

2つ目のiOSアプリケーション： ストーリーボード

目次

2つ目のiOSアプリケーションの作成について 4

At a Glance 5

モデルレイヤの設計と実装 5

マスタシーンと詳細シーンの設計と実装 5

新規シーンの作成 6

問題の解決と次のステップの考察 6

関連項目 6

入門 8

プロジェクトの新規作成 8

デフォルトプロジェクトのビルドと実行 10

ストーリーボードとそのシーンの調査 13

まとめ 16

モデルレイヤの設計 17

データユニットの決定とデータオブジェクトクラスの作成 17

Data Controllerクラスの作成 22

まとめ 28

マスタシーンの設計 32

Master View Controllerシーンの設計 32

Master View Controllerの実装ファイルから不要箇所を削除 35

Master View Controllerの実装 37

アプリケーションデリゲートでのアプリケーションの設定 39

まとめ 41

詳細シーンの情報の表示 44

Detail View Controllerコードの編集 44

詳細シーンの設計 47

詳細シーンへのデータ送信 53

まとめ 54

新規項目の追加の有効化 57

新しいシーン用のファイルの作成 57

シーン追加のUIの設計 60
UI要素用のアクションとアウトレットの作成 64
ユーザの入力の取得 70
マスタシーンの編集とシーン追加への接続 72
まとめ 75

トラブルシューティング 77

コードおよびコンパイラの警告 77
ストーリーボード項目と接続 77
デリゲートメソッド名 78

次のステップ 79

ユーザインターフェイスとユーザ体験の改善 79
機能の追加 79
追加の改善 80

コードリスト 82

モデルレイヤのファイル 82
 BirdSighting.h 82
 BirdSighting.m 82
 BirdSightingDataController.h 83
 BirdSightingDataController.m 84
Master View Controllerとアプリケーションデリゲートのファイル 85
 BirdsMasterViewController.h 85
 BirdsMasterViewController.m 86
 BirdsAppDelegate.m 88
Detail View Controllerのファイル 89
 BirdsDetailViewController.h 89
 BirdsDetailViewController.m 89
Add Scene View Controllerのファイル 91
 AddSightingViewController.h 91
 AddSightingViewController.m 91

書類の改訂履歴 93

2つ目のiOSアプリケーションの作成について

「2つ目のiOSアプリケーション：Storyboards」は、ストーリーボードを使用して、ユーザが、マスタリスト内の各項目の詳細を表示したり、リストに新しい項目を追加したりできる、ナビゲーションベースのアプリケーションを作成する方法の説明になっています。完成したアプリケーションは、次のようになります。



このチュートリアルの手順を完了すると、下記の作業を行うための知識が身につきます。

- アプリケーション用のデータの表示と管理を行うモデルレイヤの設計
- ストーリーボードでのシーンとセグエの新規作成
- シーンとの間でのデータの受け渡しと検索

このチュートリアルを最大限に活用するためには、一般的なiOSアプリケーションプログラミング、特にObjective-C言語に多少慣れている必要があります。iOSアプリケーションを作成したことがない場合は、このチュートリアルを始める前に『*Start Developing iOS Apps Today*』を読み、必要ならば『*Your First iOS App*』にもざっと目を通してください。

At a Glance

「2つ目のiOSアプリケーション：Storyboards」は、「最初のiOSアプリケーション」（『*Start Developing iOS Apps Today*』の一部）で習得した知識に基づいてビルドするため、ストーリーボードが備える強力な機能にさらに習熟できます。また、アプリケーションでTable Viewを活用する方法もいくつか習得できます。

モデルレイヤの設計と実装

しっかりした設計のアプリケーションでは、アプリケーションが管理するデータオブジェクトと、それらのデータオブジェクトの管理の両方を備えたモデルレイヤが定義されています。このチュートリアルアプリケーションのモデルレイヤは非常に単純なので、このモデルレイヤの設計と実装を通じて、より複雑なアプリケーションに応用可能な概念を身につけることができます。

関連する章 “モデルレイヤの設計”（17 ページ）

マスタシーンと詳細シーンの設計と実装

このチュートリアルアプリケーションは、マスタと詳細が対になった形式に基づいています。この形式では、ユーザは、マスタリスト内の項目を選択して、その項目に関する詳細を2つ目の画面に表示します。「メール(Mail)」や「設定(Settings)」を含めて、多くのiOSアプリケーションはマスタと詳細が対になった形式に基づいています。このチュートリアルでは、ストーリーボード使用すると、いかに容易にこの形式のアプリケーションを作成して、表示したいデータを詳細画面に送信できるかを示します。

関連する章 [“マスタシーンの設計”](#) (32 ページ) と [“詳細シーンの情報の表示”](#) (44 ページ)

新規シーンの作成

Xcodeテンプレートの多くでは、デフォルトでストーリーボードに1つ以上のシーンが配置されていますが、必要に応じて新しいシーンを追加することもできます。このチュートリアルでは、シーンを追加してNavigation Controllerに埋め込んで、マスタリストに加えたい新しい情報を入力する手段をユーザに提供できるようにします。また、デリゲーションを使用してシーンから情報を取得する方法も学びます。

関連する章 [“新規項目の追加の有効化”](#) (57 ページ)

問題の解決と次のステップの考察

このチュートリアルでアプリケーションを作成する過程では、解決法がわからない問題に遭遇するかも知れません。「2つ目のiOSアプリケーション: Storyboards」には、自分のコードと比較できる完全なコードに加えて、問題解決のためのアドバイスもいくつか記載してあります。

iOSアプリケーション開発に関する知識を増やす方法はたくさんあります。このチュートリアルでは、作成したアプリケーションを改良し、新しいスキルを身につける方法をいくつか提案します。

関連する章 [“トラブルシューティング”](#) (77 ページ)、[“コードリスト”](#) (82 ページ)、[“次のステップ”](#) (79 ページ)

関連項目

[“次のステップ”](#) (79 ページ) で提案するドキュメント以外にも、優れたiOSアプリケーションの開発の参考になる文献はたくさんあります。

- iOSアプリケーションのユーザインターフェイスとユーザ体験の設計に関して推奨する方法については、『*iOS Human Interface Guidelines*』を参照してください。
- フル機能のiOSアプリケーションの作成に関する包括的なガイドについては、『*iOS App Programming Guide*』を参照してください。
- View Controllerの詳細については、『*View Controller Programming Guide for iOS*』を参照してください。
- Xcodeの多数の機能の詳細については、『*Xcode 4 User Guide*』を参照してください。

- 自分のアプリケーションをApp Storeに送る準備をする際に実行する必要がある作業については、『*Developing for the App Store*』を参照してください。

入門

このチュートリアルでiOSアプリケーションを作成するには、Xcode 4.3以降、およびiOS 5.0以降用のiOS SDKが必要です。MacにXcodeをインストールすると、iOSプラットフォームのプログラミングインターフェイスを備えたiOS SDKも入手することになります。

この章では、新しいXcodeプロジェクトを作成し、そのプロジェクトが作成するデフォルトアプリケーションをビルドおよび実行し、ストーリーボードが、ユーザインターフェイス、およびアプリケーションの動作の一部を定義する仕組みを学びます。

プロジェクトの新規作成

このチュートリアルのアプリケーションは、野鳥観察の名所のリストを表示します。ユーザが特定の名所を選択すると、その詳細が新しい画面に表示されます。ユーザは、新しい名所に関する詳細を入力することで、リストに追加することもできます。

項目のマスタリストから1つのアイテムの詳細ビューへのトランジションは、iOSアプリケーションの一般的なデザインパターンの一部です。多くのiOSアプリケーションでは、ユーザが、徐々に一連のより詳細な画面に移行することで、データの階層を1つドリルダウンできます。たとえば、「メール(Mail)」では、特定のメッセージが表示されるまで、アカウントのマスタリストから徐々に一連のより詳細な画面にドリルダウンできます。

Xcodeは「Master-Detail」テンプレートを備えているため、ユーザがデータ階層内を移動できるアプリケーションを簡単に構築できます。

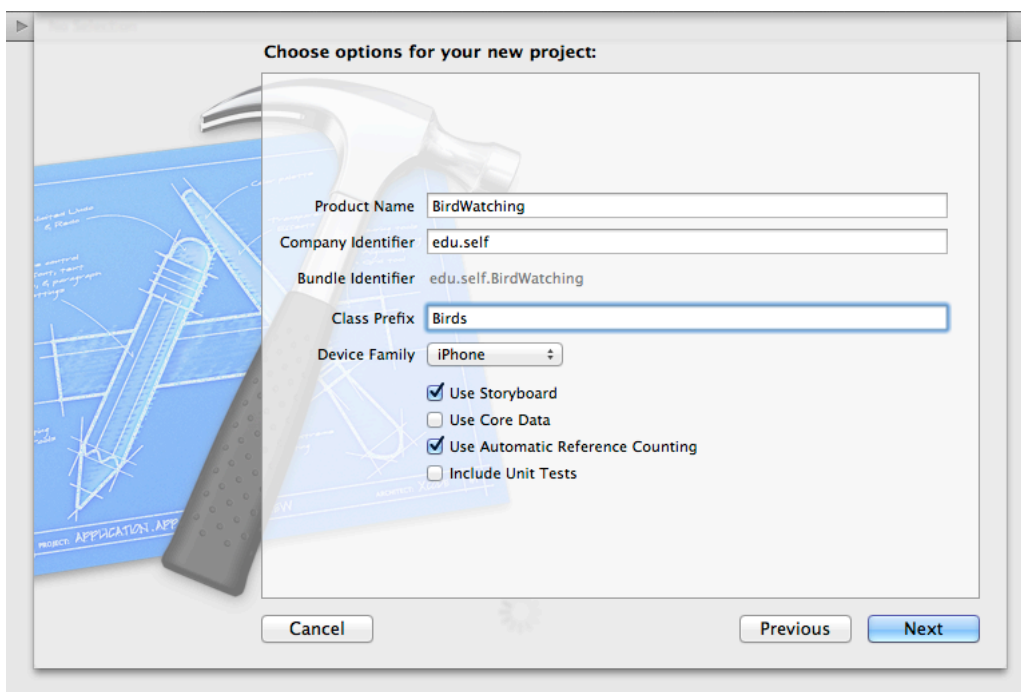
まず、「Master-Detail」テンプレートに基づくiOSアプリケーション用の新しいXcodeプロジェクトを作成して、BirdWatchingアプリケーションの開発を始めます。

1. Xcodeを開いて、「ファイル(File)」>「新規(New)」>「プロジェクト(Project)」を選択します。
2. ダイアログの左側にあるiOSセクションで、「アプリケーション(Application)」を選択します。
3. ダイアログのメイン領域で「Master-Detail Application」を選択してから、「次へ(Next)」をクリックします。

新しいダイアログが表示され、アプリケーションの名前を入力し、プロジェクトの追加オプションを選択するように指示されます。

4. ダイアログの「製品名(Product Name)」、「会社ID(Company Identifier)」、「クラスプレフィックス(Class Prefix)」の各フィールドに、下記の値を使用して入力します。
 - Product Name : BirdWatching
 - Company Identifier : 存在する場合は、会社ID会社IDがない場合は、edu.selfを使用できます。
 - Class Prefix : Birds
5. 「デバイスファミリ(Device Family)」ポップアップメニューで、iPhoneが選択状態になっていることを確認します。
6. 「ストーリーボードを使用(Use Storyboard)」と「自動参照カウントを使用(Use Automatic Reference Counting)」のオプションは選択状態のままにし、「Core Dataを使用(Use Core Data)」と「単体テストを含める(Include Unit Tests)」のオプションは未選択状態のままにします。

この情報の入力が完了したら、ダイアログは以下のようにになっているはずです。



7. 「次へ(Next)」をクリックします。

新しいダイアログが表示されて、保存する場所を指定できるようになります。
8. この新しいダイアログでプロジェクトを保存する場所を入力し、「ソース管理(source control)」オプションを未選択状態のままにして、「作成(Create)」をクリックします。

Xcodeのウインドウ（ワークスペースウインドウと呼ぶ）上にプロジェクトが開きます。デフォルトでは、ワークスペースウインドウには「BirdWatching」ターゲットに関する情報が表示されません。

デフォルトプロジェクトのビルドと実行

テンプレートベースのプロジェクトと同様に、変更を行う前に新しいプロジェクトをビルドして実行できます。テンプレートが備える機能を理解しやすくなるため、このようにすることをお勧めします。

1. Xcodeツールバーの「スキーマ(Scheme)」ポップアップメニューに、「BirdWatching」>「iPhone 5.0 Simulator」が表示されていることを確認します。

メニューにこの選択肢が表示されない場合は、開いて「iPhone 5.0 Simulator」を選択します。

2. ツールバーの「実行(Run)」ボタンをクリックします（または、「プロジェクト(Product)」>「実行(Run)」を選択します）。

Xcodeのアクティビティビューア（ツールバーの中央）に、ビルド処理に関するメッセージが表示されます。

Xcodeがプロジェクトのビルドを完了すると、iOSシミュレータが自動的に起動します。プロジェクトを作成したときに（iPad製品ではなく）iPhone製品を指定したため、シミュレータはiPhoneに似たウィンドウを表示します。シミュレートされたiPhone画面で、シミュレータは、次のように見えるデフォルトアプリケーションを開きます。



デフォルトアプリケーションには最初から若干の機能が組み込まれています。たとえばナビゲーションバーには「Add(+)」ボタンや「Edit」ボタンがあって、マスタリストの項目を追加/削除できます。

注意 このチュートリアルでは「Edit」ボタンを使わず、また、「Add (+)」ボタンをテンプレートが作成するのとは別のやり方で作成します。テンプレートに組み込まれた、上記ボタンを処理するコードは使いませんが、当面、デフォルトアプリケーションをテストできるよう、ボタンはそのままにしておいてください。

シミュレータ上で、ナビゲーションバーの右端にある「Add (+)」ボタンをクリックすると、項目（デフォルトでは現在の日付と時刻）がリストに追加されます。この項目を選択すると、詳細画面が現れます。デフォルトの詳細画面は次のように表示されます。



詳細画面が表示されたら、ナビゲーションバーの左端に戻るボタンが表示され、タイトルが前画面のタイトル（この場合は「Master」）と共に示されていることに注意します。戻るボタンは、Master-Detail 階層を管理する Navigation Controller によって自動的に用意されます。戻るボタンをクリックして、マスターリスト画面に戻ります。

次のセクションでは、アプリケーションのユーザインターフェイスを表現し、その動作の多くを実装する上でストーリーボードが果たす中心的な役割について説明します。ここで、iOSシミュレータを終了します。

シミュレータを終了すると、Xcodeのワークスペースウインドウ下部に「Debug」領域が開きます。チュートリアルでは、このペインは使いません。画面を広く使えるよう、このペインは閉じておくといでしょう。

- ツールバーの「Debug View」ボタンをクリックしてください。

「Debug View」ボタンは「View」ボタン群の中央にあって、次のような外観をしています：



ストーリーボードとそのシーンの調査

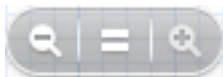
ストーリーボードは、アプリケーションの画面と、画面間のトランジションを表します。BirdWatchingアプリケーションのストーリーボードには少数の画面しかありませんが、より複雑なアプリケーションの中には、複数のストーリーボードを持ち、そのそれぞれが別々の画面サブセットを表すものもあります。

XcodeプロジェクトナビゲータでMainStoryboard.storyboardをクリックして、BirdWatchingアプリケーションのストーリーボードをキャンバスに開きます。スクロールしなくても3つの画面がすべて見えるように、拡大縮小レベルを調整する必要がある場合があります。

Do one of the following:

- キャンバスの右下角にあるズームコントロールをクリックします。

ズームコントロールは次のように見えます。

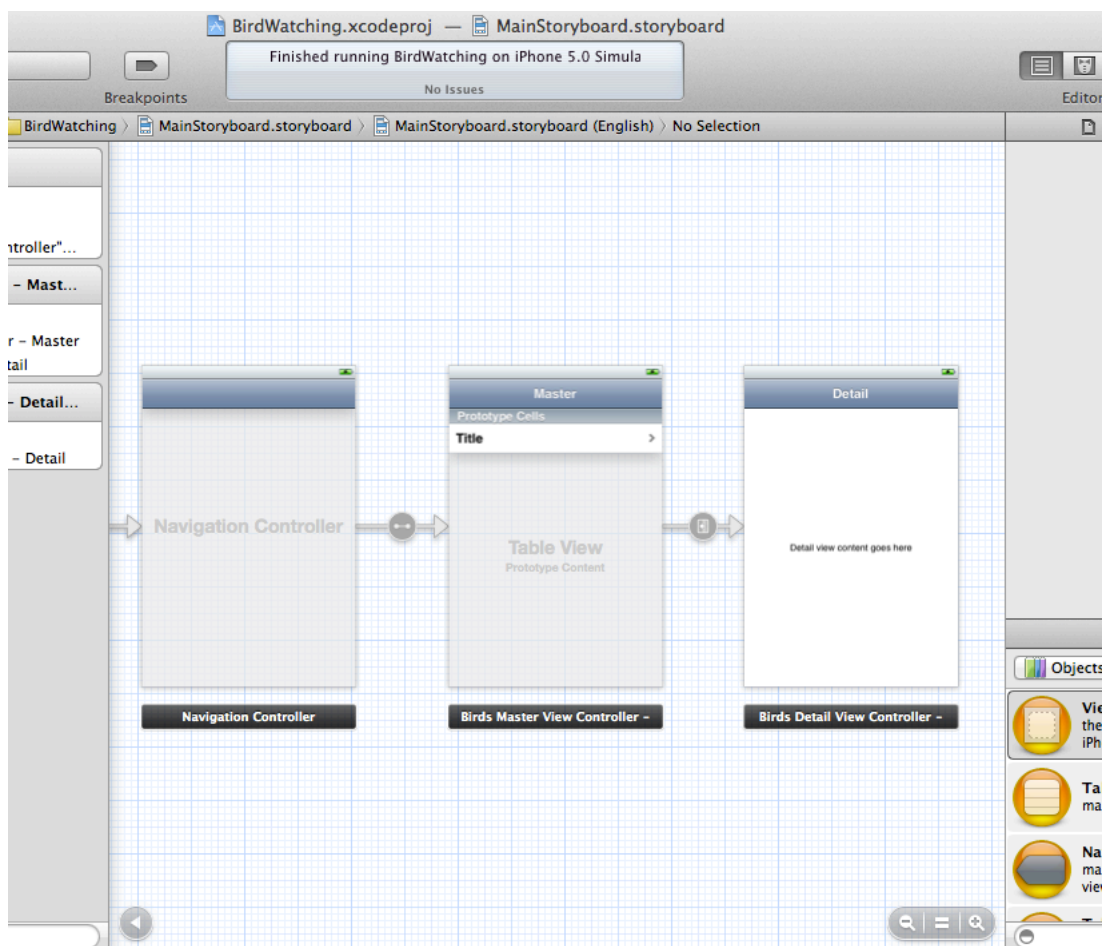


なお、中央のズームコントロールは、直近の拡大縮小レベルと100%を交互に切り替えるトグルスイッチの役割を果たします。

- 「Editor」 > 「Canvas」 > 「Zoom」を選択してから、適切な拡大縮小コマンドを選びます。
- キャンバスをダブルクリックしてください。

キャンバスのダブルクリックは、100%と50%の拡大縮小レベルを交互に切り替えるトグルスイッチの役割を果たします。

プロジェクトの3つの画面が一度に見えるようにキャンバスを拡大縮小します。次のようになるはずです。



「Master-Detail Application」テンプレートのデフォルトストーリーボードには、3つのシーンと2つのセグエがあります。

シーンは、View Controllerが管理する画面上のコンテンツ領域を表します。View Controllerは、コンテンツの画面描画を担当する一連のビューを管理するオブジェクトです（ストーリーボードに関する場合、シーンとView Controllerは同義語です）。デフォルトストーリーボードの左端のシーンは、Navigation Controllerを表します。Navigation Controllerは、自らのビューに加えて、一連のほかのView Controllerも管理するため、**コンテナView Controller**と呼ばれます。たとえば、デフォルトアプリケーションのNavigation Controllerは、アプリケーションの実行時に表示されるナビゲーションバーと戻るボタンに加えて、マスタと詳細のView Controllerも管理します。

セグエは、あるシーン（**ソース**）から次のシーン（**遷移先**）へのトランジションを表します。たとえば、デフォルトプロジェクトでは、マスタシーンがソースで、詳細シーンが遷移先です。デフォルトアプリケーションでテストしたときは、マスタリストで**Detail**項目を選択したときに、ソースから遷移先へセグエをトリガしました。この場合、セグエは**push**セグエであり、遷移先シーンがソースシー



ン上を右から左にスライドします。キャンバス上で**push**セグエはこのように見えます。

セグエは一方方向の遷移で、遷移先シーンは、表示に先立ってそのインスタンスが生成されます。2つの**View Controller**間にセグエを作成すると、遷移先シーンの設定をカスタマイズし、データを渡すことができるようになります（**“詳細シーンへのデータ送信”**（53 ページ）で実際にやってみます）。遷移先シーンから遷移元にデータを送る方法については、**“ユーザの入力の取得”**（70 ページ）でさらに詳細に学びます。

関係は、シーン間の接続の1種です。キャンバスでは、関係は、アイコンが次のようになっている点



を除けば、セグエと同様に見えます。

デフォルトストーリーボードには、**Navigation Controller**とマスタシーン間の関係があります。この場合、関係は、**Navigation Controller**によるマスタシーンの包含を表します。デフォルトアプリケーションが実行されると、**Navigation Controller**は、自動的にマスタシーンをロードして、画面の上部にナビゲーションバーを表示します。

