Advanced 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: 3.1
Last Update: 11/26/2017

Classroom materials for an advanced-level course that provides an accelerated learning environment for students who have roughly 6 months to a year or more of experience on the iOS platform. Includes comprehensive lab exercise instructions and solution code.

Copyright Notice

Advanced iOS Development

STUDENT GUIDE

SECTION 1
Introduction

About The Author

Jonathan Lehr
Co-Founder and VP Training, About Objects

Background:

- Developed Objective-C/NeXTSTEP enterprise apps for Fannie Mae and telecom giant MCI in the 90s.
- Began developing and teaching Objective-C and AppKit classes for enterprise developers in 1992.
- Has been developing and delivering iOS training curriculum since mid-2009.

About You

Things I'd like to know…

- How long have you been working with iOS?
- What was your main previous language?
- What topics are of most interest?
- What's your favorite TV show? (Just kidding.)
- Plus, anything you can share about your current or upcoming projects.

SECTION 2

Xcode

Workspaces

- Containers for grouping related Xcode projects
  - Share a common index for symbols and files
  - Share common build products directories
- For example, a workspace might contain:
  - An iPhone app project
  - An iPad app project
  - One or more Command Line Tool projects
  - One or more Static Library projects

Library Projects

- iOS currently supports only static libraries.
- Library projects often provide common resources in additional to executable code.
  - Add **Bundle** targets to package common resources for reuse.
- Use **Aggregate** targets to build custom products, e.g., bundles.

Static libraries help enforce decoupling between layers, improve testability, and make it much easier to reuse common code across multiple projects.

A static library can be accompanied by one or more resource bundles if it provides common resources such as storyboards, nib files, core data models, images, and so forth.

Lab: Library Project

- Create a **ReadingList** workspace in Xcode.
- Add a **Library** project named **ReadingListModel** to the workspace.
  - Add a **Unit Test** target to the **ReadingListModel** project.
- Add a simple **Person** class, implement a couple of properties, and create an XCUnit test case to create and manipulate instances of **Person**.

Configuring Targets and Schemes

- A build **target** lets you define a product that has its own custom
  - build settings
  - build phases
  - dependencies
- A build **scheme** lets you specify
  - one or more targets
  - a build configuration (e.g., **Debug**, **Release**)
  - environment variables and command line parameters

Adding Subprojects

- Drag the library's .**xcodeproj** into the target project (we'll assume it's an app project from here on in).
    - Usually works best if you drag from a Finder window rather than an Xcode window.
- Add the library's products to the app's **Target Dependencies** build phase.
- Configure the app project's header path so that the library's headers are visible.

- Add the static library product to **Link Binary With Libraries** build phase.
- Add other resources to the **Copy Bundle Resources** phase.
- If necessary, configure **Other Linker Flags** in **Build Settings**:

```
-all_load
-ObjC
```

Documentation

- Xcode 5 can present formatted code comments in Quick Help.
  - Can be formatted several ways:

```
/**
 * Uses the data source's built-in fetchedResultsController
 * to perform a fetch.
 * @param error If an error occurs, upon return contains an
 * instance of NSError describing the error.
 * @return `YES` if the attempted fetch was successful;
 * otherwise `NO`.
 */
- (BOOL)fetch:(NSError *__autoreleasing *)error
{
    ...
}
```

- The following syntaxes can be used interchangeably:

```
/**
 Multi-line comment.
 */

/*!
 Multi-line comment.
 */

/// Single line comment.
```

Appledoc

- Open source tool for generating docsets from code comments.
- Automatically installs docset where Xcode installs its other docsets
- Makes docset visible in Xcode's documentation browser
  - also visible in tooltips

XCUnit

- Xcode's built-in unit testing framework.
- Fork of OCUnit open source project, which was essentially a clone of JUnit.
- Xcode integration provides flexible management and running of unit tests.
- Subclass XCTestCase, and implement test methods with a **-(void)testXxx** signature.
- C macros for assertions, for example:

```
#define XCTAssertNil(a1, description, ...)
#define XCTAssertNotNil(a1, description, ...)

#define XCTAssertTrue(expression, description, ...)
#define XCTAssertFalse(expression, description, ...)

#define XCTAssertEqualObjects(a1, a2, description, ...)
#define XCTAssertEquals(a1, a2, description, ...)
```

Tip: Use **[NSBundle bundleForClass:[self class]]** to access bundle resources.

Objective-C and Foundation

- Inheritance
- Dispatch Mechanism
- Initializers
- Protocols
- Dynamic Checking
- Copying Pitfalls
- Categories
- Arrays and Dictionaries
- Property Lists
- Key-Value Coding

Instance Variable Inheritance

Instance data structures are composited at compile time by combining the inherited instance variable declarations into a single **struct**, conceptually like those shown in the following table.

How Compiler Translates ivar Declarations (Conceptual)

Declaration	Compiler-generated `struct`

`@interface NSObject <NSObject>`
`{`
`  Class isa;`
`}`
	`struct NSObject {`
	`  Class isa;`
	`};`

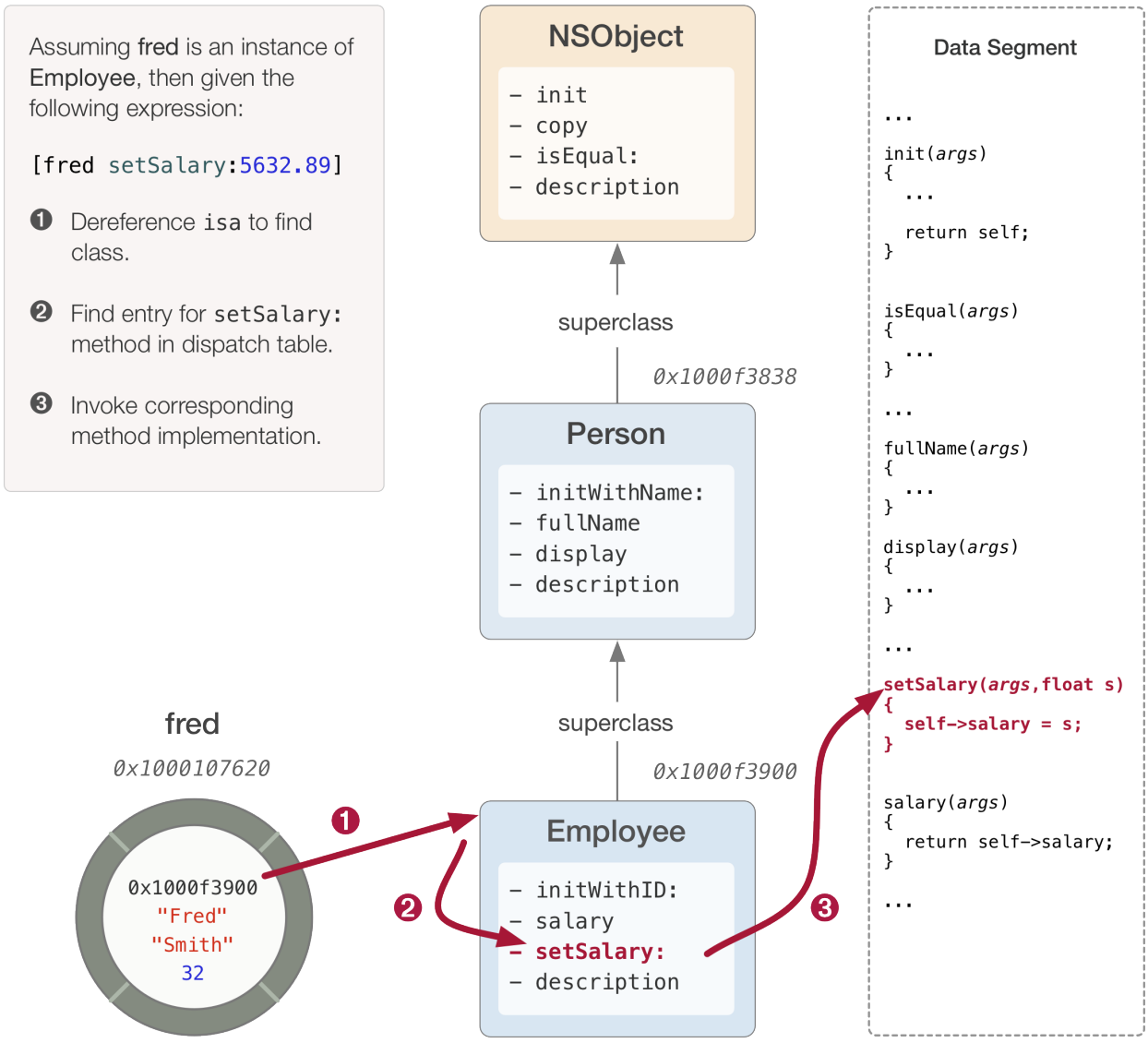
`@interface Pet : NSObject`
`{`
`  NSString *_name;`
`}`
	`struct Pet {`
	`  Class isa;`
	`  NSString *_name;`
	`};`

`@interface Dog : Pet`
`{`
`  NSString *_breed;`
`}`
	`struct Dog {`
	`  Class isa;`
	`  NSString *_name;`
	`  NSString *_breed;`
	`};`

Message Dispatching

Messages are dispatched dynamically by the Objective-C runtime system, as shown in the figure below. This is often referred to as *dynamic binding*.

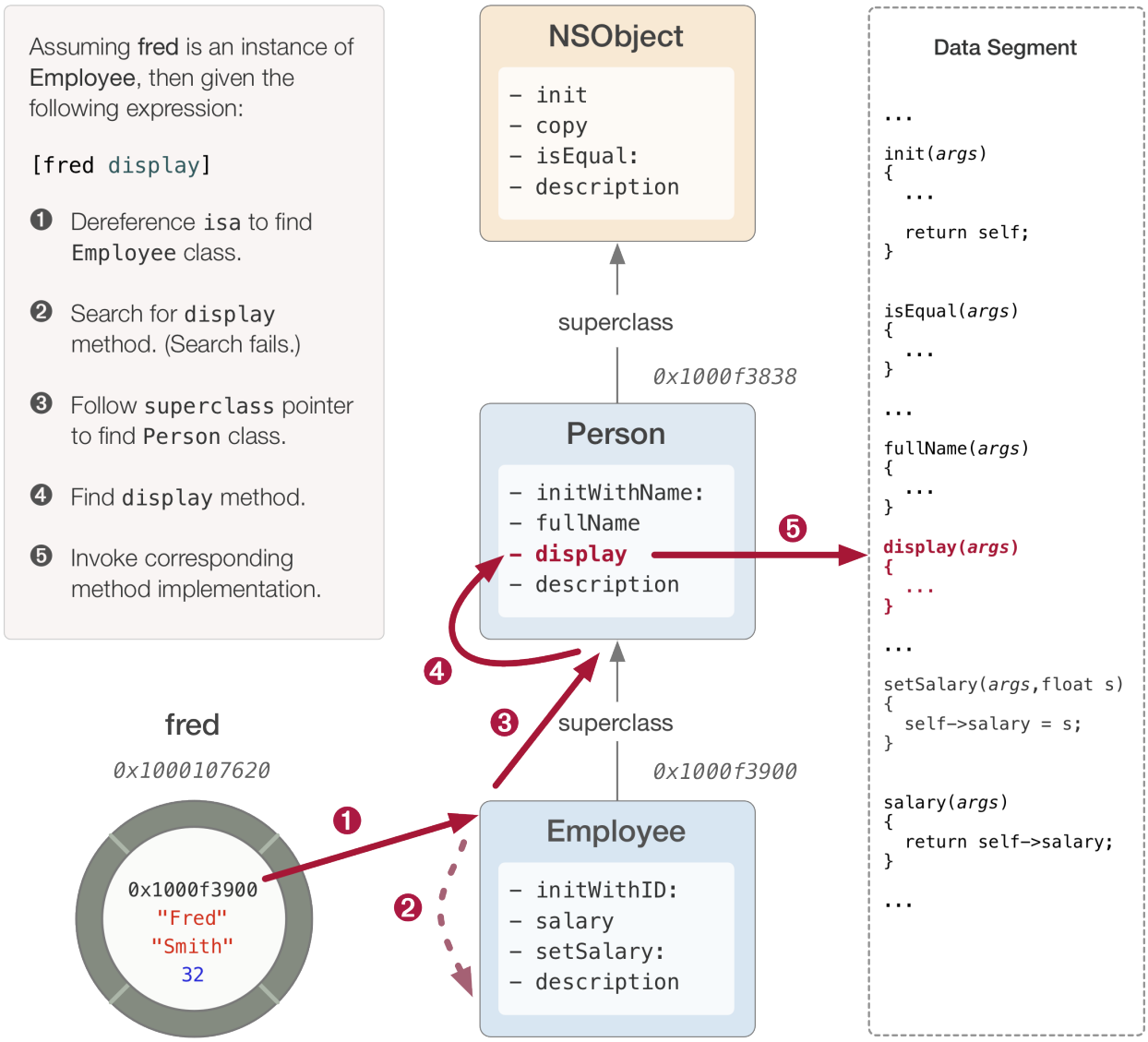
Dispatching a Message



Method Inheritance

Method inheritance is implemented by the Objective-C runtime system's message dispatching mechanism, as shown in the figure below.

Dispatching an Inherited Method



Initializer Methods

- Must do as follows:
  - Call its superclass's *designated initializer*.
  - Replace the current value of `self` with the value returned by the superclass initializer.
  - If initialization fails, must return `nil`.
  - Otherwise, must return the object that was initialized (ordinarily, `self`).
- Designated initializer
  - Generally the `init...` method that takes the most arguments.
  - Performs actual initialization; other `init...` methods in the same class directly or indirectly call this one.
  - Should be called by subclass designated initializers.

Initializer Example

```
- (id)initWithName:(NSString *)aName
{
    // Capture value returned by super init.
    self = [super init];

    // If return value is nil, object has already been
    // deallocated, so short-circuit initialization.
    if (self == nil)
        return nil;

    // Perform custom initialization.
    _name = [aName copy];

    // Return the initialized object. Note that in theory
    // this can an object other than self.
    return self;
}
```

Protocols

Objective-C protocols are similar in nature to Java interfaces.

- Protocols declare (but don't implement) a group of methods that specify an **API contract**.
- By default, classes are required to provide implementations of methods declared in protocols they adopt.
  - However, protocols can have optional sections.
  - Compiler doesn't enforce implementation of the optional protocol methods.

Declaring a Protocol

A protocol can be declared in a separate **.h** file, or incorporated in the **.h** file of a related class.

- Note that a protocol can adopt other protocols.

Declaring a Protocol

```
// Protocol methods are required by default.
@protocol Vocalizing

- (void)growl;
- (void)snarl;

@end
```

Declaring Optional Protocol Methods

```
// All vocalizing objects growl and snarl; some may bark and/or whine.
@protocol Vocalizing

- (void)growl;
- (void)snarl;

@optional
- (void)bark;
- (void)whine;

@end
```

Adopting a Protocol

- A class can adopt one or more protocols by listing them in angle brackets in the class declaration.

```
// Declaration can specify a list of protocols.

@interface Widget : NSObject <NSCoding, NSCopying>
```

- For example, the **NSCoding** protocol declares methods for serializing and deserializing instances.
  - Any class that adopts **NSCoding** must implement the two methods it declares.

```
// Given the following declaration...
//
@interface Person : NSObject <NSCoding>
// ...

// ...the compiler will warn if the Person class
// doesn't implement both of the following methods.
//
@implementation Person

- (id)initWithCoder:(NSCoder *)aDecoder
{
    // ...
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
    // ...
}
```

The NSObject Protocol

Declares the minimum set of behaviors needed for anything of type `id` to behave correctly with Foundation objects.

- Adopted by **NSObject** and **NSProxy** classes.
- Methods were all originally declared in **NSObject** class; later moved to protocol so declarations could be shared.
- Includes essential introspection methods.

Dynamic Checking

The **NSObject** protocol includes several methods that are frequently used to perform dynamic checking of such things as an object's:

- Membership in a given class hierarchy.
- Ability to respond to a particular message.
- Protocol conformance.

Dynamic Checking Examples

- Checking class hierarchy membership:

```
NSMutableArray *people = [NSMutableArray array];
NSArray *objs = @[obj1, obj2, obj3];

for (id currObj in objs)
{
    if ([currObj isKindOfClass:[Person class]]) {
        [people addObject:currObj];
    }
}
```

- Checking ability to respond to a given selector:

```
for (id currObj in objs)
{
    if ([currObj respondsToSelector:@selector(fullName)]) {
        NSLog(@"%@", [currObj fullName]);
    }
}
```

- Checking protocol conformance:

```
for (id currObj in objs)
{
    if ([currObj conformsToProtocol:@protocol(Vocalizing)]) {
        [currObj snarl];
        [currObj growl];

        if ([currObj respondsToSelector:@selector(bark)]) {
            [currObj bark];
        }
    }
}
```

Protocols as Type Qualifiers

A *protocol list* can be used to qualify an object type declaration.

- For variable assignments and argument passing, compiler checks **protocol conformance** as well as type.

```
id<Vocalizing> obj1 = [[Dog alloc] init]; // Okay
id<Vocalizing> obj2 = @"Hello";           // Warning

Person<Vocalizing> *p1 = [[Employee alloc] init]; // Okay
Person<Vocalizing> *p2 = [[Person alloc] init];   // Warning
```

- When type-checking receiver of a message:
  - **If receiver is dynamically typed**: compiler assumes receiver has *only* the methods declared in the protocols(s).
  - **If receiver is statically typed**: compiler assumes receiver has additional methods declared in the protocol(s).

Type Qualifier Examples

Example: Dynamic Type Qualified with Protocol

```
SEL mySelector = @selector(growl);

id obj1 = [[Dog alloc] init];
if ([obj1 respondsToSelector:mySelector])  // Okay.
{
    // Do stuff...
}

// Here, qualifier narrows declared type.
//
id<Vocalizing> obj2 = [[Dog alloc] init];  // Too narrow.
if ([obj2 respondsToSelector:mySelector])  // Compile error.
{
    // Do stuff...
}

id<Vocalizing, NSObject> obj3 = [[Dog alloc] init];
if ([obj3 respondsToSelector:mySelector])  // Okay.
{
    // Do stuff...
}
```

Example: Static Type Qualified with Protocol

```
// Given Employee class declared as follows...

@interface Employee : Person <Vocalizing>

// ...

Person *emp1 = [[Employee alloc] init];  // Too narrow.
[emp1 growl];                            // Compile error.

// Here, qualifier widens declared type.
//
Person<Vocalizing> *emp2 = [[Employee alloc] init];
[emp1 growl];                            // Okay
```

Copying

- NSObject declares the following instance methods:

```
- (id)copy;        // Returns immutable copy.
- (id)mutableCopy; // Returns mutable copy.
```

- Above are implemented as cover methods for methods declared in protocols **that are not adopted by** NSObject.

```
@protocol NSCopying

- (id)copyWithZone:(NSZone *)zone;

@end

@protocol NSMutableCopying

- (id)mutableCopyWithZone:(NSZone *)zone;

@end
```

Copying Pitfalls

- NSObject implements `copy` and `mutableCopy` as cover methods for `copyWithZone:` and `mutableCopyWithZone:`.
  - However, doesn't implement either protocol.
  - Therefore, sending `copy` or `mutableCopy` triggers unrecognized selector exception.
- While most Foundation classes adopt **NSCopying**, only class clusters adopt **NSMutableCopying**.

