses.append(course)
            }
        }
        return .Success(courses)
    }
    else { return .Failure( error!) }
}

// in fetchCoursesUsingCompletionHandler(_:)
if response.statusCode == 200 {
    result = self.resultFromData(data)
}
```

Lay out the interface

- change window title to RanchForecast
- add table view
    - 2 columns -> Next Start Date, Title
    - Date Formatter on date column
        - Date Style -> Medium
- add array controller
    - Bindings Inspector
        - Content Array -> File's Owner
        - Model Key Path -> courses
- table view Bindings Inspector
    - Content -> Array Controller
    - Controller Key -> arrangedObjects
    - Selection Indexes -> Array Controller
    - Controller Key -> selectionIndexes
- text field of each table view cell's Bindings Inspector
    - Value -> Table Cell View

    - Model Key Path -> objectValue.nextStartDate and objectValue.title

Opening URLs

- set URL double-click to open app with URL
- tableView doubleAction is set programmatically
- in MainWindowController

```
@IBOutlet var tableView: NSTableView!
@IBOutlet var arrayController: NSArrayController!
```

- ctrl-click File's Owner in .xib
    - tableView -> table view
    - arrayController -> array controller
- in windowDidLoad()

```
tableView.target = self
tableView.doubleAction = Selector("openClass:")
```

- NSWorkspace represents Finder (can open URLs in default web browser)

```
func openClass(sender: AnyObject!) {
    if let course = arrayController.selectedObjects.first as? Course {
        NSWorkspace.sharedWorkspace().openURL(course.url)
    }
}
```

- advantage of array controller is that it can provide objects in a different sort order from the model (prefer selectedObjects over NSTableView's selectedRow)

Safely Working with Untyped Data Structures

- app code assumes structure of parsed data
    - courseDict has a key named upcoming whose value is an array
    - that array has at least one object in it, a dictionary
    - that dictionary has a key named start_date which is a string
- can use if let optionals to improve code safety

```
if let title = courseDict["title"] as? String,
   let urlString = courseDict["url"] as? String,
   let upcomingArray = courseDict["upcoming"] as? [NSDictionary],
   let nextUpcomingDict = upcomingArray.first,
   let nextStartDateString = nextUpcomingDict["start_date"] as? String {
```

```
        ...
        return Course(title: title, url: url, nextStartDate: nextStartDate)
    }
    return nil
```

- error checking often results in ugly code but is necessary

For the More Curious: Parsing XML

- high-level
  - NSXMLDocument and NSXMLNode
  - takes NSData -> XML tree
- can traverse tree manually or use XPath queries
- NSXMLParser is less heavy weight
  - makes calls to delegate as it parses and encounters structures
- only NSXMLParser is available on iOS

Challenge: Improve Error Handling

- add error handling to courseFromDictionary to return nil if format is not as expected and test

Challenge: Add a Spinner

- add indeterminate progress indicator and animate while fetching

Challenge: Parse the XML Courses Feed

- update RanchForecast to fetch XML from http:// bookapi.bignerdranch.com/ courses.xml

29. Unit Testing

- can divide into functions, methods, or classes

Testing in Xcode

- each Xcode project has main dir, tests dir, and Products dir
- subclasses XCTestCase
- automatically runs methods prefixed with test
- commonly used assert functions

```
XCTAssertTrue(expression)
XCTAssertEqual(testResultValue, expectedValue)
XCTAssertNotNil(expression)
XCTFail()
```

- can supply custom failure messages
- see Assertions Listed by Category in Testing with Xcode guide

Your First Test

- create CourseTests.swift
- to run, open Product menu and select Test (Command + u)

```
import XCTest
import RanchForecast

class CourseTests: XCTestCase {

    func testCourseInitialization() {
        let course = Course(
            title: Constants.title,
            url: Constants.url,
            nextStartDate: Constants.date
        )
        XCTAssertEqual(course.title, Constants.title)
        XCTAssertEqual(course.url, Constants.url)
        XCTAssertEqual(course.nextStartDate, Constants.date)
    }
}
```

- add Constants.swift

```
class Constants {
    static let urlString = "http://training.bignerdranch.com/classes/test-course"
    static let url = NSURL(string: urlString)!
    static let title = "Test Course"
    static let date = NSDate()
}
```

- note: need to explicitly import RanchForecast and make all tested components public to access them from tests

A Note on Literals in Testing

- prefer constants over literals to avoid errors

Creating a Consistent Testing Environment

- create ScheduleFetcherTests.swift

```
import XCTest
import RanchForecast

class ScheduleFetcherTests: XCTestCase {

    var fetcher: ScheduleFetcher!


    override func setUp() {
        super.setUp()
        fetcher = ScheduleFetcher()
    }


    override func tearDown() {
        fetcher = nil
        super.tearDown()
    }


    func testCreateCourseFromValidDictionary() {
            let course: Course! = fetcher.courseFromDictionary(Constants.validCour
            XCTAssertNotNil(course)
            XCTAssertEqual(course.title, Constants.title)
            XCTAssertEqual(course.url, Constants.url)
            XCTAssertEqual(course.nextStartDate, Constants.date)
    }


    func testResultFromValidHTTPResponseAndValidData() {
        let result = fetcher.resultFromData(Constants.jsonData,
        response: Constants.okResponse,
        error: nil)

        switch result {
        case .success(let courses):
            XCTAssert(courses.count == 1)
            let theCourse = courses[0]
            XCTAssertEqual(theCourse.title, Constants.title)
            XCTAssertEqual(theCourse.url, Constants.url)
            XCTAssertEqual(theCourse.nextStartDate, Constants.date)
        default:
            XCTFail("Result contains Failure, but Success was expected.")
        }
    }
}
```

- make ScheduleFetcher components public

- use setUp and tearDown for fixtures
- modify Constants.swift