キャンバスには、各シーンのグラフィカルな表現、その中身、およびその接続が表示されますが、ドキュメントアウトラインにこの情報の階層リストを表示することもできます。ドキュメントアウトラインペインが、プロジェクトナビゲータとキャンバスの間に現れます。そうならない場合は、キャンバスの左下隅にある「ドキュメントアウトライン(Document Outline)」ボタンをクリックしてくださ

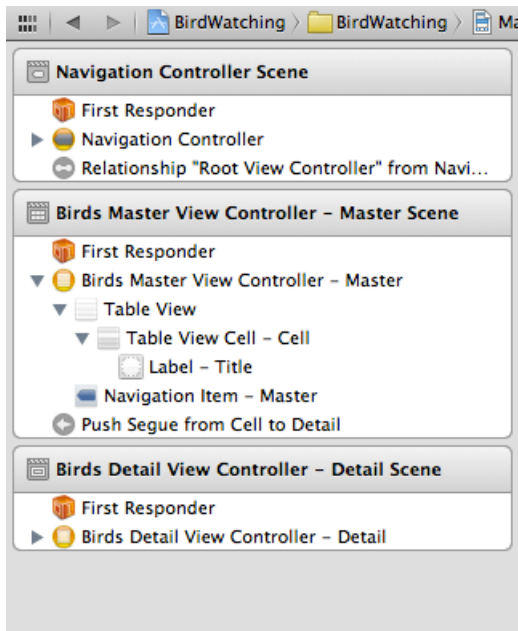


い。「ドキュメントアウトライン(Document Outline)」ボタンは次のような形をしています。

場合によっては、キャンバス上で選択するより、ドキュメントアウトラインで要素を選択する方が容易です。

1. 必要であれば、ドキュメントアウトラインペインをサイズ変更して、要素の名前が読めるようにします。
2. **View Controller**セクションで各要素の隣にある三角形をクリックします。

たとえば、Birds Master View Controllerセクションに要素の階層を表示しているときは、次のようになります。



まとめ

この章では、Xcodeを使用して、「Master-Detail Application」テンプレートに基づくiOSアプリケーションを新規作成しました。このデフォルトアプリケーションを実行すると、「メール(Mail)」や「連絡先(Contacts)」など、iPhone上のナビゲーションベースのほかのiOSアプリケーションと同様に動作することがわかりました。

Xcodeキャンバス上でテンプレートを備えたストーリーボードを開くと、そのストーリーボードとその中身を調べるいくつかの方法がわかりました。また、ストーリーボードのコンポーネントと、それらが表すアプリケーション部分についても学びました。

モデルレイヤの設計

どのようなアプリケーションでも、何らかの種類のデータを扱います。このチュートリアルでは、**BirdWatching**アプリケーションでは、一連の野鳥観察イベントを扱います。**Model-View-Controller (MVC)** 設計パターンに従うためには、このデータの表示と管理を行うクラスを作成する必要があります。

この章では、アプリケーションが扱うデータの1次ユニットを表すクラスを設計します。次に、このデータユニットのインスタンスを作成および管理する別のクラスを設計します。この2つのクラスは、データのマスタリストと共に、**BirdWatching**アプリケーションのモデルレイヤを形成します。

注意 Master-Detailテンプレートには、デフォルトアプリケーションの「Add (+)」ボタンがクリックされたときに追加する、プレースホルダデータの配列が定義されています。この後のステップで、テンプレートに組み込まれている配列を、この章で作成するクラスに置き換えます。

データユニットの決定とデータオブジェクトクラスの作成

データオブジェクトクラスを設計するには、まずアプリケーションの機能を調べて、扱う必要があるデータのユニットを明確にします。**BirdWatching**アプリケーションの機能を考察すると、**Bird**クラスと**Bird Sighting**クラスを定義しようという考えに至ります。**Bird Sighting**オブジェクトは、特定の野鳥観察イベントを表し、特定の野鳥オブジェクトに関する知識を持っているのに対し、**Bird**オブジェクトは1種類の野鳥を表すに過ぎません。さらに、**Genus**クラスや**Habitat**クラスも定義して、野鳥に関してさらに詳細な情報を追跡できるようにすることもできます。ただし、このチュートリアルをできるだけ単純にするため、ここでは、野鳥名、観察地、日付を表すプロパティが入った単一のデータオブジェクトクラスを定義するだけにします。

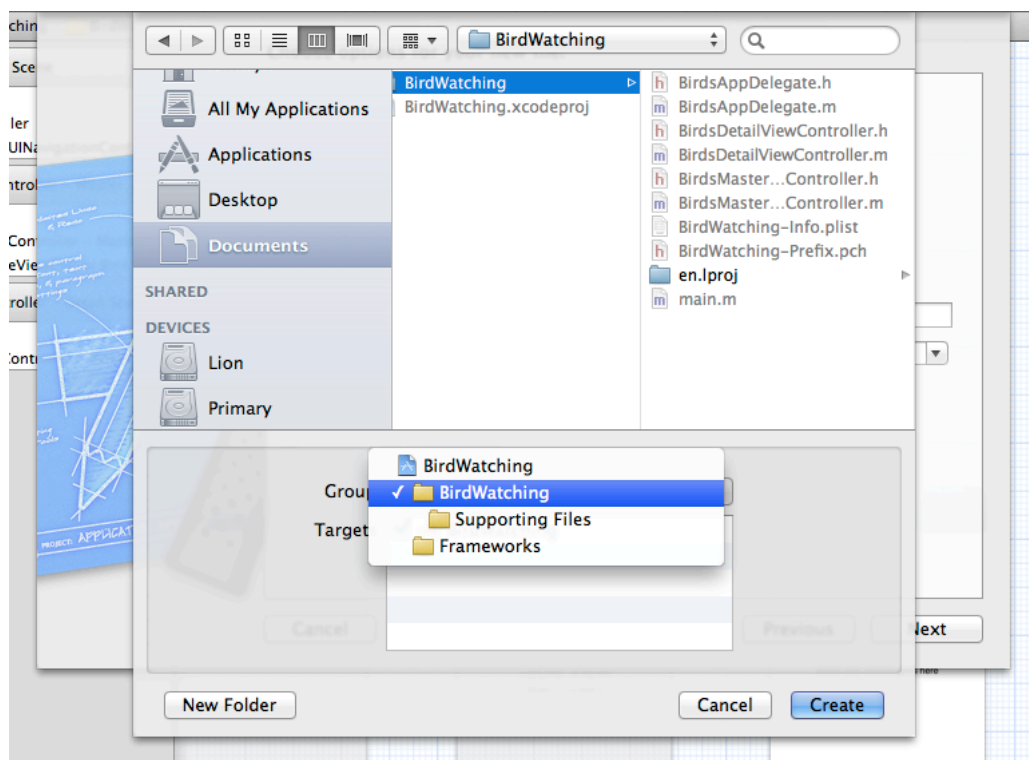
BirdWatchingアプリケーションが使用できる**Bird Sighting**オブジェクトを作成するためには、プロジェクトにカスタムクラスを加える必要があります。**Bird Sighting**オブジェクトは機能をほとんど必要としないため（たいていの場合は、**Objective-C**オブジェクトとして動作するだけで済みます）、**NSObject**の新しいサブセットを作成します。

1. Xcodeで、「ファイル(File)」 > 「新規(New)」 > 「ファイル(File)」を選びます（または、**Command-N** キーを押します）。
2. 表示されるダイアログで、ダイアログの左側にあるiOSセクションから「Cocoa Touch」を選択します。

3. ダイアログのメイン領域で「Objective-C」クラスを選択してから、「次へ(Next)」をクリックします。
4. ダイアログの次ペインで、「クラス(Class)」フィールドにクラス名BirdSightingを入力し、「～のサブクラス(Subclass of)」ポップアップメニューで「NSObject」を選択して、「次へ(Next)」をクリックします。

慣習として、データオブジェクトクラスの名前は、そのクラスが表す物に付ける名前なので、名詞になります。

5. 次に表示されるダイアログで、「グループ(Group)」ポップアップメニューからBirdWatchingフォルダを選択します。



6. ダイアログで「作成(Create)」をクリックします。

Xcodeが2つの新規ファイルを作成し、BirdSighting.hとBirdSighting.mと名付けてから、プロジェクトのBirdWatchingフォルダにそれらのファイルを追加します。デフォルトでは、Xcodeは、ワークスペースウィンドウのエディタ領域にある実装ファイル（つまり、BirdSighting.m）を自動的に開きます。

BirdSightingクラスには、野鳥観察を定義した3つの情報を保管する手段が必要です。このチュートリアルでは、宣言済みプロパティを使用してこれら3つのアイテムを表します。このクラスでは、自らの新しいインスタンスを、個々の野鳥観察を表す情報で初期化する必要もあります。これを実行するには、カスタムイニシャライザメソッドをこのクラスに追加します。

まず、プロパティとカスタムイニシャライザメソッドをヘッダファイルで宣言します。

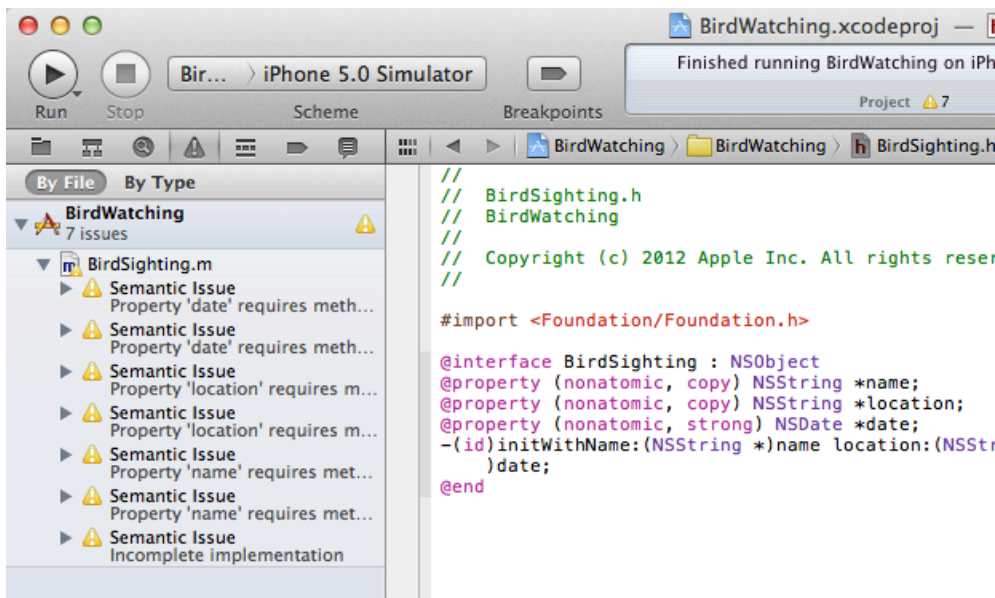
1. プロジェクトナビゲータでBirdSighting.hを選択して、エディタ領域に開きます。
2. @interfaceと@endのステートメント間に次のコードを追加します。

```
@property (nonatomic, copy) NSString *name;  
@property (nonatomic, copy) NSString *location;  
@property (nonatomic, strong) NSDate *date;
```

3. 3番目のプロパティ定義の後に、次のコード行を追加します。

```
-(id)initWithName:(NSString *)name location:(NSString *)location  
date:(NSDate *)date;
```

このコード行を追加すると、Xcodeのアクティビティビューア（ツールバーの中央およびジャンプバー内）に警告アイコンが現れます。ナビゲータセクタバーの警告ボタン（プロジェクトナビゲータの上部中央）をクリックして、警告の内容を表示します。次のようになるはずです。



この場合、Xcodeは、ここで追加した3つのプロパティにはアクセサメソッドがないこと、BirdSightingクラスの実装はinitWithNameメソッドがないため不完全であること、を注意しています。この問題は、次で解決します。

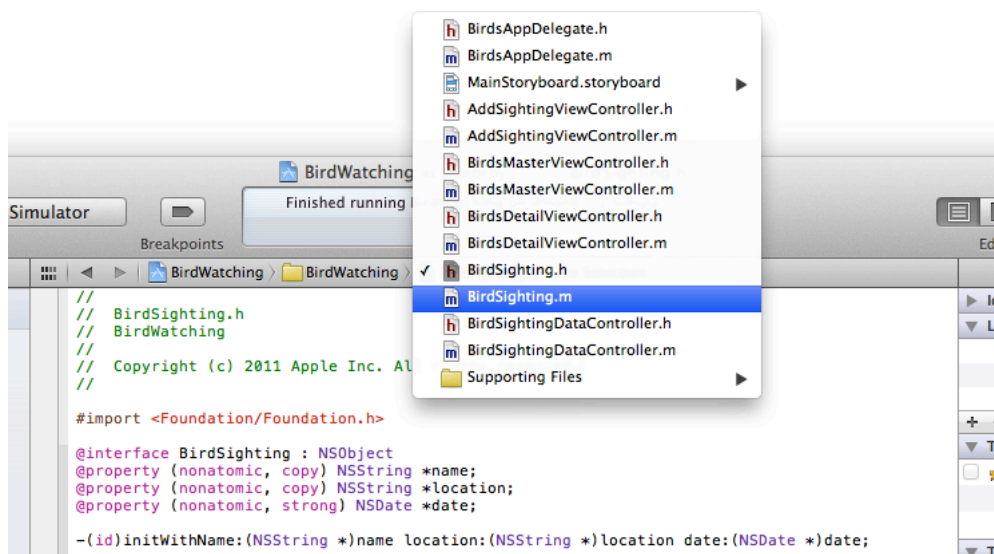
宣言済みのプロパティを使用しているため、プロパティごとのアクセサメソッドをコンパイラに合成させることができます。アクセサメソッドを合成して、カスタムイニシャライゼーションメソッドをBirdSighting実装ファイル内に作成するようにコンパイラに指示するコードを作成します。

1. エディタ領域の上部にあるジャンプバーでBirdSighting.hをクリックし、BirdWatchingプロジェクト内のファイルをリスト表示するメニューを開きます。

ジャンプバーには、必要な場合はファイル内での現在位置を含めて、プロジェクト内の現在位置を表すパス形式の表現が示されています。

2. ジャンプバーのメニューで、BirdSighting.mを選択します。

次のようになるはずです。



3. コードエディタで、BirdSighting.mの@implementationと@endのステートメント間に、プロパティの1つのための@synthesizeステートメントを入力します。

たとえば、nameプロパティから始めた場合、コード行は次のようになるはずです。

```
@synthesize name = _name;
```

@synthesizeコード行のnameにアンダースコアを追加して、_nameを、nameプロパティのインスタンス変数の名前として使用するようコンパイラに指示します。クラスでは_nameというインスタンス変数を宣言していないため、このコード行では、コンパイラに合成させるための指示もしています。

慣習として、インスタンス変数名にプレフィックスとしてアンダースコアを付けることで、インスタンス変数に直接アクセスすべきではないとわかるようにします。代わりに、合成したアクセサメソッドを使用すべきです（例外として、インスタンス変数を直接取得してinitメソッドに設定できます）。

学問的な観点から言えば、インスタンス変数への直接アクセスを避けると**カプセル化**の維持が容易になりますが、実用上のメリットも2つあります。

- 一部のCocoaテクノロジー（特に、キー値コーディング）は、アクセサメソッドの使用に依存するほか、アクセサメソッドの適切な命名にも依存します。アクセサメソッドを使用しないと、アプリケーションは、標準的なCocoa機能を活用できない場合があります。
- 一部のプロパティ値は、オンデマンドで作成されます。インスタンス変数を直接使用しようとすると、nilまたは初期化されていない値を取得する場合があります（View Controllerのビューは、オンデマンドで作成されるプロパティ値の格好の例です）。

（カプセル化の詳細に関心がある場合は、「プロパティの保存とアクセス」 in *Cocoa Fundamentals Guide*を参照してください）

4. @synthesizeステートメントのセミコロンの前で、最初のプロパティ名の後にカンマを追加して、残り2つのプロパティを入力します。

完成した@synthesizeステートメントは、次のようになるはずです。

```
@synthesize name = _name, location = _location, date = _date;
```

5. @synthesizeステートメントの後、@endの前に、initWithNameメソッドの入力を始めます。
パラメータの順序で混乱するなど、よくある間違いを避けることができるため、コード補完はメソッドシグネチャの入力時には特に有用です。ヘッダファイルで宣言したメソッドシグネチャの補完をXcodeが提供したら、-(id)iの入力を開始して、Returnキーを押します。
6. メソッドシグネチャの後に次のコードを入力して、initWithNameメソッドを実装します。

```
{  
    self = [super init];  
    if (self) {  
        _name = name;  
    }  
}
```

```
        _location = location;
        _date = date;
        return self;
    }
    return nil;
}
```

このステップのコードをすべて入力すると、警告インジケータは消えます。

注意 初期化メソッドは、プロパティにアクセスするためにアクセサメソッドを使用すべきではない2つの場所のうちの1つです（もう1か所は`dealloc`メソッド内です）。コードの残りでは、オブジェクトのプロパティの取得または設定を行う`self.name`など、アクセサメソッドを使用します。

`BirdSighting`クラスは、`BirdWatching`アプリケーションのモデルレイヤの一部です。`BirdSighting`クラスをモデルオブジェクトの純粋な表現として保存するためには、新しい`BirdSighting`オブジェクトをインスタンス化し、それらが含まれる集合を制御する方法を知っている`Controller`クラスも設計する必要があります。`Data Controller`クラスを作成すると、データモデルの実装方法について何も知らなくても、アプリケーション内のその他のオブジェクトが個々の`BirdSighting`オブジェクトとマスタリストにアクセスできるようになります。次の手順では、`Data Controller`クラスを作成します。

Data Controllerクラスの作成

`Data Controller`クラスは、通常、アプリケーション用データのロード、保存、アクセスを担当します。`BirdWatching`アプリケーションは、データのロードや保存のためにファイルにアクセスすることはありませんが、それでも、新しい`BirdSighting`オブジェクトを作成して、それらの集合を管理できる`Data Controller`が必要です。特に、このアプリケーションには次のクラスが必要です。

- すべての`BirdSighting`オブジェクトを保持するマスタ集合を作成するクラス
- 集合内の`BirdSighting`オブジェクトの数を返すクラス
- `BirdSighting`オブジェクトを集合内の特定位置に返すクラス
- ユーザからの入力を使用して新しい`BirdSighting`オブジェクトを作成し、集合に追加するクラス

BirdSightingクラスで行った処理と同様に、NSObjectから継承してプロジェクトに追加する新しいクラス用のインターフェイスと実装ファイルを作成します。

1. 「ファイル(File)」 > 「新規(New)」 > 「ファイル(File)」を選びます（または、Command-Nキーを押します）。
2. 表示されるダイアログで、ダイアログの左側にあるiOSセクションから「Cocoa Touch」を選択します。
3. ダイアログのメイン領域で「Objective-C」クラスを選択してから、「次へ(Next)」をクリックします。
4. ダイアログの次ペインでクラス名BirdSightingDataControllerを入力し、「サブクラス(Subclass)」ポップアップメニューで「NSObject」が選択状態になっていることを確認してから、「次へ(Next)」をクリックします。
5. 次に表示されるダイアログで、「グループ(Group)」ポップアップメニューからBirdWatchingフォルダを選択し、「作成(Create)」をクリックします。

このチュートリアルでは、BirdSightingオブジェクトの集合を配列で表します。**配列**とは、アイテムを順に並べたリストを保持し、リスト内の特定位置のアイテムにアクセス可能なメソッドを備えた集合オブジェクトのことです。BirdWatchingアプリケーションでは、ユーザが新しい野鳥観察をマスタリストに追加できるため、成長可能な配列である可変配列を使用します。

BirdSightingDataControllerクラスでは、そのクラスが作成するマスタ配列用のプロパティが必要です。

1. プロジェクトナビゲータまたはジャンプバーで選択して、BirdSightingDataController.hをエディタで開きます。
2. @interfaceと@endのステートメント間に次のコード行を追加します。

```
@property (nonatomic, copy) NSMutableArray *masterBirdSightingList;
```

masterBirdSightingListプロパティ定義のcopy属性に注意してください。代入にオブジェクトのコピーを使用することを指定します後で、配列のコピーも可変になるようにするカスタムsetterメソッドを実装ファイルに作成します。

メソッドの宣言を追加する必要があるので、ヘッダファイルはまだ閉じないでください。このセクションの先頭で述べたとおり、Data Controllerが実行する必要があるタスクが4つあります。これらのタスクのうちの3つは、BirdSightingオブジェクトのリストに関する知識を取得したり、新しいオブジェクトをリストに追加したりする手段をほかのオブジェクトに与えます。ところが、「マスタ集合

の作成(create the master collection)」タスクは、Data Controllerオブジェクトのみしかその存在を知る必要がないタスクです。ほかのオブジェクトにこのメソッドを公開する必要はないので、ヘッダファイルで宣言する必要はありません。

1. BirdSightingDataController.hファイルのプロパティ宣言の後に、次のコード行を追加します。

```
- (NSUInteger)countOfList;  
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex;  
- (void)addBirdSightingWithName:(NSString *)inputBirdName location:(NSString *)inputLocation;
```

ここで、objectInListAtIndex:メソッドの左のガターにXcodeが赤いエラーアイコンを表示していることに注意してください。このインジケータをクリックすると、Xcodeが「Expected a type (型が必要)」というメッセージを表示します。つまり、XcodeはBirdSightingを有効な戻り値の型として認識していません。これを解決するには、シンボルをクラスと見なすようにコンパイラに指示するステートメントである、**前方宣言**を追加する必要があります。実際には、プロジェクトがこのクラスの定義をどこかほかの場所に置くことをコンパイラに約束します。

