iOS Development

iOS 11 • Xcode 9

STUDENT GUIDE

ABOUTOBJECTS

Contact

About Objects, Inc.
11911 Freedom Drive
Suite 700
Reston, VA 20190

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

Course Information

Author: Jonathan Lehr
Revision: 5.1
Last Update: 11/13/2017

A rapid introduction to iOS development, with comprehensive coverage of Xcode, Interface Builder, and the UIKit framework.

Copyright Notice

iOS Development

STUDENT GUIDE

CHAPTER ONE

Introduction

About the Author

Jonathan Lehr

Co-founder and VP, Training

About Objects

jonathan@aboutobjects.com

ABOUTOBJECTS

Cocoa touch

Cocoa touch Apple's brand name for the iOS development environment.

The most commonly used frameworks are:

- Foundation – low-level, non-UI components
- UIKit – common UI components
- Core Graphics – 2D drawing C library
- Core Animation – 2.5D animation

iOS Design Goals

Create apps that…

- Are easy and rewarding to use
- Behave like first-class citizens of the platform
- Are cost-effective to implement
- Run efficiently
- Look great!

iOS Design Pitfalls

- Insufficient collaboration between Design and Engineering
- Designing highly custom UI without understanding cost
- Ignoring platform's well-defined user interaction patterns and visual vocabulary

Undesirable Outcomes

- Apps that are less stable, have more defects, run inefficiently
- Apps that confuse users or fail to meet expectations
- Unnecessary implementation complexity
- Future maintenance headaches
- Budget impact of needless customizations
- Risk of breakage by subsequent iOS releases

Agenda

- Working with Xcode and the iOS Simulator
- Memory management
- The View Hierarchy and Responder Chain patterns
- Drawing and animation
- Blocks (AKA closures or lambda expressions)
- Adaptive layout and view resizing
- View controllers and view loading
- Controls (text fields, buttons) and the Target-Action pattern
- Working with text input and the system keyboard
- Creating and editing nib files in Interface Builder
- Managing static and dynamic table views
- Understanding the Delegate and Data Source patterns
- Navigation controllers, navigation bars, and bar items
- Working with storyboards and segues

Additional Topics

- Debugging
- Performance testing
- Concurrency
- Persistence (file-based, Core Data, web services)
- Key-Value Coding
- Persistence (plist/JSON-based, Core Data, REST)
- Managing concurrency with Grand Central Dispatch and NSOperation/NSOperationQueue

CHAPTER TWO

MRR and Declared Properties

CHAPTER THREE

Intro to UIKit

Xcode Concepts

- Xcode projects
- What is an app?
- What is a build?
- Interface Builder

Anatomy of an Xcode Project

- Navigator
- File system contents
- Interface Builder documents
- Utilities
  - Inspectors
  - Libraries
- Image catalogs
- Build targets and schemes
- Running an app