WARNING: It's unsafe to send a **copy** message to an object of unknown type unless you first check for protocol conformance.

Categories

- A category allows additional methods to be declared and/or implemented on an existing class.
  - Allows splitting lengthy class implementations across multiple files.
  - Allows grouping of method declarations and implementations based on their cohesiveness, regardless of which classes they belong to.
- Also allows adding methods to framework classes.
- Potential pitfall:
  - A category's method implementations are added to the target class **at run time**, when category is loaded.
  - If method name matches existing method in the target class, **original implementation will be replaced**.
- To avoid pitfall, prefix method names with project prefix.

Category Example

Adding a category to NSArray

```
// Declaring category.
// Typically in NSArray+XYZAdditions.h, but could be anywhere.

@interface NSArray (XYZAdditions)

- (NSArray *)XYZ_reversedArray;    // Note use of XYZ_ prefix.

@end


// Implementing category.
// Typically in NSArray+XYZAdditions.m, but could be in any .m file.

@implementation NSArray (XYZAdditions)

- (NSArray *)XYZ_reversedArray
{
    NSMutableArray *newArray = [NSMutableArray arrayWithCapacity:
                                [self count]];

    for (id currObj in [self reverseObjectEnumerator]) {
        [newArray addObject:currObj];
    }

    return newArray;
}

@end


// Using category.

#import "NSArray+XYZAdditions.h"

- (void)doStuffBackwardsWithObjects:(NSArray *)someObjs
{
    NSArray *reversedObjs = [someObjs XYZ_reversedArray];

    // Do stuff with reversedObjs...
}
```

NSArray Literals

Modern Objective-C provides literal expressions for creating and initializing instances of NSArray.

- Also supports array subscript notation.
- Compiler translates literal syntax into method invocations.

Example: NSArray Literal Syntax

```
//
// Compiler translates into code that calls arrayWithObjects:count:.
//
NSArray *objs = @[ @"Foo", @42, @"Bar" ];

for (int i = 0; i < [objs count]; i++)
{
    //
    // Translates subscript into call to objectAtIndexedSubscript:.
    //
    NSLog(@"%@", objs[i]);
}
```

Example: Mutable Arrays

```
//
// Using array literal as argument to a factory method.
//
NSMutableArray *objs2 = [NSMutableArray arrayWithArray:@[@1, @2, @3]];

// Modifying array elements.
objs2[0] = @42;

//
// Another technique for creating a mutable array from a literal.
//
// Note: object copying behavior explored in detail in next section.
//
NSMutableArray *objs3 = [@[@1, @2, @3] mutableCopy];
```

Enumerating Arrays

Foundation declares the **NSFastEnumeration** protocol for objects that support enumeration (iteration) behavior.

- All Foundation collection classes conform to **NSFastEnumeration**.
- Compiler translates `for ... in` syntax into the necessary calls to methods declared in the protocol.
- Custom classes can support fast enumeration by adopting (implementing) the protocol.

Example: Fast Enumeration

```
NSArray *objs = @[ @"Foo", @42, @"Bar" ];
//
// Enumerating an array.
//
for (id currObj in objs)
{
    NSLog(@"%@", currObj);
}

//
// Assuming p1, p2, and p3 are Person objects...
//
NSArray *people = @[ p1, p2, p3 ];
//
// Using a strongly typed loop variable.
//
for (Person *currPerson in people)
{
    NSLog(@"%@", [currPerson firstName]);
}
```

NSDictionary

An **NSDictionary** is a keyed collection of objects *of any type* (i.e. type `id`).

- Each entry in a dictionary is a *key-value pair*.
- Dictionaries can only contain objects; can't contain:
  - non-object types, e.g., **int**, **float**, **struct**.
  - **nil** values. Instead, use singleton instance of **NSNull** to represent **nil** values.
- Instances are immutable.
  - Mutable subclass: **NSMutableDictionary**.

NSDictionary Examples

- Inserting objects in a mutable dictionary.

```
NSMutableDictionary *info = [NSMutableDictionary dictionary];

[info setObject:@"Fred" forKey:@"firstName"];
[info setObject:@32 forKey:@"age"];
```

- Retrieving objects from a dictionary.

```
NSLog(@"name: %@", [info objectForKey:@"firstName"]);
NSLog(@"age: %@",  [info objectForKey:@"age"]);
```

- NSDictionary literal expression.

```
NSDictionary *info = @{ @"name" : @"Fred",
                        @"age"  : @32 };
```

- Enumerating a dictionary's keys.

```
for (NSString *key in info)
{
    NSLog(@"key: %@, value: %@", key, [info objectForKey:key]);
}
```

Property List Files

Dictionaries, arrays, strings, and several other types of Foundation objects can serialize and deserialize themselves to/from files using one of several **property list** formats.

- Allows nesting of dictionaries and arrays.
    - Supports modeling of data hierarchies.
- Default property list format is XML.
- Alternative formats:
    - Binary (faster, but opaque).
    - Text (more human-readable, but supports fewer types, e.g., no support for NSNumber, NSDate, NSData.)

Property List Examples

Writing and reading a dictionary to/from a .plist file:

```
// Given the following...

NSDictionary *info = @{ @"name"   : @"Fred W. Smith",
                        @"age"    : @37,
                        @"phones" : @{@"Work" : @"703-123-4567",
                                      @"Home" : @"301-987-6543"},
                        @"kids"   : @[@"Bob", @"Alice", @"Jill"] };


// Write to plist file.

[info writeToFile:@"/tmp/info.plist" atomically:YES];


// Initialize new instance with contents of plist.

NSDictionary *newInfo = [NSDictionary dictionaryWithContentsOfFile:
                         @"/tmp/info.plist"];
```

Key-Value Coding

General mechanism that adds dictionary-like semantics to **NSObject**.

- **NSKeyValueCoding** category (informal protocol) API built on the following two 'primitive' methods:

```
- (id)valueForKey:(NSString *)key;
```

```
- (void)setValue:(id)value forKey:(NSString *)key;
```

- Allows all objects to be treated with dictionary semantics.
- Used heavily in sophisticated mechanisms throughout all of Apple's frameworks and tools.

SECTION 3
Core Data

Overview

- Core Data is an object-relational mapping (ORM) framework for Cocoa and Cocoa touch.
- Primarily a mechanism for fetching and storing object graphs in database tables.
- However, object graph management features can be highly useful even when data is not persisted.
- Supports the following persistent storage types on iOS:
  - SQLite relational database (incremental)
  - binary (atomic)
  - in-memory
  - custom (atomic or incremental)

Core Data Features

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

CHAPTER 1
SQLite

SQLite Overview

- SQLite is a high performance, ACID-compliant database.
- It's a C library linked directly into your app, rather than a server running as a separate process.
- SQLite store files are typically created and managed in the **Library/Application Support** directory in the app sandbox container directory.
- Note: you can locate the **Library** directory by printing the value of the following expression:

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

Commands

- To view the contents of a SQLite file from the command line:

```
$ sqlite3 myFile.sqlite
sqlite>
```

- You will now be in an interactive SQLite terminal session.

Commonly Used SQLite Commands

Command	Description
`.help`	List commands
`.databases`	List names of databases
`.tables`	List names of tables
`.schema <tablename>`	List CREATE statements
`.headers on|off`	Toggle table headers on or off

Example

Working with SQLite

```
$ sqlite3 foo.sqlite
```

Viewing Tables and Schemas

```
sqlite> .tables
ZEVENT        Z_METADATA    Z_PRIMARYKEY

sqlite> .schema zevent
CREATE TABLE ZEVENT ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER,
ZTIMESTAMP TIMESTAMP );
sqlite>
```

Viewing Data

```
sqlite> .headers on
sqlite> select * from zevent;
Z_PK  Z_ENT          Z_OP  ZTIM
----  -------------  ----  ----
1     1              1     376079792.93203
sqlite>
```

Explaining a Query Plan

```
sqlite> .explain query plan select * from zevent;
sele  order          from  deta
----  -------------  ----  ----
0     0              0     SCAN TABLE zevent (~1000000 rows)
sqlite>
```

Quitting

```
sqlite> .quit
$
```

CHAPTER 2

Core Data Basics

Main Components

Core Data provides:

- A base class for model objects, **NSManagedObject**
- Model files (**.xcdatamodel**) for mapping managed objects and their properties to database tables and columns
- Controller classes for:
  - Setting up and tearing down persistent stores
  - Coordinating multiple stores
  - Performing persistence operations
  - Managing model objects in transactional contexts
  - Managing fetch results

Managed Objects

- Core Data automatically fetches and saves instances of `NSManagedObject` and its subclasses.
- `NSManagedObject` instances are capable of:
  - Providing storage and validation for persistent values (**attributes**).
  - Maintaining associations between objects (**relationships**).
- Changes to one or more managed objects can be saved by simply sending a **save** message to a Core Data controller.

CHAPTER 3

Managed Object Model

Managed Object Model

- A managed object model contains *entities* that define mappings between:
  - Managed objects and database tables
  - Managed object properties and database columns
- A managed object model is stored in an .xcdatamodel file, an XML file containing a list of **entity descriptions**.
  - Compiled into a binary `.mom` file during builds.
  - The contents of a `.mom` are encapsulated at runtime in instances of `NSManagedObjectModel`.
  - The model's entity descriptions are encapsulated at runtime in instances of `NSEntityDescription`.

Entity Descriptions

- An entity description contains:
  - A name, and the name of a managed object class.
  - A list of property descriptions, divided into **attributes** and **relationships** (at runtime, instances of `NSAttributeDescription` and, `NRelationshipDescription` respectively).
- Entity descriptions can also define behaviors:
  - Validation rules (e.g. min value/max value)
  - Delete rules
  - Relationship cardinality and navigability

Model Versions

- Xcode stores model files in an `.xcmodeld` directory
  - Individual files represents model versions.
  - Versions used to support database migrations.
  - During builds, XML files are compiled to binary, and stored as `.mom` files in a `.momd` directory.
- Xcode's ***Model Editor*** is a graphical tool for editing model files.

Demo: Creating a Model

Lab: Model Editor

The instructor will guide you through creating a data model using the following steps:

1. Add a new file using the Data Model template. Name the new file **EbooksModel** and open it in the model editor.
2. Add an entity named **Book**. In the Inspector, set Codegen to **Class Definition**. Then configure the following attributes:
    2.1. **iTunesId** of type **Integer 32**.
    2.2. **rating** of type **Integer 16**.
    2.3. **title** of type **String**.
    2.4. **year** of type **Integer 16**.
3. Add an entity named **Author**. In the Inspector, set Codegen to **Class Definition**. Then configure the following attributes:
    3.1. **iTunesId** of type **Integer 32**.
    3.2. **name** of type **String**.
    3.3. **rating** of type **Integer 16**.
4. Configure the following relationships:
    4.1. Add a **books** relationship to **Author** with **Book** as the destination entity. In the Relationship Inspector, set Type to **To Many**, Delete Rule to **Cascade**, and Arrangement to **Ordered**.
    4.2. Add an **author** relationship to **Book**, with **Author** as the destination entity. In the Relationship Inspector, set Inverse to **books**, Type to **To One**, and Delete Rule to **Nullify**.

CHAPTER 4

Core Data Stack

Core Data Stack

- Graph of controllers needed at runtime to work with managed objects.

- You can programmatically create instances as follows:

    1. **`NSManagedObjectModel`**. Instance serves as a wrapper for the model stored in **.momd** file.
    2. **`NSPersistentStoreCoordinator`**. Initialize with the managed object model from **step 1** to manage one or more persistent stores (typically, SQLite databases).
    3. **`NSPersistentStore`**. Use the store coordinator from Step 2 to create a store object initialized with a file URL pointing to a SQLite database in the app sandbox.
    4. **`NSManagedObjectContext`**. Serves as the primary controller for working with model objects (fetch, save, undo, etc.). Set its `persistentStoreCoordinator` property to point to the store coordinator from Step 2.

Demo: Setting up a Core Data stack.

Managed Object Model

- Model files are typically found in a target's bundle.
- Use `Bundle(for:)` to locate the bundle for a given class.
- You can use `mergedModel(from:)` to coalesce separate model files into a single instance of `NSManagedObjectModel`.

Creating an instance of NSManagedObjectModel:

```
let bundle = Bundle(for: Book.self)
let mom = NSManagedObjectModel.mergedModel(from: [bundle])!

// Accessing entities
let entities = mom.entitiesByName
let bookEntity = entities["Book"]

print("""
    Entity name: \(bookEntity?.name ?? "none")
    MO class name: \(bookEntity?.managedObjectClassName ?? "none")
    """)
```

Lab: Managed Object Model

Write unit tests to experiment with a managed object model.

1. Add a new file **ManagedObjectModelTests** as follows:

    1.1. From Xcode's **File** menu select **New -> File.** In the Template Chooser, select **iOS** at the top, select the **Unit Test Case Class** template. Click **Next,** enter **ManagedObjectModelTests** as the file name, and click **Next.** In the Save panel, click **Create**.

    1.2. In the new test case, add a property named **model** of type `NSManagedObjectModel!`.

    1.3. Write a `setUp()` method that uses the `NSManagedObjectModel` class method `mergedModel(from:)` to load the model from the framework target's bundle and populate the **model** property. (Hint: use `bundle(for:)` to ensure the bundle is loaded from the correct target.).

2. Write a test that asserts that **model** property configured in Step 1 is not nil. Run the test to make sure you can successfully load the model.

3. Write a test that loops through the model's entity descriptions. For each entity, print its managed object class name, and each of its attribute description's attribute value class names, as well as each of its relationship description's destination entity names. (The test doesn't need to assert anything.)

Solution Code

```
import XCTest
import CoreData
@testable import EbooksModel

class ManagedObjectModelTests: XCTestCase
{
    let modelName = "Ebooks"
    let storeName = "EbooksTest"
    var model: NSManagedObjectModel!

    override func setUp() {
        super.setUp()
        let bundle = Bundle(for: ModelController.self)
        model = NSManagedObjectModel.mergedModel(from: [bundle])!
    }

    override func tearDown() {
        super.tearDown()
    }

    func testPersistentContainer() {
        XCTAssertNotNil(model)
    }

    func testEntityDescriptions() {
        for entity in model.entities {
            print("""
                Entity: \(entity.name ?? "nil")
                ----------------------
                \tClass name: \(entity.managedObjectClassName ?? "nil")
                \tAttributes
                """)
            for (key, attribute) in entity.attributesByName {
                print("\t\t\(key): \(attribute.attributeValueClassName ?? "nil")")
            }
            print("\tRelationships")
            for (key, relationship) in entity.relationshipsByName {
                print("\t\t\(key): \(relationship.destinationEntity?.name ?? "nil")")
            }
            print()
        }
    }
```

Persistent Container

- `NSPersistentContainer` is a recent addition to Core Data that acts as a wrapper for a Core Data stack, simplifying setup and management.
- You initialize a persistent container with a managed object model, and the name of a persistent store.
- The persistent container then creates an instance of `NSPersistentStoreCoordinator`, and provides a convenience method you can call to load one or more persistent stores.

Configuring a Persistent Container

```
guard let mom = NSManagedObjectModel.mergedModel(from: [Bundle.main]) else {
    return
}

let container = NSPersistentContainer(name: "EbooksTest",
                                      managedObjectModel: mom)
```

Working with a Persistent Container

- `NSPersistentContainer` provides properties for accessing the managed object model, a persistent store coordinator, and a shared main queue managed object context.
- Also provides convenience API for loading stores and performing background tasks.

Loading Stores and Performing Background Tasks

```
// Loads any stores that have not already been successfully added to
// the container. Completion handler is called once for each store.
open func loadPersistentStores(completionHandler block:
          @escaping (NSPersistentStoreDescription, Error?) -> Swift.Void

// Returns a new, private queue managed object context.
open func newBackgroundContext() -> NSManagedObjectContext

// Creates a temporary, private queue managed object context, which
// then executes the provided closure.
open func performBackgroundTask(_ block:
          @escaping (NSManagedObjectContext) -> Swift.Void)
```

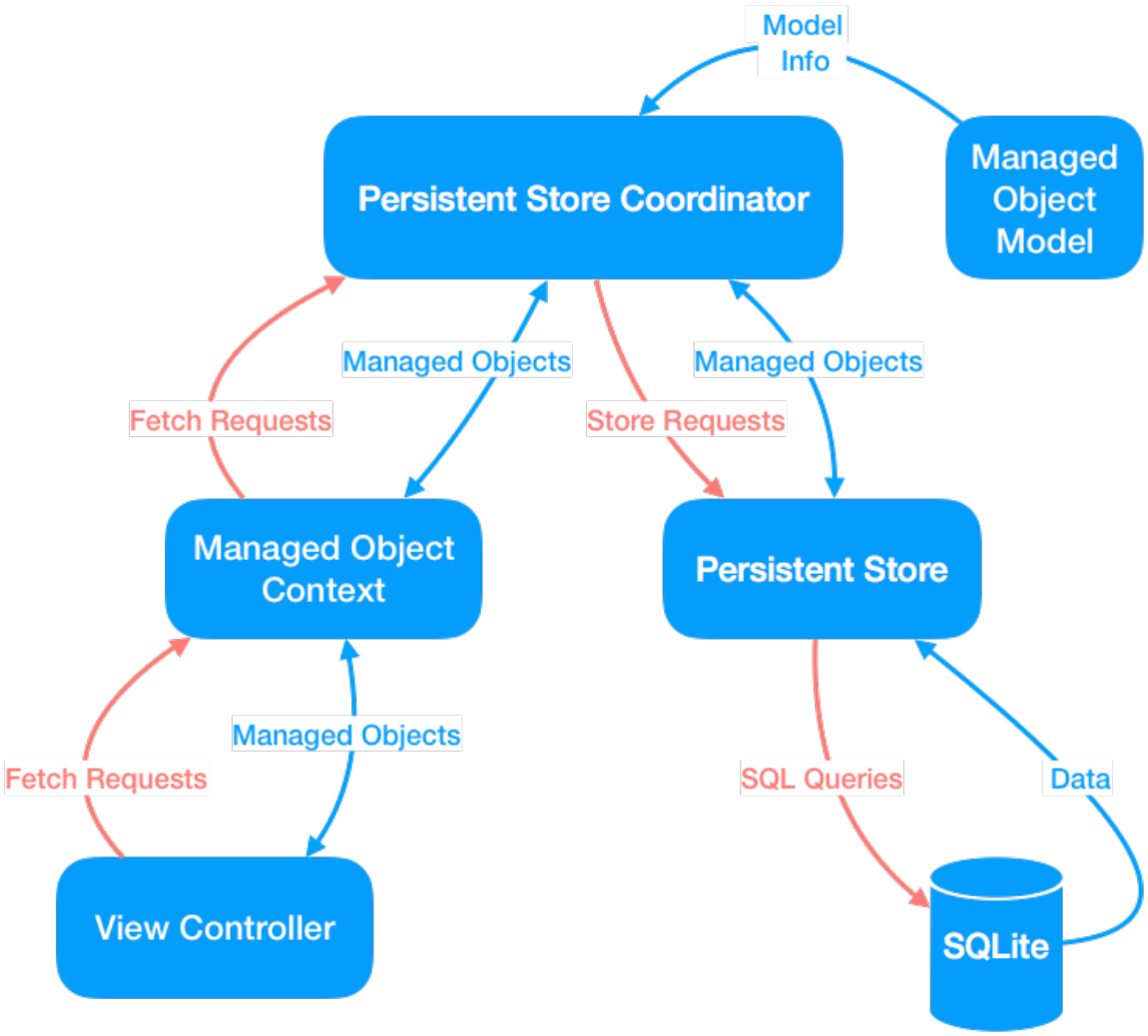
Example

```
let container = NSPersistentContainer(name: "Authors",
                                      managedObjectModel: mom)

// Load one or more NSPersistentStore objects to access SQLite stores.
// (We'll cover this in more detail shortly.)
container.loadPersistentStores { store, error in
    print(error ?? "Persistent store loaded:\n\(store)")
}

// Once store have been loaded, it's easy to start working with
// the main queue context in the `viewContext` property.
//
// For example, you could insert an MO in the context
// (Assume 'author' is an MO that already exists):
container.viewContext.insert(author)

// Here we're printing a count of managed objects currently
// registered in the context.
print(container.viewContext.registeredObjects.count)
// Prints "1"
```

Persistent Store Coordinator

- A persistent store coordinator is used by one or more managed object contexts to access model information and to save and retrieve object graphs.
- Manages multiple stores, coalescing them to present a single facade for fetching and saving across stores.
- Serializes persistence operations to be performed in the database.
  - Typically, separate threads that need to do persistence would have their own persistent store coordinators.
  - If multiple threads need to access the same persistent store coordinator, they would need to explicitly lock and unlock it.