2. BirdSightingクラスの前方宣言を追加します。

ヘッダファイルの先頭にある#importと@interfaceのステートメント間に、次の行を入力します。

```
@class BirdSighting;
```

数秒後、エラーアイコンが消えます。ただし、アクセサメソッドをまだ合成していないため、ヘッダファイルの場合と同様、黄色の警告アイコンはプロパティ宣言の隣に残ります。

これでヘッダファイルの作業を終えたので、BirdSightingDataController.mファイルをエディタで開いて、クラスの実装を開始します。

前述のとおり、Data Controllerは、マスタリストを作成して、その中にプレースホルダアイテムを入れる必要があります。これを行う良い方法の1つは、このタスクを行うメソッドを記述してから、Data Controllerのinitメソッド内で呼び出すことです。

1. BirdSightingヘッダファイルをインポートして、Data Controllerメソッドがこの型のオブジェクトを参照できるようにします。

#import "BirdSightingDataController.h"ステートメントの後に、次のコード行を追加します。


```
#import "BirdSighting.h"
```

2. @interfaceステートメントを@implementationステートメントの前に追加して、リスト作成メソッドを宣言します。

@interfaceステートメントは次のようになるはずです。

```
@interface BirdSightingDataController ()  
- (void)initializeDefaultDataList;  
@end
```

@interface BirdSightingDataController ()コードブロックは、クラス拡張として呼び出されます。**クラス拡張**では、このクラスにプライベートなメソッドを宣言できます（詳細は「[拡張 in The Objective-C Programming Language](#)」を参照）。

3. @implementationステートメントの後に次のコード行を追加して、ヘッダファイルで宣言したプロパティ用にアクセサメソッドを合成します。

```
@synthesize masterBirdSightingList = _masterBirdSightingList;
```

4. @synthesizeステートメントの後に次のコード行を入力して、リスト作成メソッドを実装します。

```
- (void)initializeDefaultDataList {  
    NSMutableArray *sightingList = [[NSMutableArray alloc] init];  
    self.masterBirdSightingList = sightingList;  
    [self addBirdSightingWithName:@"Pigeon" location:@"Everywhere"];  
}
```

initializeDefaultDataListメソッドは以下を行います。まず、新しい可変配列をsightingList変数に割り当てます。次に、[“Data Controllerの3つのデータアクセスメソッドを宣言するには、以下の手順を実行します。”](#) (? ページ) で宣言したaddBirdSightingWithName:Location:メソッドにいくつかのデフォルトデータを渡し、このメソッドが新しいBirdSightingオブジェクトを作成してマスタリストに追加します。

マスタリストプロパティ用にアクセサメソッドを合成しましたが、デフォルトsetterメソッドをオーバーライドして、新しい配列が可変のままになるようにする必要があります。デフォルトでは、setterメソッドのメソッドシグネチャは、set**PropertyName**です（プロパティ名がsetterメソッド名に含まれ

ている場合、プロパティ名の最初の文字は大文字になります）。カスタムアクセサメソッドを作成するときは、適切な名前を使用するように注意してください。そうしないと、カスタムメソッドではなく、デフォルトメソッドが呼び出されてしまいます。

- BirdSightingDataController.mの@implementationブロックに、次のコード行を追加します。

```
- (void)setMasterBirdSightingList:(NSMutableArray *)newList {
    if (_masterBirdSightingList != newList) {
        _masterBirdSightingList = [newList mutableCopy];
    }
}
```

デフォルトでは、新しいObjective-Cクラスファイルを作成するとき、Xcodeは、initメソッドのサブ実装を含めません。ほとんどのオブジェクトは、[super init]を呼び出す以外に何もする必要がないからです。このチュートリアルでは、Data Controllerクラスはマスタリストを作成する必要があります。

- BirdSightingDataController.mの@implementationブロックに、次のコード行を入力します。

```
- (id)init {
    if (self = [super init]) {
        [self initializeDefaultDataList];
        return self;
    }
    return nil;
}
```

このメソッドは、selfに、スーパークラスのイニシャライザから返された値を割り当てます。[super init]に成功すると、このメソッドは、先ほど記述したinitializeDefaultDataListメソッドを呼び出し、新たに初期化された自らのインスタンスを返します。

これで、マスタリストを作成し、プレースホルダアイテムを入れて、自らの新たなインスタンスを初期化する能力をData Controllerクラスに与え終わりました。次に、ヘッダファイルで宣言した3つのデータアクセサメソッドを実装して、マスタリストと対話する能力をその他のオブジェクトに与えます。

- countOfList
- objectInListAtIndex:

- `addBirdSightingWithName:location:`

1. 次のファイル行を入力して、`countOfList`メソッドを実装します。

```
- (NSUInteger)countOfList {  
    return [self.masterBirdSightingList count];  
}
```

`count`メソッドは、配列内のアイテムの総数を返す`NSArray`メソッドです。

`masterBirdSightingList`は、`NSArray`から継承する`NSMutableArray`型なので、このプロパティは`count`メッセージに応答できます。

2. 次のファイル行を入力して、`objectInListAtIndex:`メソッドを実装します。

```
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex {  
    return [self.masterBirdSightingList objectAtIndex:theIndex];  
}
```

3. 次のファイル行を入力して、`addBirdSightingWithName:location:`メソッドを実装します。

```
- (void)addBirdSightingWithName:(NSString *)inputBirdName location:(NSString  
*)inputLocation {  
    BirdSighting *sighting;  
    NSDate *today = [NSDate date];  
    sighting = [[BirdSighting alloc] initWithName:inputBirdName  
location:inputLocation date:today];  
    [self.masterBirdSightingList addObject:sighting];  
}
```

このメソッドは、ユーザが作成した名前と場所を今日の日付と共に`initWithName:location:date:`メソッドに送信して、新しい`BirdSighting`オブジェクトを作成して初期化します。次に、このメソッドは、新しい`BirdSighting`オブジェクトを配列に追加します。

この時点でアプリケーションをビルドして実行することもできますが、**View Controller**は実装済みのデータモデルに関して何も知らないため、初めて実行したときから何も変化していません。次の章では、マスタ**View Controller**とアプリケーションデリゲートファイルを編集して、アプリケーションがプレースホルダデータを表示できるようにします。

まとめ

この章では、BirdWatchingアプリケーション用のデータレイヤを設計し、実装しました。MVC設計パターンに従って、アプリケーションが動作時に使用するデータの保存と管理を行うクラスを作成しました。

チュートリアルこの時点で、プロジェクトには、BirdSightingとBirdSightingDataControllerの両方のクラスのインターフェイスファイルと実装ファイルが入っているはずです。自分のプロジェクトのコードの正確さを確認できるように、この4つのファイルのコードを下記に示しておきます。

BirdSighting.hのコードは次のようになります。

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

@interface BirdSighting : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *location;
@property (nonatomic, strong) NSDate *date;

-(id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate *)date;

@end
```

BirdSighting.mのコードは次のようになります。

```
#import "BirdSighting.h"

@implementation BirdSighting
@synthesize name = _name, location = _location, date = _date;

- (id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate *)date
{
    self = [super init];
    if (self) {
        _name = name;
        _location = location;
    }
}
```

```
        _date = date;
        return self;
    }
    return nil;
}

@end
```

BirdSightingDataController.hのコードは次のようになります。

```
#import <Foundation/Foundation.h>
@class BirdSighting;

@interface BirdSightingDataController : NSObject

@property (nonatomic, copy) NSMutableArray *masterBirdSightingList;

- (NSUInteger)countOfList;
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex;
- (void)addBirdSightingWithName:(NSString *)inputBirdName location:(NSString *)inputLocation;

@end
```

BirdSightingDataController.mのコードは次のようになります。

```
#import "BirdSightingDataController.h"
#import "BirdSighting.h"

@interface BirdSightingDataController ()
- (void)initializeDefaultDataList;
@end

@implementation BirdSightingDataController

@synthesize masterBirdSightingList = _masterBirdSightingList;
```

```
- (void)initializeDefaultDataList {
    NSMutableArray *sightingList = [[NSMutableArray alloc] init];
    self.masterBirdSightingList = sightingList;
    [self addBirdSightingWithName:@"Pigeon" location:@"Everywhere"];
}

- (void)setMasterBirdSightingList:(NSMutableArray *)newList {
    if (_masterBirdSightingList != newList) {
        _masterBirdSightingList = [newList mutableCopy];
    }
}

- (id)init {
    if (self = [super init]) {
        [self initializeDefaultDataList];
    }
    return self;
}

- (NSUInteger)countOfList {
    return [self.masterBirdSightingList count];
}

- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex {
    return [self.masterBirdSightingList objectAtIndex:theIndex];
}

- (void)addBirdSightingWithName:(NSString *)inputBirdName location:(NSString *)inputLocation {
    BirdSighting *sighting;
    NSDate *today = [NSDate date];

    sighting = [[BirdSighting alloc] initWithName:inputBirdName
location:inputLocation date:today];
```

```
        [self.masterBirdSightingList addObject:sighting];  
    }  
  
@end
```

マスタシンの設計

BirdWatchingアプリケーションを使い始めたとき、ユーザが最初に目にする画面には、野鳥観察のマスタリストが表示されます。この章では、アプリケーションの起動時に表示されるデフォルトの野鳥観察項目を作成するコードを記述し、マスタリストの外観を設計します。

Master View Controllerシンの設計

このチュートリアルでは、これまでたくさんのコードを記述してきました。ここでは、キャンバスでいくつかの設計作業をします。プロジェクトナビゲータ（またはジャンプバー）で `MainStoryboard.storyboard` を選択して、キャンバスにストーリーボードを開きます。

テーブルは、大量と少量のどちらのデータを表示するにも効率的でカスタマイズ可能な方法なので、iOSアプリケーションでは、しばしばTable Viewを使用して項目のリストを表示します。たとえば、「メール(Mail)」、「設定(Settings)」、「連絡先(Contacts)」、「ミュージック(Music)」では、さまざまなタイプのテーブルを使用して情報を表示します。このセクションでは、マスタシンの全体的な外観を指定し、テーブル行のレイアウトを設計します。

1. キャンバスで、（必要に応じて）拡大縮小レベルを調整して、マスタシーンに焦点を当てます。
2. Navigation Barでタイトル（ここでは「Master」）をダブルクリックして、Bird Sightingsと入力します。
3. シーンの中央をクリックして、Table Viewを選択します。

また、ドキュメントアウトラインペインのBirds Master View ControllerセクションでTable Viewを選択して、Table Viewを選択することもできます。

4. 必要ならば「Utilities View」ボタンをクリックして、ユーティリティ領域を開きます。
5. ユーティリティ領域で、「Attributes」ボタンをクリックしてAttributesインスペクタを開きます。
6. AttributesインスペクタのTable Viewセクションで、「スタイル(Style)」ポップアップメニューに「プレーン(Plain)」が表示されていることを確認します。

マスタシーンを見ればわかるとおり、プレーンスタイルテーブルの行がTable Viewの幅全体に拡大します。どの側でも背景の外観が見えるマージンを残すため、グループスタイルテーブルの行はTable Viewの端から内側に少し引っ込んでいます（グループスタイルテーブルの例を見たい場合は、iOSベースのデバイスで「設定(Settings)」を開いてください）。

Storyboardsは、テーブルの行（セルと呼びます）の外観を設計する2つの便利な方法を備えています。

- **Dynamic prototypes**では、1つのセルを設計してから、そのセルをテーブル内のほかのセルのテンプレートとして使用できます。動的プロトタイプは、テーブル内の複数のセルが同一のレイアウトを使用して情報を表示すべきときに使用します。
- **Static cells**では、セルの総数を含めて、テーブルの全体的なレイアウトを設計できます。静的セルは、表示される情報を問わずにテーブルの外観が変化しないときに使用します。

マスタシーンでは、ユーザが追加する項目数を問わず、同一レイアウトを使用してそれぞれの野鳥観察項目を表示したいので、マスタシーンのテーブル用にプロトタイプセルを設計します。アプリケーションを実行すると、野鳥観察項目を表示する際の必要に応じて、このプロトタイプセルのコピーが作成されます。

この手順に備えて、キャンバスはまだ開いたままになっているはずであり、**Attributes**インスペクタではまだマスタシーンの**Table View**に焦点が当てられているはずです。

1. **Attributes**インスペクタの「**Table View**」セクションで、「**コンテンツ(Content)**」ポップアップメニューから「**動的プロトタイプ(Dynamic Prototypes)**」を選択します。
2. キャンバスで**Table View Cell**を選択して、**Attributes**インスペクタにテーブルセル情報を表示します。
3. **Attributes**インスペクタの「**Table View Cell**」セクションで、「**Identifier**」テキストフィールドのデフォルト値を変更します。

「**Identifier**」フィールドの値は**再利用識別子**と呼ばれ、テーブル用に新しいセルを作成するとき、適切なプロトタイプセルを識別する手段をコンパイラに提供します。慣習として、セルの再利用識別子はそのセルに入っている中身を説明すべきなので、このチュートリアルでは、デフォルト値（すなわちCell）をBirdSightingCellで置き換えます。なお、この再利用IDはコード内のほかの場所でも必要になるため、コピーしておくことをお勧めします。

4. **Attributes**インスペクタの「**Table View Cell**」セクションで、「**スタイル(Style)**」ポップアップメニューから「**サブタイトル(Subtitle)**」を選択します。

組み込み**Subtitle**スタイルによって、左揃えの2つのラベルがセルに表示されます。大きな太字のフォントの「**Title**」と、小さなグレイのフォントの「**Subtitle**」です。組み込みセルスタイルを使用すると、これらのラベルとセルのプロパティ間の接続が自動的に作成されます。**Subtitle**スタイルを使用するセルでは、`titleLabel`プロパティはタイトルを表し、`detailTextLabel`プロパティはサブタイトルを表します。

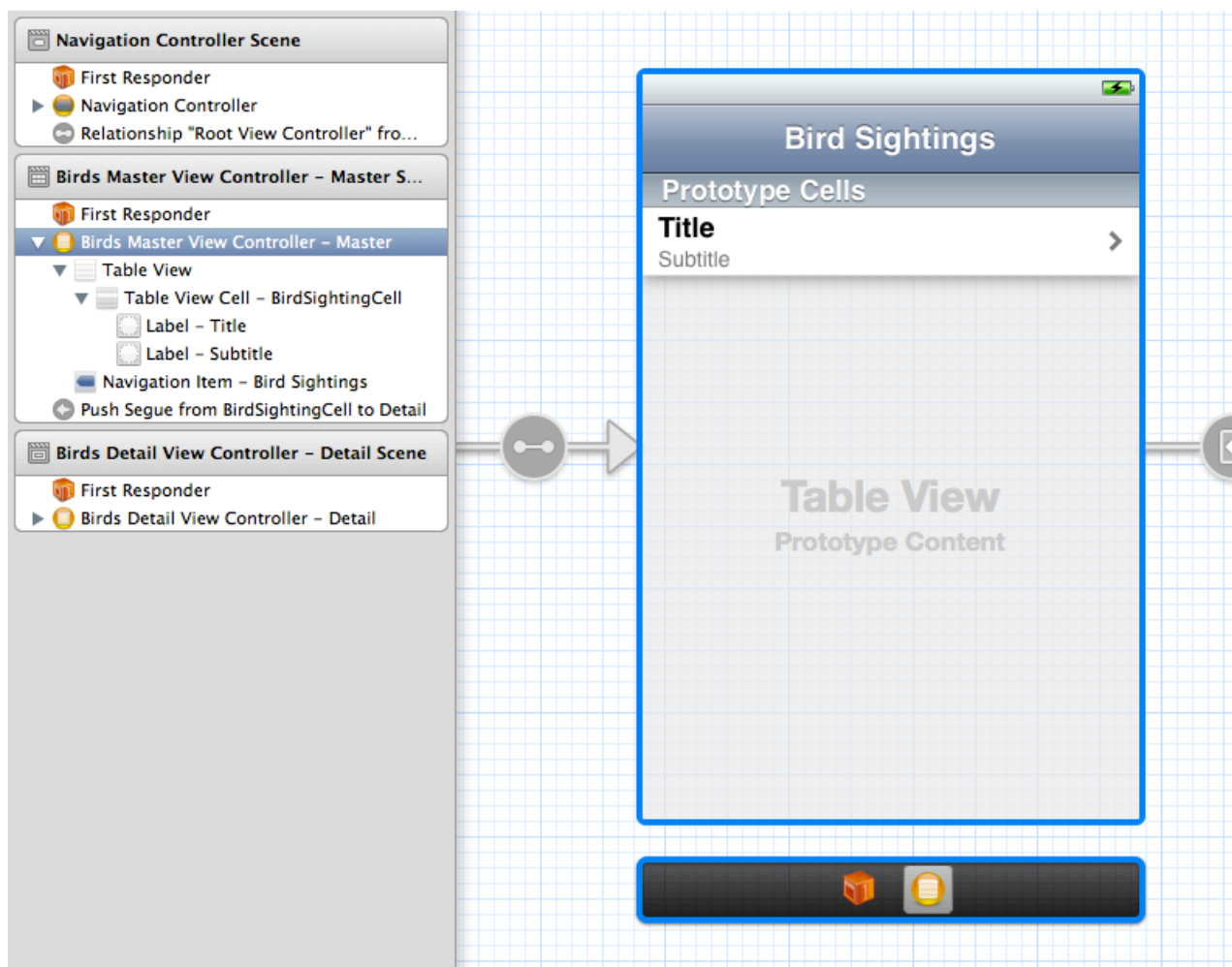
5. 必要ならば、**Attributes**インスペクタで、「**アクセサリ(Accessory)**」ポップアップメニューから「**ディスクロージャインジケータ(Disclosure indicator)**」を選択します。

アクセサリは、テーブルセルに表示できる標準的なユーザインターフェイス (UI) 要素です。ディスクローディングゲータアクセサリ (「>」のように見えます) は、アイコンをタップすると関連情報が新しい画面に表示されることをユーザに知らせます。

マスタシーンのコードファイルを編集する前に、キャンバス上のシーンがコードの `MasterViewController` クラスを表していることを確認します。この手順を忘れると、アプリケーションを実行しても、コードやキャンバスで行ったカスタマイズが表示されません。

1. ドキュメントアウトラインで、「Birds Master View Controller - Master」を選択します。

次のとおり、選択したマスタシーンがキャンバスに青色の輪郭線で描かれます。



2. ユーティリティ領域の最上部にある「Identity」ボタンをクリックして、Identityインスペクタを開きます。

3. Identityインスペクタで、「Class」フィールドにBirdsMasterViewControllerが含まれていることを確認します。

Master View Controllerの実装ファイルから不要箇所を削除

Master View Controllerが野鳥観察のマスタリストを表示するためのコードを記述する前に、テンプレートに組み込まれているコードを一部削除します。たとえば、（BirdWatchingアプリケーションにはマスタリストの編集機能がないので）「Edit」ボタンは必要ありません。また、（先に作成したデータモデルクラスを使うので）デフォルトの`_objects`配列も不要です。

不要なコードは完全に削除してしまっても構いませんが、複数行にわたる注釈記号（「/*」および「*/」）を使って、この部分をコンパイラが無視するように指示してもよいでしょう。この方が、将来アプリケーションの実装を変更するのが容易になります。たとえば、Master-Detailテンプレートのうち、Master View Controllerの実装ファイルに記述されている、`moveRowAtIndexPath`テーブルおよび`canMoveRowAtIndexPath`テーブルの表示メソッドを注釈にします。並べ替えが可能なテーブルは、どのようなアプリケーションにも必要というわけではないからです。

1. プロジェクトナビゲータで、BirdsMasterViewController.mを選択します。
2. `_objects`配列（プライベート変数）の宣言を注釈にします。

独自のデータモデルクラスを設計したので、テンプレートに組み込まれている`_objects`配列は必要ありません。注釈記号を施すと、BirdsMasterViewController.mの先頭部分は次のようになります。

```
#import "BirdsMasterViewController.h"
/*
@interface BirdsMasterViewController () {
    NSMutableArray *_objects;
}
@end
*/
@implementation BirdsMasterViewController
```

するとXcode上にはいくつか問題点が表示されます。ひとつには、テンプレートに組み込まれているMaster View Controllerのコードに、`_objects`を参照している箇所がいくつかあるからです。これは該当するコードを注釈にして解消します。それ以外の問題点は、独自メソッドを実装する際に解決します。

3. `insertNewObject`メソッドを注釈にします。

`insertNewObject`メソッドは、新しいオブジェクトを`_objects`配列に追加し、テーブルを更新します。しかしこのメソッドは不要です。後で、独自のデータモデルオブジェクトとマスタリストを、別のやり方で更新するからです。

`insertNewObject`メソッドを注釈にすると、問題点を示す3つのインジケータが消えます。

4. `tableView:commitEditingStyle:forRowAtIndexPath:`メソッドを注釈にします。

このチュートリアルでは、テンプレートに組み込まれた「Edit」ボタンは使わないので、`commitEditingStyle`メソッドに、テーブルの行を追加/削除する機能を実装する必要はありません。

`commitEditingStyle`メソッドを注釈にすると、問題点を示すインジケータが1つ消えます。

5. `viewDidLoad`メソッドの中身を注釈にします。

`viewDidLoad`メソッドには、`[super viewDidLoad]`を呼び出している箇所のみ残します。それ以外のコードを注釈にすると、`viewDidLoad`メソッドは次のようになります。

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // ビューを（一般にnibから）ロードした後、追加のセットアップ。
    /*
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(insertNewObject:)];
    self.navigationItem.rightBarButtonItem = addButton;
    */
}
```