Key Role Players

`UIView`

- Drawing and animation
- Hit testing and event distribution
- UIViewController

`UIViewController`

- Manages a view hierarchy
- Coordinates view and model objects

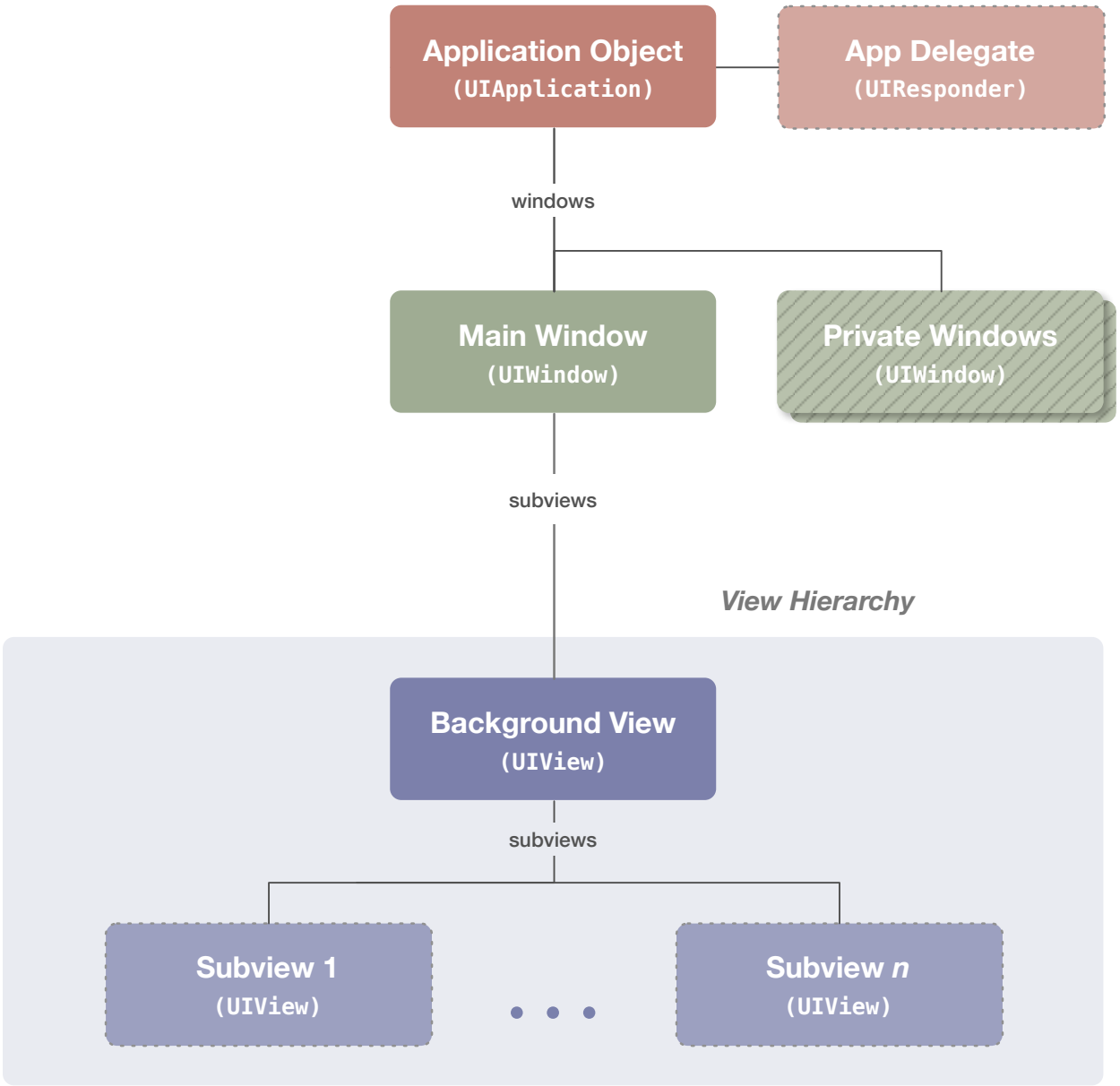
`UIWindow`

- An app ordinarily creates a single window object.
- Views must be in a window's **subviews** array to appear on-screen.