```
class Constants {
    static let urlString = "http://training.bignerdranch.com/classes/test-course"
    static let url = URL(string: urlString)!
    static let title = "Test Course"

    static let dateString = "2014-06-02"
    static let dateFormatter: DateFormatter = {
        let formatter = DateFormatter()
        formatter.dateFormat = "yyyy-MM-dd"
        return formatter
    }()

    static let date = dateFormatter.date(from: dateString)!

    static let validCourseDict = [
        "title" : title,
        "url": urlString,
        "upcoming" : [["start_date": dateString]]
    ] as [String : Any]

    static let coursesDictionary = ["courses" : [validCourseDict]]

    static let okResponse = HTTPURLResponse(
        url: url,
        statusCode: 200,
        httpVersion: nil,
        headerFields: nil
    )

    static let jsonData = try! JSONSerialization.data(
        withJSONObject: coursesDictionary,
        options: []
    )
}
```

Sharing Constants

- common to use same constants in multiple tests
- fetcher property is also a fixture
- ensure set up and tear down of fixtures always happens to avoid dependencies between tests (mainly for shared fixtures)

Refactoring for Testing

- common to need to refactor to facilitate testing

- for long methods and related functionality within callbacks, move the code to a method to allow for testing (e.g. completion handler in fetcher)

```
func testResultFromValidHTTPResponseAndValidData() {
    let result = fetcher.resultFromData(
        Constants.jsonData,
        response: Constants.okResponse,
        error: nil
    )
    switch result {
    case .Success(let courses):
        XCTAssert(courses.count == 1)
        let theCourse = courses[0]
        XCTAssertEqual(theCourse.title, Constants.title)
        XCTAssertEqual(theCourse.url, Constants.url)
        XCTAssertEqual(theCourse.nextStartDate, Constants.date)
    default:
        XCTFail(" Result contains Failure, but Success was expected.")
    }
}
```

For the More Curious: Access Modifiers

- internal
  - default
  - visible from all files within same module but not from files in a separate module
  - mainly used to be explicit (since default)
- public
  - visible from any file in a module that imports the public entity
- private
  - only visible within same file
- there are interrelated constraints that occur based off of entity interactions and access
- testing and API design are in conflict due to need for public access

For the More Curious: Asynchronous Testing

- use XCTestExpectation

```
func testDoWork() {
    let expectation = self.expectationWithDescription("doWork completes")
    worker.doWorkInBackgroundWithCompletion { (success, error) in
        XCTAssertTrue(success)
        XCTAssertNil(error)
        expectation.fulfill()
    }
```

```
    self.waitForExpectationsWithTimeout(1, nil)
}
```

- XCTest also provides
  - expectationForNotification(_: object:handler:)
  - keyValueObservingExpectationForObject(_: keyPath:expectedValue:) for KVO

Challenge: Make Course Implement Equatable

- add extension

```
public func = =( lhs: Course, rhs: Course) -> Bool
```

Challenge: Improve Test Coverage of Web Service Responses

- two more tests for resultFromData(_: response:error:)
  i. call the method with a 404 response and no data
  ii. call the method with a 200 response and invalid JSON data

Challenge: Test Invalid JSON Dictionary

- create test to check that courseFromDictionary(_:) returns nil

30. View Controllers

- UIs went from many windows to more commonly one window
- use NSViewController to manage many nested hierarchical views

NSViewController

- view hierarchy is lazily loaded
- when view property is accessed, the view controller calls loadView
- can override loadView to create the view programmatically (ch 31)

```
class NSViewController : NSResponder {
    var nibName: String? { get }
    var representedObject: AnyObject?
    var title: String?
    var view: NSView func loadView()

    // Added in OS X 10.10:
    func viewDidLoad()
```

```
    func viewWillAppear()
    func viewDidAppear()
    func viewWillDisappear()
    func viewDidDisappear()
    ...
}
```

Starting the ViewControl Application

- create ViewControl Xcode project
  - uncheck Use Storyboards, Create Document-Based Application, and Use Core Data
  - delete MainWindow window
- create ImageViewController (subclass of NSViewController and with Create XIB file)

```
var image: NSImage? override
var nibName: String? { return "ImageViewController" }
```

- always need to override in Swift even though the filename is defaulted
- in .xib ctrl-click File's Owner
  - connections panel
    - view -> view in XIB (already connected)
- add Image View and size to full screen (use Auto Layout to pin edges)
  - Scaling -> Proportionally Up or Down
  - Bindings Inspector
    - Value -> File's Owner
    - Controller Key -> image
- similar to window controllers so far but view controller provide more flexibility

Windows, Controllers, and Memory Management

- weak window outlet was changed to strong to avoid deallocation and closing
- view controllers and views maintain strong reference (window -> contentViewController)
- top-level objects loaded from NIBs have a +1 retain count
  - managed when window and view controller load their windows and views

Container View Controllers

- master/detail -> items and view of item (left and right)
- common in navigators
- container view controllers provide this (NSSplitViewController)
- also provides NSTabViewController and NSTabView

- using view controllers provides modularity and reuse like this
- iOS makes heavy use of container view controllers (added to OSX in 10.10)

Add a Tab View Controller

- in AppDelegate