6. `canEditRowAtIndexPath`テーブルビューメソッドの戻り値を変更します。

デフォルトでは、**Master-Detail**テンプレートは、**Master View Controller**のテーブルビューを編集できるようにになっています。マスタシーンで「Edit」ボタンを押せるようにはしないので、デフォルトの戻り値をNOに変更します。

以上の変更を施すと、`canEditRowAtIndexPath`メソッドは次のようになるはずです。

```
- (BOOL)tableView:(UITableView *)tableView
canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    // 指定された項目を編集できるようにしないならばNOを返す。
    return NO;
}
```

不要なコードを注釈にし、`canEditRowAtIndexPath`メソッドに修正を施しても、なおいくつか問題点を示すインジケータが残っています。そのうち2つは次のステップで解決します。

Master View Controllerの実装

通常、アプリケーションでは、Table ViewのデータソースメソッドをTable View Controllerクラスに実装して、テーブルにデータを表示します。少なくとも、Table View Controllerクラスは`numberOfRowsInSection`と`cellForRowAtIndexPath`メソッドを実装します。

野鳥観察のマスタリストを表示するため、Master View Controllerはこの2つのデータソースメソッドを実装しています。ただし、Master View Controllerは、まず、モデルレイヤオブジェクトが扱うデータにアクセスできなければなりません。

1. プロジェクトナビゲータで、`BirdsMasterViewController.h`を選択します。
2. `BirdSightingDataController`クラスの前宣言を追加します。

`#import`ステートメントの後に、次のコード行を追加します。

```
@class BirdSightingDataController;
```

3. Data Controllerプロパティを宣言します。

ヘッダファイルの`BirdsMasterViewController`を編集して、次のようにします。

```
@interface BirdsMasterViewController : UITableViewController
@property (strong, nonatomic) BirdSightingDataController *dataController;
@end
```

4. ジャンプバーで、`BirdsMasterViewController.m`を選択します。
5. `BirdsMasterViewController.m`で、`#import "BirdsDetailViewController.h"`の後に次のコード行を追加して、モデルレイヤクラスのヘッダファイルをインポートします。

```
#import "BirdSightingDataController.h"
#import "BirdSighting.h"
```

6. **Data Controller**プロパティ用にアクセサメソッドを合成します。

BirdsMasterViewController.mの@implementationステートメントの後に、次のコード行を追加します。

```
@synthesize dataController = _dataController;
```

これで、**Master View Controller**がモデルレイヤからのデータにアクセスできるようになったため、そのデータを**Table View**のデータソースに渡せます。BirdsMasterViewController.mはまだ開いたままになっているはずです。

1. numberOfRowsInSectionメソッドを実装して、配列内のBirdSightingオブジェクトの数を返します。

numberOfRowsInSectionのデフォルトの実装を次のコードに置き換えます。

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [self.dataController countOfList];
}
```

numberOfRowsInSectionメソッドの新しい実装には_objects配列を参照している箇所がないので、問題点を示すインジケータが1つ消えます。

2. cellForRowAtIndexPathメソッドを実装して、プロトタイプから新しいセルを作成し、適切なBirdSightingオブジェクトからのデータを格納します。

cellForRowAtIndexPathのデフォルトの実装をコードに置き換えます。

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"BirdSightingCell";

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
```

```
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }

    UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];

    BirdSighting *sightingAtIndex = [self.dataController
    objectInListAtIndex:indexPath.row];

    [[cell.textLabel] setText:sightingAtIndex.name];
    [[cell.detailTextLabel] setText:[formatter stringFromDate:(NSDate
    *)sightingAtIndex.date]];

    return cell;
}
```

新しい実装には`_objects`配列を参照している箇所がないので、問題点を示すインジケータがもう1つ消えます。

TableViewオブジェクトは、テーブルの行を表示する必要があるたびに`cellForRowAtIndexPath`メソッドを呼び出します。`cellForRowAtIndexPath`メソッドのこの実装では、新しいテーブルセルの作成に使用するプロトタイプセルを識別し、日付の表示に使用する形式を指定します。次に、再利用可能なテーブルセルがない場合は、キャンバスで設計したプロトタイプセルに基づいて新しいセルを作成します。最後に、マスタリストでユーザがタップした行に関連付けられた`BirdSighting`オブジェクトを取得して、その新しいセルのラベルを野鳥観察情報で更新します。

以上の作業を終えると、残った問題点は`prepareForSegue`メソッド内だけになります。このメソッドは後で編集します。

アプリケーションデリゲートでのアプリケーションの設定

デフォルトでは、アプリケーションデリゲートは、単に、がウィンドウオブジェクトにアクセスできるようにするウィンドウプロパティを合成します。アプリケーションデリゲートで多数のタスクを実行することはできる限り避けて、代わりに**View Controller**により多くの責任を与えるようにすべきです。このチュートリアルでは、アプリケーションデリゲートは、**Data Controller**オブジェクトを初期化して、**Master View Controller**に渡します。

プロジェクトナビゲータ（またはジャンプバー）で`BirdsAppDelegate.m`を選択して、エディタで開きます。`didFinishLaunchingWithOptions`メソッドしか実装する必要はないため、アプリケーションデリゲートのほとんどのコードスタブは無視できます。特に、新しい**Data Controller**オブジェクト

をMaster View ControllerのdataControllerプロパティに割り当てられるようにするため、このメソッドを実装する必要があります（このプロパティは、“[Master View Controllerがモデルレイヤ内のデータにアクセスできるようにするには、以下の手順を実行します。](#)”（? ページ）で作成済みです）。

1. BirdsAppDelegate.mで、Data ControllerとMaster View Controllerのヘッダファイルをインポートします。

既存の#import "BirdsAppDelegate.h"行の後に、次のコード行を入力します。

```
#import "BirdSightingDataController.h"
#import "BirdsMasterViewController.h"
```

2. didFinishLaunchingWithOptionsメソッドのスタブ実装を、次のコード行に置き換えます。

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UINavigationController *navigationController = (UINavigationController
    *)self.window.rootViewController;

    BirdsMasterViewController *firstViewController =
    (BirdsMasterViewController *)[navigationController viewControllers]
    objectAtIndex:0];

    BirdSightingDataController *aDataController =
    [[BirdSightingDataController alloc] init];

    firstViewController.dataController = aDataController;

    return YES;
}
```

このdidFinishLaunchingWithOptionsメソッドの最大の機能は、Data Controllerオブジェクトを初期化して、マスタシーンのdataControllerプロパティに割り当てることです。ただし、この割り当てができるようになる前に、このメソッドは、まず、Master View Controllerへの参照を取得しなければなりません。

マスタシーンのView Controllerへの参照を取得するため、didFinishLaunchingWithOptionsメソッドは、自らのwindowプロパティを使用して、ウインドウのRoot View Controllerオブジェクトを取得します。このアプリケーションでは、最初のView ControllerがUINavigationControllerオブジェクトであることがわかっているため、次のコードを使用して、ウインドウのRoot View Controllerオブジェクトを適切な型にキャストします。


```
UINavigationController *navigationController = (UINavigationController
*)self.window.rootViewController;
```

ここで、**View Controller**の**Navigation Controller**のスタックの最下部にある**Master View Controller**への参照が必要です。ただし、**UINavigationController**オブジェクトは**Root View Controller**プロパティを備えていないため、別の手法を使用する必要があります。スタックの最下部にある**View Controller**への参照を取得したことを確認するため、次のコード行を使用して、**Navigation Controller**の**viewControllers**配列の零番目のメンバを取得します。

```
BirdsMasterViewController *firstViewController = (BirdsMasterViewController
*)[[navigationController viewControllers] objectAtIndex:0];
```

この時点でアプリケーションをビルド、実行しようとしても失敗してしまいます。prepareForSegue メソッドで **_objects** 配列を使っている、という問題点を解消していないからです。しかしこれは、次の章を終えた時点では解消され、ビルド、実行できるようになります。

まとめ

この章では、動的プロトタイプセルを活用して、マスタシーンのリストの行のレイアウトを設計しました。次に、モデルレイヤ内のデータにアクセスできるようにし、**Table View**のデータソースメソッドの実装でその情報を使用して、マスタシーンの**View Controller**を実装しました。

最後に、アプリケーションデリゲートで2つのアプリケーション設定タスクを実行しました。特に、**BirdSightingDataController**オブジェクトを実装して、マスタシーン**View Controller**のプロパティに割り当てました。

チュートリアル of この時点で、**BirdsMasterViewController.h** ファイルのコードは次のようになっています。

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

@class BirdSightingDataController;

@interface BirdsMasterViewController : UITableViewController
@property (strong, nonatomic) BirdSightingDataController *dataController;

@end
```

BirdsMasterViewController.mファイル内のコードは、次のようになっているはずです（このチュートリアルで注釈化あるいは編集しなかったメソッドは示していません）。

```
#import "BirdsMasterViewController.h"
#import "BirdsDetailViewController.h"
#import "BirdSightingDataController.h"
#import "BirdSighting.h"

@implementation BirdsMasterViewController

@synthesize dataController = _dataController;

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [self.dataController countOfList];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"BirdSightingCell";

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }

    UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];

    BirdSighting *sightingAtIndex = [self.dataController
    objectInListAtIndex:indexPath.row];
    [[cell.textLabel] setText:sightingAtIndex.name];
    [[cell.detailTextLabel] setText:[formatter stringFromDate:(NSDate
    *)sightingAtIndex.date]];
    return cell;
}
```

```
- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    // 指定された項目を編集できるようにしないならばNOを返す。
    return NO;
}
@end
```

BirdsAppDelegate.mファイル内のコードは、次のようになっているはずです。

```
#import "BirdsAppDelegate.h"
#import "BirdSightingDataController.h"
#import "BirdsMasterViewController.h"

@implementation BirdsAppDelegate

@synthesize window = _window;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UINavigationController *navigationController = (UINavigationController *)self.window.rootViewController;
    BirdsMasterViewController *firstViewController = (BirdsMasterViewController *)[navigationController viewControllers] objectAtIndex:0;

    BirdSightingDataController *aDataController = [[BirdSightingDataController alloc] init];
    firstViewController.dataController = aDataController;
    return YES;
}
@end
```

詳細シーンの情報の表示

ディスクローディングデータが入ったテーブル行をタップするとき、iOSユーザはその行項目に関する詳細（または追加機能）が表示されることを期待します。BirdWatchingアプリケーションでは、マスタシーンのテーブル内の各セルに、野鳥観察アイテムとディスクローディングデータが表示されます。ユーザがその項目をタップすると、野鳥観察の名前、日付、場所を示す詳細シーンが表示されます。

この章では、選択した野鳥観察に関する情報が表示される詳細シーンを作成します。

Detail View Controllerコードの編集

詳細シーンには、選択したテーブルセルに関連付けられたBirdSightingオブジェクトからの情報を表示すべきです。マスタシーンで行った作業とは異なり、まずDetailViewControllerコードを記述してから、後の手順でキャンバス上にシーンを設計します。

1. ジャンプバーでBirdsDetailViewController.hを選択します。
2. #importステートメントの後に、BirdSightingクラスの次の宣言を追加します。

```
@class BirdSighting;
```

3. デフォルトのDetail View Controllerの親クラスをUITableViewControllerに変更します。
この作業が終了したら、編集後の@interfaceステートメントは次のようになるはずです。

```
@interface BirdsDetailViewController : UITableViewController
```

なお、デフォルトの詳細シーンを汎用View ControllerからTable View Controllerに変更したのは、設計上の判断です。つまり、アプリケーションを正常に実行する上でこの変更は必要ありません。この設計判断の最大の理由は、テーブルを使用して野鳥観察の詳細を表示すると、マスタシーンとの一貫性があるユーザ体験を提供できるからです。2つ目の理由は、静的セルを使用してテーブルを設計する方法を学ぶ機会を与えられるからです。

4. BirdSightingオブジェクトとそのプロパティを参照するプロパティを宣言します。

テンプレートが提供するデフォルトの`detailItem`と`detailDescriptionLabel`プロパティ宣言は必要ありません。これらを次のプロパティ宣言に置き換えます。

```
@property (strong, nonatomic) BirdSighting *sighting;  
@property (weak, nonatomic) IBOutlet UILabel *birdNameLabel;  
@property (weak, nonatomic) IBOutlet UILabel *locationLabel;  
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
```

デフォルトプロパティを置き換えると、Xcodeには警告およびエラーのインジケータが現れます。新しいプロパティのアクセサメソッドがないこと、旧プロパティを参照している箇所が残っていることが原因です。

詳細シーンの実装ファイルで、先ほどの手順で追加したプロパティ用にアクセサメソッドを合成して、デフォルトメソッドのうちの2つを実装する必要があります。まず、アクセサメソッドを合成します。

1. ジャンプバーで`BirdsDetailViewController.m`を選択します。

プロパティ宣言が削除されたため、Xcodeは、`detailItem`と`detailDescriptionLabel`シンボルの各使用箇所の隣に赤色の問題インジケータを表示します。

2. ファイルの先頭に次のコード行を追加して、`BirdSighting`ヘッダファイルをインポートします。

```
#import "BirdSighting.h"
```

3. 既存の`@synthesize`ステートメントを次のコード行に置き換えます。

```
@synthesize sighting = _sighting, birdNameLabel = _birdNameLabel,  
locationLabel = _locationLabel, dateLabel = _dateLabel;
```

`@synthesize`ステートメントを更新すると、Xcodeがフラグを立てた問題の一部が解消されますが、すべてが解消されるわけではありません。先ほど削除した`detailItem`と`detailDescriptionLabel`プロパティを参照する2つのデフォルトメソッドが残っています。これらのメソッドは次で更新します。

デフォルトでは、`BirdsDetailViewController`実装ファイルには、`setDetailItem`と`configureView`メソッドのスタブ実装が含まれています。なお、`setDetailItem`メソッドは、テンプレートが提供した（および、[“Detail View Controllerのヘッダファイルをカスタマイズするには、以下の手順を実行し](#)

ます。” (? ページ) で削除した) detailItemプロパティのカスタムsetterメソッドです。このテンプレートが、Xcodeが合成可能なデフォルトsetterではなく、カスタムsetterメソッドを使用する理由は、setDetailItemがconfigureViewを呼び出せるからです。

このチュートリアルでは、詳細項目はBirdSightingオブジェクトなので、sightingプロパティがdetailItemプロパティの代わりを務めます。この2つのプロパティは同様の役割を果たすので、setDetailItemメソッドの構造体を使用すると、sightingプロパティのカスタムsetterを簡単に作成できます。

- setDetailItemメソッドを次のコードに置き換えます。

```
- (void)setSighting:(BirdSighting *) newSighting
{
    if (_sighting != newSighting) {
        _sighting = newSighting;

        // ビューを更新する
        [self configureView];
    }
}
```

configureViewメソッド (新しいsetSightingメソッドとデフォルトのviewDidLoadメソッドの両方で呼び出されます) は、詳細シーンのUIを特定の情報で更新します。このメソッドを編集して、選択したBirdSightingオブジェクトからのデータで詳細シーンが更新されるようにします。

- デフォルトのconfigureViewメソッドを次のコードに置き換えます。

```
- (void)configureView
{
    // ユーザインターフェイスを詳細アイテム用に更新する
    BirdSighting *theSighting = self.sighting;

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
}
```

```
if (theSighting) {
    self.birdNameLabel.text = theSighting.name;
    self.locationLabel.text = theSighting.location;
    self.dateLabel.text = [formatter stringFromDate:(NSDate
*)theSighting.date];
}
}
```

この節の作業を終えても、問題点を示すインジケータはなお残っています。元の `detailDescriptionLabel` プロパティが `viewDidUnload` メソッドに使われているからです。 `viewDidUnload` メソッドは、ビューに強い参照がある場合に、そのクリーンアップ処理を実装するべき箇所です。このチュートリアルでは、詳細コントローラのビューが、[“Detail View Controllerのヘッダファイルをカスタマイズするには、以下の手順を実行します。”](#) (? ページ) で作成した、 `sighting` プロパティの強い参照を保持しています。 `viewDidUnload` メソッドを編集して、 `sighting` プロパティに対する強い参照を解除するようにします。

- `viewDidUnload` メソッドを次のように編集します。

```
- (void)viewDidUnload
{
    self.sighting = nil;
    [super viewDidUnload];
}
```

次のセクションでは、詳細シーンをキャンバス上でレイアウトします。

詳細シーンの設計

前のセクションでは、テンプレートが提供する `Detail View Controller` の親クラスを `UIViewController` から `UITableViewController` に変更しました。このセクションでは、キャンバス上のデフォルト詳細シーンを、オブジェクトライブラリからの新しい `Table View Controller` に置き換えます。

1. ジャンプバーで `MainStoryboard.storyboard` を選択して、キャンバス上に開きます。
2. キャンバスで詳細シーンを選択して、**Delete** キーを押します。
3. `Table View Controller` をオブジェクトライブラリからドラッグして、キャンバス上でドロップします。

4. キャンバス上で新しいシーンを選択状態にしたままで、ユーティリティ領域の「Identity」ボタンをクリックして、Identityインスペクタで開きます。
5. Identityインスペクタの「カスタムクラス(Custom Class)」セクションで、「クラス(Class)」ポップアップメニューからBirdsDetailViewControllerを選択します。

キャンバスからシーンを削除すると、そのシーンへのすべてのセグエも消えます。ユーザが項目を選択したときにマスタシーンが詳細シーンにトランジションできるように、マスタシーンから詳細シーンへのセグエを確立し直す必要があります。

1. マスタシーンのテーブルセルから詳細シーンまで、Controlキーを押しながらドラッグします。
表示される半透明のパネルで、「プッシュ(Push)」を選択します。プッシュセグエでは、新しいシーンが以前のシーンの上に画面の右端からスライドします。

なお、Xcodeは、詳細シーンにナビゲーションバーを自動的に表示します。ソースシーン（つまり、マスタシーン）がNavigation Controller階層の一部であることをXcodeが知っているからです。そのため、Xcodeはシーン内でナビゲーションバーの外観をシミュレートして、レイアウトを設計しやすくします。

2. キャンバスでセグエを選択します。
3. Attributesインスペクタで、「Identifier」フィールドにカスタムIDを入力します。

慣習として、セグエが行うことを説明する識別子を使用するのが最善です。このセグエは詳細シーンに野鳥観察の詳細を表示するので、このチュートリアルではShowSightingDetailsを使用します。

シーンが別の遷移先シーンにトランジションできる場合は、各セグエに一意的識別子を与えて、コード内で区別できるようにすることが重要です。このチュートリアルでは、マスタシーンが詳細シーンと（[“新規項目の追加の有効化”](#)（57 ページ）で作成した）追加シーンにトランジションできるので、この2つのセグエを区別する手段を用意する必要があります。

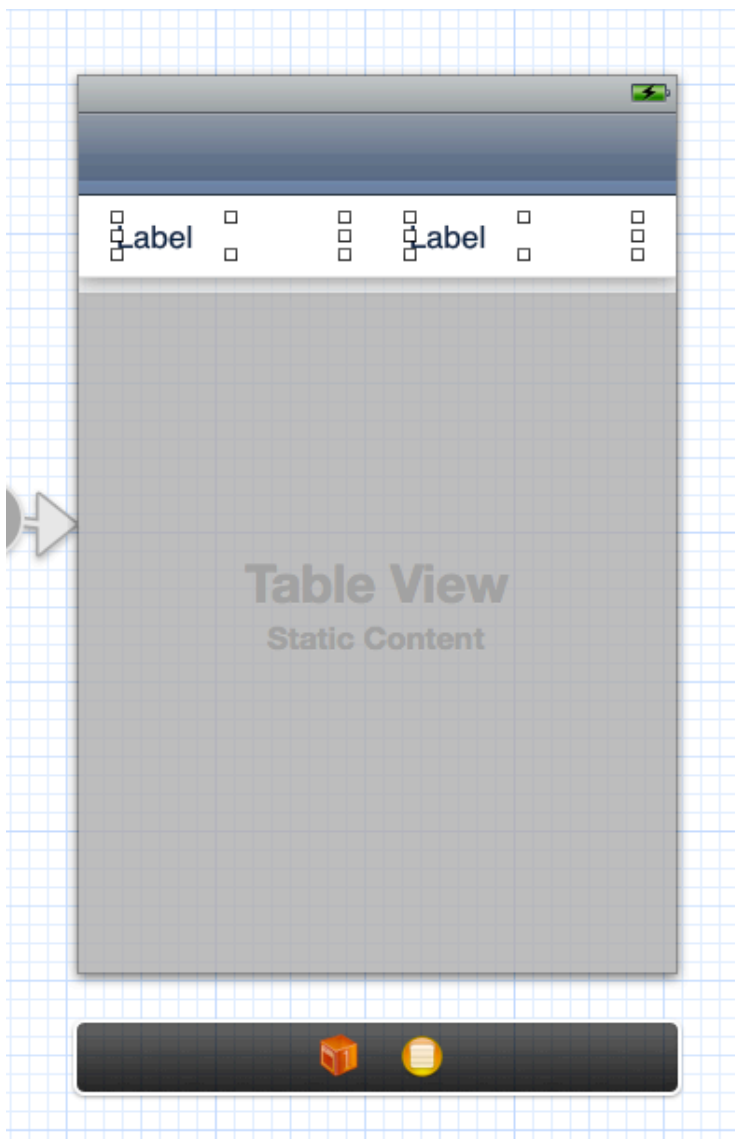
野鳥観察の詳細はさまざまですが、必ず同一の形式で表示されなければなりません。特に、名前、日付、場所です。詳細テーブルに異なるレイアウトで情報を表示することはないので、静的テーブルセルを使用すると、このレイアウトをキャンバス上で設計できます。