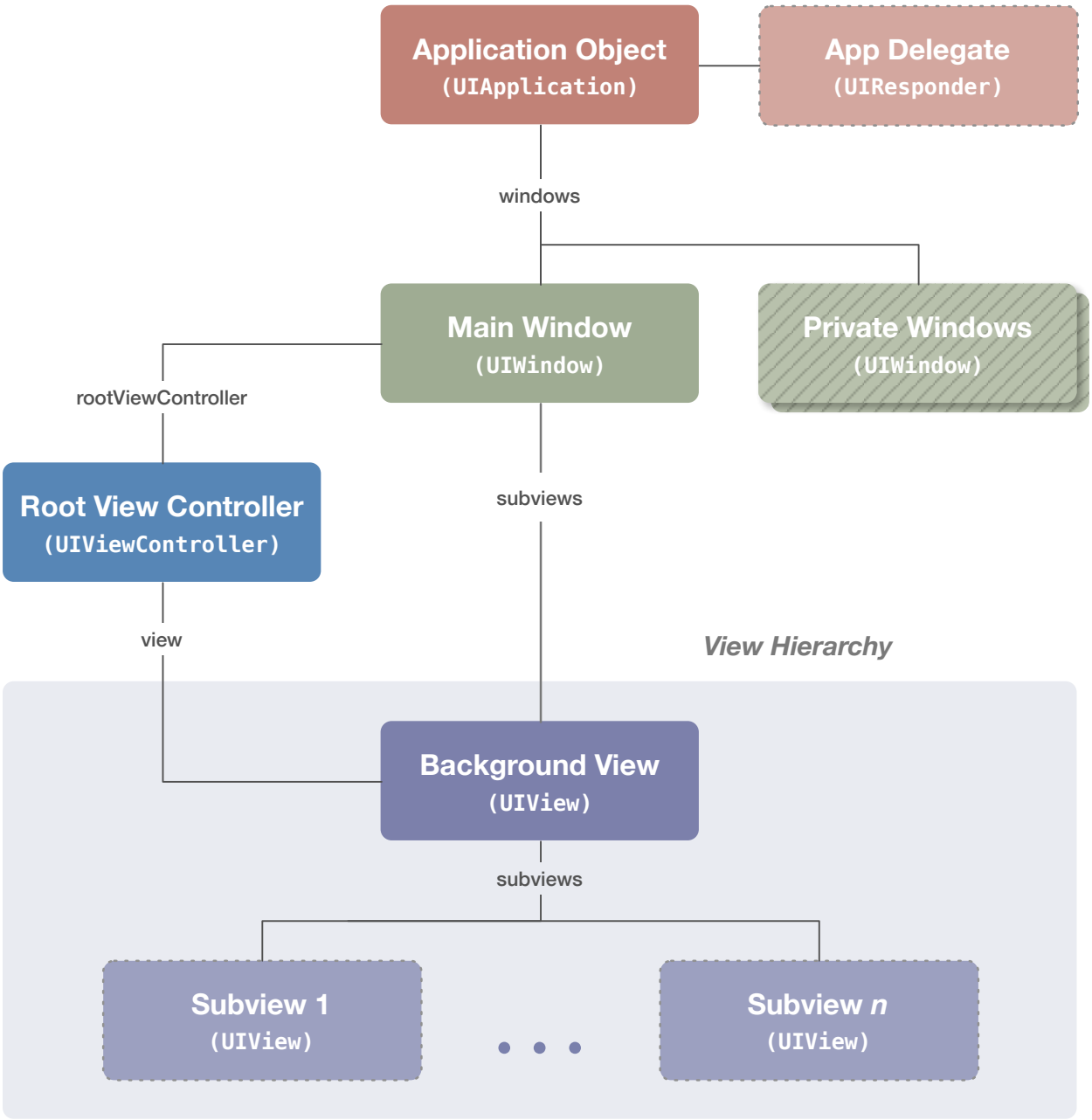
`UIApplication`

- Singleton
- Manages your app's windows

Initial App Components

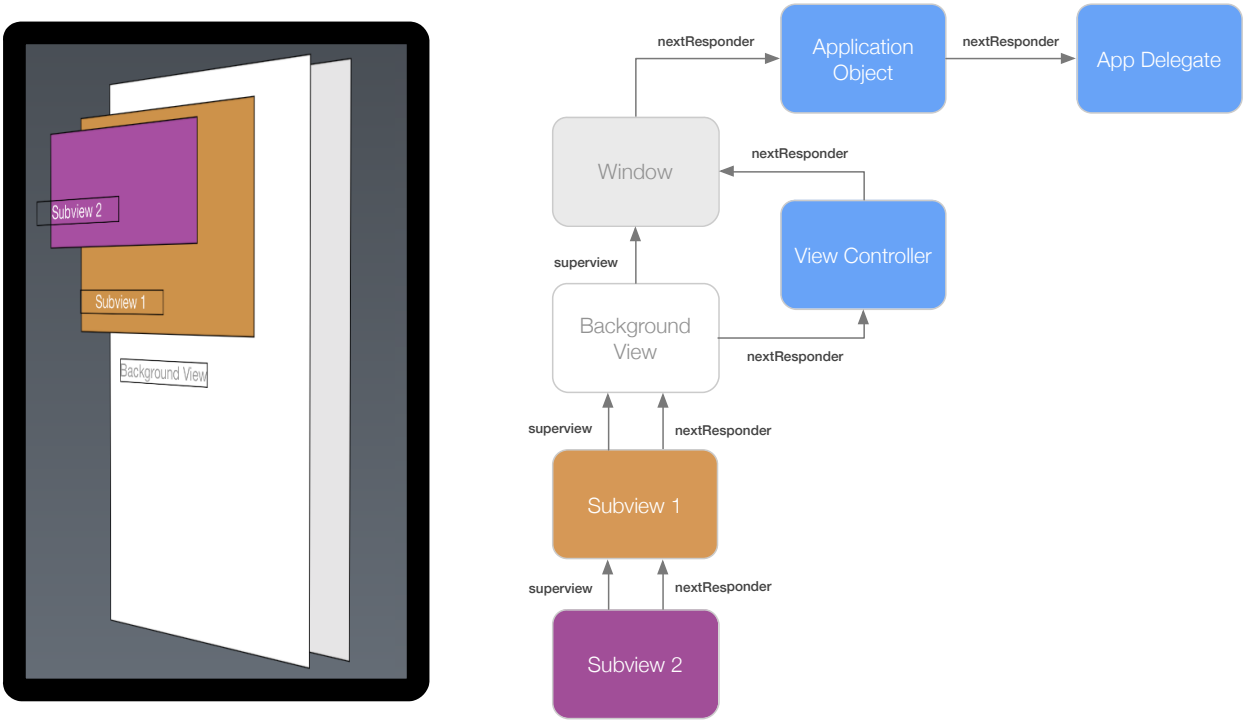


Where Root View Controller Fits



View Hierarchy

- Views can contain other views (**subviews**), and have knowledge of their parent view (**superview**).
- The order of the view hierarchy determines *compositing order*.



A view hierarchy, with off-screen objects

Windows

- iOS windows are specialized views.
- Generally covered with other views, so not visible to user.
- Your app typically manages a single window containing the subviews that are currently needed on screen.
- If your app supports external displays, each would have its own window.

Application

- Each app has one instance (singleton) of `UIApplication`

```
// Swift
UIApplication.shared

// Objective-C
[UIApplication sharedApplication]
```

- Integrates app with OS environment
- Manages app's
  - Windows
  - Status Bar
  - Event loop

App Delegate

- Implements `UIApplicationDelegate` Protocol
  - All methods optional
- `UIApplication` sends messages to delegate concerning:
  - App launch and termination
  - Memory Warnings
  - Changes in orientation
  - Becoming the active application and being deactivated

CHAPTER FOUR

View Hierarchy and Responder Chain

Views

- Instances of `UIView` class
- Inherit methods for responding to touch, motion, and keyboard events from `UIResponder`
- Define methods for:
  - Drawing content
  - Managing geometry
  - Managing *view hierarchy*
  - Animation

View Geometry

Views provide several properties for managing their geometry:

frame – view's size and position (location of upper-left corner relative to its superview)

bounds – size and position of view's content (relative to itself)

center – location of the view's center relative to its superview's origin

transform – specifies transformations (scale, rotate, translate) of the view's underlying geometry

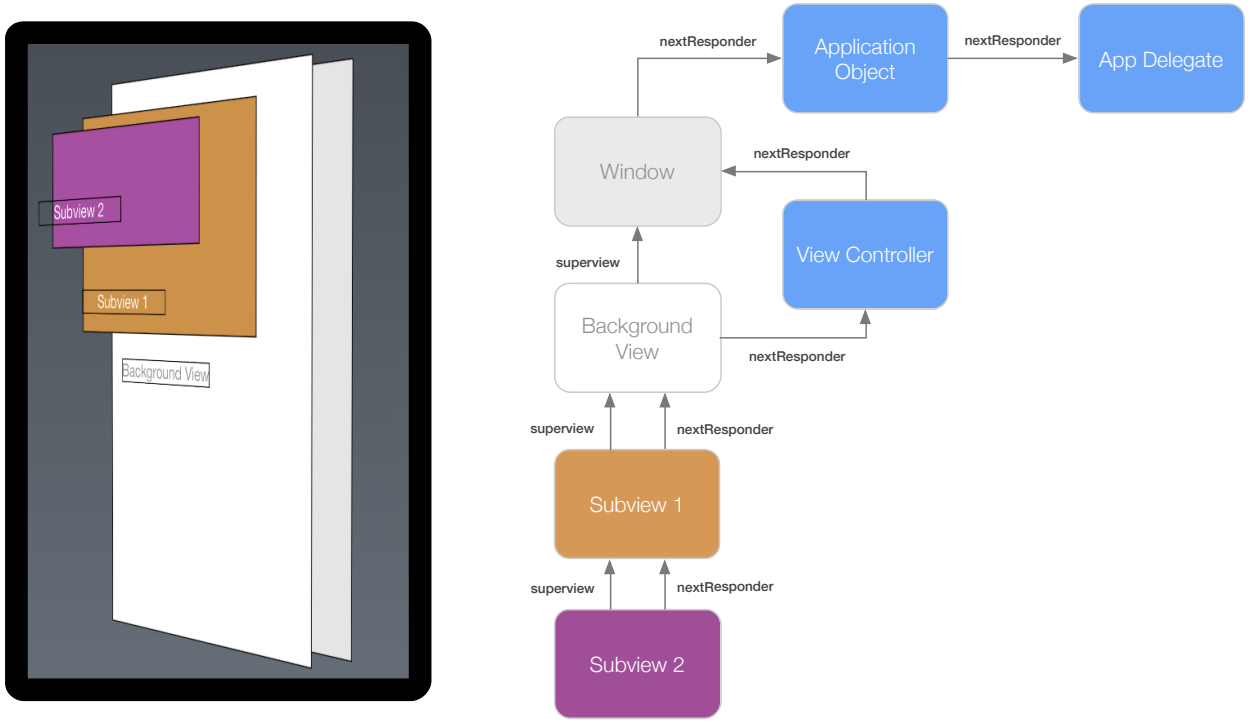
Note that changing the size of the frame changes the size of the bounds, and vice versa. Also, changing the origin of the frame changes the location of the center, and vice versa.