Where NSPersistentStoreCoordinator Sits in the Core Data Stack



Persistent Store

- The following persistent storage types are supported by `NSPersistentStore` on iOS:
  - SQLite relational database (incremental)
  - binary (atomic)
  - in-memory
  - custom (atomic or incremental)
- Instances of `NSPersistentStore` handle the low-level details of storing and retrieving objects from the database (or other store type).

Lab: Persistent Container

Write unit tests to experiment with a persistent container and its Core Data stack.

1. Add a new unit test case named **PersistentContainerTests**.
2. In the new test case, bring over the properties and `setUp` and `tearDown` methods from the previous lab.
    2.1. Add a property named **container** of type `NSPersistentContainer!`.
    2.2. Add code to `setUp` to set the container property to a new instance of `NSPersistentContainer` initialized with the managed object model, and the store name **EbooksTest**.
    2.3. Send a message to the new container to load its persistent stores. (This will cause it to load a single store based on the store name you provided in the initializer.)
3. Write a test method to verify that the container's default directory URL ends with "Application Support".
4. Write a test method that accesses the `NSPersistentStore` instance loaded in the `setUp` method. Verify that the persistent store URL ends with the store name you provided, and that it's type is "SQLite".
5. Write a test that verifies that the container's managed object model is the same as the one you initialized the container with.
6. Write a test that verifies that the container's `viewContext` property contains a managed object context with a concurrency type of `.mainQueueConcurrencyType`.

Solution Code

```
import XCTest
import CoreData
@testable import EbooksModel

class PersistentContainerTests: XCTestCase
{
    let modelName = "Ebooks"
    let storeName = "EbooksTest"
    var container: NSPersistentContainer!
    var model: NSManagedObjectModel!

    override func setUp() {
        super.setUp()
        let bundle = Bundle(for: ModelController.self)
        model = NSManagedObjectModel.mergedModel(from: [bundle])!
        container = NSPersistentContainer(name: storeName, managedObjectModel: model)
        container.loadPersistentStores { store, error in
            print(error ?? "Persistent store loaded:\n\(store)")
        }
    }

    override func tearDown() {
        super.tearDown()
    }

    func testDirectoryURL() {
        let dirName = NSPersistentContainer.defaultDirectoryURL().lastPathComponent
        XCTAssertEqual(dirName, "Application Support")
    }

    func testStoreNameAndType() {
        let storeCoordinator = container.persistentStoreCoordinator
        let store = storeCoordinator.persistentStores.first!
        let name = store.url!.deletingPathExtension().lastPathComponent
        XCTAssertEqual(name, storeName)
        XCTAssertEqual(store.type, "SQLite")
    }

    func testModel() {
        let model = container.persistentStoreCoordinator.managedObjectModel
        XCTAssertEqual(model, self.model)
    }

    func testContext() {
        XCTAssertEqual(container.viewContext.concurrencyType,
                       NSManagedObjectContextConcurrencyType.mainQueueConcurrencyType)
    }
```

```
}
```

CHAPTER 5

Managed Object Contexts

Managed Object Context

- Instance of `NSManagedObjectContext.`
- Manages a collection of managed objects (instances of `NSManagedObject`).
  - Implements the **Key-Value Observing** (KVO) informal protocol to monitor updates, insertions, and deletions.
  - Provides a transactional context for persistence operations.
- Responsibilities include:
  - Life-cycle management of its managed objects
  - Tracking changes
  - Fetching and saving
  - Uniquing
  - Validation
  - Managing concurrency
  - Undo/redo management

Managed Objects

- Instances of `NSManagedObject`
- Ordinarily, managed objects are inserted in a managed object context during initialization.

```
// Designated initializer.
public init(entity: NSEntityDescription,
            insertInto context: NSManagedObjectContext?)

// Returns a new object, inserted into managedObjectContext. Only legal to
// call on subclasses of NSManagedObject that represent a single entity in
// the model.
@available(iOS 10.0, *)
public convenience init(context moc: NSManagedObjectContext)
```

- Note that the designated initializer's context argument above is nullable.
  - That would allow you to initialize a managed object without a reference to its context.
  - You could then insert the managed object later by sending an `insert(_:)` message to its context.

Viewing SQL

- Core Data is capable of logging the SQL it generates for database operations to the console.
- To enable SQL logging
  1. Select **Product > Scheme > Edit Scheme…**
  2. Configure the following command line argument:

```
-com.apple.CoreData.SQLDebug 1
```

Lab: Managed Object Context

Create a unit test case to experiment with managed objects and contexts.

1. Add a new unit test case named **ManagedObjectContextTests** as follows:

    1.1. In the new test case, bring over the properties and `setUp` and `tearDown` methods from the previous lab.

    1.2. Modify the `setUp` method to send a message to the container's persistent store coordinator to destroy its persistent stores. (This will drop all tables in the database.) Be sure to check that the store file exists before making the call; otherwise, this will always fail the first time you run the tests after a full clean.

2. Write a test that inserts a new instance of `NSManagedObject` (the base, class, not a subclass), initialized with the **Book** entity, in the container's view context.

    2.1. Without casting, check the instance's type. It should be `Book`.

    2.2. Prior to inserting, verify that the `insertedObjects` and `registeredObjects` arrays are empty.

    2.3. Upon insertion, verify that both arrays have counts of **1**.

3. Write a test that creates an instance of `Book` in the container's view context, and do as follows:

    3.1. Set the book's title to a non-empty string.

    3.2. Save the context and verify that the call didn't throw.

    3.3. Add a breakpoint at the end of the test. Run the test, and while stopped at the breakpoint, find the SQLite database in the filesystem and verify that a row matching the object's state exists.

    3.4. Let the test continue to the end to allow the `tearDown` method to execute, and then check the database — it should now be empty.

4. Write a attempts to save a book instance similar as in the previous step. However, this time, do as follows:

    4.1. Inspect the **title** attribute of the **Book** entity in Xcode's Model Editor. Uncheck **optional** to make the property value mandatory. Also, check **Min Length**, and set the validation to **1**.

    4.2. Create a book instance in the view context, but don't set its title.

4.3. Call `validateForSave()` on the book, and verify that it throws an error.

4.4. Call `save()` on the context, and verify that it also throws an error.

Solution Code

```
import XCTest
import CoreData
@testable import EbooksModel

class ManagedObjectsContextTests: XCTestCase
{
    let modelName = "Ebooks"
    let storeName = "EbooksTest"
    var container: NSPersistentContainer!
    lazy var managedObjectModel: NSManagedObjectModel! =
        NSManagedObjectModel.mergedModel(from: [Bundle(for: Book.self)])

    override func setUp() {
        super.setUp()
        container = NSPersistentContainer(name: storeName,
                                          managedObjectModel: managedObjectModel)
        container.loadPersistentStores { store, error in
            print(error ?? "Persistent store loaded:\n\(store)")
        }
        print("\(NSPersistentContainer.defaultDirectoryURL())\n")
    }

    override func tearDown() {
        if let dbUrl = container.persistentStoreDescriptions.first?.url,
            FileManager.default.fileExists(atPath: dbUrl.path) {
            try! container.persistentStoreCoordinator
                .destroyPersistentStore(at: dbUrl, ofType: "sqlite", options: nil)
        }
        super.tearDown()
    }

    func testInsertManagedObject() {
        XCTAssertEqual(container.viewContext.insertedObjects.count, 0)
        XCTAssertEqual(container.viewContext.registeredObjects.count, 0)
        let entity = managedObjectModel.entitiesByName["Book"]!
        let mo = NSManagedObject(entity: entity, insertInto: nil)
        container.viewContext.insert(mo)
        print(mo)
        XCTAssertTrue(mo is Book)
        XCTAssertEqual(container.viewContext.registeredObjects.count, 1)
        XCTAssertEqual(container.viewContext.insertedObjects.count, 1)
    }
```

```
func testSaveBook() {
    let book = Book(context: container.viewContext)
    book.title = "A Title"

    do {
        try container.viewContext.save()
    }
    catch let error {
        XCTFail("Unable to save book. Error was \(error)")
    }
}

func testValidateBookForSave() {
        let book = Book(context: container.viewContext)

        do {
            try container.viewContext.save()
            XCTFail("Should have failed validation because title not optional, and min
length must be >= 1")
        }
        catch let error {
            print(error)
        }
    }

    func testSaveBook() {
        let book = Book(context: container.viewContext)
        book.title = "A Title"

        do {
            try container.viewContext.save()
        }
        catch let error {
            XCTFail("Unable to save book. Error was \(error)")
        }
    }
}
```

Managed Object Properties

- `NSManagedObject` provides opaque, dictionary-like, generic storage for property values.
  - Allows Core Data to easily and efficiently realize *faults* (covered in next section).
  - Allows `NSManagedObject` to be used without subclassing.
- `NSManagedObject` provides API access to primitive values.

Primitive Accessors

```
// Swift
open func primitiveValue(forKey key: String) -> Any?
open func setPrimitiveValue(_ value: Any?, forKey key: String)

// KVC - overridden to access generic dictionary storage unless
// subclasses explicitly provide accessors
open func value(forKey key: String) -> Any?
open func setValue(_ value: Any?, forKey key: String)

// Objective-C
- (nullable id)primitiveValueForKey:(NSString *)key;
- (void)setPrimitiveValue:(nullable id)value forKey:(NSString *)key;

// KVC - overridden to access generic dictionary storage unless
// subclasses explicitly provide accessors
- (nullable id)valueForKey:(NSString *)key;
- (void)setValue:(nullable id)value forKey:(NSString *)key;
```

Synthesized Subclasses

- Core Data synthesizes opaque, entity-specific subclasses in memory at runtime to provide strongly-typed computed properties as wrappers for primitive values.

- By default, Core Data uses `NSObject` behavior to add method implementations at runtime for properties annotated with `@NSManaged` (Swift) or `@dynamic` (Objective-C).

Dynamically Resolving Instance Methods

```
// Swift
open class func resolveInstanceMethod(_ sel: Selector!) -> Bool

// Objective-C
+ (BOOL)resolveInstanceMethod:(SEL)sel;
```

Lab: Managed Object Properties

Create a unit test case to experiment with primitive properties.

1. Add a new unit test case named **ManagedObjectContextTests**, and bring over the properties and `setUp` and `tearDown` methods from the previous lab.
2. Write a test that inserts a new instance of `NSManagedObject` (the base, class, not a subclass), initialized with the **Book** entity, in the container's view context.
    2.1. Try to set its `title` property. (This should fail to compile.)
    2.2. Set the object's primitive value for the key **title** to **"A"**.
    2.3. Verify that the primitive value for the key **title** is now **"A"**.
3. Write a test that inserts a new instance of `NSManagedObject` in the context as in the previous step, and verify that it can respond to the selector **"title"**. Note that you will have to use the Foundation C function `NSSelectorFromString` to convert the literal string **"title"** into a selector that can be passed to the Swift `responds(to:)` method.
4. Write a test that inserts a new `Book` into the view context. Set the value of its `title` property to **"A"**, and then call `setPrimitiveValue(_:forKey:)` to change the value to **"B"**. Verify that the value of the `title` property is now **"B"**.

Solution Code

```
import XCTest
import CoreData
@testable import EbooksModel

class ManagedObjectsPropertiesTests: XCTest
{
    let modelName = "Ebooks"
    let storeName = "EbooksTest"
    var container: NSPersistentContainer!
    var model: NSManagedObjectModel!

    override func setUp() {
        super.setUp()
        let bundle = Bundle(for: ModelController.self)
        model = NSManagedObjectModel.mergedModel(from: [bundle])!
        container = NSPersistentContainer(name: storeName, managedObjectModel: model)
    }

    override func tearDown() {
        super.tearDown()
    }

    func testMOPrimitiveValues() {
        let mo = NSManagedObject(context: container.viewContext)
        // mo.title = "A"
        mo.setPrimitiveValue("A", forKey: "title")
        XCTAssertEqual(mo.primitiveValue(forKey: "title") as! String, "A")
    }

    func testMORespondsTo() {
        let mo = NSManagedObject(context: container.viewContext)
        // XCTAssertTrue(mo.responds(to: #selector(title)))
        XCTAssertTrue(mo.responds(to:NSSelectorFromString("title")))
    }

    func testBookPrimitiveValues() {
        let book = Book(context: container.viewContext)
        book.title = "A"
        book.setPrimitiveValue("B", forKey: "title")
        XCTAssertEqual(book.title, "B")
    }
}
```

CHAPTER 6

Custom Managed Object Classes

Managed Object Subclasses

- You can subclass `NSManagedObject`, but not its dynamically generated subclasses.
- The name of the subclass to be used for a given entity can be specified in the model file; otherwise `NSManagedObject` is assumed.
- If your subclasses will be referenced from Objective-C code, you should prefix class names per Objective-C naming guidelines.

Primitive Properties

- `NSManagedObject` contains dictionary-like storage for its properties.
  - Among other things, this makes it easy for Core Data to realize faults.
  - Allows you to use `NSManagedObject` directly, without subclassing.
- `NSManagedObject` provides an API for accessing primitive values. Because it's common to use subclasses with either synthesized or generated accessors, you should rarely need to use this API directly.

Property Accessors

- Overridden Obj-C property accessors must not call **super**.
- Properties annotated with `@NSManaged` will be synthesized at runtime (the Swift equivalent of `@dynamic`).
- Accessors must provide **KVO**-like notifications. If you write your own accessors, they must invoke the relevant access and change notification methods. (Core Data disables ordinary KVO notifications for instances of NSManagedObject in favor of its own custom KVO-style API.)

KVO-like Access and Change Notifications

```
// NSManagedObject KVO-style access notifications
open func willAccessValue(forKey key: String?)
open func didAccessValue(forKey key: String?)

// NSManagedObject KVO-style change notifications
open func willChangeValue(forKey key: String)
open func didChangeValue(forKey key: String)

open func willChangeValue(forKey inKey: String,
          withSetMutation inMutationKind: NSKeyValueSetMutationKind,
          using inObjects: Set<AnyHashable>)

open func didChangeValue(forKey inKey: String,
          withSetMutation inMutationKind: NSKeyValueSetMutationKind,
          using inObjects: Set<AnyHashable>)
```

Usage Example

```
public var firstName: String?
{
    get {
        willAccessValue(forKey: "firstName")
        defer { didAccessValue(forKey: "firstName") }
        return primitiveValue(forKey: "firstName") as? String
    }
    set {
        willChangeValue(forKey: "firstName")
        setPrimitiveValue(newValue, forKey: "firstName")
        didChangeValue(forKey: "firstName")
    }
}
```

Collection Properties

- The default accessor type for `NSManagedObject` to-many relationships is `NSSet`. For ordered relationships, the accessor type is `NSOrderedSet`.

- You can use KVC methods to modify contents of immutable sets and ordered sets. The following methods return a mutable proxy for the underlying collection.