デフォルトでは、新しいTable View Controllerシーンには、プロトタイプベースのセルを使用するテーブルが入ります。詳細テーブルのレイアウトを設計する前に、コンテンツのタイプを変更します。

1. キャンバスで、詳細シーンのTable Viewを選択します。
2. Attributesインスペクタで、「コンテンツ(Content)」ポップアップメニューから「静的セル(Static Cells)」を選択します。

Table Viewのコンテンツタイプを動的プロトタイプから静的セルに変更すると、結果のテーブルには自動的に3つのセルが入ります。偶然にも、アプリケーションに表示したいテーブル行の数も3つです。各セルの中身を別々にレイアウトすることもできますが、各セルは同一のレイアウトを使用するので、1つを設計してからほかにもコピーして作成する方が簡単です。

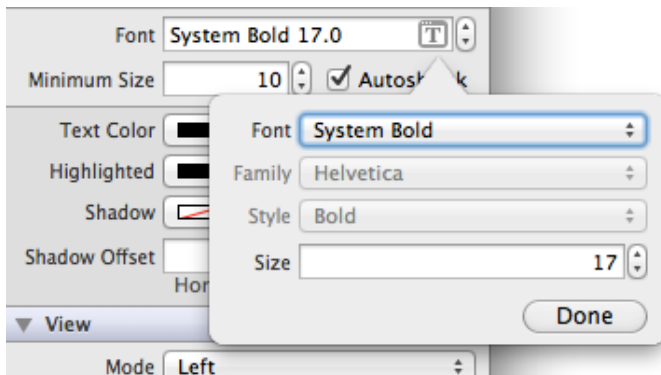
1. セルのうちの2つを選択し、Deleteキーを押して削除します。
削除するセルはどれでも構いませんが、隣り合ったセルを削除する方が簡単です。
2. (必要に応じて) 残りのセルを選択し、それらがAttributesインスペクタの「スタイル(Style)」ポップアップメニューに入っていることを確認しますCustom。
3. 一度に1つずつ、オブジェクトライブラリから2つのラベルをドラッグし、セルにドロップします。
4. 各ラベルのサイズを変更して、次のように並べます。



5. 左側のラベルを選択し、配置とフォントを変更します。

Attributesインスペクタで「右揃え(right Alignment)」ボタンをクリックし、ラベルのテキストを右揃えにします。

また、このインスペクタで、「フォント(Font)」フィールドの内側にある「T」というラベルのボタンをクリックします。



表示されるパネルで、「フォント(Font)」ポップアップメニューから「システムボールド(System Bold)」を選択します。

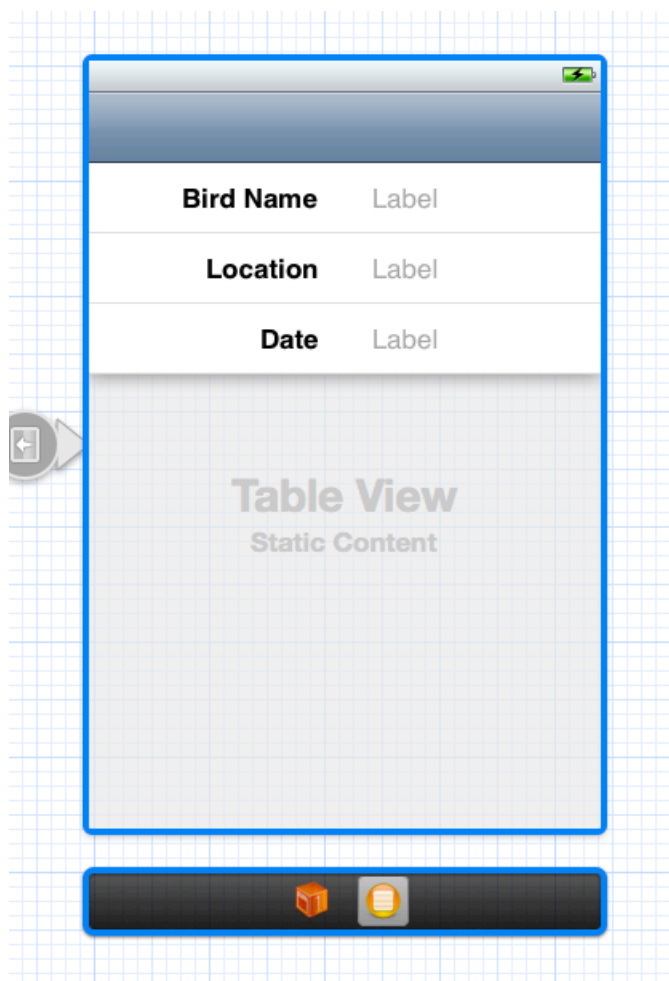
6. 右のラベルを選択し、フォントの色を変更します。

Attributesインスペクタで、「テキストカラー(Text Color)」ポップアップメニューから「ライトグレイカラー(Light Gray Color)」を選択します（ポップアップメニューを開く際は、色付きの矩形ではなく矢印の付近をクリック）。

7. ドキュメントアウトラインで「Table Viewセクション(Table View Section)」を選択します。
8. Attributesインスペクタの「Table Viewセクション(Table View Section)」領域で、上下の矢印を使用して行数を3に増やします。
9. 各セルの左のラベルを編集して、適切な説明が表示されるようにします。

上部のセルにBird Nameと入力し、中央のセルにLocationと入力し、下部のセルにDateと入力します。

テーブルのセルのレイアウトを終えると、詳細シーンは次のようになるはずです。

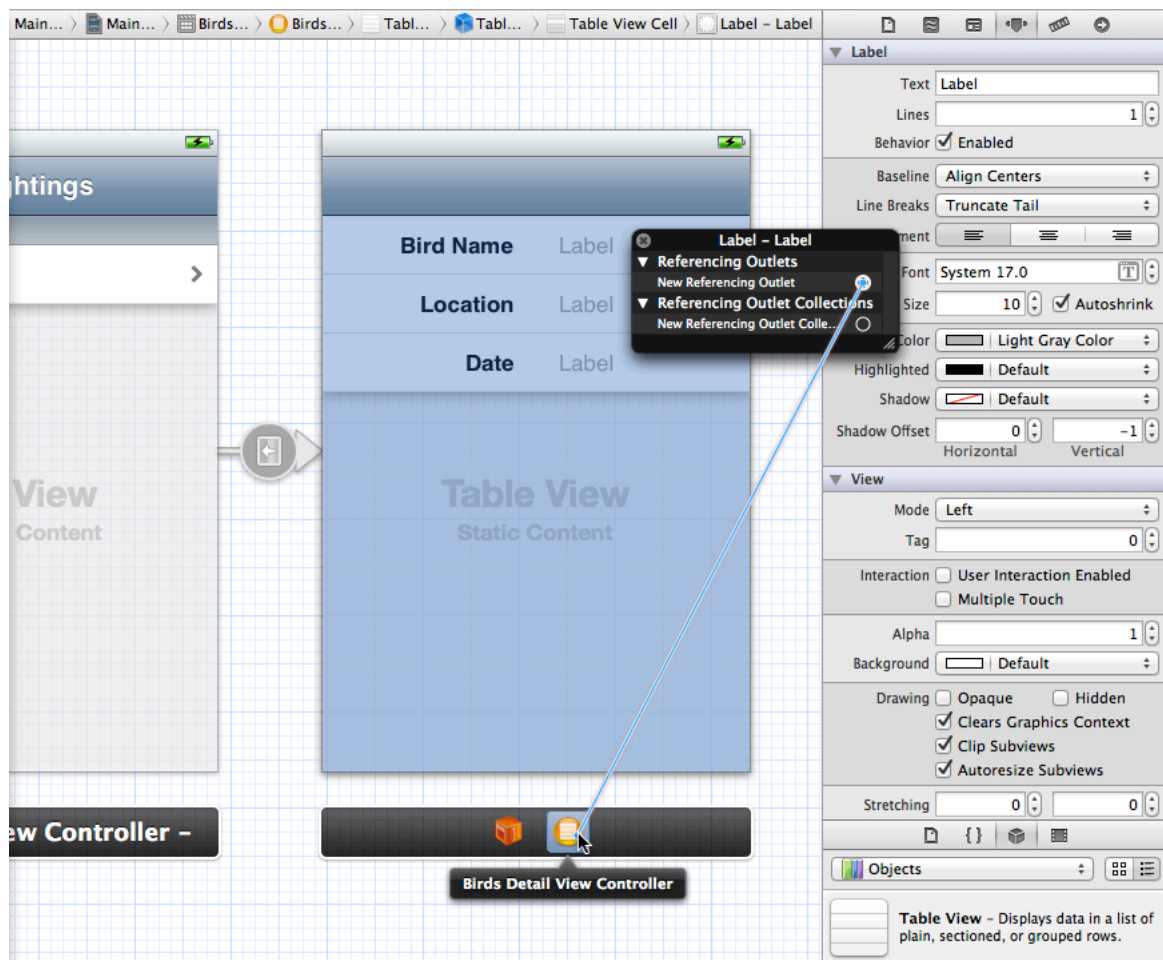


アプリケーションを実行したときに、各セルの右のラベルに、選択状態のBirdSightingオブジェクト内の情報の1つが表示されるようにします。すでに、Detail View Controllerに3つのプロパティを与え、それぞれがBirdSightingオブジェクトの1つのプロパティにアクセスするようにしてあります。この手順では、詳細シーンの各ラベルをDetail View Controllerの適切なプロパティに接続します。

1. Bird Nameセルの右のラベルを選択し、Controlキーを押しながらクリックします。
2. 表示される半透明のパネルで、New Referencing Outlet項目の白丸からシーンドックのBirdsDetailViewControllerオブジェクトまで、Controlキーを押しながらドラッグします。

シーンドックとは、シーンの下にあるバーのことです。シーン（またはその中の任意の要素）を選択すると、通常は次の2つのプロキシオブジェクトが表示されます。最初のレスポンドを表すオレンジ色の立方体と、そのシーンのView Controllerオブジェクトを表す黄色の球体です。また場合によっては、シーンドックにそのシーンの名前を表示することもできます。

Controlキーを押しながらドラッグすると、次のようになるはずです。



3. Controlキーを押しながらのドラッグを終えたときに表示される「接続(Connections)」パネルで、`birdNameLabel`を選択します。
4. Locationセルの右のラベルについても手順1、2、3を実行しますが、今回は「接続(Connections)」パネルで`locationLabel`を選択します。
5. Dateセルの右のラベルについても手順1、2、3を実行しますが、今回は「接続(Connections)」パネルで`dateLabel`を選択します。

詳細シーンのUIのすべての要素をコードで接続したように思えますが、マスタリストで選択した項目を表す`BirdSighting`オブジェクトに、詳細シーンからアクセスすることはやはりできません。これは、次の手順で解決します。

詳細シーンへのデータ送信

Storyboardsでは、prepareForSegueメソッドを介してシーン間でデータを簡単に受け渡すことができます。このメソッドは、最初のシーン（ソース）が次のシーン（遷移先）にトランジションしようとしているときに呼び出されます。ソースのView Controllerは、prepareForSegueを実装して、ビューに表示すべきデータを遷移先のView Controllerに渡すなどの設定タスクを実行できます。

注意 テンプレートに組み込まれた、prepareForSegueのデフォルト実装では、詳細View ControllerのデフォルトのsetDetailItemメソッドを使っています。setDetailItemをsetSightingに置き換えた（[“野鳥観察プロパティ用のカスタムsetterメソッドを作成するには、以下の手順を実行します。”](#)（? ページ））ので、XcodeにはsetDetailItemの実装がない旨のエラーが表示されます。この問題点は、次のステップでprepareForSegueメソッドを実装する際に解消します。

prepareForSegueメソッドの実装では、マスタシーンと詳細シーン間のセグエに付与したIDが必要になります。このチュートリアルでは、IDはすでにこのメソッドのコードリストの一部になっているので、アプリケーションを一から記述するときは、このセグエのAttributesインスペクタからIDをコピーする必要があります。

1. ジャンプバーでBirdsMasterViewController.mを選択して、このファイルをエディタで開きます。
2. 詳細ビューのヘッダファイルを忘れずにインポートしてください。
ファイルの先頭は次のようなコードになっているはずです。

```
#import "BirdsDetailViewController.h"
```

3. @implementationブロックのデフォルトのprepareForSegueメソッドを、次のコードで置き換えます。

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"ShowSightingDetails"]) {
        BirdsDetailViewController *detailViewController = [segue
destinationViewController];

        detailViewController.sighting = [self.dataController
objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];
    }
}
```

しばらくすると残っていたインジケータもすべて消えます。

プロジェクトをビルドして実行します。シミュレータ上では、マスタシーンに、`BirdSightingDataController`クラスで作成したプレースホルダデータが表示されるはずです。マスタシーンには「Edit」ボタンも「Add (+)」ボタンも表示されません。これを作成するコードを `BirdsMasterViewController.m` から削除したからです。

マスタシーンで、プレースホルダ項目を選択してください。マスタシーンが詳細シーンにトランジションするため、設計したテーブル設定内の項目に関する方法が詳細シーンに表示されます。画面の左上に表示される戻るボタンをクリックして、マスタシーンに戻ります。

次の章では、ユーザが新しい野鳥観察に関する情報を入力して、マスタリストに追加できる新しいシーンを設計します。ここで、iOSシミュレータを終了します。

まとめ

この章では、テンプレートが提供する `DetailViewController` をカスタマイズして、選択した野鳥観察項目に関する詳細が表示されるようにしました。まず、`DetailViewController` のコードを編集して、親クラスを `UITableViewController` に変更し、野鳥観察の詳細を参照するプロパティを追加して、UIを更新するメソッドを実装しました。キャンバスで、テンプレートが提供する詳細シーンを `TableViewController` に置き換えました。この作業を終えたときには、マスタシーンから詳細シーンへのセグエを作成し直す必要があることを学びました。詳細シーンには常には同じ設定のデータを表示すべきなので、静的コンテンツベースのテーブルを使用して、3つのテーブルセル（野鳥観察の詳細ごとに1つずつ）をレイアウトしました。

最後に、**Master View Controller** で `prepareForSegue` メソッドを実装しました。このメソッドでは、マスタリストでのユーザの選択内容に関連付けられた `BirdSighting` オブジェクトを詳細シーンに渡すことができます。

チュートリアルはこの時点で、`BirdsDetailViewController.h` のコードは次のようになっています。

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

@class BirdSighting;

@interface BirdsDetailViewController : UITableViewController

@property (strong, nonatomic) BirdSighting *sighting;
@property (weak, nonatomic) IBOutlet UILabel *birdNameLabel;
@property (weak, nonatomic) IBOutlet UILabel *locationLabel;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
```

```
@end
```

BirdsDetailViewController.mのコードは次のようになっているはずです（このチュートリアルでは編集しなかったテンプレート提供のコードは示していません）。

```
#import "BirdsDetailViewController.h"
#import "BirdSighting.h"

@interface BirdsDetailViewController ()
- (void)configureView;
@end

@implementation BirdsDetailViewController

@synthesize sighting = _sighting, birdNameLabel = _birdNameLabel, locationLabel =
_locationLabel, dateLabel = _dateLabel;

- (void)setSighting:(BirdSighting *) newSighting
{
    if (_sighting != newSighting) {
        _sighting = newSighting;

        [self configureView];
    }
}

- (void)configureView
{
    BirdSighting *theSighting = self.sighting;

    // Cache a date formatter to create a string representation of the date object.

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }

    if (theSighting) {
        self.birdNameLabel.text = theSighting.name;
        self.locationLabel.text = theSighting.location;
        self.dateLabel.text = [formatter stringFromDate:(NSDate *)theSighting.date];
    }
}

...
@end
```

BirdsMasterViewController.mのprepareForSegueメソッドは、次のようになっているはずです（このチュートリアルの前の方でこのファイルに加えた編集内容は示していません）。

```
#import "BirdsDetailViewController.h"

...

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"ShowSightingDetails"]) {
        BirdsDetailViewController *detailViewController = [segue
destinationViewController];

        detailViewController.sighting = [self.dataController
objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];

    }
}

...

@end
```


新規項目の追加の有効化

チュートリアルはこの時点では、BirdWatchingアプリケーションは、マスタシーンのプレースホルダアイテムと、詳細シーンのアイテムに関する追加情報を表示します。ストーリーボードを使用してこの共通アプリケーション設計を簡単に設計し、実装する方法はすでに学びました。

この章では、ユーザが新しい野鳥観察に関する情報を入力して、それをマスタリストに追加できる新しいシーンを設計します。次の方法を習得します。

- 新しいNavigation Controller階層をストーリーボードに追加する方法
- シーンをモーダルに表現する方法
- ソースシーンが遷移先シーンからデータを取得できるカスタムプロトコルを設計する方法

新しいシーン用のファイルの作成

この手順では、シーン追加を管理するView Controller用にインターフェイスファイルと実装ファイルを作成します。その後、キャンバス上でシーン追加を設計します。この手順で行うView Controllerの実装はそれほど多くありませんが、プロジェクトにあらかじめ必要なファイルを追加しておいてから、キャンバス上でシーンに対して作業すると便利です。キャンバスから適切なクラスファイルにドラッグして、プロパティ宣言とアクションメソッドを設定できるからです。

1. 「ファイル(File)」 > 「新規(New)」 > 「ファイル(File)」を選びます（または、Command-Nキーを押します）。
2. 表示されるダイアログで、左側にあるiOSセクションから「Cocoa Touch」を選択します。
3. ダイアログのメイン領域で「Objective-C」クラスを選択してから、「次へ(Next)」をクリックします。
4. ダイアログの次ペインで、「クラス(Class)」フィールドにAddSightingViewControllerと入力し、「サブクラス(Subclass)」ポップアップメニューからUITableViewControllerを選択します。
各入力テキストフィールドはテーブルセルに表示されるので、シーン追加をTable View Controllerのサブクラスにします。
5. 「Targeted for iPad」と「With XIB for user interface」オプションが未選択状態になっていることを確認してから、「次へ(Next)」をクリックします。

6. 次に表示されるダイアログで、「グループ(Group)」ポップアップメニューからBirdWatchingフォルダを選択し、「作成(Create)」をクリックします。

ここで、ストーリーボードファイルにシーン追加を作成します。

1. ジャンプバーでMainStoryboard.storyboardを選択します。

作業スペースを広げるため、キャンバスの左下にある次の形のボタンをクリックして、ドキュメン

トアウトラインを閉じます。



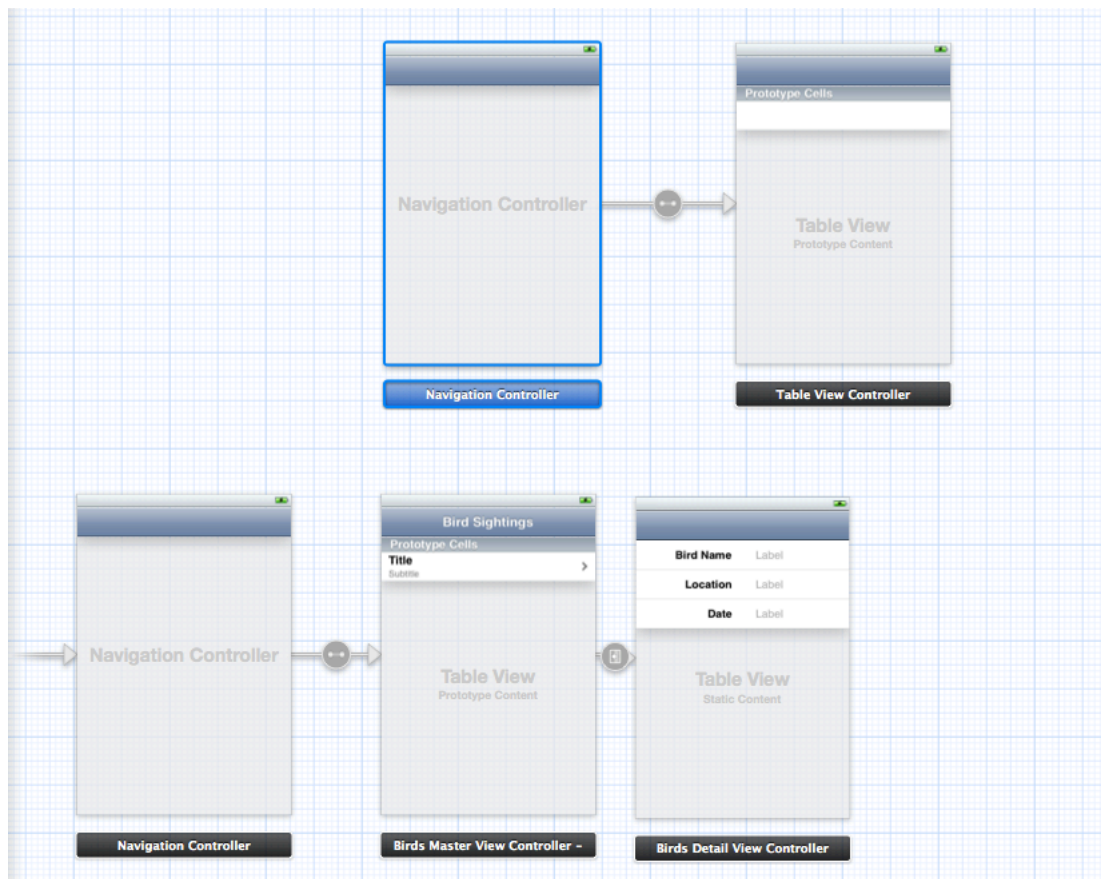
2. Table View Controllerをオブジェクトライブラリからドラッグして、キャンバス上でドロップします。