Responders

- Subclasses of `UIResponder`
- Respond to touch and motion events
- Implement **Responder Chain**
- Provide
  - Input View
  - Undo Manager

View Hierarchy

- Views can contain other views (**subviews**), and have knowledge of their parent view (**superview**).
- The order of the view hierarchy determines *compositing order*.
- `UIView` objects automatically propagate messages to their subviews



A view hierarchy, with off-screen objects

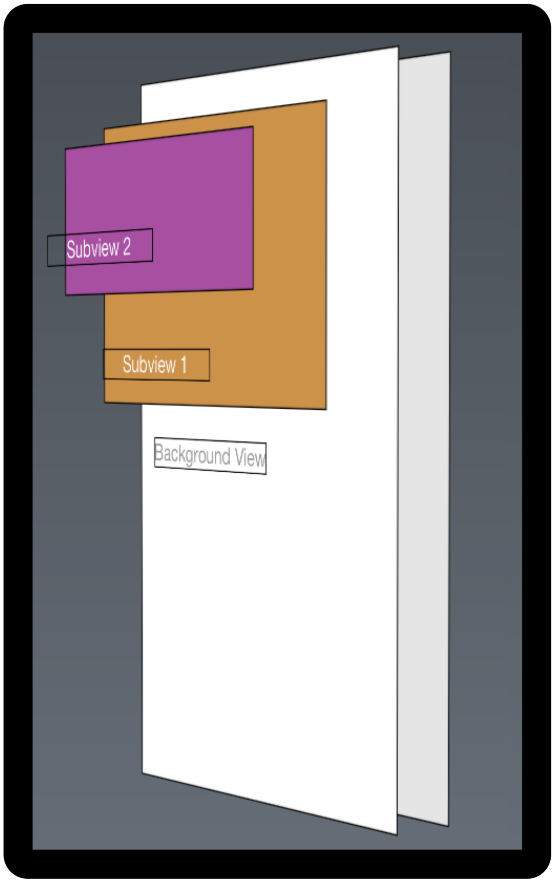
UIResponder

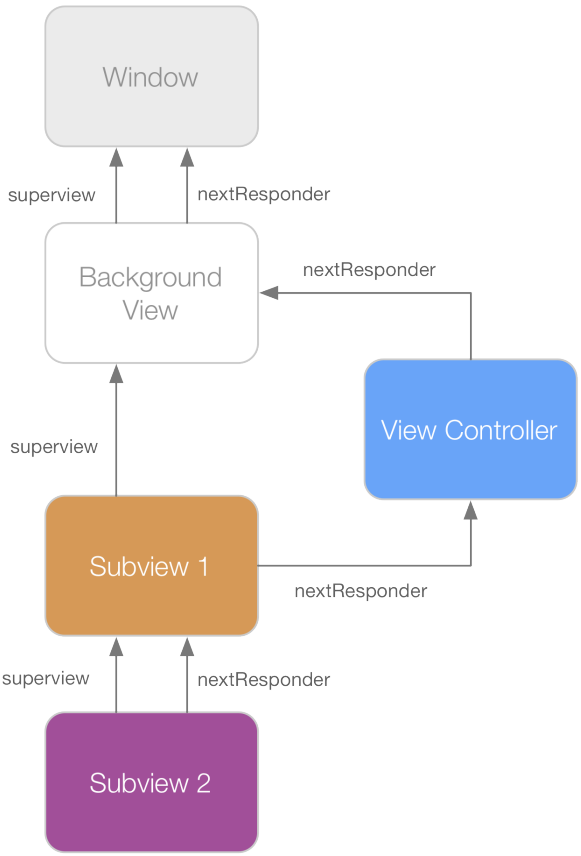
- Abstract superclass for objects that respond to touches and motion events
- Manages input view (keyboard)
- Provides undo manager (`NSUndoManager`)
- Manages **Responder Chain**
- UIResponder event-handling methods simply forward messages to the *next responder*
  - For example, if you add a plain UIView to a CoolView's subviews array, it will forward all `touchesMoved:withEvent:` messages to the CoolView.
- A responder can handle an event phase message and still forward it to the next responder

Responder Chain

- Like **View Hierarchy**, but chained bottom-up instead of top-down
- Event messages note handled by current responder are forwarded to next responder
- Allows other types of responders to be inserted in chain

Responder Chain





UIView

- Subclass of `UIResponder`
- Defines a rectangular area for drawing and extension point for custom drawing (the `drawRect:` method)
- Responsibilities
  - Drawing
  - Animation
  - Responding to events (touches, etc.)

UIView Responsibilities

- Managing subviews (adding, removing, reordering)
- Hit testing
- Drawing and animation
- Laying out subviews
- Autoresizing
- (Plus responsibilities inherited from UIResponder)

The Status Bar

- The status bar is a separate window managed by the framework.
- Transparent background with light or dark opaque content.
- Can be ordered on and off screen.
- Typically, a navigation bar or other non-scrolling content is positioned behind the status bar.
- **Important:** the status bar can double in height (for example to display the in-call status bar).