KVC Mutable Set Accessors

```
// Swift:
open func mutableSetValue(forKey key: String) -> NSMutableSet
open func mutableOrderedSetValue(forKey key: String) -> NSMutableOrderedSet

// Objective-C:
- (NSMutableSet *)mutableSetValueForKey:(NSString *)key;
- (NSMutableSet *)mutableOrderedSetValueForKey:(NSString *)key;
```

Usage Example

```
// Swift:
let nicknames = author.mutableOrderedSetValue(forKey: "nicknames")
nicknames.add("Fred")
nicknames.remove("Freddy")

// Objective-C:
NSMutableSet *nicknames = [author mutableSetValueForKey:@"nicknames"];
[nicknames addObject:@"Fred"];
[nicknames removeObject:@"Freddy"];
```

Code Generation

Xcode can use the information in a data model to generate `NSManagedObject` subclasses and extensions that contain:

- Strongly-typed properties, for example:

```
@NSManaged public var name: String?
@NSManaged public var books: NSOrderedSet?
```

- KVC accessors for collections, for example:

```
@objc(addBooksObject:)
@NSManaged public func addToBooks(_ value: Book)
```

- Convenience API, for example:

```
@nonobjc public class func fetchRequest() -> NSFetchRequest<Author>
```

Generated Class Example

Author+CoreDataClass.swift

```
import CoreData

@objc(Author)
public class Author: NSManagedObject { }
```

Author+CoreDataProperties.swift

```
extension Author {

    @nonobjc public class func fetchRequest() -> NSFetchRequest<Author> {
        return NSFetchRequest<Author>(entityName: "Author")
    }

    @NSManaged public var iTunesId: Int32
    @NSManaged public var name: String?
    @NSManaged public var rating: Int16
    @NSManaged public var books: NSOrderedSet?

}

// MARK: Generated accessors for books
extension Author {

    @objc(insertObject:inBooksAtIndex:)
    @NSManaged public func insertIntoBooks(_ value: Book, at idx: Int)

    @objc(removeObjectFromBooksAtIndex:)
    @NSManaged public func removeFromBooks(at idx: Int)

    @objc(insertBooks:atIndexes:)
    @NSManaged public func insertIntoBooks(_ values: [Book], at indexes: NSIndexSet)

    @objc(removeBooksAtIndexes:)
    @NSManaged public func removeFromBooks(at indexes: NSIndexSet)

    @objc(replaceObjectInBooksAtIndex:withObject:)
    @NSManaged public func replaceBooks(at idx: Int, with value: Book)

    @objc(replaceBooksAtIndexes:withBooks:)
    @NSManaged public func replaceBooks(at indexes: NSIndexSet, with values: [Book])

    @objc(addBooksObject:)
    @NSManaged public func addToBooks(_ value: Book)

    @objc(removeBooksObject:)
    @NSManaged public func removeFromBooks(_ value: Book)

    @objc(addBooks:)
    @NSManaged public func addToBooks(_ values: NSOrderedSet)

    @objc(removeBooks:)
    @NSManaged public func removeFromBooks(_ values: NSOrderedSet)
```

```
}
```

Manual Code Generation

- Xcode can generate classes and extensions based on information you provide (class names, attributes, relationships, etc.) in the data model.

- However, you would either need to manually regenerate or else hand-modify these files whenever you modify the model.

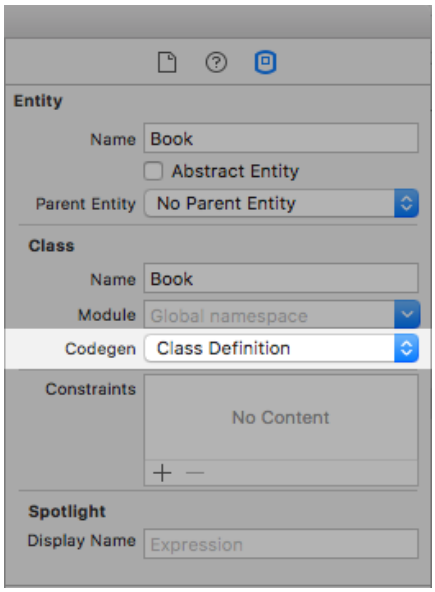
- To manually generate code:

  1. Open the data model

  2. Select **Editor > Create NSManagedObject Subclass…**.

Dynamic Code Generation

- Alternatively, you can elect to have Xcode dynamically regenerate classes and/or extensions whenever you modify the data model.
- Dynamically generated code is placed in the Derived Data directory, and will not be directly visible in your project.
- To configure dynamic code generation, open the data model, and select the entity you want to configure. In the Inspector, select **Class Definition** from the **Codegen** dropdown.



Demo: Generating Classes

Change and Undo Management

- The context uses KVO to track changes to managed objects.
  - Includes insertions and deletions, changes to relationships, as well as changes to property values.
  - Its `hasChanges` property allows you to check prior to calling `save()` to avoid incurring unnecessary overhead.
  - The `NSManagedObject` property `hasPersistentChangedValues`, serves a similar purpose.
- Provides API for undo/redo, rollback, etc.

Managing NSManagedObjectContext State

```
open var undoManager: UndoManager?

open func undo()
open func redo()
open func reset()
open func rollback()

open var hasChanges: Bool { get }
open func save() throws

open func refresh(_ object: NSManagedObject, mergeChanges flag: Bool)
```

Faults

- A fault is a placeholder for a managed object whose property values (other than objectID) have not yet been fetched.
- Faults allow Core Data to avoid performing unnecessary joins, and conserve memory and CPU by deferring memory allocation for storage of property values.
- When a fault fires, its data is dynamically fetched in. (You can observe this at runtime by enabling SQL logging.)

Lab: Managed Object Subclasses

Create a unit test case to experiment with managed object subclasses.

1. Add a new unit test case named **ManagedObjectSubclassTests**, and bring over the properties and `setUp` and `tearDown` methods from the previous lab.
2. Write a test that inserts a new instance of `Author` in the container's view context, and then verifies that the author object is in the context's registered objects array.
3. Write a test that inserts two new instances of `Author`, `author1` and `author2`, in the container's view context, their names to **"A"** and **"B"**, respectively.
    3.1. Verify that the value of author1's `isDeleted` property is **false**.
    3.2. Verify that the context's inserted and registered objects arrays contain 2 objects, and that it's deleted objects array is empty.
    3.3. Call the context's `delete(_:)` method, passing author1, and verify that the deleted objects now contains one object, and that the other arrays's counts are unchanged.
    3.4. Call the context's `save(_:)` method, and verify that the deleted and inserted objects arrays are now empty, while the registered objects array now contains only author2.
4. Write a test that inserts an Author instance and saves the context, and does as follows:
    4.1. Call the context's `count(for:)` method with the Author type's built-in fetch request, and verify that the resulting count agrees with the count of the registered objects.
    4.2. Call the context's `reset()` method, and verify that the registered objects count is now zero, while the result of calling `count(for:)` is still one.
    4.3. Insert another Author instance, and verify that the registered objects count is now one, while the result of calling `count(for:)` is now two.
5. Write a test that adds an Author to the context, and verifies that the value of its `hasPersistentChangedValues` property is **false**.
    5.1. Change one or more property value, and then verify that the value of the `hasPersistentChangedValues` property is now **true**.
    5.2. Save the context, and verify that the author's objectID is different after the save.
6. Write a test that adds a new Author to the context, sets its `iTunesId` property, and then saves and resets the context.

6.1. Call the context's `fetch(_:)` method with the `Author` type's built-in fetch request.

6.2. Verify that the resulting array contains one author, and that the author it contains is not the same instance as the original, but that it has the same managed object ID.

7. Write a test that adds a new Author to the context, and then adds two instances of `Book` to the author.

7.1. Verify that the books's `author` property contains a reference to the author, and the inserted objects array contains three items.

7.2. Save the context, and then call the context's `count(for:)` method with the `Book` type's built-in fetch request. Verify that the result is equal to the number of elements in the author's `books` property.

8. Write a test that adds a new Author to the context, and verifies that it is not a fault, and that its managed object id is temporary.

8.1. Save the context and then reset it. Then fetch the author using the Author type's built-in fetch request. Verify that the fetched author *is* a fault, that its object id *is not* temporary, and that its **books** relationship is a fault.

8.2. Initialize a local let constant with the first item in the author's `books` collection. Assert that the book is a fault, that the fetched author is no longer a fault, and that and that the author's **books** relationship is no longer a fault.

8.3. Access the book's title property, and then verify that the book is no longer a fault.

Solution Code

```
import XCTest
import CoreData
@testable import EbooksModel

class ManagedObjectsSubclassTests: XCTestCase
{
    var container: NSPersistentContainer!
    var context: NSManagedObjectContext { return container.viewContext }

    lazy var model: NSManagedObjectModel! =
        NSManagedObjectModel.mergedModel(from: [Bundle(for: Book.self)])

    override func setUp() {
        super.setUp()
        container = NSPersistentContainer(name: storeName,
                                          managedObjectModel: model)
        container.loadPersistentStores { store, error in
            print(error ?? "Persistent store loaded:\n\(store)")
        }
    }

    override func tearDown() {
        if let dbUrl = container.persistentStoreDescriptions.first?.url,
            FileManager.default.fileExists(atPath: dbUrl.path) {
            try! container.persistentStoreCoordinator
                .destroyPersistentStore(at: dbUrl, ofType: "sqlite", options: nil)
        }
        super.tearDown()
    }
}

// MARK: Inserting, Deleting, and Counting
extension ManagedObjectsSubclassTests
{
    func testInsertAuthor() {
        let author = Author(context: container.viewContext)
        XCTAssertTrue(container.viewContext.registeredObjects.contains(author))
    }

    func testDeleteAuthor() {
        let author1 = Author(context: context)
        author1.name = "A"
        let author2 = Author(context: context)
        author2.name = "B"
        XCTAssertFalse(author1.isDeleted)
        XCTAssertEqual(context.insertedObjects.count, 2)
        XCTAssertEqual(context.registeredObjects.count, 2)
```

```
        XCTAssertEqual(context.deletedObjects.count, 0)

        // Calling `delete` marks author1 as deleted. Note, however, that author1
        // remains in the insertedObjects and registeredObjects arrays pending
        // a call to `save`.
        context.delete(author1)
        XCTAssertTrue(author1.isDeleted)
        XCTAssertEqual(context.insertedObjects.count, 2)
        XCTAssertEqual(context.registeredObjects.count, 2)
        XCTAssertEqual(context.deletedObjects.count, 1)

        try! context.save()
        XCTAssertEqual(context.insertedObjects.count, 0)
        XCTAssertEqual(context.registeredObjects.count, 1)
        XCTAssertEqual(context.deletedObjects.count, 0)
    }

    func testCountQuery() {
        let author1 = Author(context: context)
        author1.name = "B"

        try! context.save()
        XCTAssertEqual(context.registeredObjects.count, 1)
        XCTAssertEqual(try! context.count(for: Author.fetchRequest()), 1)

        // Calling `reset` removes all managed objects from the context.
        context.reset()
        XCTAssertEqual(context.registeredObjects.count, 0)
        XCTAssertEqual(try! context.count(for: Author.fetchRequest()), 1)

        let author2 = Author(context: context)
        author2.name = "C"
        XCTAssertEqual(context.registeredObjects.count, 1)
        XCTAssertEqual(try! context.count(for: Author.fetchRequest()), 2)
    }
}

// MARK: Saving and Fetching
extension ManagedObjectsSubclassTests
{
      func testSaveAuthor() {
        let author = Author(context: context)
        XCTAssertFalse(author.hasPersistentChangedValues)

        author.iTunesId = 123
        author.name = "C"
        author.rating = 2
        XCTAssertTrue(author.hasPersistentChangedValues)

        let temporaryId = author.objectID

        if author.hasPersistentChangedValues {
            do {
                try context.save()
```

```
            }
            catch let error {
                XCTFail("Unable to save author due to error: \(error)")
            }
        }

        let persistentId = author.objectID
        XCTAssertNotEqual(temporaryId, persistentId)
    }

    func testFetchAuthor() {
        let iTunesId = 456
        let author1 = Author(dictionary:
            ["iTunesId": iTunesId, "name": "F", "rating": 1], context: context)
        do {
            try context.save()
        }
        catch let error {
            XCTFail("Unable to save author, error was: \(error)")
        }

        context.reset();

        do {
            let result = try context.fetch(Author.fetchRequest()) as! [Author]
            XCTAssertFalse(result.first! === author1)
            XCTAssertEqual(result.first!.objectID, author1.objectID)
            XCTAssertEqual(context.insertedObjects.count, 0)
            XCTAssertEqual(context.registeredObjects.count, 1)
        }
        catch let error {
            XCTFail("Unable to fetch previously saved author, error was: \(error)")
        }
    }
}

// MARK: Nested Books
extension ManagedObjectsSubclassTests
{
    func testManuallyAddBooksToAuthor() {
        let author = Author(context: context)
        author.name = "A"
        let book1 = Book(context: context)
        book1.title = "B"
        let book2 = Book(context: context)
        book2.title = "C"

        // Use generated KVC accessors to manage books
        author.addToBooks(book1)
        author.addToBooks(book2)

        // KVC accessors take care of behind-the-scenes details, such as
        // setting inverse relationships
        XCTAssertEqual(book1.author, author)
```

```
        XCTAssertEqual(context.insertedObjects.count, 3)

        do {
            try context.save()
        }
        catch let error {
            XCTFail("Unable to save author due to error: \(error)")
        }

        let savedBooksCount = try! context.count(for: Book.fetchRequest())
        XCTAssertEqual(savedBooksCount, author.books!.count)
    }

    func testFaultedObjects() {
        let author = Author(dictionary: authorAndBooksInfo, context: context)
        XCTAssertTrue(author.objectID.isTemporaryID)
        XCTAssertFalse(author.isFault)
        try! context.save()

        // Fetched objects are initially faults, unless the fetch request
        // specifies otherwise.
        context.reset()
        let fetchedAuthor = try! context.fetch(Author.fetchRequest()).first!
            as! Author
        XCTAssertTrue(fetchedAuthor.isFault)
        XCTAssertTrue(fetchedAuthor.hasFault(forRelationshipNamed: Author.booksKey))
        XCTAssertFalse(fetchedAuthor.objectID.isTemporaryID)

        // Accessing the author's `books` property trips the fault. Note, however,
        // that objects in the set of books are themselves faults.
        let fetchedBooks = fetchedAuthor.books!
        let book1 = fetchedBooks.firstObject as! Book
        XCTAssertTrue(book1.isFault)
        XCTAssertFalse(fetchedAuthor.isFault)
        XCTAssertFalse(fetchedAuthor.hasFault(forRelationshipNamed: Author.booksKey))

        // Accessing the book's `title` property trips the fault that represents
        // the book.
        XCTAssertEqual(book1.title, "Book 1")
        XCTAssertFalse(book1.isFault)

        // A fetched object's managed object id contains references to the object's
        // Core Data entity, as well as a URI that includes the store identifier,
        // entity name, and primary key
        XCTAssertEqual(fetchedAuthor.objectID.entity.name, "Author")
        XCTAssertEqual(fetchedAuthor.objectID.uriRepresentation().host,
                       fetchedAuthor.objectID.persistentStore?.identifier)
    }
}
```

CHAPTER 7

Fetch Requests

Fetch Requests

- Core Data performs SQL queries on your behalf
  - **NSFetchRequest** objects specify query details
  - Use **NSPredicate** to specify **WHERE** clause
- Result types
  - Managed object (default)
  - Managed object ID
  - Dictionary
  - Count

Batches and Offsets

- Fetch requests can be configured to fetch large result sets in smaller batches.

- The resulting array will automatically batch fetch the next batch of results on demand.

```
/* This breaks the result set into batches.  The entire request will be
evaluated, and the identities of all matching objects will be recorded, but
no more than batchSize objects' data will be fetched from the persistent
store at a time.  The array returned from executing the request will be a
subclass that transparently faults batches on demand.  For purposes of thread
safety, the returned array proxy is owned by the NSManagedObjectContext the
request is executed against, and should be treated as if it were a managed
object registered with that context.  A batch size of 0 is treated as
infinite, which disables the batch faulting behavior.  The default is 0. */

- (NSUInteger)fetchBatchSize;
- (void)setFetchBatchSize:(NSUInteger)bsize;
```

Faults vs. Aggressive Fetching

- By default, fetched objects contain faults for all relationships.
- You can optionally specify relationships you want prefetched.

```
/* Returns/sets an array of relationship keypaths to prefetch along with the
entity for the request.  The array contains keypath strings in
NSKeyValueCoding notation, as you would normally use with valueForKeyPath.
(Prefetching allows Core Data to obtain developer-specified related objects
in a single fetch (per entity), rather than incurring subsequent access to
the store for each individual record as their faults are tripped.)  Defaults
to an empty array (no prefetching.)
*/
- (NSArray *)relationshipKeyPathsForPrefetching;
- (void)setRelationshipKeyPathsForPrefetching:(NSArray *)keys;
```

- You can also specify that the fetched objects themselves should initially be faults.

```
/* Returns/sets if the objects resulting from a fetch request are faults.  If
the value is set to NO, the returned objects are pre-populated with their
property values (making them fully-faulted objects, which will immediately
return NO if sent the -isFault message.)  If the value is set to YES, the
returned objects are not pre-populated (and will receive a -didFireFault
message when the properties are accessed the first time.)  This setting is
not utilized if the result type of the request is
NSManagedObjectIDResultType, as object IDs do not have property values.
Defaults to YES.
*/
- (BOOL)returnsObjectsAsFaults;
- (void)setReturnsObjectsAsFaults:(BOOL)yesNo;
```

Lab: Fetch Requests

- Create an XCUnit test case class to experiment with fetch requests.
- Tests:
  - Create a fetch request to fetch objects for a given entity. Send **executeFetchRequest:** to the context, and check the array of returned MOs to verify that all the objects for that entity were fetched.
  - Set the fetch request's batch size to a value smaller than the length of the previously returned array, and execute the fetch again. Check the array's count. Enumerate through the array, and watch the console to see if the fault trips when you exhaust the first batch of results.
  - Configure the fetch request to return objects as faults. After executing a new fetch, check the first object in the array to verify that its relationships are fault objects. Try to trip the faults by accessing properties of the related objects.

Predicates

- Use to specify search criteria
- Initialize with format string

```
+ (NSPredicate *)predicateWithFormat:(NSString *)f, ...;
```

- Can also be used to filter collections in memory

```
- (NSArray *)filteredArrayUsingPredicate:(NSPredicate *)p;
```

- Or to test an object's state

```
- (BOOL)evaluateWithObject:(id)object;
```

EXAMPLE

```
NSFetchRequest *f = [NSFetchRequest fetchRequestWithEntityName:@"Book"];
f.predicate = [NSPredicate predicateWithFormat:@"title = %@", title];
```

Lab: Predicates

- Create an XCUnit test case class to experiment with predicates.
- Tests:
  - Create a fetch request, adding a predicate specifying that the value of one of the target entity's string properties must begin with a provided prefix. Send **executeFetchRequest:** to the context, and check the array of returned MOs to verify that only objects matching the predicate were returned.

CHAPTER 8

Working with Data Sources

Fetched Results Controllers

- Designed for integration with classes that implement data source protocols such as **UITableViewDataSource** and **UICollectionViewDataSource**.

- Sends notification message to its delegate as managed objects are added, deleted, or modified in its managed object context (see the **NSFetchedResultsControllerDelegate** protocol).

  - Allows the data source to dynamically synchronize the state of the UI as the state of the context changes.

Lab: Fetched Results Controllers

- Create an XCUnit test case class to experiment with fetch results controllers.
- Tests:
  - Create a fetched results controller initialized with a fetch request that specifies a fetch of all the objects for a given entity. Send the fetched results controller a **performFetch:** message, and then check its fetched objects array.
  - In your test case class, implement several methods from the **NSFetchedResultsControllerDelegate** protocol. The implementations can simply log their argument values. Create a new fetched results controller and set its delegate property to **self** (i.e., the test case instance). Perform a fetch, and then try modifying the context by inserting, deleting, and modifying objects. Verify that the delegate methods are called as expected.