```
func applicationDidFinishLaunching(aNotification: NSNotification) {
    let flowViewController = ImageViewController()
    flowViewController.title = "Flow"
    flowViewController.image = NSImage(named: NSImageNameFlowViewTemplate)
    let columnViewController = ImageViewController()
    columnViewController.title = "Column"
    columnViewController.image = NSImage(named: NSImageNameColumnViewTemplate)
    let tabViewController = NSTabViewController()
    tabViewController.addChildViewController(flowViewController)
    tabViewController.addChildViewController(columnViewController)
    let window = NSWindow(contentViewController: tabViewController)
    let window = NSWindow(contentViewController: flowViewController)
    window.makeKeyAndOrderFront(self)
    self.window = window
}
```

- when a tab is selected the view controller asks the corresponding child view for its view and displays that

View Controllers vs. Window Controllers

- window controllers work well for simple, relatively static views
- for more complex applications use window controllers and view controllers together
    - window controller is thin and focused on window display and global actions
    - view controllers are responsible for their own area of UI
- there is a cost to view controllers -> specifically inter-view communication

Considerations for OSX 10.9 and Earlier

- prior to 10.10
    - view controllers were not part of the responder chain
    - view life cycle methods such as viewDidLoad(), viewWillAppear(), viewWillDisappear(), were not available
    - Cocoa did not provide any container view controllers
    - NSWindow’s contentViewController was not available.
- workarounds

```
extension NSViewController {
    func patchResponderChain() {
        if view.nextResponder != self {
            let resp = view.nextResponder
            view.nextResponder = self
            nextResponder = resp
        }
    }
}

// usage
override func viewDidLoad() {
    super.viewDidLoad()
    box.contentView = someViewController.view
    someViewController.patchResponderChain()
}
```

Challenge: SpeakLineViewController

- change SpeakLine to use a view controller

Challenge: Programmatic View Controller

- create ImageViewController purely programmatically
  - override loadView() to build the view(s)
  - assign the root of that hierarchy to the view property
  - do not call super.loadView(), as it will attempt to load the view hierarchy from a NIB file
  - loadView() is called to lazily load the view hierarchy the first time the view property is accessed

Challenge: Add a Window Controller

- create an NSWindowController subclass called MainWindowController
  - move the code for instantiating the view controller within it
  - NSWindowController also has a contentViewController property (mirrors the window's contentViewController)

31. View Swapping and Custom Container View Controllers

View Swapping

- tab view view changing is called view swapping
  - addSubview
  - removeFromSuperview

```
class NerdBox: NSView {
    var contentView: NSView?

    func showContentView(view: NSView) {
        contentView?.removeFromSuperview()
        addSubview(view)
        contentView = view
    }
}
```

- can use NSBox to handle constraints and auto resizing

NerdTabViewController

- displays a child view for each child view controller
  - insertChildViewController(_: atIndex:)
  - removeChildViewControllerAtIndex(_:).

```
import Cocoa class NerdTabViewController : NSViewController {
    var box = NSBox()
    var buttons: [NSButton] = []

    func selectTabAtIndex(index: Int) {
        assert(contains( 0..<childViewControllers.count, index), "index out of ran
        for (i, button) in enumerate(buttons) {
            button.state = (index == i) ? NSOnState : NSOffState
        }
        let viewController = childViewControllers[index] as! NSViewController
        box.contentView = viewController.view
    }

    func selectTab(sender: NSButton) { let index = sender.tag selectTabAtIndex(ind

    override func loadView() {
        view = NSView()
        reset()
    }
```

```
    func reset() { }

    override func insertChildViewController(
        childViewController: NSViewController,
        atIndex index: Int
    ) {
        super.insertChildViewController(childViewController, atIndex: index)
        if viewLoaded { reset() }
    }

    override func removeChildViewControllerAtIndex(index: Int) {
        super.removeChildViewControllerAtIndex(index)
        if viewLoaded { reset() }
    }
}
```

- NSBox (box) holds contents of the chosen tab
- buttons used for changing between tabs (triggers selectTab(_:)) from tag property

```
func reset() {
    view.subviews = []
    let buttonWidth: CGFloat = 28
    let buttonHeight: CGFloat = 28
    let viewControllers = childViewControllers as! [NSViewController]
    buttons = map(enumerate(viewControllers)) { (index, viewController) -> NSButto
        let button = NSButton()
        button.setButtonType(.ToggleButton)
        button.translatesAutoresizingMaskIntoConstraints = false
        button.bordered = false
        button.target = self
        button.action = Selector("selectTab:")
        button.tag = index
        button.image = NSImage(named: NSImageNameFlowViewTemplate)
        button.addConstraints([
            NSLayoutConstraint(
                item: button,
                attribute: .Width,
                relatedBy: .Equal,
                toItem: nil,
                attribute: .NotAnAttribute,
                multiplier: 1.0,
                constant: buttonWidth
            ),
            NSLayoutConstraint(
                item: button,
                attribute: .Height,
                relatedBy: .Equal,
                toItem: nil,
                attribute: .NotAnAttribute,
                multiplier: 1.0,
                constant: buttonHeight
            )]
        )
```

```
        return button
    }
    let stackView = NSStackView()
    stackView.translatesAutoresizingMaskIntoConstraints = false
    stackView.orientation = .Horizontal
    stackView.spacing = 4
    for button in buttons { stackView.addView(button, inGravity: .Center) }
    box.translatesAutoresizingMaskIntoConstraints = false
    box.borderType = .NoBorder
    box.boxType = .Custom
    let separator = NSBox()
    separator.boxType = .Separator
    separator.translatesAutoresizingMaskIntoConstraints = false
    view.subviews = [stackView, separator, box]
    let views = ["stack": stackView, "separator": separator, "box": box]
    let metrics = ["buttonHeight": buttonHeight]
    func addVisualFormatConstraints(visualFormat:String) {
        view.addConstraints(
            NSLayoutConstraint.constraintsWithVisualFormat(
                visualFormat,
                options: .allZeros,
                metrics: metrics,
                views: views
            )
        )
    }
    addVisualFormatConstraints("H: |[ stack] |")
    addVisualFormatConstraints("H: |[ separator] |")
    addVisualFormatConstraints("H: |[ box( > = 100)] |")
    addVisualFormatConstraints("V: |[ stack(buttonHeight)][separator(== 1)][box(>=
    if childViewControllers.count > 0 { selectTabAtIndex(0) }
}
```

- NSStackView arranges one ore more views in order horizontally or vertically with a gravity to indicate where they should be positioned
- in AppDelegate