Auxiliary Window

- Transparent window for alert views and action sheets
- Owned and managed by application object
- Private subclass of `UIWindow`

CHAPTER FIVE

Drawing and Animation

Views and Layers

- Subviews are drawn on top of parent view (**superview**) in the order in which they appear in **subviews** array
- UIView has **layer** property of type `CALayer`
- `CALayer` is like a lightweight `UIView`
- Has **sublayers** property for nested instances
- Very efficient for animation; 100s of layers can be animated simultaneously

Quartz

- iPhone's 2D drawing engine
- Stroking and filling lines, polygons, text
- Drawing and painting
- Transparency, shadows, anti-aliasing
- Shading and color management
- PDF document creation and metadata access

Custom Drawing

- Override `-drawRect:` in a subclass of UIView

```
// Swift
open func draw(_ rect: CGRect)

// Objective-C
- (void)drawRect:(CGRect)rect;
```

  - Typically, use Core Graphics API to do custom drawing
  - For CALayer, override `-drawInContext:`

- **Note:** You never call these methods: they're called automatically by the framework
- If data rendered in a view changes, send the following message:

```
// Swift
open func setNeedsDisplay()

// Objective-C
- (void)setNeedsDisplay;
```

Framework Drawing Behavior

- At the end of each event cycle, framework propagates `-display` message down the view hierarchy
- If a view needs to redraw itself, its -display method will call its own `-drawRect:` method
- You never call either of these methods

Core Animation

- iPhone SDK's animation engine
  - Defines hierarchy of animation layers within a `UIView`
  - Includes flexible layout manager
  - Provides sophisticated 2D and 2.5D animation effects through high-level APIs
- High Performance
  - Animations run in a separate thread
  - Data structure is light enough to permit simultaneous animation of hundreds of layers

CALayer

- Parent view provides event-handling
- Layer provides content, plus geometry, timing, and visual properties
- Each view has a root layer — an instance of `CALayer` — in its `layer` property
- Layers can have sublayers just as views can have subviews
- Sublayers can only be added programmatically

UIView Animation

- `UIView` defines a set of class methods that hide some Core Animation details
- Simplifies animating changes to view properties, such as **frame**, **bounds**, **transform**, etc.
- Use `beginAnimations:context:` and `commitAnimations` to define beginning and end of animation blow (similar to DB transaction)

Animation Block

- Block begins with call to UIView class method, `beginAnimations:context:`
  - Both arguments are optional
- Block ends with call +commitAnimations, which spawns a separate thread to run the animations
- Changes to any *animatable properties* of a view inside the block are animated automatically

```
// Swift
//
UIView.beginAnimations(nil, context: nil)

// Modify animatable properties of one or more views...

UIView.commitAnimations()

// Objective-C
//
[UIView beginAnimations:nil context:NULL];

// Modify animatable properties of one or more views...

[UIView commitAnimations];
```

Animation Example

- An easy way to move a view is to change its center property
- Making the change in an animation block will automatically animate the move

```
CGPoint location = [self center];
location.x += 80.0;
location.y += 240.0;

// Beginning of animation block
[UIView beginAnimations:nil context:NULL];
[UIView setAnimationDuration:2.5]; // Make animation take 2.5
seconds

[self setCenter:location];

[UIView commitAnimations];
// End of animation block
```

CHAPTER SIX

Interface Builder and Nib Files

Interface Builder

- Interface Builder (IB) is a graphical editor for objects.
- Provides a visual interface for creating and configuring objects without writing code.
- IB allows you to instantiate and modify objects dynamically via drag-and-drop.
- Objects are then stored in the file system so they can be reloaded by your app at runtime.

How IB Works

- Dynamically instantiates objects when you:
  - Drag objects from **Library** into editor or sidebar
  - Copy and paste in editor or sidebar
- Sends messages to instances when you:
  - Set properties in **Inspector**
  - Drag lines to make connections
  - Drag objects to add/remove/reorder subviews
- Sends messages to classes and objects dynamically. For example:
  - Sends alloc and `initWithFrame:` to create instances of UIView (and its subclasses)
  - Sends messages like `setBackgorundColor:`, `setFrame:`, etc. when you modify values in Inspector
  - Sends messages like `addSubview:`, `removeFromSuperview:`, etc. in response to drag-and-drop

IB Documents

There are two kinds of IB documents: nib files and storyboards.

Nib Files

- Documents that contains 'freeze-dried' (serialized) objects. May contain off-screen objects in addition to UI components.
- Xcode compiles XIB (XML Interface Builder) documents into binary nib files.
- **ibtool** is a command-line tool that compiles IB documents; can also be used to localize nib files.

Storyboards

- Documents used to define multiple 'scenes' in an application, as well as the transitions (**segues**) from one scene to the next.
- Xcode compiles storyboard XML documents into sets of binary nib files.

IB Attributes

Term	Definition
`IBAction`	Exposes a method as a connection point between user interface elements and app code
`IBOutlet`	Exposes a symbol as a connection point for sending messages from app code to a user interface element.
`IBDesignable`	Lets Interface Builder know that it should render the view directly in the canvas. This allows you to see how your custom views will appear without building and running the app.
`IBInspectable`	Enables creation and access to **user-defined runtime attributes** inside the identity inspector.

The Main Nib File

- App's *main nib file* loaded automatically by application object
- **File's Owner** in main nib must be an instance of `UIApplication`
  - Allows application object to get connected to objects in the nib file when the nib gets loaded

File's Owner

- File's Owner is the object that will load the nib at runtime
- Usually a subclass of `UIViewController`
- However, main nib always loaded by application object (instance of `UIApplication`)

Connecting Objects

- 'Connecting' means setting a property or instance variable of one object to point to another object
- How to connect:
  - Right-click object whose property you want to connect to bring up Heads Up Display
  - Draw line from circle next to the property to the object it should point to

Loading Nibs Programmatically

- Use `NSBundle` class method
  - `mainBundle`
- Returns `NSBundle` object that represents project's top-level directory
  - `pathForResource:ofType:inDirectory:` returns full path name for specified file
  - Draw line from circle next to the property to the object it should point to

Nib File Loading

- Unarchived by subclass of `NSCoder`
- Objects that conform to `NSCoding` protocol receive - `initWithCoder:` during unarchiving; allotter receive `init`
- All objects receive `awakeFromNib` after being unarchived

```swift
// Swift
func loadBallView() {
    let objs = Bundle.main.loadNibNamed("Foo", owner: self, options: [:])
    if (objs == nil) {
        print("Unable to load nib file named Foo.nib")
    }
}
```

```objc
// Objective-C
- (void)loadBallView
{
    NSArray *objs = [[NSBundle mainBundle] loadNibNamed:@"Foo"
                                                  owner:self
                                                options:nil];
    if (objs == nil) {
        NSLog(@"Unable to load nib file named Foo.nib");
    }
}
```