CHAPTER 9

Model Versioning & Data Migration

The mogenerator Tool

- Alternative tool for generating managed object classes.
- Generates pair of classes per entity:
  - Base class regenerated when model changes.
  - Stub subclass you can customize; not regenerated.
- Additional features, such as:
  - Primitive accessor declarations.
  - Convenience methods such as **entityName**, **insertInManagedObjectContext:**, etc.
  - Wrapper methods for to-many relationships that return mutable collections.
- Can be integrated in Xcode by adding **Aggregate Target** with a **Run Script** build phase.

MOGenerator on Github:

https://github.com/rentzsch/mogenerator

How To:

http://macindie.com/2012/05/mogenerator-your-core-data-projects-bff/

SECTION 4

Cocoa touch Design Patterns

CHAPTER 1

Working with Nib Files

Main Window Nib

- Three ways to configure initial objects in app at launch time.
  - Programmatically
  - Main storyboard
  - Main nib file
- Typical main nib file contents:
  - Main window
  - Root view controller
  - App delegate

View Controllers

- **view** property lazily initialized by calling **loadView**
  - By default, loads the view from a nib file
  - Can be overridden to load view programmatically
  - Override **viewDidLoad** to do additional initialization

NSBundle and UINib

- Use **NSBundle** to access bundle resources

- Accessing the app bundle

```
+ (NSBundle *)mainBundle;

// Use bundleForClass: for classes that are sometimes loaded
// from a bundle other than the app's main bundle -- for example
// from a unit test bundle.
+ (NSBundle *)bundleForClass:(Class)aClass;
```

- Locating resources

```
- (NSString *)pathForResource:(NSString *)name
                       ofType:(NSString *)ext;

- (NSString *)pathForResource:(NSString *)name
                       ofType:(NSString *)ext
                  inDirectory:(NSString *)subpath;
```

- Loading nib files

```
@interface NSBundle(UINibLoadingAdditions)
- (NSArray *)loadNibNamed:(NSString *)name
                    owner:(id)owner
                  options:(NSDictionary *)options;
@end

@interface NSObject(UINibLoadingAdditions)
// Sent to each object loaded from a nib after all the objects
// in the nib archive have been loaded
- (void)awakeFromNib;
@end
```

Loading Table View Cells

- Use **UITableView** methods to register classes and/or nib files to use when creating cells dynamically

- Methods for dequeueing cells and header/footer views by reuse identifier:

```
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier;

// Dequeues a cell for a given index path
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier
                           forIndexPath:(NSIndexPath *)indexPath;

// Dequeues a view to use for header and footer views
- (id)dequeueReusableHeaderFooterViewWithIdentifier:(NSString *)identifier;
```

- Registering class or nib file to use to dequeue cells or header/footer views by reuse identifier:

```
// Registering nib or class to use for table view cells

- (void)registerNib:(UINib *)nib
forCellReuseIdentifier:(NSString *)identifier;

- (void)registerClass:(Class)cellClass
forCellReuseIdentifier:(NSString *)identifier;

// Registering nib or class to use for section headers and footer views

- (void)registerNib:(UINib *)nib
forHeaderFooterViewReuseIdentifier:(NSString *)identifier;

- (void)registerClass:(Class)aClass
forHeaderFooterViewReuseIdentifier:(NSString *)identifier;
```

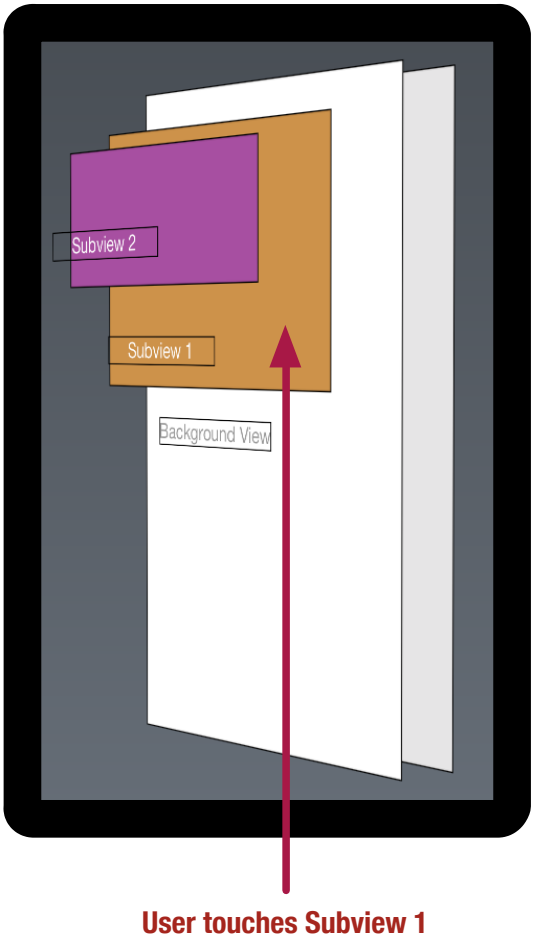
CHAPTER 2

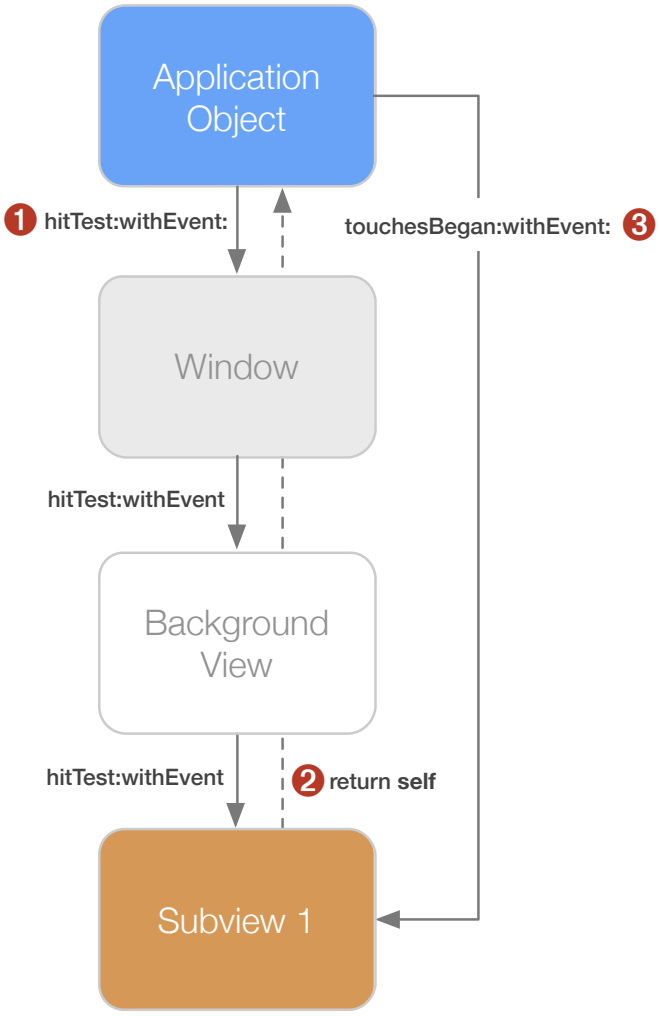
The View Hierarchy

Hit Testing

- The **application object** is responsible for dispatching event messages
- To dispatch touch events, application object:
  - Asks the window to determine which view got touched
  - Sends a touch event message directly to the touched view
- View hierarchy provides a recursive **hit test** method to determine which view got touched

```
// Recursively calls -pointInside:withEvent:
- (UIView *)hitTest:(CGPoint)point
           withEvent:(UIEvent *)event;

// Returns YES if point is in bounds
- (BOOL)pointInside:(CGPoint)point
          withEvent:(UIEvent *)event;
```





View and Layer Drawing

- Instances of **UIView** have a built-in image of **CALayer**.
  - Acts as root of a potential tree of sublayers
  - By default, provides the view's drawing and animation behavior
  - Given opportunity to redraw once per event/drawing cycle; by default, only redraws if dirty
- **CALayer** objects are capable of performing their drawing operations directly in the graphics hardware.
- Layers are capable of delegating drawing behavior to their parent view.
  - If you override a view's **drawRect:** method, it will be called automatically by the view's layer.
  - Note that the view's drawing will be done in the CPU rather than in graphics hardware.

View Geometry

UIView properties that affect geometry:

frame – origin and size of view; coordinates of origin are relative to upper left corner of superview; changing the frame's size automatically changes size of bounds

bounds – origin and size of view's drawing area; origin expresses where drawn content lies relative to the upper left corner of the drawing area; changing the bounds's size automatically changes size of frame

center – expresses where center of frame is located relative to superview's origin; changing the center automatically changes the frame's origin

transform – applies an affine transform (scale, translate, and/or rotate) to view's geometry; rotation changes frame size independent of bounds size

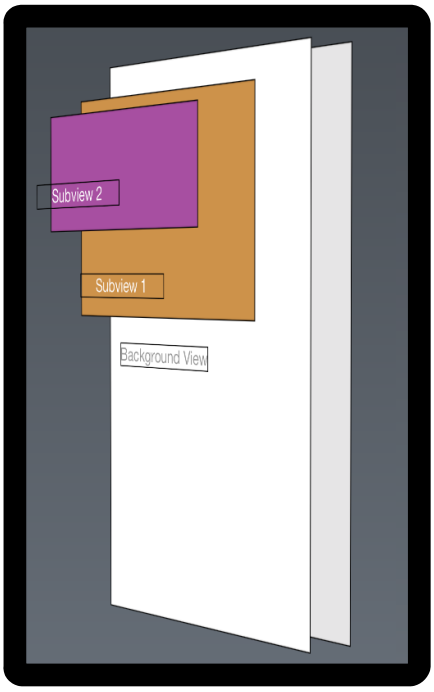
NOTE: If transform is anything other than **CGAffineTransformIdentity**, result of changing the **frame** property is undefined.

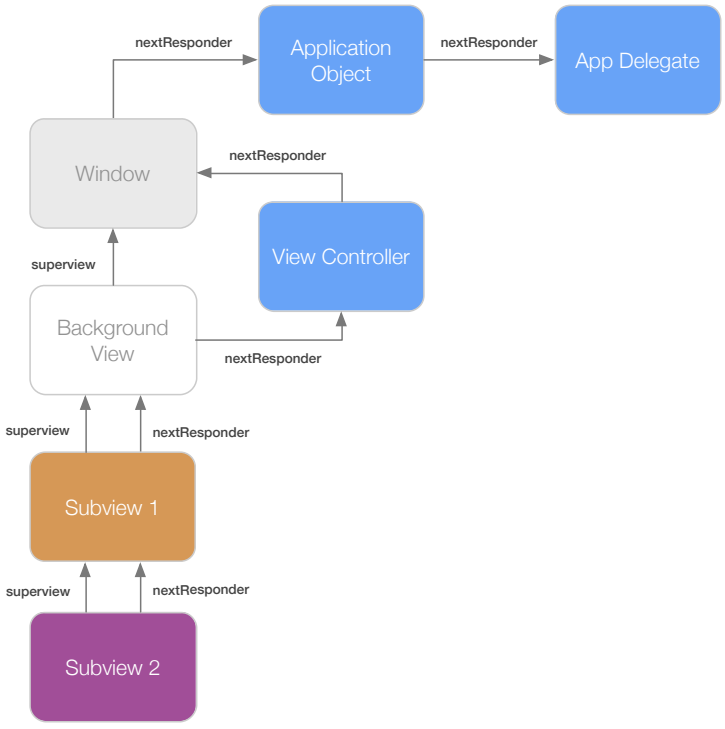
CHAPTER 3

The Responder Chain

Next Responder

- **UIResponder** defines **nextResponder** property
  - Allows responder objects to be chained together.
  - Touch and keyboard event messages automatically forwarded to next responder in the chain.
- Ordinarily, a responder that handles a given event doesn't forward it.
  - However, custom implementations can do both.





First Responder

- **UIWindow** uses its **_firstResponder** ivar to point to the view (if any) that should receive keyboard events.
- Application object dispatches keyboard event message to its **keyWindow**'s first responder.
- Most responder subclasses refuse to become first responder to avoid stealing focus from text objects.

Input Accessory View

- **UIResponder** defines properties **inputView** and **inputAccessoryView**.
- Input view is typically system keyboard.
- If input accessory view is non-**nil**, automatically presented immediately above input view whenever input view is on screen.
- Calls to the accessors for these properties automatically forwarded up the responder chain.

SECTION 5
Storyboards

CHAPTER 1

Storyboard Overview

Table Views

- **UITableViewCell**
  - Subclass of **UIView**
  - Draws an individual row
- **UITableView**
  - Inherits scrolling behavior from **UIScrollView**
  - Requests rows dynamically from data source
  - Draws table view and tracks selection
- **UITableViewController**
  - Subclass of **UIViewController**
  - Populates **view** property with instance of **UITableView**

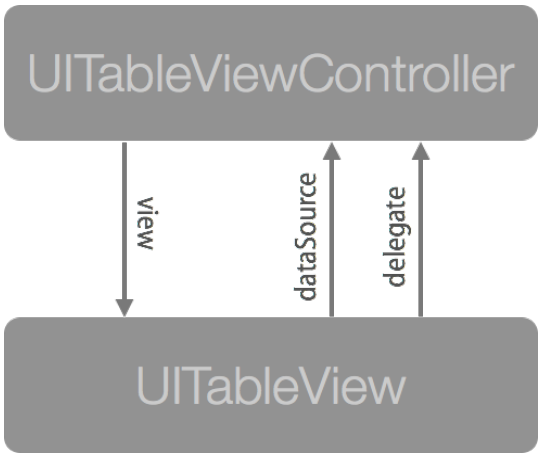
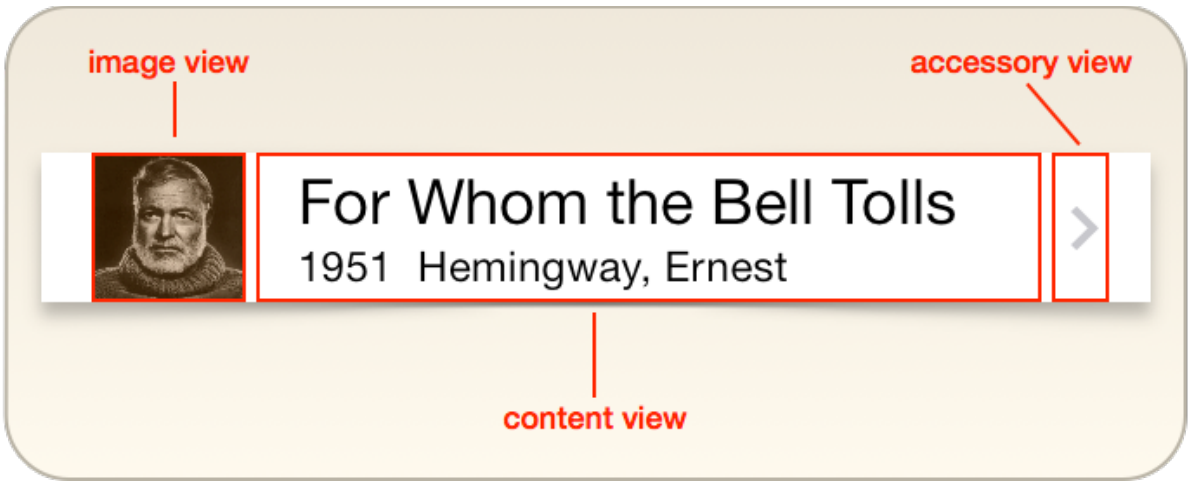


Table View Cells

content view – Freely customizable by adding subviews. Alternatively, select one of several pre-defined styles. NOTE: 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-defined styles.

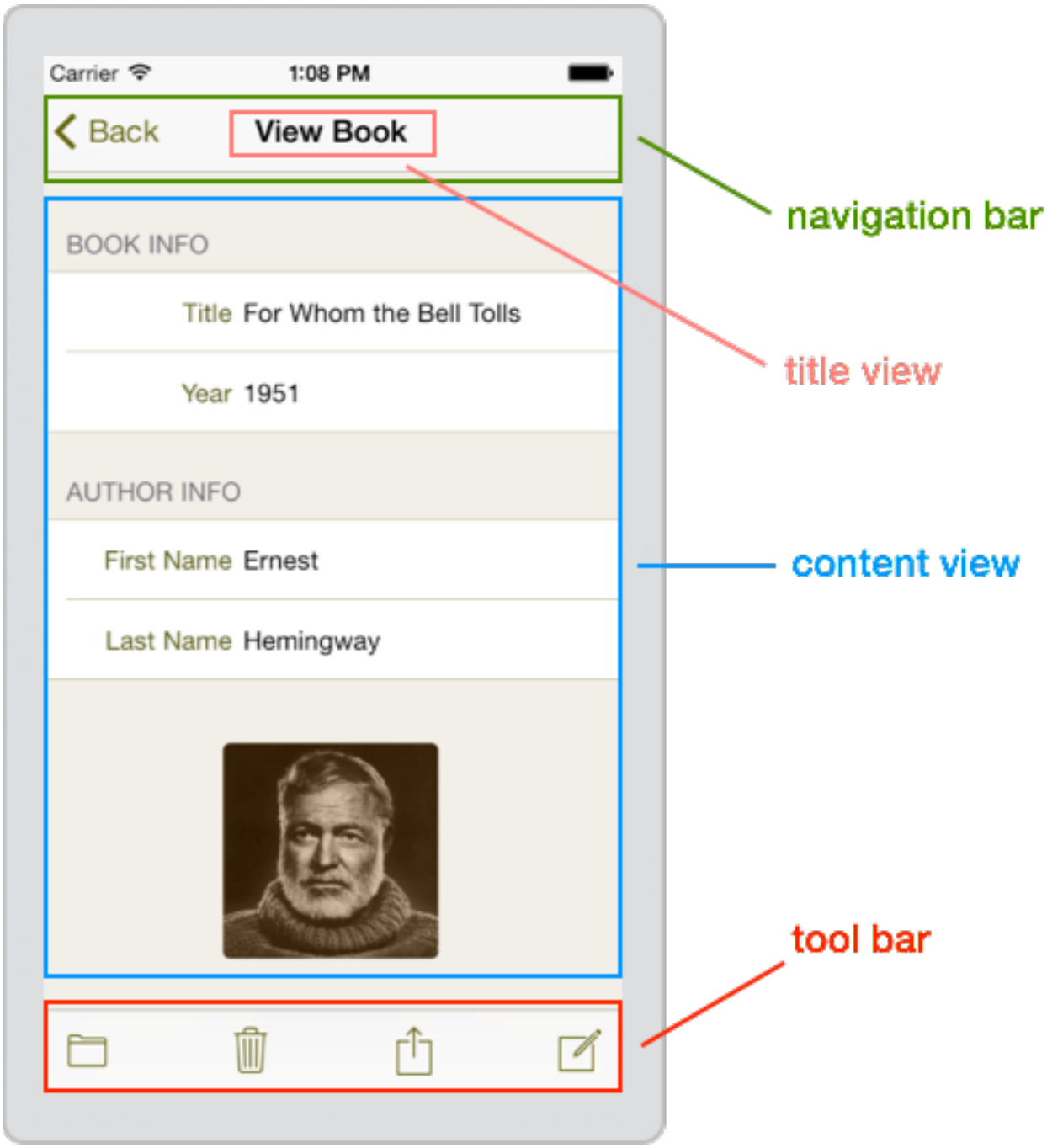


Navigation

- Navigation Controllers
- Navigation Bar Items
- Modal Segues
- Unwind Segues

Navigation Controllers

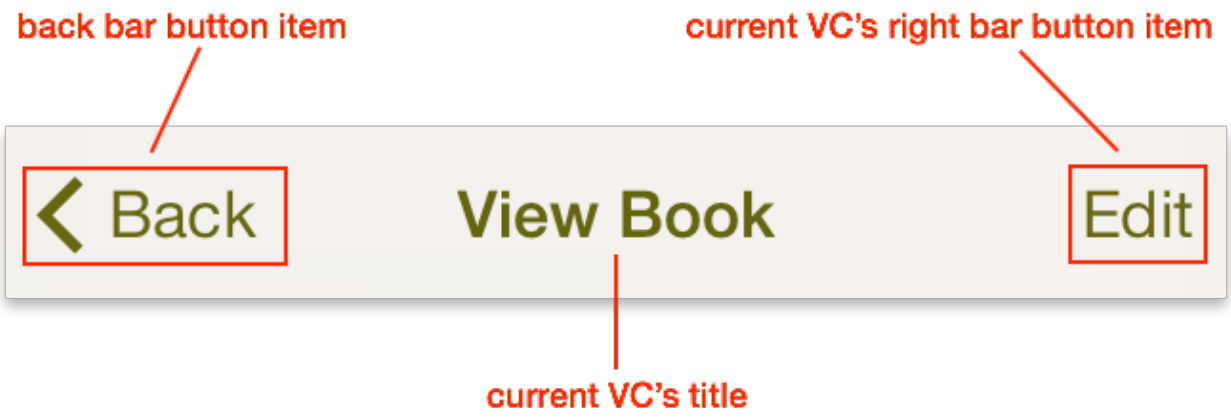
- UINavigationController has several configurable subviews:
  - **navigation bar** – Presents the top VC's navigation items.
  - **title view** – Presents top VC's title property.
  - **content view** – Presents the top VC's view.
  - **tool bar** – Presents the top VC's toolbar items.