```
let tabViewController = NerdTabViewController()
```

Adding Tab Images

- one option to set the view for each tab view

```
func addChildViewController(childViewController: NSViewController, tabImage: NSIma
```

- not a great option -> need to manage the the closed view and does not work with existing NSViewControllers

- could use representedObject but this causes issues
  - better to create a strongly-typed property to specific to model object than to use AnyObject?
- in NerdTabViewController

```
@objc protocol ImageRepresentable { var image: NSImage? { get } }

// in reset()
X button.image = NSImage(named: NSImageNameFlowViewTemplate) X
if let viewController = viewController as? ImageRepresentable { button.image = vie
else { button.title = viewController.title! }

// ImageViewController
class ImageViewController: NSViewController, ImageRepresentable {
    ...
```

Challenge: Boxless NerdTabViewController

- replace the NSBox in NerdTabViewController with a plain NSView
  - rather than setting the box's contentView -> add selected child view controller's view as a subview
  - use Auto Layout to ensure that the view resizes with the container view
  - be sure to remove the previously selected view controller's view

Challenge: NerdSplitViewController

- use an ordinary NSView and not NSSplitView to create a custom split view controller
- only use two child view controllers and split horizontally or vertically
- create a pane for each child view controller

Challenge: Draggable Divider

- need a way to receive child view events on the draggable divider and then update constraints in response
- NSViewControllers are in the responder chain
  - implement mouseDragged(_:) in the NerdSplitViewController
  - or use an NSPanGestureRecognizer added to a view representing the divider

32. Storyboards

- organized by scenes (windows and views)
- connected by segues (relationships between scenes)

- can be used for simple connected views (only two views)
- can also be used to manage a complex graph of views and view controllers (previously done programmatically)

A New UI for RanchForecast

- add a web view to navigate to URL
- create RanchForecastSplit
    - check Use Storyboards
- existing relationship shows "Relationship "window content" to View Controller"
    - contents of the view controller's view will be shown as window's content
- delete the existing ViewController in the storyboard and delete the ViewController.swift file
- drag a Vertical Split View Controller onto the canvas (below the Window Controller)
- ctrl-drag from the blue Window Controller icon -> split view controller
    - Relationship Segue window
        - window content
- some editing is not possible when the storyboard editor is zoomed out
- copy in ScheduleFetcher and Course files (with Copy items if needed)

Adding the course list

- create CourseListViewController

```
import Cocoa

class CourseListViewController: NSViewController {
    dynamic var courses: [Course] = []
    let fetcher = ScheduleFetcher()

    @IBOutlet var arrayController: NSArrayController!

    override func viewDidLoad() {
        super.viewDidLoad()
        fetcher.fetchCoursesUsingCompletionHandler { (result) in
            switch result {
            case .Success(let courses):
                print("Got courses: \(courses)")
                self.courses = courses
            case .Failure(let error):
                print("Got error: \(error)")
                NSAlert(error: error).runModal()
                self.courses = []
            }
        }
```

```
    }
}
```

- in storyboard select narrow view controller
    - Identity Inspector
        - Class -> CourseListViewController
- add array controller to Course List View Controller
    - arrayController outlet on Course List View Controller -> array controller
    - NOTE: important to add to correct scene within storyboard
- array controller Bindings Inspector
    - Content Array -> Course List View Controller's self.courses
- add a table to Course List View Controller Scene
    - table view is View Based and has two columns
- Configure Auto Layout constraints for the table view
    - stretch the table view to take up space of parent view
    - pin on all four sides
    - set minimum width for scroll view
        - click Pin
        - check Width
        - click "Add 1 Constraint"
    - select created constraint
        - Constraints
            - Width - (174) - Bordered Scroll View - Table View -> Attributes Inspector
                - Relation -> Greater Than or Equal
- table view Bindings Inspector
    - Content -> Array Controller's arrangedObjects controller key
    - Selection Indexes -> Array Controller's selectionIndexes
- Double-click header of first column -> "Date"
- date text field Bindings Inspector
    - Value -> Table Cell View
    - Model Key Path -> objectValue.nextStartDate
- drop a Date Formatter on the text field
- Double-click header of first column -> "Title"
- title text field Bindings Inspector
    - Value -> Table Cell View
    - Model Key Path -> objectValue.title

Adding the web view

- create WebViewController.swift

```
import Cocoa
import WebKit

class WebViewController: NSViewController {
    var webView: WKWebView { return view as! WKWebView }

    override func loadView() {
        let webView = WKWebView()
        view = webView
    }

    func loadURL(url: NSURL) {
        let request = NSURLRequest(URL: url)
        webView.loadRequest(request)
    }
}
```

- by overriding loadView, the view from the storyboard isn't used
- in storyboard
  - right, wider view controller Identity Inspecotr
    - Class -> WebViewController

Connecting the Course List Selection with the Web View

- need to call loadURL on web view when the table view selection changes
  - directly couple the two view controllers by adding a reference from CourseListViewController to WebViewController
    - simplest method but causes tight coupling which is bad
  - post a notification from the CourseListViewController when the selection changes and observe that notification in the WebViewController
    - reduces coupling
    - one-way instead of bidirectional
    - allows update of multiple parties
  - create a CourseListViewControllerDelegate protocol and conform WebViewController to it
    - direct but could couple sibling views
  - add a closure property to the CourseListViewController to be called when the selection changes
    - good -> requires avoiding a strong reference cycle
  - use KVO to observe the array’s selectedObjects property
    - avoid relying on KVO for simple tasks
    - often not apparent the KVO is used in code
    - compile does not check it

- choice to use
    - create a third view controller which will contain the master and detail view controllers
        - parent view manages communication between the two views

Creating the CourseListViewControllerDelegate

- in CourseListViewController.swift