Loading a Nib File

- Use UINib if you want contents cached to memory
  - Subsequent calls to `instantiateWithOwner:options:` won't have to reread nib from file system

```
- (void)loadMyNib
{
    UINib *nib = [UINib nibWithNibName:@"MyNib" bundle:nil];

    // Either do this...
    [nib instantiateWithOwner:self options:nil];
    // (Owner's outlets have now been set.)

    // ...or this...
    NSArray *objs = [nib instantiateWithOwner:self options:nil];
    // ...and do something with objs.
}
```

Storyboards

Storyboards are documents used to define multiple 'scenes' in an application, as well as the transitions (**segues**) from one scene to the next. Xcode compiles storyboard XML documents into sets of binary nib files.

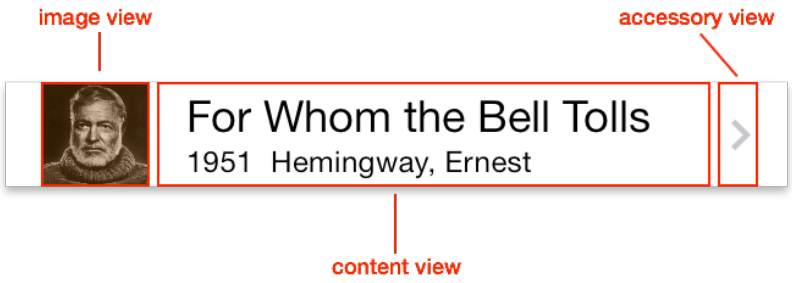
CHAPTER SEVEN

Table Views and Navigation

Table Views

- Present a scrollable list of cells
  - Each cell represents a row of information
  - Table view tracks selected row (or rows if multiple selection enabled)
- Rows can be organized in sections
- Two styles – plain (below, left) and grouped (below, right):

Table View Cells



- Present a single row of information
- Contain built-in image view, and customizable content and accessory views

Table View Protocols

- `UITableViewDataSource` protocol declares methods used by a `UITableView` to request cells dynamically:

```
// Swift
public func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell

// Objective-C
- (UITableViewCell *)tableView:(UITableView *)
    cellForRowAtIndexPath:(NSIndexPath *)indexPath;
```

- `UITableViewDelegate` protocol notifies **delegate** of user interactions

- For example when user selects a row, table view will send:

```
// Swift
optional public func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath)

// Objective-C
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath;
```

UITableViewController

- Adopts both table view protocols:

```
// Swift
open class UITableViewController : UIViewController, UITableViewDelegate,
UITableViewDataSource {
    // ...
}

// Objective-C
@interface UITableViewController : UIViewController
<UITableViewDelegate, UITableViewDataSource>
```

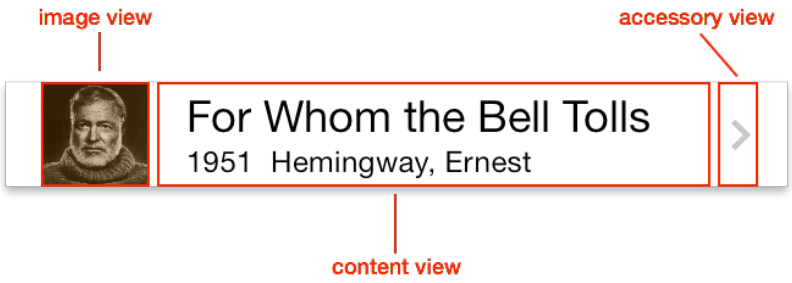
- Provides `tableView` property for strongly typed access to `view` property

```
// Swift
open var tableView: UITableView!

// Objective-C
@property(nonatomic, strong, null_resettable) UITableView *tableView;
```

- By default, a table view's `dataSource` property points to the same object as its `delegate` property, typically the table view's controller.
  - You can change this if you want the data source to be separate from the controller.

Anatomy of a Cell



- **content view** – Freely customizable by adding subviews. Alternatively, you can select one of several pre-de ned styles. **NOTE:** content view may be resized during autorotation and editing

- **image view** – Lazily initialized instance of UIImageView. Read-only, but you can set the image view's image property

- **accessory view** – Freely customizable. Alternatively, you can select one of several pre-de ned styles.

Refreshing Cell Data

- View controller is responsible for notifying its table view(s) to refresh cells when underlying model data changes

- Send reloadData to table view to reload all visible rows. For finer-grained control, use more specific methods, for example:

```
- (void)insertRowsAtIndexPaths:(NSArray *)indexPaths
              withRowAnimation:(UITableViewRowAnimation)animation;
- (void)deleteRowsAtIndexPaths:(NSArray *)indexPaths
              withRowAnimation:(UITableViewRowAnimation)animation;
- (void)reloadRowsAtIndexPaths:(NSArray *)indexPaths
              withRowAnimation:(UITableViewRowAnimation)animation;
```

- For multiple simultaneous insertions and/or deletions, surround the above (and similar) with calls to the following methods:

```
- (void)beginUpdates;
- (void)endUpdates;
```

Working with Static Cells

- Storyboards are ideal for developing static table views because nearly all the work can be done in the Storyboard Editor.

- If you're working with a nib file, you can create UITableViewCell instances in the nib and connect them to outlets of the File's Owner
  - However, requires writing code in cellForRowAtIndexPath: to figure out which cell gets returned for which index path

Using the Grouped Style

- Grouped table view style is commonly used when displaying detailed information

- When you need a table view to display more than one section (the default), implement `numberOfSectionsInTableView:` to return the number of sections you prefer

- Override `viewWillAppear:` to set the controller's title property if you want the title to change based on selection. (Remember, title is automatically presented by `UINavigationController` and `UITabBarController`, and the like.)