Navigation Bar Items

- VC's **navigationItem** property returns lazily initialized instance of **UINavigationItem**.
  - Container for VC's bar button items, title, prompt, etc.
- Nav controller presents top VC's navigation items in its navigation bar.

- 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 the current VC's view into the content view.
  - Swaps toolbar items (if toolbar visible).

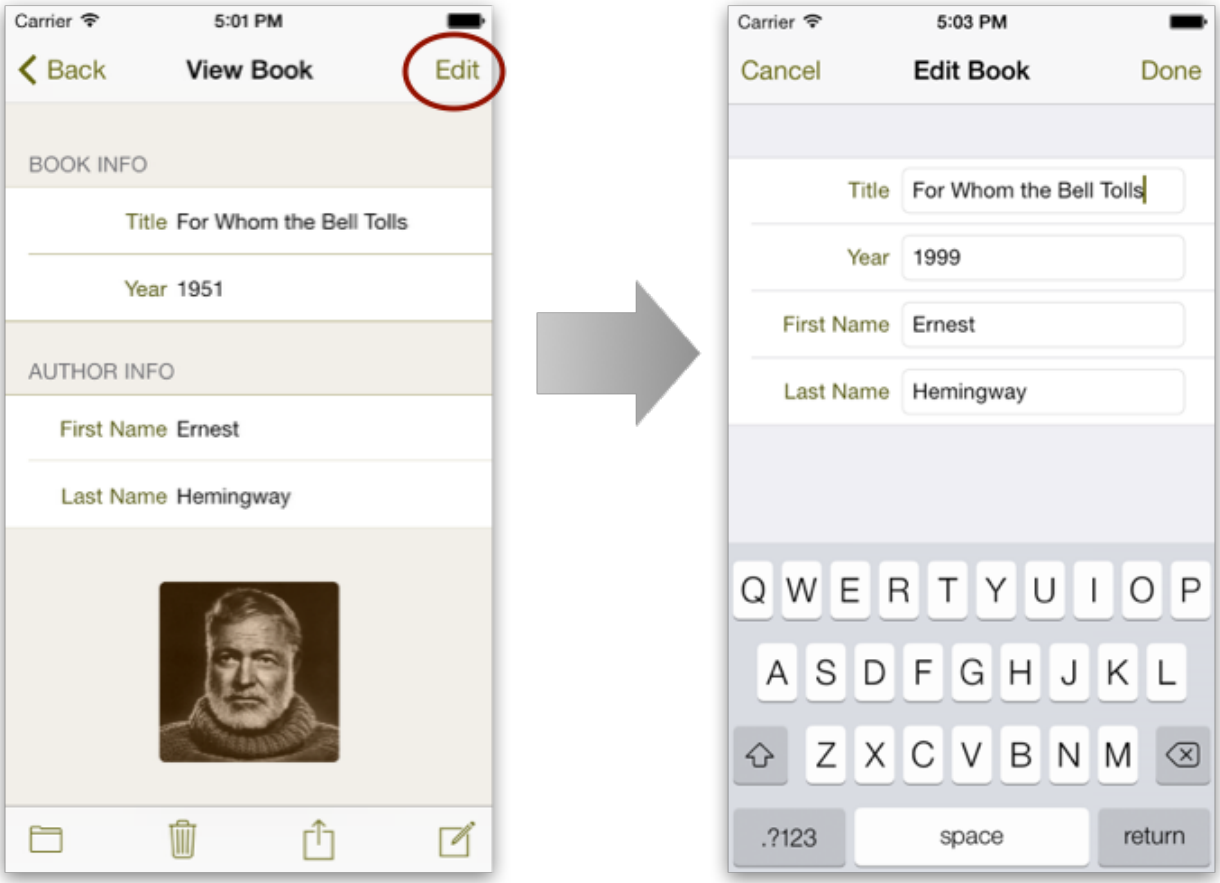


Modal Segues

- Modal segues often used to present input views.

- Related API:

```
- (void)presentViewController:(UIViewController *)viewControllerToPresent
                     animated:(BOOL)flag
                   completion:(void (^)(void))completion;

// The completion handler, if provided, will be invoked after the dismissed
// controller's viewDidDisappear: callback is invoked.
//
- (void)dismissViewControllerAnimated:(BOOL)flag
                           completion:(void (^)(void))completion;

@property(nonatomic,readonly) UIViewController *parentViewController;

// The view controller that was presented by this view controller or its nearest
ancestor.
@property(nonatomic,readonly) UIViewController *presentedViewController;

// The view controller that presented this view controller (or its farthest ancestor.)
@property(nonatomic,readonly) UIViewController *presentingViewController;
```



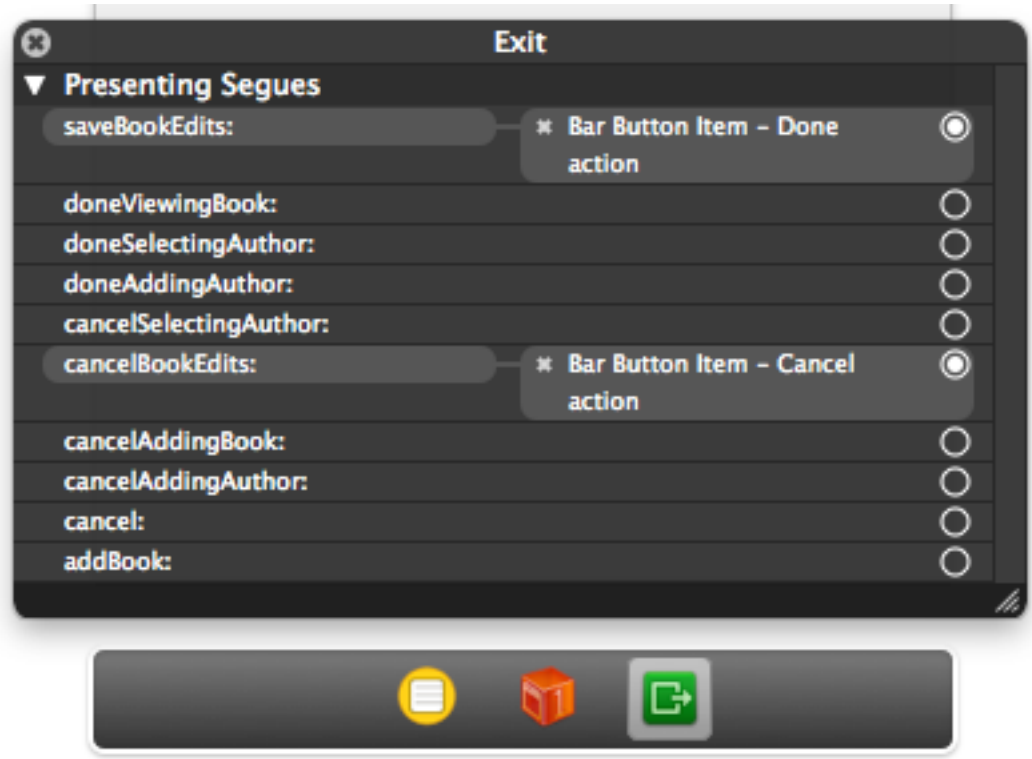
Unwind Segues

- Declare methods with the following signature:

```
- (IBAction)myUnwindSegue:(UIStoryboardSegue *)segue;
```

- Method implementation can be empty.

- All unwind segue methods in the chain of parent view controllers can be connected via the green **Exit** icon.



CHAPTER 2

Integrating Core Data

Fetched Results Controllers

- Configured with instance of **NSFetchRequest**
- Performs fetches and manages result set
  - Rows automatically grouped by section
  - Controller analyzes fetch request to pre-compute section info; caches results to avoid re-computation
  - *If caching enabled, call **+deleteCacheWithName:** before mutating fetch request*
- Controller monitors changes to managed object context
  - Automatically updates result set
  - Notifies delegate
- Automatically purges cache under memory pressure

Adopting the Delegate Protocol

- Syncing animations

```
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller
{
    [self.tableView beginUpdates];
}

- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
{
    [self.tableView endUpdates];
}
```

- Handling section-level changes

```
- (void)controller:(NSFetchedResultsController *)controller
  didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
           atIndex:(NSUInteger)sectionIndex
     forChangeType:(NSFetchedResultsChangeType)type
{
    switch(type) {
        case NSFetchedResultsChangeInsert:
            [self.tableView insertSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                          withRowAnimation:UITableViewRowAnimationFade];
            break;
        case NSFetchedResultsChangeDelete:
            [self.tableView deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                          withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}
```

- Handling row-level changes

```
- (void)controller:(NSFetchedResultsController *)controller
   didChangeObject:(id)anObject
       atIndexPath:(NSIndexPath *)indexPath
     forChangeType:(NSFetchedResultsChangeType)type
      newIndexPath:(NSIndexPath *)newIndexPath
{
    switch(type) {
        case NSFetchedResultsChangeInsert:
            [self.tableView insertRowsAtIndexPaths:@[newIndexPath]
                                  withRowAnimation:UITableViewRowAnimationFade];
            break;
        case NSFetchedResultsChangeDelete:
            [self.tableView deleteRowsAtIndexPaths:@[indexPath]
                                  withRowAnimation:UITableViewRowAnimationFade];
            break;
        case NSFetchedResultsChangeUpdate:
            [self populateCell:[self.tableView cellForRowAtIndexPath:indexPath]
                   atIndexPath:indexPath];
            break;
        case NSFetchedResultsChangeMove:
            [self.tableView deleteRowsAtIndexPaths:@[indexPath]
                                  withRowAnimation:UITableViewRowAnimationFade];
            [self.tableView insertRowsAtIndexPaths:@[newIndexPath]
                                  withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}
```

Data Source Implementation

- Providing row and section info

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return self.fetchedResultsController.sections.count;
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    return [self.fetchedResultsController.sections[section] name];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    id<NSFetchedResultsSectionInfo> info =
        self.fetchedResultsController.sections[section];
    return info.numberOfObjects;
}
```

- Populating cells

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"MyID"];
    RELBook *book = [self.fetchedResultsController objectAtIndexPath:indexPath];
    cell.textLabel.text = book.title;

    return cell;
}
```

- Handling edits

```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        id obj = [self.fetchedResultsController objectAtIndexPath:indexPath];
        [self.fetchedResultsController.managedObjectContext deleteObject:obj];
    }

    [self save:nil];
}
```

SECTION 6

Designing the Controller Layer

Data Sources

- Goals:
  - Reducing complexity of view controller layer
  - Minimizing code duplication
  - Achieving better separation of concerns
  - Fostering reuse
- Approach:
  - Move implementations of **UITableViewDataSource** and **NSFetchedResultsControllerDelegate** protocols to a separate class
  - Subclass as needed for specific usages

Separating the Data Source

- By default, **dataSource** and **delegate** properties both point to the same object.
  - Typically the view controller that owns the table view.
  - However, they're not required to point to the same object.
- Downside of implementing both protocols in a view controller:
  - *Adds to view controller size and complexity.*
  - *Makes common functionality hard to reuse.*
- Consider migrating the **UITableViewDataSource** implementation to a subclass of **NSObject**.
  - Implementation can be shared with other view controllers.
  - Data sources can be configured directly in Interface Builder.

Sample DataSource API

- Declarations from the ReadingList example project:

```
@interface RELDataSource : NSObject <UITableViewDataSource, NSFetchedResultsControllerDelegate>

@property (strong, nonatomic) IBOutlet UITableView *tableView;

- (NSString *)reuseIdentifierForCellAtIndexPath:(NSIndexPath *)indexPath;

/** Must be overridden by subclasses to provide array of NSSortDescriptor objects. */
- (NSArray *)sortDescriptors;
/** May be overridden by subclasses to provide cache name. Default is `nil`. */
- (NSString *)cacheName;
/** May be overridden by subclasses to provide entity name. Default is `nil`. */
- (NSString *)entityName;
/** May be overridden by subclasses to provide section name keypath. Default `nil`. */
- (NSString *)sectionNameKeyPath;
/** May be overridden by subclasses to provide NSPredicate object. Default `nil`. */
- (NSPredicate *)predicate;
/** May be overridden by subclasses to customize fetchedResultsController. */
- (void)configureFetchedResultsController;
@property (strong, nonatomic) NSFetchedResultsController *fetchedResultsController;

- (id)objectAtIndexPath:(NSIndexPath *)indexPath;

/** May be overridden by subclasses to customize managedObjectContext. */
- (void)configureManagedObjectContext;
@property (strong, nonatomic) NSManagedObjectContext *managedObjectContext;

/** Working with the Managed Object Context **/
- (BOOL)fetch:(NSError * __autoreleasing *)error;
- (BOOL)save:(NSError * __autoreleasing *)error;
- (void)reset;

@end
```

CHAPTER 1

Working with Multiple Storyboards

Tab Bar Controller and Child View Controllers

- Create separate storyboards containing groups of closely related scenes.
- Tab bars present a good opportunity for dividing up a larger storyboard into separate storyboards.
  - Consider creating a separate storyboard for each tab.
- Use **UIStoryboard** to load storyboards dynamically.

```
UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"MyStoryboard"
                                                     bundle:nil];

UIViewController *controller = [storyboard instantiateInitialViewController];
```

- To access a specific scene in the storyboard:

```
controller = [storyboard instantiateViewControllerWithIdentifier:@"MyVC"];
```

CHAPTER 2

Preferences and Defaults

Registering Defaults

- Use **NSUserDefaults** to register app defaults.
  - Consider externalizing to plist
- For example:

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"AppDefaults"
                                                 ofType:@"plist"];
NSMutableDictionary *defaultsDict = [NSMutableDictionary
                                    dictionaryWithContentsOfFile:path];

if (defaultsDict[RELFetchBatchSizeKey] == nil)
    defaultsDict[RELFetchBatchSizeKey] = @(RELFetchBatchSize);

if (defaultsDict[RELiTunesSearchBatchSizeKey] == nil)
    defaultsDict[RELiTunesSearchBatchSizeKey] = @(RELiTunesSearchBatchSize);

if (defaultsDict[RELUseAutolayoutNameKey] == nil)
    defaultsDict[RELUseAutolayoutNameKey] = @(RELUseAutolayout);

[[NSUserDefaults standardUserDefaults] registerDefaults:defaultsDict];
```

Storing User Preferences

- Store user preferences and runtime app configuration in **NSUserDefaults**.
- Manages in-memory key-value stores for different domains, including
  - Global domain
  - Argument domain
  - Registration domain
  - Application domain
- User domain values synced to file system periodically.
  - Send a **synchronize** message to flush to file system immediately.

CHAPTER 3

Working Across Tabs and Contexts

Posting Notifications and Registering Observers

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

Working with Multiple Contexts

- **NSManagedObjectContext** supports creation of child context.
  - Set child's **parentContext** property.
- Saving child causes it to merge changes into parent context.
  - **Important:** Does *not* write to persistent store.
- Child can have its own instance of **NSUndoManager**.
- Changes to child context don't affect parent until **save:** is called on child.

SECTION 7
Web Services

CHAPTER 1

Concurrency

Run Loops

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

Run Loop Modes

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

  Default: **NSDefaultRunLoopMode**

  Modal: **NSModalPanelRunLoopMode**

  Event Tracking: **NSEventTrackingRunLoopMode**

  Common Modes: **NSRunLoopCommonModes**
- You can also define custom run loop modes.

Multithreading

- In iOS, every thread must have its own:
  - autorelease pool;
  - run loop.
  - If you create your own threads, you're responsible for creating and managing these yourself.
- Threading code is hard to write and debug, and threads are resource intensive. Avoid creating your own threads by hand.
- Instead use **Grand Central Dispatch** or higher-level APIs that are layered on top of GCD (e.g. **NSOperation**).

Grand Central Dispatch

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

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

NSOperation

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

Custom Operations

- Custom subclasses should override the inherited **main** method to define the behavior they will carry out when executed.
  - Note: subclasses intended to be used standalone (rather than in a queue) to provide concurrent operations must override additional methods in order to correctly manage changes to internal state.

- Implementations of **main** should:
  - Swallow all exceptions.
  - Call **[self isCancelled]** periodically to check for cancellation.
  - Generally, store their result in a property, to be retrieved after execution completes.
  - May post notifications to alert interested observers of success or failure.
- **NSOperation** provides built-in support for **KVO**, so state changes are directly observable.

KVO Notifications

- If you override any of the following methods, you have to ensure your implementations are KVO-compliant.