```
protocol CourseListViewControllerDelegate: class {
    func courseListViewController(viewController: CourseListViewController, select
}

// in CourseListViewController class
weak var delegate: CourseListViewControllerDelegate? = nil

@IBAction func selectCourse(sender: AnyObject) {
    let selectedCourse = arrayController.selectedObjects.first as! Course?
    delegate?.courseListViewController(self, selectedCourse: selectedCourse)
}
```

- in storyboard
    - ctrl-drag table view -> Course List View Controller -> selectCourse:

Creating the parent view controller

- create MainSplitViewCOntroller.swift

```
import Cocoa

class MainSplitViewController: NSSplitViewController, CourseListViewControllerDele
    var masterViewController: CourseListViewController {
        let masterItem = splitViewItems[0] as! NSSplitViewItem
        return masterItem.viewController as! CourseListViewController
    }

    var detailViewController: WebViewController {
        let masterItem = splitViewItems[1] as! NSSplitViewItem
        return masterItem.viewController as! WebViewController
    }

    let defaultURL = NSURL(string: "http://www.bignerdranch.com/")!

    override func viewDidLoad() {
        super.viewDidLoad()
        masterViewController.delegate = self
        detailViewController.loadURL(defaultURL)
    }

    // MARK: CourseListViewControllerDelegate
    func courseListViewController(
```

```
        viewController: CourseListViewController,
        selectedCourse: Course?
    ) {
        if let course = selectedCourse { detailViewController.loadURL(course.url)
        else { detailViewController.loadURL( defaultURL) }
    }
}
```

- in storyboard
- select Split View Controller within Split View Controller Scene
    - Identity Inspector
        - Class -> MainSplitViewController

For the More Curious: How is the Storyboard Loaded?

- managed by Info.plist
    - NSMainStoryboardFile -> Main
- scenes can be loaded programmatically using NSStoryboard
- one scene can be loaded programmatically using the identifier

```
if let storyboard = NSStoryboard(name: "Main", bundle: nil) {
    if let vc = storyboard.instantiateControllerWithIdentifier("Palette") as? NSVi
        paletteWindow = NSWindow(contentViewController: vc)
        paletteWindow.makeKeyAndOrderFront(nil)
    }
}
```

33. Core Animation

- animation in Ch. 17 works for relatively simple, static animations
- Core Animation relies on composing a view by creating layers rather than drawing
    - images
    - Bezier paths
    - modify size, opacity, etc
- relies on OpenGL and can be useful when not animating

CALayer

- base class for all kinds of layers
- arranged in a hierarchy
- set layer in view (layer-hosting view)
- can draw into a CALayer
- more common to set properties on the layer

- not typically subclassed
- not part of the responder chain
- all event handling has to be custom code

```
class CALayer: NSObject {
    var bounds: CGRect
    var position: CGPoint
    var zPosition: CGFloat
    var frame: CGRect
    var opacity: CGFloat
    var hidden: Bool
    var mask: CALayer!
    var borderWidth: CGFloat
    var borderColor: CGColor!
    var cornerRadius: CGFloat
    var shadowOpacity: CGFloat
    var shadowRadius: CGFloat
    var shadowOffset: CGSize
    var shadowColor: CGColor!
    var actions: [NSObject : AnyObject]!
    // Defaults to nil!
    var delegate: AnyObject!
    // NSObject (CALayerDelegate)
    // ...
}
```

- CATransaction can be used to group and synchronize multiple animations, as well as to disable animations temporarily

Scattered

- create an application called Scattered
  - Use Storyboards
- in ViewController.swift

```
class ViewController: NSViewController {
    var textLayer: CATextLayer!
    var text: String?

    func addImagesFromFolderURL(folderURL: NSURL) { }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Set view to be layer-hosting:
        view.layer = CALayer()
        view.wantsLayer = true
        let textContainer = CALayer()
        textContainer.anchorPoint = CGPoint.zeroPoint
        textContainer.position = CGPointMake(10, 10)
        textContainer.zPosition = 100
```

```
        textContainer.backgroundColor = NSColor.blackColor().CGColor
        textContainer.borderColor = NSColor.whiteColor().CGColor
        textContainer.borderWidth = 2
        textContainer.cornerRadius = 15
        textContainer.shadowOpacity = 0.5
        view.layer!.addSublayer(textContainer)
        let textLayer = CATextLayer()
        textLayer.anchorPoint = CGPoint.zeroPoint
        textLayer.position = CGPointMake(10, 6)
        textLayer.zPosition = 100
        textLayer.fontSize = 24
        textLayer.foregroundColor = NSColor.whiteColor().CGColor
        self.textLayer = textLayer
        textContainer.addSublayer(textLayer)
        // Rely on text's didSet to update textLayer's bounds:
        text = "Loading..."
        let url = NSURL(fileURLWithPath: "/Library/Desktop Pictures")!
        addImagesFromFolderURL(url)
    }

    // remove override var represenedObject
```

- when app launches
  - configure view to be layer-hosting (ordering is important)
    - assign layer property
    - wantsLayer = true
  - configure two layers
    - textContainer -> CALayer
    - textLayer -> CATextLayer
- uses anchorPoint of (0, 0) for lower left corner (defaults to 0.5, 0.5)
- layers have a frame property, but more common to set position and bounds independently
- set text property's didSet