シーン追加には、「Done」ボタンと「Cancel」ボタンを配置して、ユーザが入力内容を保存したり、追加操作を取り消したりできるようにする必要があります。通常、「Done」ボタンと「Cancel」ボタンは、画面の上部にあるナビゲーションバーに属します。シーン追加がナビゲーション階層の一部になっている必要はありませんが（追加したシーンへのトランジションをユーザに許さないため）、ナビゲーションバーに配置するもっとも簡単な方法は、シーン追加をNavigation Controllerに埋め込むことです。シーンをNavigation Controllerに埋め込むと、次のような利点も生じます。

- 主要コンテンツビューがスクロールしても、ナビゲーションバーがスクロールして画面から消えることはありません。このチュートリアルでこれは問題になりませんが（シーン追加のTableViewにはセルが2つしかないため）、知っておくと有用です。
- シーン追加から追加したシーンへのトランジションまで現在の設計を拡張するのが容易になります。BirdWatchingアプリケーションの拡張を可能にする方法については、“次のステップ”（79ページ）を参照してください。

1. キャンバスでシーン追加を選択します。
2. 「エディタ(Editor)」>「Embed In」>「Navigation Controller」を選択します。

XcodeがNavigation Controllerシーンをキャンバスに追加し、関係によってシーン追加に自動的に接続します。次のようになるはずです。



ここまで、Xcodeが、特に次の2つの問題について警告し続けていることに注意してください。

- AddSightingViewControllerの実装が不完全であるおそれがあります。
- プロトタイプテーブルセルに再利用識別子が必要です。
- 今のところ、先にキャンバスに追加したシーンには到達できません。

最初の問題は、Xcodeが、Table Viewのデータソースメソッド用のスタブを新しい UITableViewController実装ファイルに自動的に入れるせいで発生します（“[Table ViewデータソースメソッドをMaster View Controllerに実装するには、以下の手順を実行します。](#)”（? ページ）で、これらのメソッドの一部をMaster View Controller用に作成済みです）。ただし、ユーザが入力した野鳥観察項目をいくつでも表示できるようにしなければならないマスタシーンのテーブルとは異なり、シーン追加のテーブルには常に同一設定のコンテンツが表示されます。そのため、マスタシーンのテーブルのときのように動的プロトタイプをベースにするのではなく、シーン追加のテーブルは、詳

細シーンのテーブルのときと同様、静的コンテンツをベースにできます。静的コンテンツをテーブルのベースにすると、**Table View Controller**が**Table View**のデータソースのニーズを自動的に処理するため、データソースメソッドを実装する必要はありません。

2番目の問題は、新しい**Table View Controller**シーンをキャンバスにドラッグしたときに、デフォルトでプロトタイプセルがベースになるせいで発生します。次のセクションでシーン追加のテーブルのベースを静的コンテンツに変更すると、静的セルには再利用識別子は不要なので、この問題も解決します。

3つめの問題は、シーンを追加しても、それだけではアプリケーションのほかのシーンに接続されないことが原因です。この問題は、マスタシーンからシーン追加へのセグエを作成する際に解消します。

シーン追加に**Table View**のデータソースメソッドは必要ないので、`AddSightingViewController.m`の次のスタブは削除（またはコメントアウト）しても構いません。

- `numberOfSectionsInTableView`
- `numberOfRowsInSection`
- `cellForRowAtIndexPath`
- `didSelectRowAtIndexPath`（これはテンプレート上で初めから注釈になっているかも知れません）

注意 静的セルを使用して詳細シーンのテーブルを設計したときには、テンプレートが提供するクラスファイルに詳細シーン用のデータソースメソッドスタブはなかったので、Xcodeが同様の警告を表示することはありませんでした。スタブがなかったのは、テンプレートが提供するデフォルト詳細シーンが、**UITableViewController**ではなく、**UIViewController**に基づくからです。

後でキャンバスで**Table View**のスタイルを指定するので、先に進む前に、デフォルト**`initWithStyle`**メソッドを削除（またはコメントアウト）しても構いません。

シーン追加のUIの設計

シーン追加をレイアウトする前に、まずシーン追加を管理するカスタム**View Controller**クラスを指定します。これを指定しないと、この章で行う作業が実行アプリケーションに何も反映されなくなります。

1. ジャンプバーで**`MainStoryboard.storyboard`**を選択します。

2. キャンバスでシーン追加を選択します。

シーン追加が埋め込まれたNavigation Controllerではなく、必ずシーン追加自体を選択します。

3. Identityインスペクタで、「クラス(Class)」ポップアップメニューからAddSightingViewControllerを選択します。

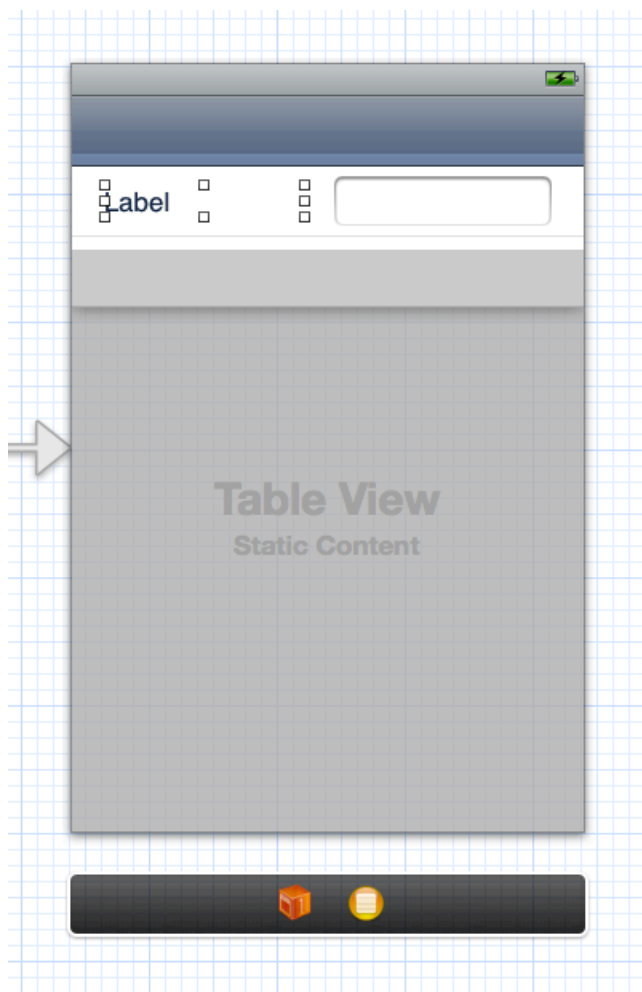
詳細シーンのときと同様、必ず同一のレイアウトを使用して入力テキストフィールドを表示することになるので、シーン追加のTable View用の静的セルを設計します。その前に、Table Viewのコンテンツタイプを動的のプロトタイプから静的セルに変更しなければなりません。

1. 必要に応じてキャンバス上のシーン追加をダブルクリックし、拡大します。
2. Table Viewを選択して、Attributesインスペクタを開きます。
3. Attributesインスペクタで、「コンテンツ(Content)」ポップアップメニューから「静的セル(Static Cells)」を選択します。

シーン追加には、野鳥名と観察場所に1つずつ、合計2つのテーブルセルが必要です。入力すべき情報がユーザにわかるように、各セルにはラベルとテキスト入力フィールドが必要です。この手順では、まず上部のテーブルセルをレイアウトしてから、その中身を複製して、下部のセルのレイアウトが容易になるようにします。

1. キャンバスで、シーン追加のTable Viewにある3つのデフォルトセルのうちの1つを選択し、Deleteキーを押します。
2. 一度に1つずつ、ラベルとテキストフィールドをオブジェクトライブラリからドラッグし、上部テーブルセルの上でドロップします。

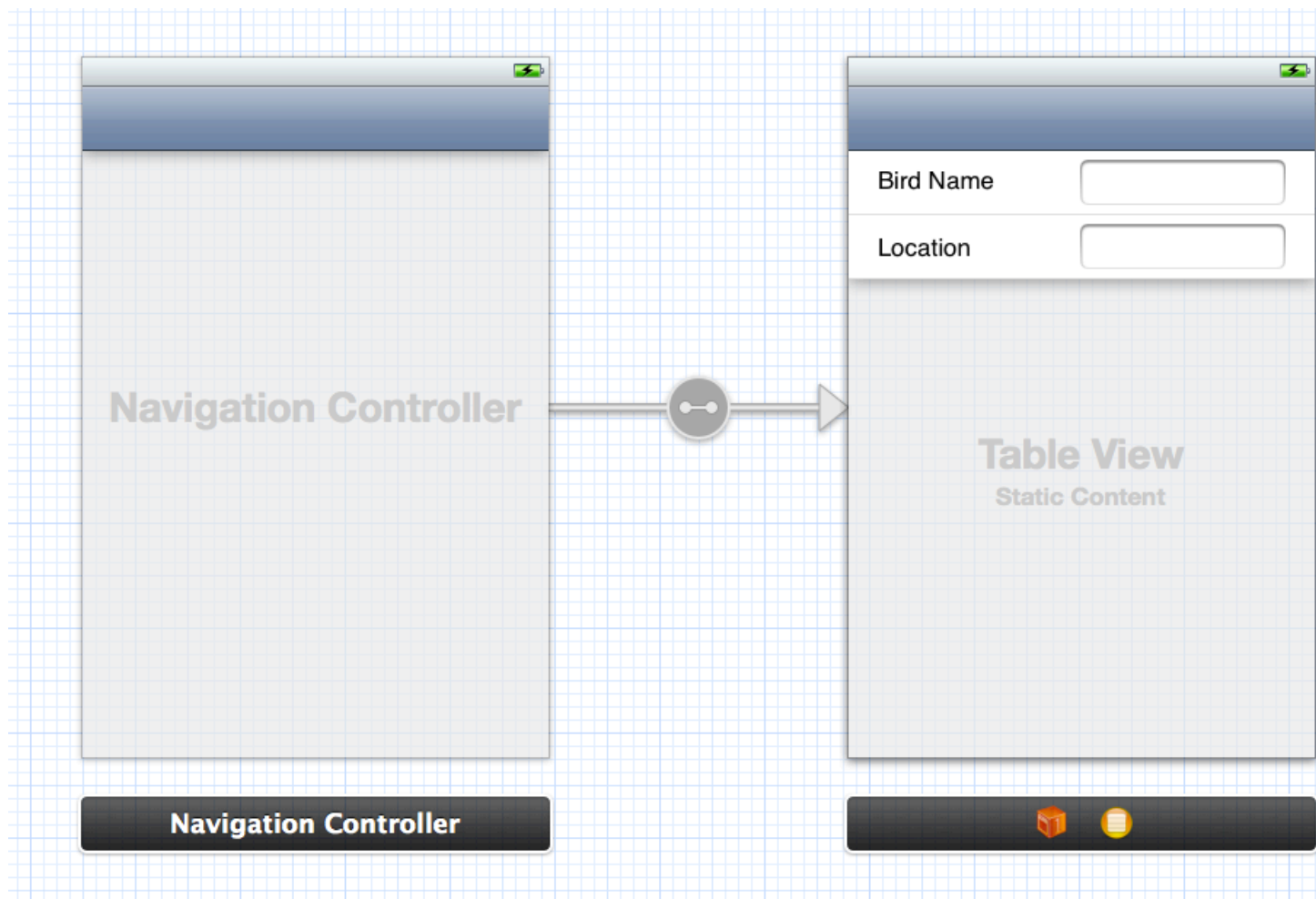
3. 上部セルで、ラベルとテキストフィールドのサイズを変更して、次のようにします。



4. テキストフィールドを選択し、Attributesインスペクタで次のように選択します。
- 「大文字化(Capitalization)」ポップアップメニューで「単語(Words)」を選択します。
 - 「キーボード(Keyboard)」ポップアップメニューが「デフォルト(Default)」に設定されていることを確認します。
 - 「Return Key」ポップアップメニューで「Done」を選択します。
5. ラベルを選択し、Command-Dキーを押して複製します。
6. 複製したラベルを下部セルまでドラッグし、上部セルのラベルに合わせて配置します。
7. 上部セルのテキストフィールドを複製して、下部セルまでドラッグし、上部セルのテキストフィールドに合わせて配置します。
8. 上部セルのラベルをダブルクリックし、デフォルトのテキストをBird Nameに置き換えます。

9. 下部セルのラベルをダブルクリックし、デフォルトのテキストをLocationに置き換えます。

シーン追加のテーブルセルのレイアウトを終えると、キャンバスは次のようになるはずです。



注意 キャンバスの表示を縮小すると、シーン追加のView Controllerのタイトル（すなわち「Add Sighting View Controller」）が、上に示したプロキシオブジェクトではなく、シーンドックに表示されます。

- ここで、シーン追加のナビゲーションバーに「Cancel」ボタンと「Done」ボタンを配置する必要があります。
1. 一度に1つずつ、2つのバーボタン項目をオブジェクトライブラリからドラッグし、ナビゲーションバーの各端に1項目ずつドロップします。

バーボタン項目をドロップする位置に関して、それほど神経質になる必要はありません。ナビゲーションバーの中央よりどちらかの端に近い位置にボタンをドロップすれば、Xcodeが自動的に適切な位置に配置します。

2. 左側のバーボタンを選択し、**Attributes**インスペクタが開いていることを確認します。
3. **Attributes**インスペクタの「バーボタン項目(Bar Button Item)」セクションで、「識別子(Identifier)」ポップアップメニューから「キャンセル(Cancel)」を選択します。
4. 右側のバーボタンを選択し、「識別子(Identifier)」ポップアップメニューから「完了(Done)」を選択します。

UI要素用のアクションとアウトレットの作成

シーン追加の**View Controller**は、ユーザが入力した情報を取得するため、先ほどのセクションで追加した2つのテキストフィールドにアクセスする必要があります。同様に、**View Controller**は、ユーザが「Cancel」ボタンと「Done」ボタンをタップしたときに適切に応答しなければなりません。すでにプロジェクトにAddSightingViewControllerコードファイルを追加済みなので、キャンバス上のこれらの要素と**View Controller**のヘッダファイル間を接続するときに、Xcodeに適切なコードを追加させることができます。

View Controllerとテキストフィールドが実行時に通信できるようにするため、シーン追加の各テキストフィールドでは、**View Controller**コードにアウトレットが必要です。各要素から**View Controller**のヘッダファイルまでControlキーを押しながらドラッグして、入力テキストフィールド用のアウトレットを用意します。

1. Xcodeツールバーの「ユーティリティ(Utility)」ボタンをクリックしてユーティリティ領域を非表示にし、「アシスタントエディタ(Assistant editor)」ボタンをクリックして「アシスタントエディタ(Assistant editor)」ペインを表示します。

「アシスタントエディタ(Assistant editor)」ボタンは、（「ユーティリティ(Utility)」ボタンを含む）3つのビューボタンの左端にあり、次のような形をしています。

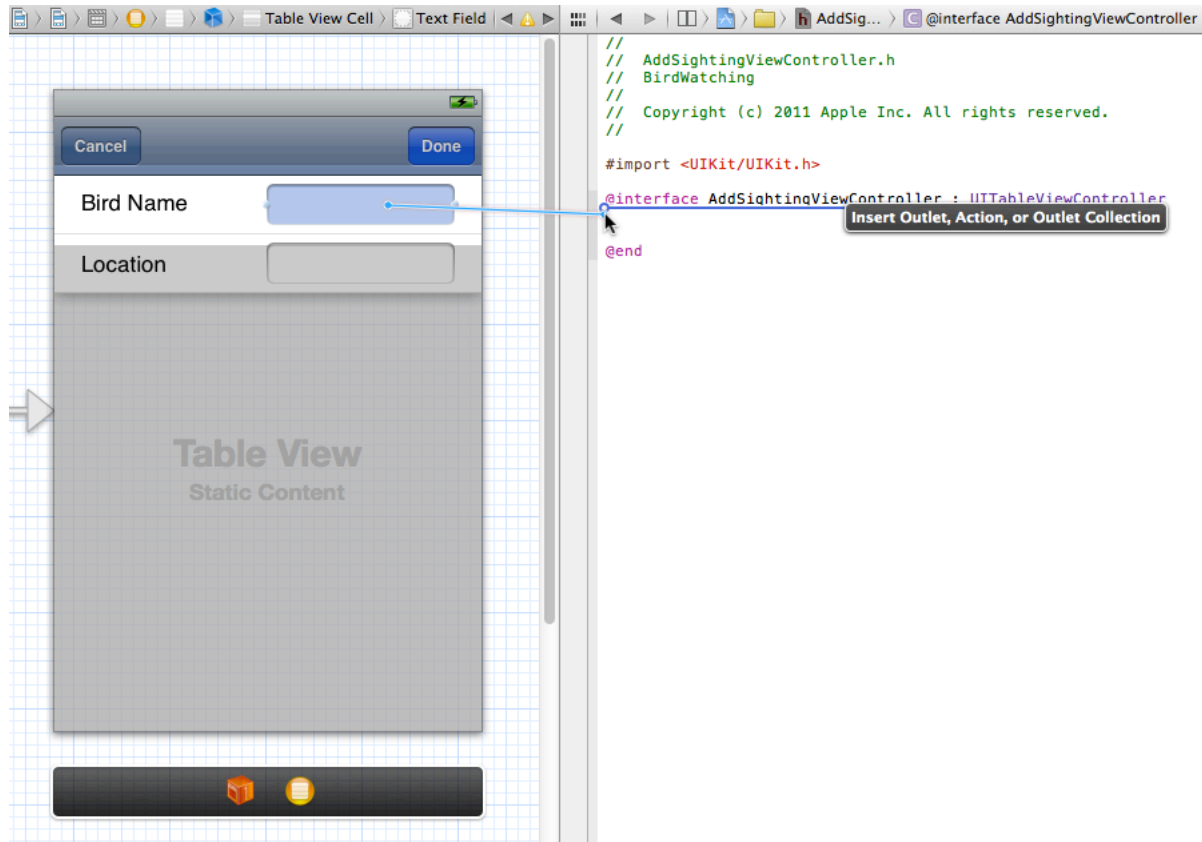


2. Assistantが**View Controller**ヘッダファイル（つまり、AddSightingViewController.h）を表示したことを確認します。

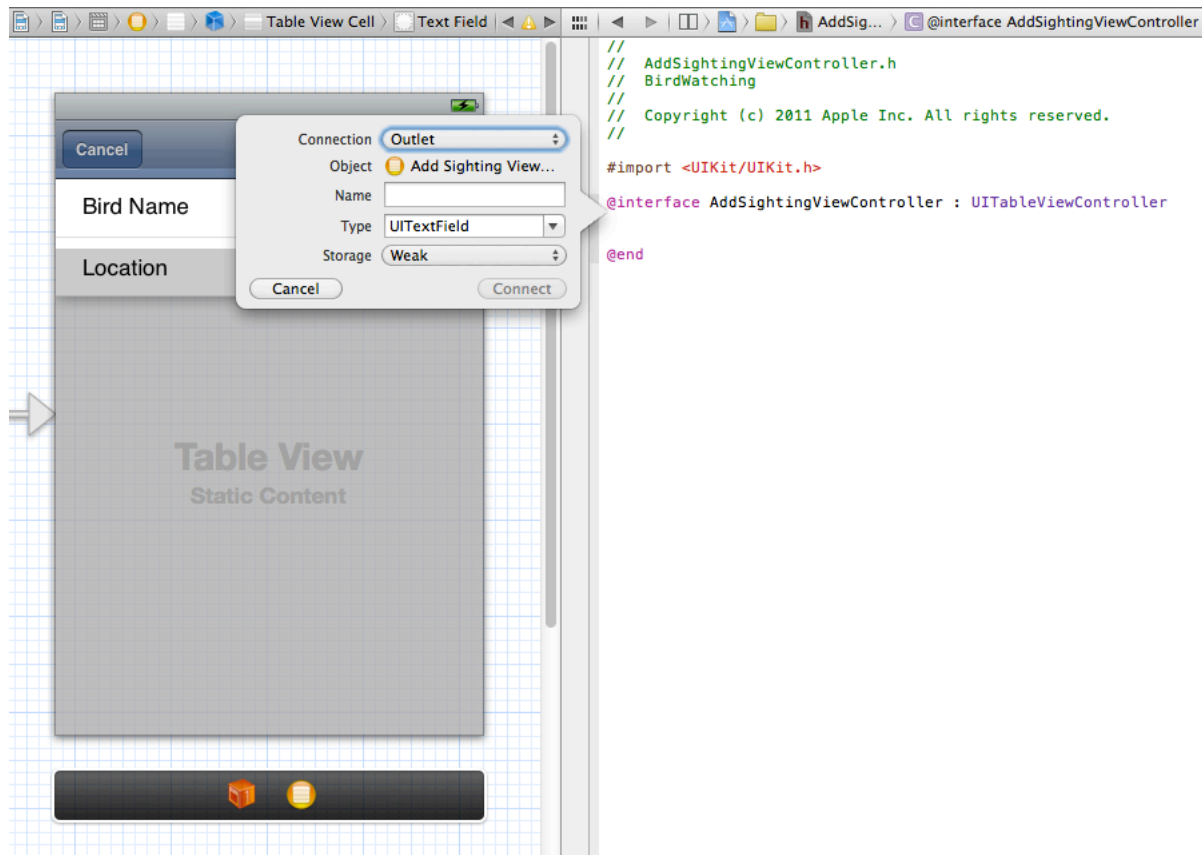
Assistantが別のファイルを表示した場合は、シーン追加のIDをAddSightingViewControllerに設定するのを忘れたか、シーン追加がキャンバス上で選択されていない可能性があります。