```
- (BOOL)isCancelled;
- (BOOL)isConcurrent;
- (BOOL)isExecuting;
- (BOOL)isFinished;
- (BOOL)isReady;

- (NSArray *)dependencies;
- (NSOperationQueuePriority)queuePriority;
- (void (^)(void))completionBlock;
```

NSOperationQueue

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

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

- Managing a queue's operations:

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

- Supporting concurrent operations:

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

The URL Loading System

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

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

URL Connections

- **NSURLConnection** has three related delegate protocols:
  - **NSURLConnectionDelegate** – asynchronous callbacks sent to the delegate during authentication challenges, as well as to notify the delegate of connection failure.
  - **NSURLConnectionDataDelegate** – asynchronous callbacks sent to the delegate as the connection is downloading data.
  - **NSURLConnectionDownloadDelegate** – asynchronous sent to notify the delegate about download progress.

- Asynchronous loading:
  - Implement delegate callbacks; or
  - Use block-based API

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

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

URL Sessions

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

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

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

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

    - Provides global session via **+sharedSession** method.

- You can also create your own sessions.
- Custom sessions can be configured as background sessions.
    - Background sessions managed by iOS.
    - Can continue download when app is not running.
    - iOS will relaunch app when download completes.

Key-Value Coding

- Often used to map values from web service calls.
- Declared in **NSKeyValueCoding** informal protocol
  - Objective-C categories that add methods to **NSObject** and collection classes.
- KVC primitive methods:

```
- (id)valueForKey:(NSString *)key;
- (void)setValue:(id)value forKey:(NSString *)key;
```

- KVC uses introspection to set and get property values.
  - Capable of directly accessing instance variables
  - Uses accessor methods, if available

Value Transformers

- Subclass **NSValueTransformer** to define data transformations.
- Implementations can be uni- or bi-directional.
- Directly supported in Core Data models.

```
- (void)setAttributeValuesWithiTunesDictionary:(NSDictionary *)iTunesValues
{
    NSArray *propertyKeys = [self.class iTunesMappingDictionary].allKeys;
    NSDictionary *attributes = self.entity.attributesByName;

    for (NSString *propertyKey in propertyKeys)
    {
        NSString *iTunesKey = [self.class iTunesMappingDictionary][propertyKey];
        id value = iTunesValues[iTunesKey];

        if (value == nil) {
            continue;
        }

        NSAttributeDescription *description = attributes[propertyKey];
        NSString *targetClassName = description.attributeValueClassName;
        if (targetClassName == nil) {
            targetClassName = description.userInfo[@"targetClass"];
        }

        NSString *transformerName = RELTransformerNameForClassName(targetClassName);
        NSValueTransformer *transformer = [NSValueTransformer
                                           valueTransformerForName:transformerName];
        if (transformer != nil) {
            value = [transformer transformedValue:value];
        }

        [self setValue:value forKey:propertyKey];
    }
}
```

Reachability

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

Testing Tips

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

SECTION 8
Localization

Lab – Localization

OVERVIEW

Use Localization to configure locale specific strings, images and the app name of a new project. **Note** - this lab was written for Xcode 5.1 and Apple continues to refine how localization is implemented with storyboards - some modifications may be necessary depending on your version of Xcode.

Part 1

1. Create a new iPhone only Xcode project of type **Single View Application** named **Internationalization** using **IAL** as the class prefix (for **I**nternationalization **A**nd **L**ocalization).
    - Product Name: **Internationalization**
    - Organization Name and Company Identifier: Your choice
    - Class Prefix: **IAL**
    - Devices: **iPhone**
2. **Add** the **UI elements** to the storyboard as in the following screenshot: ScreenShot1.png
    - A **UITextField** at y position 40, the right edge aligned with the right layout guide and the default width and height.
    - A **UILabel** with its horizontal center the same as the UITextField, its left edge aligned with the left layout guide, width 140, the default height and the text **Author's Name:**
    - A **UIButton** at y position 80, width 80, centered horizontally, the title **Go** and the default height.
    - A **UILabel** at y position 125, left edge aligned with the left layout guide, right edge aligned with the right layout guide, the text centered and the default height. Under the **Attributes** inspector set the **Lines** property to **0** to support multi-line text.
    - A **UIImageView** at y position 170, width and height 95 and centered horizontally.
3. As of Xcode 5.1, **Base Localization** is on by default for projects created with a storyboard such as when using the **Single View Application** template.
4. Add a **French localization** as follows:
    - Click the **+** icon in the **Localizations** section of the project info. ScreenShot2.png

- Select **French** from the popup that appears. Click **Finish** to accept the default settings.

5. **Add images** as follows:

   - Because the **English** and **French** images will end up with the same names we add them one language at a time.
   - Create an Xcode **group** named **ImagesEnglish**.
   - Select the **ImagesEnglish** group and then **File->Add Files to "Internationalization"…** from the Xcode File menu.
   - Navigate to the filesystem location provided by your instructor and select **Hemingway.png** and **Hemingway@2x.png** in the **ImagesEnglish** folder (not the folder itself) and ensure that the files and options in the dialog match ScreenShot3.png
   - Click **Add** to add the images.

6. **Localize images** as follows:

   - Select **Hemingway.png** in the **ImagesEnglish** group in the files navigator.
   - Click **Localize…** in the file inspector in the **Localizations** section. ScreenShot5.png
   - In the window that appears, select **Base** from the popup and click **Localize.**
   - Repeat for the **Hemingway@2x.png** image, also in the **ImagesEnglish** group.

7. Add **french** images by repeating step 5 substituting **ImagesFrench** for **ImagesEnglish**.

8. Localize the **french** images by repeating step 6 using the images in the **ImagesFrench** group and **French** for the popup.

9. Configure the **UIImageView** to display an image as follows:

   - Select the **storyboard** then select the **UIImageView**.
   - In the **Attributes Inspector** of the **UIImageView** select **Hemingway.png** from the **Image** popup. Note that even though there are four Hemingway images only one is available in the popup. The iOS image loading mechanism will select the appropriate one at run time based on the device and user language selection.

10. Test

    - ScreenShot4.png shows the files inspector.
    - Inspect the images in the **ImagesEnglish** group and notice that there's no beret on his head. Inspect the images in the **ImagesFrench** group and notice the beret on his head.
    - Run the app and verify the no beret image appears.
    - Change the language to **French** as follows:

- Run the **Settings** app in the simulator.
- Navigate to **General->International->Language** and select **Français**.

- Run the app and verify that the beret image appears.

Part 2

Localize storyboard strings and position UI elements with Auto Layout.

1. ScreenShot6.png and ScreenShot7.png show the before and after for this section. Note that the **ObjectID** values and order of the UI elements may be different from yours.
2. Click the small triangle next to **Main.storyboard** to disclose its contents if necessary.
3. **Main.Storyboard (Base)** is the base version of the storyboard and is edited visually. **Main.strings (French)** is a file containing string mappings to use when a user selects **French** as their language. Each entry is a pair of lines of text. The first line is a C language style comment **/* ... */**. The second line of each pair is used for a runtime substitution.
4. Modify (translate) your **Main.strings (French)** file:
   - Change **Go** to **Aller** in the non-commented line.
   - Change **Author's Name:** to **Entrer le nom de l'auteur:** in the non-commented line.
5. Test.
   - Run the app, switching between **French** and **English** and verify that the label and button text change. Note that the line beginning with **Entrer** is truncated - that will be fixed with **Auto Layout** next.
6. Use **Auto Layout** to make the UI elements adjust their sizes and positions automatically based on their contents and the device rotation.
   - There are multiple ways to make **Auto Layout Constraints**. One way is to **control-drag** similar to creating a target/action pair for a button.
   - **Control-drag** from the **UITextField** to the **Author's Name UILabel**. A floating view will appear. Select **Center Y** by clicking it. ScreenShot8.png
   - The new colored lines that appear are because once you begin adding **constraints**, Interface Builder uses that color to warn you when the constraints don't match the positions of the UI elements and we still have work to do.
   - Select the **Author's Name UILabel** and click the **Pin** constraints button at the bottom of the storyboard editor pane (Move the cursor around and use **Tool Tips** if necessary). Add the **constraints** as in ScreenShot9.png, ScreenShot10.png, ScreenShot11.png and make sure **Update Frames** is set to **Items of New Constraints**. **Note 1** - configure **all 3 constraints** in the popover before clicking **Add Constraints**. **Note 2 -** Undo works well

when experimenting with constraints.

- Select the **UITextField**, click the **Pin Constraints** button and add **Use Standard Value constraint** to pin the right edge of the **UITextField** to its' **Superview** making sure to select **Items of NewConstraints** for **Update Frames** as above. ScreenShot12.png
- Select the **Author's Name UILabel** again and select the **Size Inspector** on the right **Utilities** Xcode column. Set both the **Horizontal** and **Vertical** values of the **Content Compression Resistance Priority** to **751**. The goal is to make sure that the values are larger than the **UITextField** values so the **UILabel** maintains its size. ScreenShot12A.png
- Test. Run the app and see that in portrait and landscape for both languages the label resizes to fit its text, the text field resizes to fill the rest of the width, and there is spacing before the label and after the text field.

7. Add additional **Auto Layout Constraints** to center the rest of the UI items.
   - Use the **shift key** to multiple select the **Go Button**, the **UIImageView**, and the **UILabel** between the two. Click the **Align** constraints button at the bottom of the storyboard editor pane and add a **Horizontal Center in Container** constraint for the 3 views making sure **None** is selected for for **Update Frames** ScreenShot13.png
   - Select only the **Go Button** and make a **pin constraint** for the **Top Edge** to the **UITextField** making sure to select **Items of NewConstraints** for **Update Frames** ScreenShot14.png
   - Select only the **UILabel** below the **Go Button** and make **3 pin constraints** to pin the **Top Edge** to the **Go Button**, the **Left Edge** to the **View** with the **Standard** value and the **Right Edge** to the **View** with the **Standard** value making sure to select **Items of NewConstraints** for **Update Frames**. ScreenShot15.png and/or ScreenShot15A.png
   - Select the **UIImageView** and make a **Top Edge pin constraint** to the **UILabel** above it making sure to select **Items of NewConstraints** for **Update Frames** ScreenShot16.png
   - Test. Run the app in both **French** and **English** in both **Portrait** and **Landscape** and verify that the newly centered elements are centered with both languages and orientations.

Part 3

Localize strings not in the storyboard.

1. Create a property named **nameTextField** connected to the **UITextField**. Remember that you can **control-drag** from the **UITextField** into the **class extension** of **IALViewController** using the **assistant** editor to create and connect the property in one step.
2. Create a property named **approvalLabel** connected to the **UILabel** below the **Go Button**.

3. Create a method named **goButtonTouched** and have the **UIButton** trigger it.

4. Create **Localizable.strings** files for localization as follows:

   - Go to the Xcode **File->New->File…** menu
   - Select **iOS . Resource . Strings File** and click **Next**.
   - Name the file **Localizable.strings** and click **Create**.
   - Select the new **Localizable.strings** file if necessary, select **Identity and Type** from the **Utilities** on the right column of Xcode and click **Localize…**
   - Select **Base** in the dialog that opens and click **Localize** to create the english strings file. ScreenShot17.png
   - Select the **Localizable.strings** file again if necessary and click **French** in the **Localization** section of the **Identity and Type** inspector to create a french strings file. ScreenShot18.png
   - Test. Navigate to your **Internationalization** project folder in the **Finder**, drill down far enough into the subfolders to verify that both the **Base.lproj** and **fr.lproj** folders contain a **Localizable.strings** file.

5. Implement the **goButtonTouched** method using **NSLocalizedString** to:

   - Fetch a localized string
   - Combine that with the name from the **nameTextField**
   - Display the combined string in the **approvalLabel**
   - Code pasted below…

6. Use **genstrings** to extract strings from your **.m** files as follows:

   - Use the **cd** command in the **Terminal application** to navigate into your **Internationalization** project folder, drilled down into the folder where an **ls** command shows **Base.lproj** among others (Note, after typing **cd** and a space in **Terminal** you can drag and drop a folder from the **Finder** into the **Terminal** to get the path to the folder.
   - Type the following: **genstrings *.m -o Base.lproj** and then the return key.
   - Type the following: **genstrings *.m -o fr.lproj** and then the return key.
   - The previous two **genstrings** commands **overwrite** the **Localizable.strings** files.
   - Note: The **genstrings** command will **overwrite** the destination file unless you add the **-a** option which tells it to append what it finds to the output file. Using -a might give you duplicate entries to delete, but has the advantage of not stomping things you've already translated.
   - Update the **French** version **Localizable.strings** file to have **Approuve** on the right side

of the = sign.

- Update the **Base** version **Localizable.strings** file to have **Approves** on the right side of the = sign.
- Test. Run the app in both languages and verify **Hemingway Approves** and **Hemingway Approuve** appear as appropriate when you enter text in the **UITextField** and tap the **UIButton**.

```
    NSString *localizedPrefix = NSLocalizedString(@"ApprovalPrefix",
@"Approval string prefix");

    self.approvalLabel.text = [NSString stringWithFormat:@"%@:
Hemingway %@", self.nameTextField.text, localizedPrefix];

    [self.view endEditing:YES];
```

Part 4

Localize the app name displayed on the springboard. **CFBundleDisplayName** is the key used by the **Internationalization-info.plist** file for display on the springboard. Changing the name to display is as simple as adding entries to the localized files

1. Select **InfoPlist.strings (English)** and add a **CFBundleDisplayName** entry (Pasted below).
2. Select **InfoPlist.strings (French)** and add a **CFBundleDisplayName** entry.
3. Test. Verify the springboard shows the correct app name for each language.

English: `CFBundleDisplayName = "La La";`

French: `CFBundleDisplayName = "Ooh La La";`

Bonus Lab 1

Use the **ibtool** command to extract strings from an already localized storyboard after modifications. Xcode does not automatically update the **Main.strings (French)** storyboard localization file. The **ibtool** command can be used to extract strings from your base storyboard that you can then paste into each of the storyboard strings files you have localized.

1. Use the **cd** command in the **Terminal application** to navigate into your **Internationalization** project folder, drilled down into the **Base.lproj** folder where an **ls** command shows **Main.storyboard** among others.
2. Type: **ibtool Main.storyboard --generate-strings-file newStrings.strings** to create a file

named **newStrings.strings** containing the localizable strings from **Main.storyboard.** JONATHAN - it should be dash dash generate-strings-file but Scrivener changes it to a single dash…

3. Type: **cat newStrings.strings** to view the file that **ibtool** created. If there were new entries to localize you would paste them into each of your **Main.strings** files and localize them.

Base Storyboard

- When you localize a storyboard, all of the strings (button titles, label contents, etc) are extracted into a file for localization. Therefore, it's easiest to lay out as much of your interface as possible before localization. In the lab we will also see a way to deal with items added afterwards.
- Next item
- Next item
  - Provide access to the *appearance proxy* for a given view class.
- To customize a class's appearance globally, send the class message **appearance** to obtain the appearance proxy.

```
+ (id)appearance;
```

  - You can then send use the following method to customize its appearance:

```
+ (id)appearanceWhenContainedIn:
      (Class<UIAppearanceContainer>)ContainerClass, ...
```

Overview

When designing our view hierarchies, our layout must adapt to both internal and external changes. There are three main approaches: programmatically adjust the layout, use Autoresizing Masks or use Auto Layout. Of these, Auto Layout is the most powerful; however, it also has the steepest learning curve.

Learning Auto Layout involves learning two different skills:

- Mastering the logic puzzle behind constraint-based layout
- Becoming proficient with Auto Layout's API

Topics/Goals

By the end of this module you will gain a better understanding of the problem that Auto Layout is attempting to solve, the logic behind constraint-based layouts, and a working understanding of the API. Specifically, we will cover the following:

- The difference between internal and external changes
- The relative strengths and weaknesses of programmatically laying out views, Autoresizing Masks and Auto Layout
- The logic behind Auto Layout's constraints
- Using `NSLayoutConstraint` to programmatically create constraint
- Creating constraints in Interface Builder
- Combining Auto Layout constraints and Core Animation

Glossary

Term	Definition
External Change	A change to our view's `bounds`. External changes can be caused by auto rotation, or by the appearance of in-call or audio recording status bars.
Internal Change	A change in the size of our view due to a change in its content. Internal changes are also caused when the user changes the size of the fonts used by dynamic type.
Constraint	A linear equation that describes one part of our layout. Each constraint consists of two attributes, a constant and a multiplier. The entire set of constraints should uniquely define both the size and position of all the views in our view hierarchy.
Attribute	The part of the view we are using in our constraint. This can be any of the edges (top, bottom, left or right), as well as other options like center x, center y, baseline, leading edge and trailing edge.
Relationship	Determines how the two attributes are related in our constraint. This can be less than and equal, equal or greater than and equal.
Priority	Each constraint has a priority, which is represented as an integer between 1 and 1000. Constraints with a priority of 1000 are required. All others are optional. The engine will attempt to satisfy the constraints in descending priority order. If it cannot satisfy an optional constraint, that constraint is simply ignored.

Ambiguous Layout	The set of constraints have more than one possible solution. Either we don't have enough constraints to uniquely specify both the location and size of all our views, or we have two or more conflicting, optional constraints with the same priority, and Auto Layout does not know how to break the tie.
Conflicting Layout	The system cannot simultaneously satisfy two or more required constraints. Auto Layout throws an exception whenever it finds a conflicting layout. However, it also catches that exception, and simply spews debug info into the console.

Responding to Change

Our application's layout must respond to two different types of changes:

External Change: *Our view's bounds change, forcing us to layout its subviews*

- Rotating the user interface
- Adapting the UI for both 3.5- and 4-inch screens
- Active Call and Audio Recording bars

Internal Change: *The size of our content changes, forcing us to relayout the view*

- Translating labels into different languages
- The user adjusts the font size using dynamic type
- Supporting iOS 6 and iOS 7 controls

Three Approaches

- Programmatically Layout Our Views
- Use Autoresizing Masks
  - External Changes Only!
- Use Auto Layout
  - *Most Powerful*
  - *Most Complex*
  - ***Spoiler Alert: Use This Approach!***

Programatic Approach

```
-[UIView layoutSubviews]

-[UIViewController viewWillLayoutSubviews]

-[UIViewController viewDidLayoutSubviews]

-[UIViewController
  willRotateToInterfaceOrientation:duration:]

-[UIViewController
  didRotateToInterfaceOrientation:duration:]