```
var text: String? {
    didSet {
        let font = NSFont.systemFontOfSize(textLayer.fontSize)
        let attributes = [NSFontAttributeName : font]
        var size = text?.sizeWithAttributes(attributes) ?? CGSize.zeroSize
        // Ensure that the size is in whole numbers:
        size.width = ceil(size.width)
        size.height = ceil(size.height)
        textLayer.bounds = CGRect(origin: CGPoint.zeroPoint, size: size)
        textLayer.superlayer.bounds = CGRect(x: 0, y: 0, width: size.width + 16, h
        textLayer.string = text
    }
}

func thumbImageFromImage(image: NSImage) -> NSImage {
    let targetHeight: CGFloat = 200.0
    let imageSize = image.size
    let smallerSize = NSSize(width: targetHeight * imageSize.width / imageSize.hei
```

```
    let smallerImage = NSImage(size: smallerSize, flipped: false) { (rect) -> Bool
        image.drawInRect(rect)
        return true
    }
    return smallerImage
}

func addImagesFromFolderURL(folderURL: NSURL) {
    let t0 = NSDate.timeIntervalSinceReferenceDate()
    let fileManager = NSFileManager()
    let directoryEnumerator = fileManager.enumeratorAtURL(
        folderURL,
        includingPropertiesForKeys: nil,
        options: nil,
        errorHandler: nil
    )!
    var allowedFiles = 10
    while let url = directoryEnumerator.nextObject() as? NSURL {
        // Skip directories:
        var isDirectoryValue: AnyObject?
        var error: NSError?
        url.getResourceValue(& isDirectoryValue, forKey: NSURLIsDirectoryKey, erro
        if let isDirectory = isDirectoryValue as? NSNumber where isDirectory.boolV
            let image = NSImage(contentsOfURL: url)
            if let image = image {
                allowedFiles--
                if allowedFiles < 0 { break }
                let thumbImage = thumbImageFromImage(image)
                presentImage(thumbImage)
                let t1 = NSDate.timeIntervalSinceReferenceDate()
                let interval = t1 - t0
                text = String(format: "% 0.1fs", interval)
            }
        }
    }
}

func presentImage(image: NSImage) {
    let superlayerBounds = view.layer!.bounds
    let center = CGPoint(x: superlayerBounds.midX, y: superlayerBounds.midY)
    let imageBounds = CGRect(origin: CGPoint.zeroPoint, size: image.size)
    let randomPoint = CGPoint(x: CGFloat(arc4random_uniform(UInt32(superlayerBound
    let timingFunction = CAMediaTimingFunction(name: kCAMediaTimingFunctionEaseInE
    let positionAnimation = CABasicAnimation()
    positionAnimation.fromValue = NSValue(point: center)
    positionAnimation.duration = 1.5
    positionAnimation.timingFunction = timingFunction
    let boundsAnimation = CABasicAnimation()
    boundsAnimation.fromValue = NSValue(rect: CGRect.zeroRect)
    boundsAnimation.duration = 1.5
    boundsAnimation.timingFunction = timingFunction
    let layer = CALayer()
    layer.contents = image
    layer.actions = ["position" : positionAnimation, "bounds" : boundsAnimation]
    CATransaction.begin()
    view.layer!.addSublayer(layer)
```

```
    layer.position = randomPoint
    layer.bounds = imageBounds
    CATransaction.commit()
}
```

Implicit Animation and Actions

- many properties, when modified, have implicit animation
- many ways to customize implicit animations
- can modify actions property (CALayer, dictionary)
    - determines what to do when a property is assigned
- CABasicAnimation
    - fromValue
    - toValue
    - also position and bounds
    - to display image assign to contents
    - contentsGravity -> how contents are scaled
- CATransaction -> group several changes to be executed at once
    - surround with CATransaction.begin() and CATransaction.commit()
- use CATransaction.setDisableActions(true) to disable

More on CALayer

- use drawLayer(_: inContext:) for custom drawing using Core Graphics/Quartz
- useful subclasses
    - CATextLayer -> text
    - CAShapeLayer -> stroked or filled path
    - CAGradientLayer -> configurable gradient
    - CAOpenGLLayer -> OpenGL
- base layer is private _NSViewBackingLayer (knows to draw contents of view upon itself)

Challenge: Show Filenames

- add a text layer to each image layer to show the filename of that image
- supply an additional parameter to presentImage(_:)
- add shadows and borders to the layers

Challenge: Reposition Image Layers

34. Concurrency

Multithreading

- threads enable single application to do many things at once
- each has its own stack
- heap is shared
- in Cocoa -> main thread manages display and handles events from window manager
- threads are good for tasks like
    - long-running computations
    - hardware I/O
    - synchronous network communication
- NSOperationQueue and Grand Central Dispatch

A Deep Chasm Opens Before You

- race conditions are difficult to solve
- cannot make assumptions about when a thread will be scheduled, how long it will execute before being interrupted, other threads running, exclusivity for data access
- make effort to minimize shared access to mutable data structures

Improving Scattered: Time Profiling in Instruments

- remove all traces of allowedFiles in ViewController

Introducing Instruments

- open Product and select Profile
- click record and then stop to profile
- Time Profiler is the default
    - takes snapshot of stack repeatedly
- many details to Instruments

Analyzing output from Instruments

- two categories of blocking problems
  - CPU-bound
    - CPU is bottleneck due to computation
  - I/O-bound
- issue in Scattered is a combo of both
  - loading images and generating thumbnails in viewDidLoad blocks main thread

NSOperationQueue

- NSOperationQueue is good for processing chunks of information in the background
- collection of NSOperation
- access main queue that represents main thread NSOperationQueue.mainQueue()
- NSOperationQueue has system dictated default number of operations
  - can override with maxConcurrentOperationCount

Multithreaded Scattered

- in ViewController