Adding an Edit Button

- Set the `BarButtonItem` on the right side of the Nav Bar

```
// Return an Edit|Done button that can be used as
// a navigation item's custom view.
// Default action toggles the editing state with animation.

// Swift
override func viewDidLoad() {
    super.viewDidLoad()
    self.navigationItem.rightBarButtonItem = self.editButtonItem
}

// Objective-C
- (void)viewDidLoad {
    [super viewDidLoad];
    self.navigationItem.rightBarButtonItem = self.editButtonItem;
}
```

CHAPTER EIGHT

Navigation and Controller Layer

UINavigationController

- Manages navigation bar, content view, and tool bar.
- Manages a stack of view controllers, adding top view controller's view as subview of its own content view.

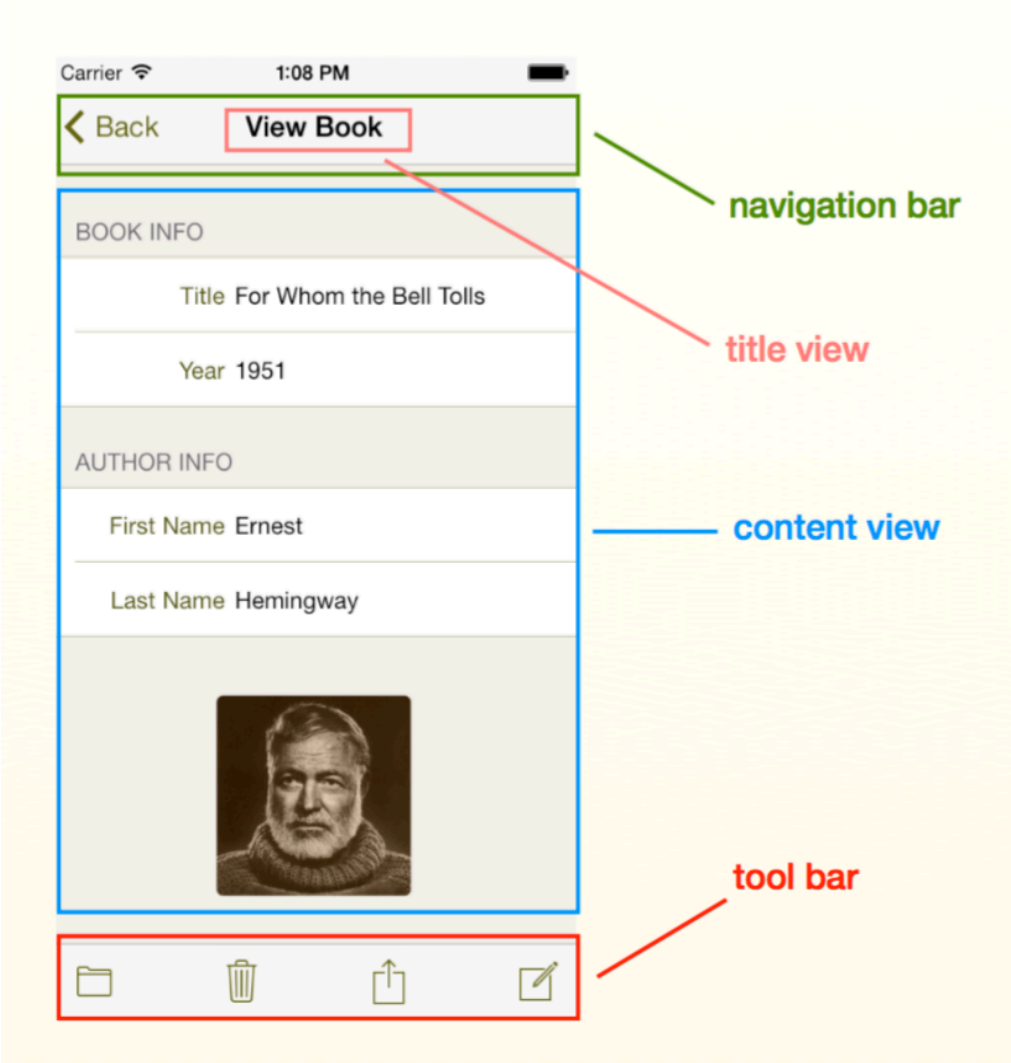
```
- (id)initWithRootViewController:(UIViewController *)rootViewController;

- (void)pushViewController:(UIViewController *) viewController
                  animated:(BOOL)animated;

- (UIViewController *)popViewControllerAnimated:(BOOL)animated;
```

A Nav Controller's Views

- UINavigationController has several configurable subviews:
  - **navigation bar** — Presents the top VC's *navigation items.*
  - **title view** — Presents top VC's `title` property
  - **content view** — Presents the top VC's view
  - **tool bar** — Presents the top VC's *toolbar items*



Navigation Bar

- Translucent; blurs content that scrolls behind it
- Typically managed by a navigation controller
- Provides views for the following:
  - Bar button items
  - Back button
  - Title
  - Prompt

Pushing a VC

- When you push a VC on the nav controller's stack, the nav controller:
  - Swaps the current VC's navigation items into the nav bar
  - Swaps the current VC's view into the content view
  - Swaps toolbar items (if toolbar is visible)

Managing the Stack

- `UINavigationController` methods for managing the stack:

```
/* Pushing and popping view controllers */

- (void)pushViewController:(UIViewController *)viewController
                  animated:(BOOL)animated;

- (UIViewController *)popViewControllerAnimated:(BOOL)animated;

- (NSArray *)popToViewController:(UIViewController *)viewController
                        animated:(BOOL)animated;

- (NSArray *)popToRootViewControllerAnimated:(BOOL)animated;

/*  Accessing the top view controller  */

@property(nonatomic,readonly,retain) UIViewController *topViewController;
// Returns modal view controller if it exists. Otherwise the top view
controller.

@property(nonatomic,readonly,retain) UIViewController *visibleViewController;

/*  Accessing the viewControllers array  */

@property(nonatomic,copy) NSArray *viewControllers;

- (void)setViewControllers:(NSArray *)viewControllers
                  animated:(BOOL)animated
```

Segues

- Segues define relationships between view controllers, or scenes, in a storyboard.
- There are two fundamental types:
  - Relationship (usually parent – child)
  - Transition
- Control drag from an element of one scene (source) to another scene (destination) to define a segue.