3. BirdNameセルのテキストフィールドからヘッダファイルのメソッド宣言領域まで、Controlキーを押しながらドラッグします。

Controlキーを押しながらドラッグすると、次のようになるはずです。



Controlキーを押しながらのドラッグを終えると、Xcodeは、先ほど作成したアウトレット接続を設定するためのPopoverを表示します。



4. Controlキーを押しながらのドラッグを終えたときに表示されるPopoverで、アウトレットを次のように設定します。
 - 「接続(Connection)」ポップアップメニューに「アウトレット(Outlet)」があることを確認します。
 - 「名前(Name)」フィールドに、birdNameInputと入力します。
 - 「タイプ(Type)」フィールドにUITextFieldがあり、「ストレージ(Storage)」ポップアップメニューにWeakがあることを確認します。
5. Popoverで「接続(Connect)」をクリックします。
6. 手順3、4、5と同様にして、Locationテーブルセルにテキストフィールド用のアウトレットを作成します。
 - テキストフィールドから、XcodeがbirdNameInput用に追加した@propertyステートメントとヘッダファイル内の@endステートメント間の領域まで、Controlキーを押しながらドラッグします。

- **Control**キーを押しながらのドラッグを終えたときに表示されるPopoverで、「接続(Connection)」メニューに「アウトレット(Outlet)」があることを確認し、「名前(Name)」フィールドに `locationInput` と入力します。
- 「タイプ(Type)」フィールドに `UITextField` があり、「ストレージ(Storage)」メニューに `Weak` があることを確認してから、「接続(Connect)」をクリックします。

Controlキーを押しながらのドラッグで、各テキストフィールドとView Controller間にアウトレット接続を確立すると、View Controllerのファイルにコードを追加するようにXcodeに指示したことになります。特に、Xcodeは次の宣言を `AddSightingViewController.h` に追加します。

```
@property (weak, nonatomic) IBOutlet UITextField *birdNameInput;
@property (weak, nonatomic) IBOutlet UITextField *locationInput;
```

`AddSightingViewController.m` に対して、Xcodeは、プロパティに適した `@synthesize` ステートメントを追加し、プロパティを `viewDidUnload` メソッドの `nil` に設定します。

```
@implementation AddSightingViewController
@synthesize birdNameInput = _birdNameInput;
@synthesize locationInput = _locationInput;
...
- (void)viewDidUnload
{
    [self setBirdNameInput:nil];
    [self setLocationInput:nil];
    [super viewDidUnload];
}
```

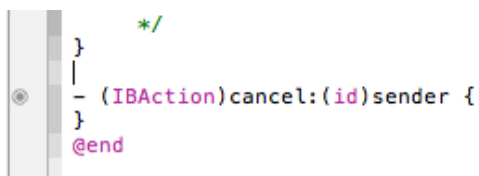
注意 Xcodeが自動的にインスタンス変数を `_variableName` (つまり、`birdNameInput = _birdNameInput`) に割り当てなかった場合は、手作業で割り当ててください。

ユーザがシーン追加の「Cancel」ボタンまたは「Done」ボタンをタップすると、そのボタンは、対応するアクションメソッドの実行方法を知っているオブジェクトにアクションメッセージを送信することができます。このチュートリアルでは、これらのボタンは、シーン追加のView Controllerにメッセージを送信します。これらのメッセージを簡単に作成したい場合は、キャンバス上のUI要素からヘッダファイルまで**Control**キーを押しながらのドラッグをしたときにコードを追加するXcodeの機能を活用できます。

この手順を行うときは、AddSightingViewController.hが「アシスタント(Assistant)」ペインにまだ表示されていることを確認します。

1. シーン追加の「Cancel」ボタンからAddSightingViewController.hのメソッド宣言領域まで（すなわち、最後の@property文の直後から、@end文の直前まで）、Controlキーを押しながらドラッグします。
2. Controlキーを押しながらのドラッグを終えたときに表示されるPopoverで、ボタンの接続を次のように設定します。
 - 「接続(Connection)」ポップアップメニューで「アクション(Action)」を選択します。
 - Nameフィールドに、cancel:と入力します（コロンを忘れないでください）。
 - Typeフィールドにidがあることを確認します。
3. Popoverで「接続(Connect)」をクリックします。

Xcodeは、AddSightingViewController.hにメソッド宣言を追加するだけでなく、実装ファイルにこのメソッドのスタブ実装も追加します。



```
    }  
    - (IBAction)cancel:(id)sender {  
    }  
@end
```

新しいメソッドの左側にある黒丸に注意してください。このアイコンは、接続が完了したことを表します。

4. 同様に、「Done」ボタン用のアクション接続も作成します。
 - キャンバス上の「Done」ボタンからAddSightingViewController.hのメソッド宣言領域まで、Controlキーを押しながらドラッグします。
 - Controlキーを押しながらのドラッグを終えたときに表示されるPopoverで、「接続(Connection)」ポップアップメニューから「アクション(Action)」を選択し、「名前(Name)」フィールドにdone:と入力します。
 - 「タイプ(Type)」フィールドにidがあることを確認して、「接続(Connect)」をクリックします。

これで、アウトレットとアクションメソッドの準備が終わったので、「標準エディタ(Standard editor)」ボタンをクリックしてAssistant editorを閉じ、キャンバス編集領域を広げます。「標準エディタ(Standard editor)」ボタンは次のような形をしています。



ユーザの入力内容を処理してシーン追加を消去するコードを記述する前に、テキストフィールドとシステムキーボード間のやり取りを管理する必要があります。『*≫YourFirstiOSApp*』 (『*Start Developing iOS Apps Today*』の一部) で学んだとおり、アプリケーションは、テキストフィールドなどのテキスト入力要素のファーストレスポンドの状態を切り替えたときの副次効果として、システムキーボードの表示と消去を行うことができます。UITextFieldDelegate プロトコルには、ユーザがキーボードを消去したいときにテキストフィールドが呼び出す `textFieldShouldReturn:` メソッドが含まれています。テキストフィールドのデリゲートの役割を果たすオブジェクトを選択し、このメソッドを実装して、テキストフィールドのファーストレスポンドの状態を無効にすることで、キーボードを消去します。BirdWatching アプリケーションでは、各テキストフィールドのデリゲートをシーン追加に設定し、必要なプロトコルメソッドを `AddSightingViewController.m` に実装します。

1. 必要に応じてキャンバス上のシーン追加をダブルクリックし、拡大します。
2. 上部のテキストフィールドをクリックして選択します。
3. テキストフィールドから、シーンドックに表示された **View Controller** プロキシオブジェクトまで、**Control** キーを押しながらドラッグします。

View Controller プロキシオブジェクトは黄色の球体で表します。

4. 表示される半透明のパネルの「アウトレット(Outlets)」セクションで、`delegate` を選択します。
5. シーン追加で下部のテキストフィールドを選択し、手順3と4を繰り返します。
6. プロジェクトナビゲータで、`AddSightingViewController.h` を選択します。
7. `@interface` コード行を次のように更新します。

```
@interface AddSightingViewController : UITableViewController
<UITextFieldDelegate>
```

このようにして `UITextFieldDelegate` を `@interface` 行に追加すると、テキストフィールドデリゲートプロトコルを採用するようにシーン追加に指示することになります。

8. プロジェクトナビゲータで、`AddSightingViewController.m` を選択します。
9. 実装ブロックに次のコードを追加して、`UITextFieldDelegate` プロトコルメソッドを実装します。

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    if ((textField == self.birdNameInput) || (textField == self.locationInput))
    {
        [textField resignFirstResponder];
    }

    return YES;
}
```

```
}
```

この実装のifステートメントによって、ユーザがどのボタンをタップした場合でも、テキストフィールドがファーストレスポンドの状態を確実に無効にします。

ユーザの入力の取得

“ストーリーボードとそのシーンの調査”（13 ページ）で学んだとおり、セグエは、ソースシーンから遷移先シーンへ単方向のトランジションを表します。セグエを使用してデータを遷移先に渡せるようになりますが（マスタシーンと詳細シーン間のセグエで行ったとおり）、セグエを使用して遷移先からソースにデータを送ることはできません。セグエを使う代わりに、データを返したくなったときに遷移先のView Controllerが呼び出すメソッドを宣言するデリゲートプロトコルを作成します。デリゲートプロトコルを定義して遷移先シーンからデータを取得する仕組みを実装すると、データをモデルオブジェクトにさかのぼって反映させるよう、アプリケーションを制御できるようになります。後述する“マスタシーンの編集とシーン追加への接続”（72 ページ）で、これらのデリゲートメソッドをソースのView Controller（つまり、マスタシーン）に実装し、そのマスタシーンをシーン追加のデリゲートとして割り当てる予定です。

1. AddSightingViewController.hで、#import行の後に次のコードを追加します。

```
@protocol AddSightingViewControllerDelegate;
```

2. @interface行の後に次のコード行を追加して、デリゲートプロパティを宣言します。

```
@property (weak, nonatomic) id <AddSightingViewControllerDelegate> delegate;
```

注意 delegateプロパティを宣言すると、アクセサメソッドを用意する必要があることをXcodeが警告します。この問題は、doneとcancelメソッドを実装するときに解決する予定です。

シーン追加にはデリゲートとして割り当てられるオブジェクトがわかりませんが、プロトコルメソッドを適切に呼び出せるようにするためには、そのオブジェクトへの参照が必要です。デリゲートへの参照をシーン追加に与える最善の方法は、デリゲートプロパティを宣言することです。

なお、<AddSightingViewControllerDelegate>をプロパティ宣言に追加すると、デリゲートプロトコルに従うオブジェクトしかプロパティに割り当てられないようにするために、コンパイラに指示することになります。

3. @end行の後に次のコード行を追加して、プロトコルメソッドを宣言します。

```
@protocol AddSightingViewControllerDelegate <NSObject>
- (void)addSightingViewControllerDidCancel:(AddSightingViewController
*)controller;
- (void)addSightingViewControllerDidFinish:(AddSightingViewController
*)controller name:(NSString *)name location:(NSString *)location;
@end
```

シーン追加は、ユーザが「Done」ボタンをタップしたときは...DidFinishプロトコルメソッド、ユーザが「Cancel」ボタンをタップしたときは...DidCancelメソッドを呼び出さなければなりません。この呼び出しを行うため、“「キャンセル(Cancel)」ボタンと「Done」ボタン用のアクションを用意するには、以下の手順を実行します。” (? ページ) でアクション接続を作成したときにXcodeが追加したdoneとcancelメソッド用のスタブを実装します (該当するメソッドスタブを見つけるためには、実装ファイルを下方にスクロールしなければならないかも知れません)。

1. プロジェクトナビゲータで、AddSightingViewController.mを選択します。
2. 2番目の@synthesizeコード行の後に次のコード行を追加して、デリゲートプロパティ用のアクセサメソッドを合成します。

```
@synthesize delegate = _delegate;
```

3. 中括弧の間に次のコード行を追加して、doneメソッドを実装します。

```
[[self delegate] addSightingViewControllerDidFinish:self
name:self.birdNameInput.text location:self.locationInput.text];
```

4. 中括弧の間に次のコード行を追加して、cancelメソッドを実装します。

```
[[self delegate] addSightingViewControllerDidCancel:self];
```

デリゲートとして割り当てられたオブジェクトに応じてシーン追加が2つのプロトコルメソッドを呼び出す仕組みに注意してください。

残ったタスクは、**MasterViewController**をシーン追加のデリゲートとして割り当て、プロトコルメソッドを実装し、マスタシーンをシーン追加に接続することだけです。これを次の手順で行います。

マスタシーンの編集とシーン追加への接続

これで、シーン追加からマスタシーンにデータを送り返せるデリゲートプロトコルの作成が終わったので、このプロトコルに従うべきクラスを識別して、そのメソッドを実装する必要があります。慣習として、シーンをモーダルに表現する**View Controller**は、シーンを消去する**View Controller**でもなければなりません。このチュートリアルでは、シーン追加を表示し、シーン追加からの新たな情報でリストを更新する方法を知っているのがマスタシーンなので、マスタシーンがプロトコルメソッドを実装します。

1. `BirdsMasterViewController.m`を編集領域に開きます。

必要に応じて「標準エディタ(Standard editor)」ボタンをクリックして「アシスタントエディタ(Assistant editor)」ペインを閉じ、スペースを広げます。

2. `@implementation`行の上に次のコード行を追加して、シーン追加のヘッダファイルをインポートします。

```
#import "AddSightingViewController.h"
```

3. 最後の`#import`ステートメントと`@implementation`ステートメントとの間に次のコード行を追加して、デリゲートプロトコルへの**View Controller**の準拠を宣言します。

```
@interface BirdsMasterViewController () <AddSightingViewControllerDelegate>
@end
```

`BirdsMasterViewController`の実装が不完全なことを、**Xcode**が警告します。デリゲートメソッドを実装すれば、この警告は消えます。

4. 実装ブロックに次のコード行を追加して、`addSightingViewControllerDidCancel`プロトコルメソッドを実装します。

```
- (void)addSightingViewControllerDidCancel:(AddSightingViewController *)controller {
    [self dismissViewControllerAnimated:YES completion:NULL];
}
```


ユーザが「Cancel」をタップしたときに入力内容を保存する必要はないので、このメソッドはそのままシーン追加を消去します。

5. 次のコード行を追加して、addSightingViewControllerDidFinishメソッドを実装します。

```
- (void)addSightingViewControllerDidFinish:(AddSightingViewController
*)controller name:(NSString *)name location:(NSString *)location {
    if ([name length] || [location length]) {
        [self.dataController addBirdSightingWithName:name location:location];
        [[self tableView] reloadData];
    }

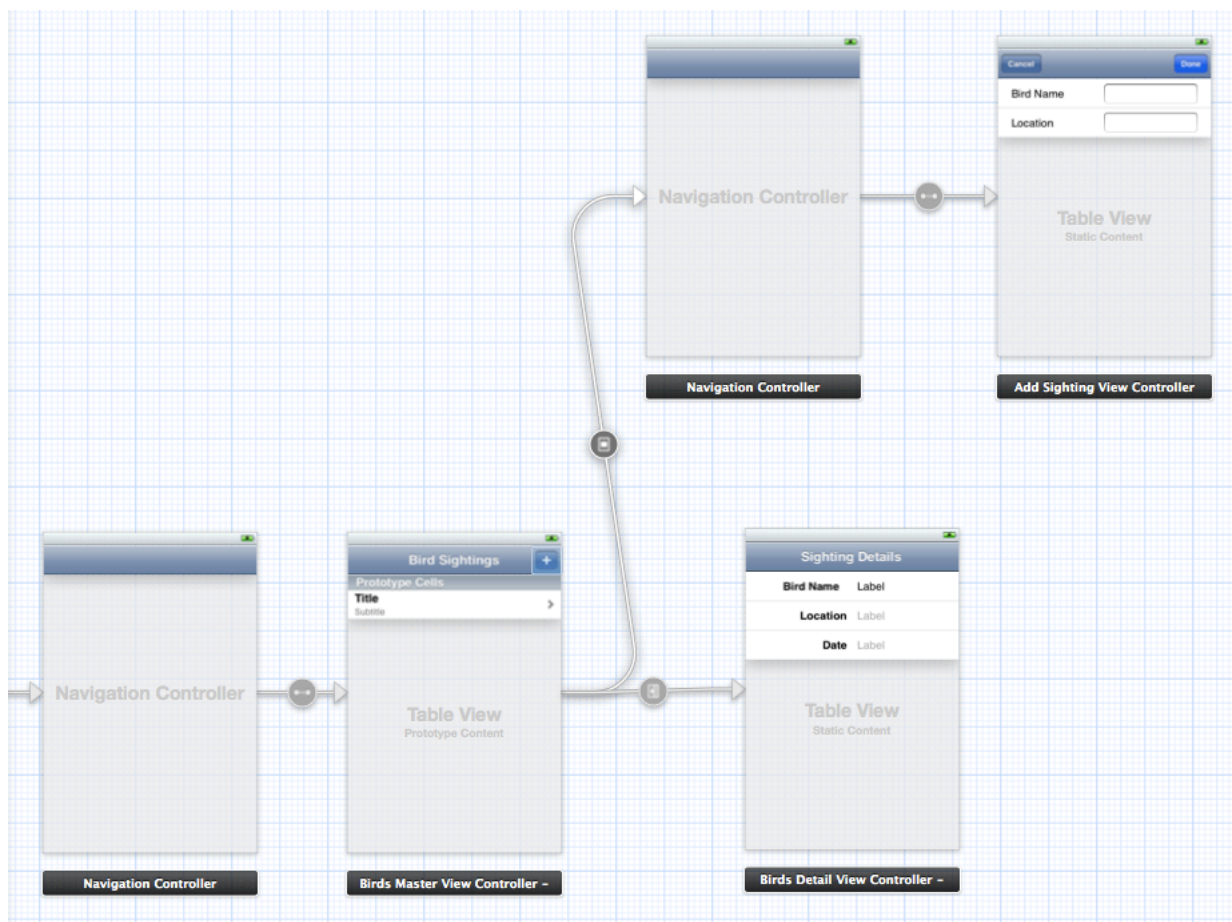
    [self dismissModalViewControllerAnimated:YES];
}
```

このメソッドは、ユーザの入力内容をData ControllerのaddBirdSightingWithName:location:メソッドに送信して、新しいBirdSightingオブジェクトが作成されるようにします。その後、シーン追加を消去して、マスタリストを更新し、その新しいアイテムを表示します。

iOSアプリケーションでは、ユーザは、通常、モーダルに表される自己完結型の画面に新しい情報を入力します。たとえば、ユーザが、iPhoneの「連絡先(Contacts)」で「追加(Add)」ボタンをタップすると、情報を入力できる新しい画面がスライドして現れます。ユーザがタスクを終了すると（またはタスクを取り消すと）、画面はスライドして消え、前の画面が再表示されます。この設計パターンに従うためには、ユーザがマスタシーンの「追加(Add)」ボタンをタップしたときにシーン追加がスライド表示し、ユーザがシーン追加の「Done」または「Cancel」ボタンをタップするとスライド消去されるようにしなければなりません。この動作は、マスタシーンからシーン追加へのモーダルなセグエを作成することで実現します。

1. ジャンプバーでMainStoryboard.storyboardを選択して、ストーリーボードファイルを開きます。
2. すべてのシーンが表示されるように縮小してから、マスタシーンをダブルクリックして拡大します。
3. 「ユーティリティ(Utility)」ボタンをクリックして、ユーティリティ領域を開きます。
4. オブジェクトライブラリからマスタシーンのナビゲーションバーの右端まで、バーボタン項目をドラッグします。
5. バーボタンを選択状態にしたままで、Attributesインスペクタを開き、「識別子(Identifier)」ポップアップメニューから「追加(Add)」を選択します。
6. キャンバスで、「追加(Add)」ボタンから、シーン追加が入っているNavigation Controllerまで、Controlキーを押しながらドラッグします。

7. 表示される半透明のパネルで、「モーダル(Modal)」を選択します。
「モーダル(Modal)」を選択した後、キャンバスは次のようになるはずです。



Xcodeでは、「追加(Add)」ボタンからではなく、マスタシーンの中央から新しいセグエが現れるように見えますが、実際の接続はこのボタンに関連付けられています。

8. キャンバスで、モーダルセグエをクリックして選択します。
9. **Attributes**インスペクタの「識別子(Identifier)」フィールドで、モーダルセグエを識別するIDを入力します。
このチュートリアルでは、`ShowAddSightingView`をIDとして使用します。

“設定情報を詳細シーンに送信するには、以下の手順を実行します。” (? ページ) で、ユーザがマスタリスト内の項目をタップしたときにマスタシーンがBirdSightingオブジェクトを詳細シーンに送信できるように、`prepareForSegue`メソッドを実装しました。シーン追加へのセグエを用意すると

き、マスタシーンは、シーン追加が表示するためのデータを送信する必要はありませんが、シーン追加のデリゲートとして自らを割り当てる必要があります。データを渡す必要はありませんが、`prepareForSegue`メソッドは、この設定タスクを行うのに適した場所です。

1. プロジェクトナビゲータで`BirdsMasterViewController.m`を選択します。
2. `prepareForSegue`のifステートメントの閉じ中括弧の後に、次のコード行を追加します。