```
let processingQueue: OperationQueue = {
    let result = OperationQueue()
    //result.maxConcurrentOperationCount = 4
    return result
}()

func addImagesFromFolderURL(_ folderURL: URL) {
    processingQueue.addOperation {
        let t0 = Date.timeIntervalSinceReferenceDate

        let fileManager = FileManager()
        let directoryEnumerator = fileManager.enumerator(
            at: folderURL,
            includingPropertiesForKeys: nil,
            options: [],
            errorHandler: nil
        )!

        while let url = directoryEnumerator.nextObject() as? URL {
            // Skip directories:

            var isDirectoryValue: AnyObject?
            do {
                try (url as NSURL).getResourceValue(&isDirectoryValue,
                    forKey: URLResourceKey.isDirectoryKey)
            } catch {
```

```
                print("error checking whether URL is directory: \(error)")
                continue
            }

            self.processingQueue.addOperation {
                guard let isDirectory = isDirectoryValue as? Bool,
                    isDirectory == false else {
                        return
                }

                guard let image = NSImage(contentsOf: url) else {
                    return
                }

                let thumbImage = self.thumbImageFromImage(image)

                OperationQueue.main.addOperation {
                    self.presentImage(thumbImage)
                    let t1 = Date.timeIntervalSinceReferenceDate
                    let interval = t1 - t0
                    self.text = String(format: "%0.1fs", interval)
                }
            }
        }
    }
}
```

- instead of creating NSOperation objects use NSOperationQueue's addOperationWithBlock (_:)

Thread synchronization

- Scattered already only had shared mutable global data access in main thread to the Core Animation layers
- for basic mutex use NSRecursiveLock

```
let lock = NSRecursiveLock()

func addImage(image: NSImage) {
    lock.lock()
    images.append(image)
    lock.unlock()
}
```

- prevents multiple threads from running the code between lock and unlock at once
- recursive -> the same thread that has the lock can run same code recursively locking and unlocking
- Cocoa provides other tools
    - NSLock

        - NSCondition
- see Advanced Mac OS X Programming

For the More Curious: Faster Scattered

- GCD maintains thread pool based on cores and system load
- faster to use two queues
    - one to load images
    - one to generate thumbnails
- avoids duplicate I/O and CPU concurrent operations

Challenge: An Even Better Scattered

- adapt Scattered to use two queues
    - use NSData to read images to speed disk I/O and pass to NSImage

35. NSTask

- each app (as a bundle) has an executable which causes a fork when the process is started
- NSTask wraps fork and exec

ZIPspector

- zipinfo shows contents of a zip file
- create ZIPspector Xcode project
    - Document based
    - no storyboard or core data
- Target
    - Info
        - Identifier -> com.pkware.zip-archive
        - Role -> com.pkware.zip-archive
- conform Document to NSTableViewDataSource

```
@IBOutlet weak var tableView: NSTableView!
var filenames: [String] = []
```

- add table view with one column named Filenames
    - Attributes Inspector
        - Editable -> `[]`

- in text field in column
  - constraints
    - left and right -> 2 points
  - Bindings Inspector
    - Bind to -> Table Cell View
    - Model Key Path -> objectValue
-

```
// MARK: - NSTableViewDataSource
func numberOfRows(in tableView: NSTableView) -> Int {
    return filenames.count
}

func tableView(_ tableView: NSTableView, objectValueFor tableColumn: NSTableColumn
    return filenames[row]
}
```

- to read

```
override func read(from url: URL, ofType typeName: String) throws {
    // Which file are we getting the zipinfo for?
    let filename = url.path

    // Prepare a task object
    let task = Process()
    task.launchPath = "/usr/bin/zipinfo"
    task.arguments = ["-1", filename]

    // Create the pipe to read from
    let outPipe = Pipe()
    task.standardOutput = outPipe

    // Start the process
    task.launch()

    // Read the output
    let fileHandle = outPipe.fileHandleForReading
    let data = fileHandle.readDataToEndOfFile()

    // Make sure the task terminates normally
    task.waitUntilExit()
    let status = task.terminationStatus

    // Check status
    guard status == 0 else {
        let errorDomain = "com.bignerdranch.ProcessReturnCodeErrorDomain"
        let errorInfo = [ NSLocalizedFailureReasonErrorKey : "zipinfo returned \(s
        let error = NSError(
            domain: errorDomain,
            code: 0,
            userInfo: errorInfo
        )
        throw error
```

```
    }

    // Convert to a string
    let string = String(data: data, encoding: String.Encoding.utf8)!

    // Break the string into lines
    filenames = string.components(separatedBy: "\n")
    print("filenames = \(filenames)")

    // In case of revert
    tableView?.reloadData()
}
```

- NSPipe to standardOutput is a buffer and can supply a file handle (fileHandleForReading)

Asynchronous Reads

- will read in background from a process that occasionally returns data (using /sbin/ping)

iPing

- create Xcode iPing
  - not document-based
  - check Use Storyboards
- ViewController.swift

```
import Cocoa

class ViewController: NSViewController {

    @IBOutlet var outputView: NSTextView!
    @IBOutlet weak var hostField: NSTextField!
    @IBOutlet weak var startButton: NSButton!
    var task: Process?
    var pipe: Pipe?
    var fileHandle: FileHandle?

    @IBAction func togglePinging(_ sender: NSButton) {
        // Is there a running task?
        if let task = task {
            // If there is, stop it!
            task.interrupt()
        } else {
            // If there isn't, start one.