View Controller Presentation

- A view controller can temporarily present another view controller on top of its own content area.
  - This is sometimes referred to as *modal* presentation.
  - When the user completes their task, the presenting view controller can dismiss the presented view controller.
- Segues that trigger view controller presentation:
  - With size classes disabled: **Modal**
  - With size classes enabled: **Show Detail**, **Present Modally**, **Present as Popover**

Unwind Segues

- Beware of loops in a storyboard! These will almost always yield undesirable and potentially dangerous behavior.
- Unwind segue destinations are defined by action methods with a custom method signature.
  - Note that the methods can be empty; i.e., they don't have to contain any code.
  - Unwind segues dismiss intervening view controllers between the source and destination.

Tool Bars

- Generally appear at the bottom of the screen.
- Contain bar buttons and spacing items.
  - May also contain other types of views.
  - Note that IB doesn't currently seem to support adding custom views to navigation controller toolbars.
- Toolbar items perform action relevant to the scene's content, e.g., play/pause, next/previous, sharing, deleting, etc.
- Should **NOT** be used for navigation.

CHAPTER NINE

Adaptive Interfaces

Designing Adaptive Interfaces

- An adaptive layout dynamically adjusts content to make the best use of available space.
- Combines the use of **auto layout** and **size classes** to dynamically reposition content as necessary.
- This can be particularly important for apps that need to support multitasking well.

Auto Layout

- Auto layout is difficult, but powerful.
- Works by defining sets of *constraints* on views to specify visual relationships for the auto layout engine.
- When auto layout is enabled, some views have an *intrinsic size* — the smallest size needed to fully display their content.
  - You can adjust their *content hugging* and *content compression resistance* priorities by specifying integer values between 0 (lowest) and 1,000 (highest).
- You can also specify constraints for:
  - Width and height (including equal widths/heights)
  - Aspect ratio
  - Edge alignment
  - Horizontal and vertical centering
- You must specify sufficient constraints for the layout engine to unambiguously size and position the view hierarchy.

Size Classes

- Size classes are a way of abstracting device screen sizes and orientations.
- There are two size classes: Compact and Regular, for both vertical and horizontal size.
  - Allows you to specify different constraints for when a view is, for example, horizontally compact, than when it's at the regular horizontal size.
- UI *traits* define coarse-grained layout differences.
  - Horizontal size class
  - Vertical size class
  - Display scale
- Traits can change dynamically, triggered by:
  - Device rotation
  - Changes to container view
  - Application logic

Default Size Classes

Device	Portrait	Landscape
iPad (all) iPad Mini	Vertical: **Regular** Horizontal: **Regular**	Vertical: **Regular** Horizontal: **Regular**
iPhone 6 Plus iPhone 6S Plus	Vertical: **Regular** Horizontal: **Compact**	Vertical: **Compact** Horizontal: **Regular**
iPhone 6 iPhone 6S iPhone 5s iPhone 5c iPhone 5 iPhone 4s	Vertical: **Regular** Horizontal: **Compact**	Vertical: **Compact** Horizontal: **Compact**

Managing Size Classes in IB

- Use the size class viewing control at the bottom of the editor to pick size classes for the current selection.
- Use the installation control at the bottom of the Attributes Inspector to add and remove configurations.
- The Attributes Inspector will display a plus next to any properties that can vary by size class.
    - Click the plus to add a setting for a different size class combination.

Preview Assistant

- Open an assistant editor for the storyboard or nib file you're currently editing.
- In the assistant editor's jump bar, click the first item in the path (ordinarily labelled **Manual**).
- In the popup menu, select **Preview**.
- Click the plus button in the lower left corner to add previews for additional form factors.
- To remove the preview for a given form factor, select it and hit the delete key.

Stack Views

- Simplifies layout of views in rows and columns.
- Manages a list of *arranged subviews*, allowing you to control axis, distribution, alignment, and spacing.
- You specify constraints for the stack view, but not for its content.
- You can nest stack views, allowing you to build complex layouts with very few constraints.
- Stack views adjust their layout dynamically whenever their arranged subviews list is modified, or when the **hidden** property of any of their arranged subviews changes.

CHAPTER TEN

Persistence and Web Services

Types of Persistence

- File-based (plist, archives, and various document types)
- Preferences (with **NSUserDefaults**)
- Relational database (C and Objective-C APIs)
- PDF (natively supported by Quartz)
- Web-based

File-Based Persistence

- Plist (property list) files
  - Can be stored as text or binary
- Archive files
  - Objects encoded (serialized) to binary files.
- Other document types
  - Well known formats such as CSV (comma-separated values)
  - Vendor or developer defined formats

Plist Persistence

- Three formats: XML, text (similar to JSON), and binary.
- Arguably the easiest, lowest cost, though there can be some minor pitfalls
- Great for quick, throwaway persistence solution
  - Allows you to create and edit data in easy-to-read text files
  - Can be used to populate object graphs in just a few simple lines of code
  - Object graphs can be written just as easily
- Can be used as a long-term solution for relatively simple and/or small data sets

Archive Files

- Conceptually, allows objects to save themselves to — and restore themselves from — a file.
- Can be useful for certain kinds of documents (for example, imagine saving a diagram created in an iPad app by saving the current state of its object model).
- Easy to implement, and relatively fool-proof when using **NSKeyedArchiver and NSKeyedUnarchiver**.
  - However, care must be taken to manage compatibility with earlier versions as object model changes.
- Nib files are an example of archive files that are contained in nearly every app.

NSURLSession

- NSURLSession and related classes provide an API for downloading content via HTTP/HTTPS.
- Provides set of delegate methods for supporting authentication and background downloads
- Provides status and progress properties
- Highly asynchronous

<<<<>>>>