-[UIViewController
  willRotateToInterfaceOrientation:duration:]

-[UIView setNeedsLayout]
```

Pros and Cons

Advantages

- Easy to understand
- No limits on how we change the layout

Disadvantages

- Extremely tedious to write
- Error prone
- Difficult to maintain
- Does not scale well

- Automates the response to external changes
- Defines how a view responds when its superview's bounds change
- Six Parameters:
  - *Top margin*
  - *Height*
  - *Bottom margin*
  - *Left margin*
  - *Width*
  - *Right margin.*
- Each parameter may be either **fixed** or **flexible**.

Basic Rules

- Fixed parameters do not change
- Flexible parameters shrink and grow as the superview's bounds change
- Left margin + width + right margin will always equal the bound's width
- Top margin + height + bottom margin will equal the bound's height

Advanced Rules

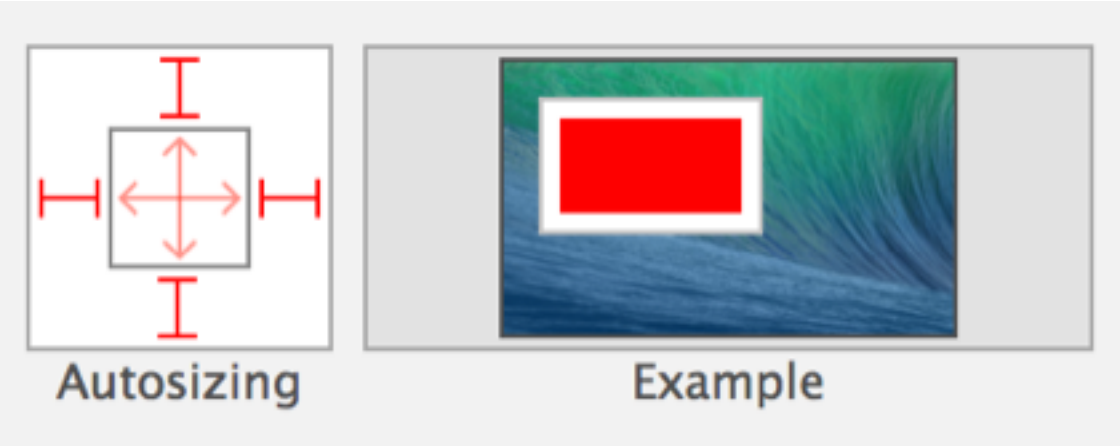
- Works best when there's only one flexible parameter per column and row
- If you have more than one flexible parameter for a given dimension, the changes will be split between them
- If you set all three to fixed, the size parameter will be ignored, and the height (or width) will change
- By default, views have fixed top and left margins, fixed height and width, and flexible bottom and right margins.

Programatically Setting the Mask

Set **UIView**'s **autoresizingMask** property to a bit mask value using the following constants:

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
    UIViewAutoresizingNone                 = 0,
    UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
    UIViewAutoresizingFlexibleWidth        = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
    UIViewAutoresizingFlexibleHeight       = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5
};
```

Setting the Mask in Interface Builder



- Use the Autosizing tool in the Size inspector
- Only available when Auto Layout is turned off
- Auto Layout is turned on by default for all nibs and storyboards

Auto Layout

When using Auto Layout,

never specify a view's geometry using

`frame`, `bounds` or `center`!

Use Constraints Instead

Constraint as Linear Equation

`Attribute_1 = Multiplier * Attribute_2 + Constant`

- Each constraint can be expressed using the linear equation shown above
- Auto Layout calculates the size and position of all our views by solving the set of linear equations
- "=" defines an equality, not assignment!

Attribute Types

- Left Edge, Right Edge, Top Edge, Bottom Edge
- Leading Edge, Trailing Edge
- Width, Height
- CenterX, CenterY
- Baseline
- Not An Attribute

Examples

Setting a constant height:

```
View's Height = 0.0 * Not An Attribute + 40.0;
```

Setting a fixed distance between two buttons:

```
Button_2's Leading Edge = 1.0 * Button_1's Trailing Edge + 8.0;
```

Give two buttons the same width:

```
Button_2's Width = 1.0 * Button_1's Width + 0.0;
```

Center a view in its superview:

```
View's CenterX = 1.0 * Superview's CenterX + 0.0;

View's CenterY = 1.0 * Superview's CenterY + 0.0;
```

Give a view a constant aspect ratio:

```
View's Height = 2.0 * View's Width + 0.0;
```

Non-Ambiguous, Non-Conflicting Layouts

- Our set of linear equations must have **one and only one possible solution**
- We must explicitly or implicitly define the following:
  - *Height*
  - *Width*
  - *X-coordinate*
  - *Y-coordinate*
- In theory this requires **two constraints per axis**

I Lied

There are three reasons why we may need a different number of constraints:

1. Constraints can also represent inequalities
    - *In general it requires at least two inequalities to replace a single equality*
3. Not all constraints are required.
    - *Optional constraints can have priorities*
4. Views may have an intrinsic size
    - *The intrinsic size is represented using optional constraints that define the view's height and width*

Inequalities

We have three options for the relationship between the left-hand side and right-hand side of our equation:

- Less than or equal
- Equal
- Greater than or equal

Priorities

- Constraints have a priority between 1 and 1,000
- Constraints with a priority of 1,000 are **required**
- All other constraints are **optional**
- Auto Layout will attempt to satisfy all the optional constraints, starting with the highest priority
  - *It will skip any optional constraints that it cannot satisfy*
  - *Even if it skips one optional constraint, it will continue trying to satisfy lower-priority optional constraints*

Intrinsic Size

- UI elements may have a natural size based on their content
- The intrinsic size can define the view's height, width or both
- These are defined as pairs of inequalities
  - *This lets us specify minimum or maximum sizes based on the content*

Content Hugging and Compression Resistance

Compression Resistance:

```
View's Height >= 0.0 * Not An Attribute +
View's Intrinsic Height;

View's Width >= 0.0 * Not An Attribute +
View's Intrinsic Width;
```

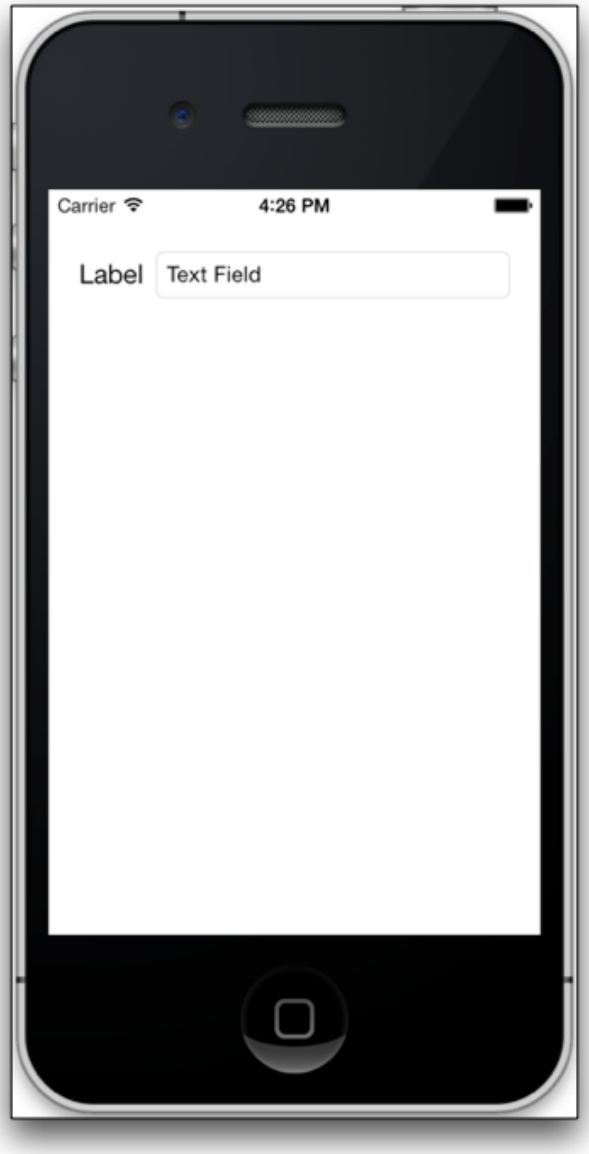
Content Hugging:

```
View's Height <= 0.0 * Not An Attribute +
View's Intrinsic Height;

View's Width <= 0.0 * Not An Attribute +
View's Intrinsic Width;
```

Auto Layout Errors

There are three common types of errors when using Auto Layout:

1. Ambiguous layout
    - *more than one solution*
2. Conflicts
    - *cannot satisfy all the required constraints*
3. Error in our logic
    - *we have a "bug" in our constraints*



1. How do we want this to resize when the view rotates?
2. What horizontal constraints do we need?
3. What vertical constraints do we need?

1. How do we want this to resize when the view rotates?

- Label's size and the margin between it's leading edge and the superview should remain the same.
- The margin between Text View's trailing edge and the superview should remain the same.
- The distance between the label and the text view should remain the same.
- Text View should stretch to fill the space.
- Both of their heights and the distance relative to the status bar should remain the same.
- The Baselines should remain aligned.

2. What horizontal constraints do we need?

- Label's Leading Edge = 1.0 * View's Leading Edge + 20.0
- Text View's Leading Edge = 1.0 * Label's Trailing Edge + 8.0
- Text View's Trailing Edge = 1.0 * View's Trailing Edge - 20.0
- We can use the Label and Text View's intrinsic size to define their widths; however, we must set the label's horizontal content hugging and compression resistance priorities higher than the text view's to make sure the text view shrinks and grows as our bounds change.

3. What vertical constraints do we need?

- Text View's Top = 1.0 * Top Layout Guide + 20.0
- Label's Baseline = 1.0 * Text View's Baseline + 0.0
- Again, we can use the view's intrinsic size for the heights.

Constraints

- Constraints are instances of NSLayoutConstraint
- Instantiate by calling:

```
+[NSLayoutConstraint
  constraintWithItem:
  attribute:
  relatedBy:
  toItem:
  attribute:
  multiplier:
  constant:]
```

- This is a direct translation of the linear equation we used in the last section

Items

- The items may either be a **UIView** instance or **UIViewController.topLayoutGuide** or **UIViewController.bottomLayoutGuide**
- Top layout guide defines the top of the controller's scene, just below any bars (status bar, navigation bar, etc.)
- Bottom layout guide defines the bottom of the controller's scene, just above any bars (tab bar, tool bar, etc.)
- **Currently there is a bug with the bottom layout guide**

Attributes

```
enum {
    NSLayoutAttributeLeft = 1,
    NSLayoutAttributeRight,
    NSLayoutAttributeTop,
    NSLayoutAttributeBottom,
    NSLayoutAttributeLeading,
    NSLayoutAttributeTrailing,
    NSLayoutAttributeWidth,
    NSLayoutAttributeHeight,
    NSLayoutAttributeCenterX,
    NSLayoutAttributeCenterY,
    NSLayoutAttributeBaseline,

    NSLayoutAttributeNotAnAttribute = 0
};
typedef NSInteger NSLayoutAttribute;
```

Relationship

```
enum {
    NSLayoutRelationLessThanOrEqual = -1,
    NSLayoutRelationEqual = 0,
    NSLayoutRelationGreaterThanOrEqual = 1,
};
typedef NSInteger NSLayoutRelation;
```

Multiplier and Constant

- These are both simply **CGFloat** values

Default Constraints

- Auto Layout automatically creates a set of default constraints based on the view's Autoresizing Mask
- As long as a view uses the default constraints, we can continue to modify its **frame**, **bounds** and **center**
- To disable the default constraints, call

-[UIView setTranslatesAutoresizingMaskIntoConstraints:] and pass **NO**

- Interface Builder does this automatically whenever we add constraints to our view hierarchy

Adding Constraints

- Add constraints to a view by calling **-[UIView addConstraint:]**
  - *Must be called on the nearest ancestor of both views in the view hierarchy*
  - *Specifically, the views in the constraint must either be the receiving view or subviews of the receiving view*
- We can also remove constraints by calling **-[UIView removeConstraint:]**

Intrinsic Size

- To set the intrinsic size in our views, override -[UIView intrinsicContentSize]
- This should return a CGSize
- By default UIView returns {UIViewNoIntrinsicMetric, UIViewNoIntrinsicMetric}
- If we need to recalculate the intrinsic size, call

-[UIView invalidateIntrinsicContentSize]

Methods for setting the priorities

Methods for setting the priorities:

- `-[UIView setContentCompressionResistancePriority:forAxis:]`
- `-[UIView setContentHuggingPriority:forAxis:]`

UIKit provides several handy constants:

- `UILayoutPriorityRequired = 1000`
- `UILayoutPriorityDefaultHigh = 750`
- `UILayoutPriorityDefaultLow = 250`
- `UILayoutPriorityFittingSizeLevel = 50`

Visual Formatting API

- Uses an ASCII-art style syntax to define constraints
- Not as powerful as the low-level API
- Let's us create multiple constraints with one method call
- Can greatly simplify your constraint creation code
- However, the compiler cannot check our syntax
    - *All typos end up becoming runtime errors*
- For more information, see Apple's *Auto Layout Guide: Visual Format Language*

Debugging Constraints

- Auto Layout throws an exception when there are conflicting constraints
  - *It also catches this error, spews out some data to the console, then attempts to "fix" the conflict by ignoring one or more constraints*
- We don't get any notification about ambiguous constraints
  - *Views either don't appear, don't appear correctly or don't resize as expected*
  - *Debugging Methods:*

```
-[UIView hasAmbiguousLayout]

-[UIView exerciseAmbiguityInLayout]

-[UIView constraintsAffectingLayoutForAxis:]
```

Auto Layout Lab 1

1. Open the Auto Layout Lab 1 project. Run the application. Note how the views do not adjust when rotated into a landscape orientation.

2. Open `KEWLCoolController.m`. Modify the code to use Auto Layout Constraints.

    - Remove all references to `frame`, `bounds` or `center`.

    - Disable the implicit constraints for the `contentView`, `controlView`, `textField` and `addButton`.

    - Add Constraints to non-ambiguously define the location and size of the `contentView`, `controlView`, `textField` and `addButton`.

Notes:

- It's often easiest to get part of the interface working, then add other elements. Start with the content and control view. Once they are working, add the controls themselves. *You may have to temporarily add an extra constraint to set the height of the control view—its height can later be calculated based on its contents.*

- Use `topLayoutGuide` and `bottomLayoutGuide` where appropriate.

- Use 8.0 points between two controls and 20.0 points between a control and its containing view.

- Don't add constraints to the cool views. We want to be able to modify their frame when we drag them around the screen—therefore, they must use implicit constraints.

- Use the textField and addButton's intrinsic size.

    - Set the `textField` and `addButton`'s content hugging and compression resistance priorities so that they have a fixed vertical height.

    - `textField`'s width should resize while the `addButton`'s width remains

constant.

Previously, in Xcode 4

Interface Builder tried to be too helpful

- Automatically generated a full set of constraints whenever we added a view
- Wouldn't let us create ambiguous or conflicting layouts
- Worked great for simple layouts--but fell apart for anything complex

Problems with this approach

- Often difficult to set the constraints we wanted
    - *Modifying the default constraints often involved moving through a temporarily invalid layout before reaching our goal*
- Any modifications to the layout might cause IB to regenerate a new set of default constraints

Xcode 5

- No longer tries to do anything for us
  - ***And that's a good thing!***
  - *Must manually set all constraints*
  - *Must reposition and resize the view after adding constraints*
- Provides a great set of tools for setting constraints
- Provides excellent feedback on our constraints

Visual Feedback

Color Codes

- Blue lines indicate non-ambiguous, non-conflicting constraints
- Orange lines indicate ambiguous layouts
- Red lines indicate conflicting layouts

Line Style

- Solid lines indicate required constraints
- Dotted lines indicate optional constraints
- Dotted orange rectangles usually indicate the view's correct position according to the current set of constraints

Interface Builder Demo

- Align, Pin, Resolve Auto Layout Issues and Resizing Behavior tools
- Editor menu (Align, Resolve Auto Layout Issues, Pin and Canvas)
- Size and Attribute Inspectors
- Top and Bottom Layout Guides
- Viewing Warnings and Errors

Best Practices

- Use the Pin and Align tools instead of "drawing" constraints
- When using the Pin tool, use the drop-down menus to select the proper view and set the standard size for the constraints
- Layout a view's constraints first, and then update its frame
- When working on complex layouts, build and test iteratively
    - You may need to use temporary constraints just to get the initial views to work right. You can replace these with actual constraints as you work through all the views.
- Use the Top and Bottom Layout Guides instead of the top and bottom of the superview.

Auto Layout Lab 2

1. Open the Auto Layout Lab 2 Project.
2. Open `MainStoryboard.storyboard`.
3. Add Auto Layout Constraints in Interface Builder
    - These should be the same constraints we added in Auto Layout Lab 1. Use that lab's solution as a guide.
    - Again, you may want to start with the Content and Control views. Once they are working, add the other constraints.

Don't Cross The Streams

- Core Animation (and UIKit Dynamics) move views by modifying their **frame** or **center**
- We should never modify the **frame** or **center**, unless we're using default constraints
- Therefor we should avoid mixing both custom constraints and Core Animation

Sane Options

- Just use the default constraints and animate the **center**, **bounds** or **frame** as normal.
- Alternatively, animate changes to the constraints themselves
  - Usually must store the constraints in outlets or properties
  - Modify the constraints constant value, or replace the constraint entirely
  - Then call **-[UIView layoutIfNeeded]** inside the animation block
  - This will animate the changes made to the constraints

Auto Layout Bonus Lab

Copy the solution from Auto Layout Lab 2 and modify its cool views.

- Give the cool view class an intrinsic size.
- Disable the implicit constraints for the cool views.
- Set the cool view's position using Auto Layout constraints.
- When you move or animate the view, update the constraint.

Auto Layout May

- When we deal with scroll views, we have two sizes
  - The size of the view's bounds
  - The size of the view's content
- Constraints between the scroll view and external views will set the view's location and its bounds size
- Constraints between the scroll view and its subviews will set the view's content size
- The only exception is the scroll view's height and width attributes, which will use the scroll view's bounds size
- We need to provide additional constraints to unambiguously specify the scroll view's location, its bounds size and its content size.

Hints for Managing Scroll Views

- Add a content view to the scroll view
    - Constrain the content view's width equal to the scroll view's width
    - Constrain the content view's height to be greater than or equal to the scroll view's height
    - Constrain the content view's height to be less than or equal to the scroll view's height, but give this constraint a low priority
- Add all of the scroll view's content to the content view
    - Constrain all its contents to the content view
    - Don't try to constrain the scroll view's content to items outside the content View. This may cause the items to have a fixed position relative to objects outside the scroll view, making them appear to "float" above the scrolling content.