            // Create a new task
            let task = Process()
            task.launchPath = "/sbin/ping"
            task.arguments = ["-c10", hostField.stringValue]
```

```
        // Create a new pipe for standardOutput
        let pipe = Pipe()
        task.standardOutput = pipe

        // Grab the file handle
        let fileHandle = pipe.fileHandleForReading

        self.task = task
        self.pipe = pipe
        self.fileHandle = fileHandle

        let notificationCenter = NotificationCenter.default
        notificationCenter.removeObserver(self)
        notificationCenter.addObserver(self,
                                       selector: #selector(ViewController.receiveDataRea
                                           name: FileHandle.readCompletionNotification,
                                         object: fileHandle)
        notificationCenter.addObserver(self,
                                       selector: #selector(ViewController.receiveTaskTer
                                           name: Process.didTerminateNotification,
                                         object: task)

        task.launch()

        outputView.string = ""

        fileHandle.readInBackgroundAndNotify()
    }

}


func appendData(_ data: Data) {
    let string = String(data: data, encoding: String.Encoding.utf8)! as String
    let textStorage = outputView.textStorage!
    let endRange = NSRange(location: textStorage.length, length: 0)
    textStorage.replaceCharacters(in: endRange, with: string)
}


func receiveDataReadyNotification(_ notification: Notification) {
    let data = notification.userInfo![NSFileHandleNotificationDataItem] as! Da
    let length = data.count

    print("received data: \(length) bytes")
    if length > 0 {
        self.appendData(data)
    }

    // If the task is running, start reading again
    if let fileHandle = fileHandle {
        fileHandle.readInBackgroundAndNotify()
    }
}
```

```
    func receiveTaskTerminatedNotification(_ notification: Notification) {
        print("task terminated")

        task = nil
        pipe = nil
        fileHandle = nil

        startButton.state = 0
    }
}
```

- open storyboard
  - Window Controller Scene -> Window
    - Title -> iPing
  - View Controller Scene
    - add text field, button, and text view
    - text view Attributes Inspector
      - Editable -> `[]`
      - Type -> Toggle
      - Title -> Start Pinging
      - Alternate Title -> Stop Pinging
    - button
      - target -> ViewController
      - action -> togglePinging
    - connect outlets to corresponding views

Challenge: .tar and .tgz Files

- extend ZIPspector to read .tar and .tgz files

36. Distributing Your App

Build Configurations

- builds so far are debug builds and contain debug symbols
- use release build for customers
- can add build configurations
- project editor -> Info
- run, test, profile, analyze, archive -> each has a build configuration
- Scheme Editor
  - Product -> Scheme -> Edit Scheme...

Preprocessor Directives: Using Build Configurations to Change Behavior

- can hardcode modified behavior into build configurations using preprocessor directives
- project editor -> Build Settings
    - Swift Compiler -> Custom Flags
        - Other Swift Flags
            - Debug -> -DDEBUG
- makes the compile-time value DEBUG only available for the Debug builds configuration
- can check in code

```
#if DEBUG
    printAlot()
#else
    print()
#endif
```

- other code is completely omitted from the build app

Creating a Release Build

- want to archive target
- Scheme Editor
    - Archive
        - Build Configuration -> Release
- when archived, appears in Organizer in Archives tab
- can extract the app bundle from the archive using export button in Archives tab (Export as a Mac Application)

A Few Words on Installers

- can store bundles in zip archive or create installer as DMG
- Mac App Store apps cannot use an installer
- advice is to avoid installers in general

App Sandbox

- sandboxing -> security method that constrains an application's interactions with the system (filesystem, network)
- Apple has required sandboxing of all apps on iOS since iOS 2.0

- sandboxing introduced to the Mac in OS X 10.7
- all applications on the Mac App Store must be sandboxed.

Entitlements

- with sandboxing permitted to access bundle files and files within container
- entitlements are opt in permissions to access more
    - network
    - contacts
    - etc
- enable App Sandbox
- add entitlements to App Sandbox

Containers

- sandboxed apps are provided a container in ~/Library/Containers
- use NSFileManager's URLsForDirectory(_: inDomains:)

Mediated file access and Powerbox

- includes access to temp files
- access to file-open/save dialog (NSOpenPanel, NSSavePanel)
- Powerbox is a system daemon that manages this and displays sheets when NSOpenPanel and NSSavePanel are called
- AppKit Open Recent menu provides similar capabilities
- see App Sandbox Design Guide

The Mac App Store

- App Store handles
    - purchasing
    - installation
    - packaging
    - distribution
- for apps not sandboxed or meeting guidelines, other distribution methods are required
- for distribution use Xcode
    - sign binary
    - provide description
    - submission via Organizer

Receipt Validation

- no OS support for license validation (unlike iOS)
- means no copy protection without special effort (using receipt validation)
- receipt contains the application's bundle identifier, its version string, and a hash of the computer's GUID
- receipts are cryptographically signed by Apple

Local receipt verification

- follow these steps
  - verify that the receipt is present
  - verify that the receipt is properly signed by Apple
  - verify that the bundle identifier in the receipt matches
  - verify that the version identifier matches
  - verify that the hash contained in the receipt matches the computer's GUID hash
- duplicate bundle and version IDs as constants in app code to avoid easy duplication
- if validation fails, exit with 173
- code for verification is low-level C
- see Receipt Validation Programming Guide

Server-based verification

- connect server to App Store server (benefits including prevention of man in the middle attack)
- follow following steps
  - read the receipt file at the location given by NSBundle.appStoreReceiptURL()
  - send the data from that file to your server
  - on server, send an HTTP POST request with a JSON body containing the receipt data to https://buy.itunes.apple.com/verifyReceipt
  - on server, interpret the response from the Mac App Store and send an appropriate response to your app
  - in app, handle the response from your server, calling exit(173) if necessary
- see Receipt Validation Programming Guide

Additional Notes

Issues with computed properties

- not stored (reduces space overhead)
- evaluated on each access