```
else if ([[segue identifier] isEqualToString:@"ShowAddSightingView"]) {  
    AddSightingViewController *addController =  
    (AddSightingViewController *)[[[segue destinationViewController]  
    viewControllers] objectAtIndex:0];  
    addController.delegate = self;  
}
```

なお、上記のコードでは、セグエの`destinationViewController`プロパティを`addController`に割り当てていません。`destinationViewController`は、実際には、シーン追加自体ではなく、シーン追加を管理している`UINavigationController`のインスタンスを参照するからです。必要になる`AddSightingViewController`のインスタンスは、**Navigation Controller**の`viewControllers`配列のインデックス0です。

プロジェクトをビルドして実行します。新しい野鳥観察情報をマスタリストに追加してみてください。すべてが設計どおりに機能するはずですが、そうならなかった場合は、この章で追加したコードを、“[まとめ](#)”（75 ページ）のコードリストと比較してみてください。それでも問題が解決しない場合は、“[トラブルシューティング](#)”（77 ページ）に記載されたデバッグのアドバイスを参照してください。

まとめ

この章では、**Table View Controller**が管理する新たなシーン用のクラスファイルを作成しました。キャンバスにこのシーンを作成したときには、新たな野鳥観察に関してユーザが入力できる2つの情報のそれぞれに対応したセルを1つずつ備えた、静的コンテンツに基づくテーブルを設計しました。この作業を行う過程で、静的コンテンツに基づくテーブルを管理する**Table View Controller**が、テーブルのデータソースのニーズを自動的に処理することを学びました。また、「Cancel」ボタンと「Done」ボタンをナビゲーションバーに表示できるようにするため、**Navigation Controller**に新たなシーンを埋め込みました。

次に、Xcodeのコード追加機能を活用して、キャンバス上の項目から`AddSightingViewController`ヘッダファイルまで**Control**キーを押しながらドラッグすることで、アクションメソッド用のプロパティ宣言、合成機能、スタブを作成しました。

最後に、遷移先シーンがソースにデータを返送できるようにするデリゲートプロトコルを定義する方法を学びました。このチュートリアルでは、**Master View Controller**にこのプロトコルのデリゲートの役割をさせ、デリゲートメソッドをBirdsMasterViewController.mに実装しました。

おめでとうございます。徐々に詳細な画面に移動したり、モーダルな画面に新しい情報を入力したりなど、もっとも一般的なiOSユーザ体験をいくつか可能にするアプリケーションの作成が完了しました。アプリケーションを正しく動作させる上で問題が生じている場合は、“[トラブルシューティング](#)”（77 ページ）に記載されたデバッグのアドバイスを参照すると共に、自分のコードを“[コードリスト](#)”（82 ページ）のリストと比較してみてください。また、より高度な課題に挑戦したい場合は、BirdWatchingアプリケーションの拡張や改良の方法を紹介した“[次のステップ](#)”（79 ページ）を参照してください。

トラブルシューティング

BirdWatchingアプリケーションが正しく機能しないトラブルが発生した場合は、この章に記載された問題解決のアプローチを試します。

コードおよびコンパイラの警告

アプリケーションが期待どおりに動作しない場合は、まず、自分のコードと“[コードリスト](#)”（82ページ）に示されたリストを比較します。

このチュートリアルコードは、どのような警告も表示されることなく、コンパイルできなければなりません。Xcodeが警告を報告した場合は、エラーを扱うときと同じ方法で扱うことをお勧めします。Objective-C言語は非常に柔軟な言語なので、エラーを引き起こす可能性がある問題があっても、せいぜい警告しか表示されない場合があります。

ストーリーボード項目と接続

アプリケーションの問題をコード内で解決することに慣れていると、ストーリーボード内のオブジェクトを検査することを忘れがちです。ストーリーボードの大きな利点は、アプリケーションオブジェクトの外観と一部の設定の両方を取得できるおかげで、コーディング量が少なくて済むことです。この利点を活かすためには、アプリケーションが期待どおりに動作しないときに、コードに加えて、ストーリーボードも検査することが重要です。

メソッドで送信したデータをシーンが受け取っていないように思える場合は、Attributesインスペクタでセグエの識別子を調べます。prepareForSegueメソッドで使ったものと同じ識別子をセグエに与えるのを忘れても、テスト中にシーンへのトランジションを行うことはできますが、送信したデータは表示されません。

prepareForSegueメソッドでセグエの識別子のスペルを間違えたときも、同じ結果になります。識別子のスペルを間違えたり、prepareForSegueメソッドで不適切な識別子を使用したりしても、Xcodeは警告しないため、これらの値を自分で調べることが重要です。

カスタムView Controllerクラスを変更してもシーンに反映されていないように見える場合は、Identityインスペクタでそのシーンのクラスを調べます。カスタムView Controllerクラスの名前が「クラス (Class)」ポップアップメニューに表示されない場合、Xcodeはその変更をシーンに適用しません。

シーン追加のナビゲーションバーにある「Cancel」ボタンまたは「Done」ボタンをクリックしても何も起こらない場合は、それらのボタンのアクションがView Controllerに接続されていない可能性があります。

入力用のテキストフィールドが表示されたのに、入力したデータがマスタリストに表示されない場合は、View Controllerのアウトレットがそのテキストフィールドに接続されていない可能性があります。

デリゲートメソッド名

デリゲートの使用に関してよくある間違いは、デリゲートメソッド名のスペルミスです。たとえデリゲートオブジェクトを正しく設定しても、そのデリゲートがメソッドの実装で正確な名前を使用していなければ、正しいメソッドを呼び出すことはできません。通常は、信頼性の高いソース（技術ドキュメントやヘッダファイルの宣言など）から、`textFieldShouldReturn:`や`addSightingViewControllerDidFinish:`など、デリゲートメソッドの宣言をコピーアンドペーストするのが最善です。

次のステップ

このチュートリアルでは、簡単ですが機能的なアプリケーションを作成しました。習得した知識を活用するため、この章で紹介したいいくつかの方法でBirdWatchingアプリケーションを拡張してみてください。

ユーザインターフェイスとユーザ体験の改善

iOSユーザの高い期待に応えるため、アプリケーションは、優れたユーザインターフェイスとユーザ体験を備えていなければなりません。無駄な装飾を追加するのは避けるのが最善ですが、一部のビューの背景に着色したり簡単な画像を表示したりすると、BirdWatchingアプリケーションの見栄えが向上する可能性があります。

iOSユーザは、一般的に、iOSベースのデバイスをどの向きでも使用できると期待するため、開発したアプリケーションをさまざまな向きでサポートすることをお勧めします。これを容易にするため、まず`shouldAutorotateToInterfaceOrientation`:メソッドについて学びます。次に、このストーリーボードのシーンを必要に応じて調整して、ビューが回転してもUI要素がすべて適切に配置されるようにします。

BirdWatchingアプリケーションでは新しい野鳥観察イベントを簡単に追加できますが、ユーザ体験は理想的とは言えません。まず、「Done」ボタンは、シーン追加が表示されるとすぐにアクティブになりますが、ユーザがキーボードのキーを押した後にアクティブになる方が便利です。また、テキストフィールドには、ユーザが入力内容をすばやく消去できる「Clear」ボタンがありません。

機能の追加

最善のiOSアプリケーションは、ユーザが主要タスクを容易かつ楽しく達成できるように、適度な量の機能のみを備えています。BirdWatchingアプリケーションでは主要タスクを容易に実行できますが、機能をいくつか追加すれば使うのがさらに楽しくなります。考慮すべき改善点は、次のとおりです。

- テキストフィールドに野鳥名を入力するようにユーザに指示する代わりに、ユーザが選択できるように野鳥名のリストを表示します。
- 今日の日付を自動的に入力する代わりに、ユーザが特定の日付を入力できるようにします。

- Location Servicesを有効にするようにユーザに指示して、アプリケーションがユーザの現在地を野鳥観察地として提案できるようにします。また、テキストフィールドに場所を入力するようにユーザに指示する代わりに、ユーザが場所を選択できるように地図を表示します。
- マスタリストの編集とソートを可能にします。これらの機能の有効化の詳細については、『*Table View Programming Guide for iOS*』を参照してください。また、先に説明したように、Master-Detail テンプレートには「Edit」ボタン（および表の配置を調整する若干の機能）が最初から組み込まれているのです。この作業は、`BirdsMasterViewController.m`の該当するコードを囲む、注釈である旨の記号を外すことから始めるとよいでしょう。
- ユーザが野鳥観察に画像を追加できるようにします。このアプリケーションで、ユーザが一連の保管画像や自分の写真から選択できるようにします。
- マスタリストを永続的に保管して、ユーザの野鳥観察画像が、アプリケーションを再起動する都度表示されるようにしてください。
- iCloudストレージを採用します。iCloudストレージをサポートすると、ユーザがアプリケーションの1つのインスタンスに加えた変更がそのユーザのほかのデバイスに自動的に反映されるため、アプリケーションのほかのインスタンスもその変更を認識できます。iCloudストレージをアプリケーションで有効にする方法については、『*Your Third iOS App: iCloud*』を参照してください。

追加の改善

iOSアプリケーション開発に習熟してきたら、BirdWatchingアプリケーションに次の変更を加えてみるとよいでしょう。

- コードを最適化します。高いパフォーマンスは、iOSの優れたユーザ体験のためにはきわめて重要です。Xcodeが提供しているさまざまなパフォーマンスツール（特に、Instruments）を使用して、必要なリソースを最小にするようにアプリケーションを調整する方法を学びましょう。
- 単体テストを追加します。テストによって、実装が変化してもメソッドがアダプタイズどおりに機能するようにします。単体テストを一から構築した新バージョンのプロジェクトを作成することもできますが、現在のプロジェクトを使用して、「ファイル(File)」>「新規(New)」>「新規ファイル(New File)」を選択し、「その他のカテゴリ(Other category)」を選んでから、「Cocoa Unit Testing Bundle」テンプレートを選択することもできます。プロジェクトを調べて、単体テストを組み込むときにXcodeが追加したものを把握します。単体テストの詳細については、『*Tools Workflow Guide for iOS*』を参照してください。
- Core Dataを使用してモデルレイヤを管理します。Core Dataの使い方を習得するためにはある程度の努力が必要ですが、モデルレイヤをサポートする際に必要になるコードを容易に効率化できます。このテクノロジーを習得するため、*Core Data Tutorial for iOS*を使用してみてください。

- アプリケーションがデバイス上で動作することを確認します。App Storeにアプリケーションを送信する際の前提条件なので、デバイスにアプリケーションをインストールしてテストするプロセスに慣れておくことをお勧めします。
- アプリケーションをユニバーサルにします。iOSユーザは、たいいていの場合、iOSベースのあらゆるタイプのデバイスで好みのアプリケーションを実行できることを期待します。アプリケーションをユニバーサルにすると、デベロッパ側の作業が必要になる可能性があります。特に、同一の基盤コードの大半を再利用できる場合でも、異なる2つのユーザインターフェイスを作成する必要があるのが一般的です。アプリケーションをユニバーサルにする際に必要な手順の詳細については、「ユニバーサルアプリケーションの作成」 in *iOS App Programming Guide* を参照してください。

コードリスト

この付録には、BirdWatchingプロジェクトのインターフェイスファイルと実装ファイルのコードリストを掲載してあります。リストには、編集しなかったコメントやメソッドは含まれていません。

モデルレイヤのファイル

このセクションには、次のファイルのリストを掲載してあります。

- BirdSighting.h
- BirdSighting.m
- BirdSightingDataController.h
- BirdSightingDataController.m

BirdSighting.h

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

@interface BirdSighting : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *location;
@property (nonatomic, strong) NSDate *date;

-(id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate *)date;

@end
```

BirdSighting.m

```
#import "BirdSighting.h"
```

```
@implementation BirdSighting
@synthesize name = _name, location = _location, date = _date;

- (id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate *)date
{
    self = [super init];
    if (self) {
        _name = name;
        _location = location;
        _date = date;
        return self;
    }
    return nil;
}

@end
```

BirdSightingDataController.h

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

@class BirdSighting;

@interface BirdSightingDataController : NSObject

@property (nonatomic, copy) NSMutableArray *masterBirdSightingList;

- (NSUInteger)countOfList;
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex;
- (void)addBirdSightingWithName:(NSString *)inputBirdName location:(NSString *)inputLocation;

@end
```

BirdSightingDataController.m

```
#import "BirdSightingDataController.h"
#import "BirdSighting.h"

@interface BirdSightingDataController ()
- (void)initializeDefaultDataList;
@end

@implementation BirdSightingDataController

@synthesize masterBirdSightingList = _masterBirdSightingList;

- (id)init {
    if (self = [super init]) {
        [self initializeDefaultDataList];
    }
    return self;
}

- (void)setMasterBirdSightingList:(NSMutableArray *)newList {
    if (_masterBirdSightingList != newList) {
        _masterBirdSightingList = [newList mutableCopy];
    }
}

- (NSUInteger)countOfList {
    return [self.masterBirdSightingList count];
}

- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex {
    return [self.masterBirdSightingList objectAtIndex:theIndex];
}
```

```
- (void)addBirdSightingWithName:(NSString *)inputBirdName location:(NSString *)inputLocation {
    BirdSighting *sighting;
    NSDate *today = [NSDate date];

    sighting = [[BirdSighting alloc] initWithName:inputBirdName
location:inputLocation date:today];
    [self.masterBirdSightingList addObject:sighting];
}

- (void)initializeDefaultDataList {
    NSMutableArray *sightingList = [[NSMutableArray alloc] init];

    self.masterBirdSightingList = sightingList;
    [self addBirdSightingWithName:@"Pigeon" location:@"Everywhere"];
}

@end
```

Master View Controllerとアプリケーションデリゲートのファイル

このセクションには、次のファイルのリストを掲載してあります。

- BirdsMasterViewController.h
- BirdsMasterViewController.m
- BirdsAppDelegate.m

BirdsMasterViewController.h

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

@class BirdSightingDataController;

@interface BirdsMasterViewController : UITableViewController
```

```
@property (strong, nonatomic) BirdSightingDataController *dataController;

@end
```

BirdsMasterViewController.m

```
#import "BirdsMasterViewController.h"
#import "BirdsDetailViewController.h"
#import "BirdSightingDataController.h"
#import "BirdSighting.h"
#import "AddSightingViewController.h"

@interface BirdsMasterViewController () <AddSightingViewControllerDelegate>
@end

@implementation BirdsMasterViewController

@synthesize dataController = _dataController;

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"ShowSightingDetails"]) {
        BirdsDetailViewController *detailViewController = [segue
            destinationViewController];

        detailViewController.sighting = [self.dataController
            objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];

    } else if ([[segue identifier] isEqualToString:@"ShowAddSightingView"]) {
        AddSightingViewController *addController = (AddSightingViewController
            *)[[segue destinationViewController] viewControllers] objectAtIndex:0;
        addController.delegate = self;
    }
}
```

```
}

- (void)addSightingViewControllerDidCancel:(AddSightingViewController *)controller
{
    [self dismissViewControllerAnimated:YES completion:NULL];
}

- (void)addSightingViewControllerDidFinish:(AddSightingViewController *)controller
name:(NSString *)name location:(NSString *)location {

    if ([name length] || [location length]) {
        [self.dataController addBirdSightingWithName:name location:location];
        [[self tableView] reloadData];
    }

    [self dismissModalViewControllerAnimated:YES];
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [self.dataController countOfList];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"BirdSightingCell";

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
}
```

```
        UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];

        BirdSighting *sightingAtIndex = [self.dataController
objectInListAtIndex:indexPath.row];

        [[cell.textLabel] setText:sightingAtIndex.name];

        [[cell.detailTextLabel] setText:[formatter stringFromDate:(NSDate
*)sightingAtIndex.date]];

        return cell;

    }

    - (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath
*)indexPath {        return NO; }

@end
```

BirdsAppDelegate.m

```
#import "BirdsAppDelegate.h"
#import "BirdSightingDataController.h"
#import "BirdsMasterViewController.h"

@implementation BirdsAppDelegate

@synthesize window = _window;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UINavigationController *navigationController = (UINavigationController
*)self.window.rootViewController;

    BirdsMasterViewController *firstViewController = (BirdsMasterViewController
*)[[navigationController viewControllers] objectAtIndex:0];

    BirdSightingDataController *aDataController = [[BirdSightingDataController
alloc] init];

    firstViewController.dataController = aDataController;
```



```
        return YES;
    }
@end
```

Detail View Controllerのファイル

このセクションには、次のファイルのリストを掲載してあります。

- BirdsDetailViewController.h
- BirdsDetailViewController.m

BirdsDetailViewController.h

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

@class BirdSighting;

@interface BirdsDetailViewController : UITableViewController
@property (strong, nonatomic) BirdSighting *sighting;
@property (weak, nonatomic) IBOutlet UILabel *birdNameLabel;
@property (weak, nonatomic) IBOutlet UILabel *locationLabel;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;

@end
```

BirdsDetailViewController.m

```
#import "BirdsDetailViewController.h"
#import "BirdSighting.h"

@interface BirdsDetailViewController ()
- (void)configureView;
@end

@implementation BirdsDetailViewController
```

```
@synthesize sighting = _sighting, birdNameLabel = _birdNameLabel, locationLabel =
_locationLabel, dateLabel = _dateLabel;

- (void)setSighting:(BirdSighting *) newSighting
{
    if (_sighting != newSighting) {
        _sighting = newSighting;

        // ビューを更新する
        [self configureView];
    }
}

- (void)configureView
{
    BirdSighting *theSighting = self.sighting;

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }

    if (theSighting) {
        self.birdNameLabel.text = theSighting.name;
        self.locationLabel.text = theSighting.location;
        self.dateLabel.text = [formatter stringFromDate:(NSDate *)theSighting.date];
    }
}

...
@end
```

Add Scene View Controllerのファイル

このセクションには、次のファイルのリストを掲載してあります。

- AddSightingViewController.h
- AddSightingViewController.m

AddSightingViewController.h

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

@protocol AddSightingViewControllerDelegate;

@interface AddSightingViewController : UITableViewController <UITextFieldDelegate>

@property (weak, nonatomic) id <AddSightingViewControllerDelegate> delegate;

@property (weak, nonatomic) IBOutlet UITextField *birdNameInput;
@property (weak, nonatomic) IBOutlet UITextField *locationInput;

- (IBAction)cancel:(id)sender;
- (IBAction)done:(id)sender;

@end

@protocol AddSightingViewControllerDelegate <NSObject>

- (void)addSightingViewControllerDidCancel:(AddSightingViewController *)controller;
- (void)addSightingViewControllerDidFinish:(AddSightingViewController *)controller
    name:(NSString *)name location:(NSString *)location;

@end
```

AddSightingViewController.m

```
#import "AddSightingViewController.h"

@implementation AddSightingViewController
```

```
@synthesize birdNameInput = _birdNameInput, locationInput = _locationInput, delegate
= _delegate;

- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    if ((textField == self.birdNameInput) || (textField == self.locationInput)) {
        [textField resignFirstResponder];
    }
    return YES;
}

- (IBAction)done:(id)sender {

    [[self delegate] addSightingViewControllerDidFinish:self
name:self.birdNameInput.text location:self.locationInput.text];
}

- (IBAction)cancel:(id)sender {

    [[self delegate] addSightingViewControllerDidCancel:self];
}

- (void)viewDidUnload
{
    [self setBirdNameInput:nil];
    [self setLocationInput:nil];
    [super viewDidUnload];
}

...
@end
```

書類の改訂履歴

この表は「2つ目のiOSアプリケーション：ストーリーボード」の改訂履歴です。

日付	メモ
2012-02-28	細かな訂正を行いました。
2012-02-16	Xcode 4.3用に更新しました。
2012-01-09	ストーリーボードを使用してMaster-Detail形式のアプリケーションを実装する方法を解説した新規ドキュメント。



Apple Inc.
© 2012 Apple Inc.
All rights reserved.

本書の一部あるいは全部を Apple Inc. から書面による事前の許諾を得ることなく複写複製（コピー）することを禁じます。また、製品に付属のソフトウェアは同梱のソフトウェア使用許諾契約書に記載の条件のもとでお使いください。書類を個人で使用する場合に限り 1 台のコンピュータに保管すること、またその書類にアップルの著作権表示が含まれる限り、個人的な利用を目的に書類を複製することを認めます。

Apple ロゴは、米国その他の国で登録された Apple Inc. の商標です。

キーボードから入力可能な Apple ロゴについても、これを Apple Inc. からの書面による事前の許諾なしに商業的な目的で使用すると、連邦および州の商標法および不正競争防止法違反となる場合があります。

本書に記載されているテクノロジーに関しては、明示または黙示を問わず、使用を許諾しません。本書に記載されているテクノロジーに関するすべての知的財産権は、Apple Inc. が保有しています。本書は、Apple ブランドのコンピュータ用のアプリケーション開発に使用を限定します。

本書には正確な情報を記載するように努めました。ただし、誤植や制作上の誤記がないことを保証するものではありません。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
<http://www.apple.com/jp/>

App Store is a service mark of Apple Inc.

iCloud is a registered service mark of Apple Inc.

Apple, the Apple logo, Cocoa, Cocoa Touch, Instruments, iPad, iPhone, Mac, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Apple Inc. は本書の内容を確認しておりますが、本書に関して、明示的であるか黙示的であるかを問わず、その品質、正確さ、市場性、または特定の目的に対する適合性に関して何らかの保証または表明を行うものではありません。その結果、本書は「現状有姿のまま」提供され、本書の品質または正確さに関連して発生するすべての損害は、購入者であるお客様が負うものとなります。

いかなる場合も、Apple Inc. は、本書の内容に含まれる瑕疵または不正確さによって生じる直接的、間接的、特殊的、偶発的、または結果的損害に対する賠償請求には一切応じません。そのような損害の可

能性があらかじめ指摘されている場合においても同様です。

上記の損害に対する保証および救済は、口頭や書面によるか、または明示的や黙示的であるかを問わず、唯一のものであり、その他一切の保証にかわるものです。Apple Inc. の販売店、代理店、または従業員には、この保証に関する規定に何らかの変更、拡張、または追加を加える権限は与えられていません。

一部の国や地域では、黙示あるいは偶発的または結果的損害に対する賠償の免責または制限が認められていないため、上記の制限や免責がお客様に適用されない場合があります。この保証はお客様に特定の法的権利を与え、地域によってはその他の権利がお客様に与えられる場合もあります。