iPhone OS Core Dataチュートリアル

iPhone

2009-09-09

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/

目次

図

はじめに

Core Dataは、スキーマ駆動のオブジェクトグラフの管理および永続化のためのフレームワークです。基本的に、Core Dataは、（Model-View-Controllerデザインパターンにおける）*Model*オブジェクトをファイルに保存したり、ファイルから復元したりするために役立ちます。これはアーカイブ（『*Archives and Serializations Programming Guide for Cocoa*』を参照）に似ていますが、Core Dataはアーカイブよりもはるかに多くの機能を提供します。その中には、次のような機能が含まれています。

- Modelオブジェクトに加わったすべての変更を管理するためのインフラストラクチャを提供します。これによって、アンドゥとリドゥ、オブジェクト間の相互関係の管理が自動的にサポートされます。
- いつの時点でもModelオブジェクトのサブセットだけをメモリ内に保持することができます。この機能は、メモリの節約が不可欠なiPhoneでは特に重要です。
- Modelオブジェクトを記述するためにスキーマを使用します。GUIベースのエディタで、Modelクラスの基本機能（Modelクラス間の関係を含む）を定義します。これによって、デフォルト値の設定や属性値の検証などの基本機能の恩恵を“何もしなくても”受けることができます。
- オブジェクトの編集セットを互いに素な状態に維持できます。この機能は、たとえば、一方のビューで破棄される可能性のある編集を行いながら、もう一方のビューに表示されるデータには影響を与えないようにしたい場合に役立ちます。
- データストアのバージョン管理および移行のためのインフラストラクチャを備えています。これを利用すると、ユーザの古いバージョンのファイルを簡単に最新バージョンにアップグレードできます。

Core Dataは、iPhone OS v3.0以降で利用できます。この文書では、iPhone OS v3.0用のツールおよび手法について説明します。

iPhoneでのCore Dataの使いかたを学ぶには、この文書を読む必要があります。この文書には、以下の内容が含まれています。

- Core Dataの基礎となる基本的なデザインパターンおよび手法
- Xcodeのデータモデリングツールの使用についての基礎
- Core Dataによって管理されるオブジェクトの作成、更新、および削除の方法、変更をデータストアにコミットする方法

重要： Core Dataは、初心者向けのテクノロジーではありません。Code Dataを使い始める前に、以下を含むiPhoneアプリケーション開発の基礎を理解する必要があります。

- XcodeおよびInterface Builderの使いかた
- Model-View-Controller、デリゲーションなどの基本的なデザインパターン
- View Controller、Navigation Controller、およびTable Viewの使いかた

これらのツールおよび手法については、このチュートリアルでは説明していません。この文書の内容は、Core Dataそのものに焦点を当てています。

必要な経験を積むためには、次のような文書を読む必要があります。

- *Your First iPhone Application*
- *Xcode Workspace Guide*
- *Cocoa Fundamentals Guide*
- *View Controller Programming Guide for iPhone OS*
- *Table View Programming Guide for iPhone OS*

この書類の構成

このチュートリアルは、次の章で構成されています。

このチュートリアルのソースコードは、*Locations*サンプルコードに含まれています。

さあ、始めましょう

この章の目標は、これから作成するアプリケーションについて説明した後に、Xcodeプロジェクトを作成し、Xcodeのプロジェクトテンプレートで何が提供されるのか、その基本を理解することです。

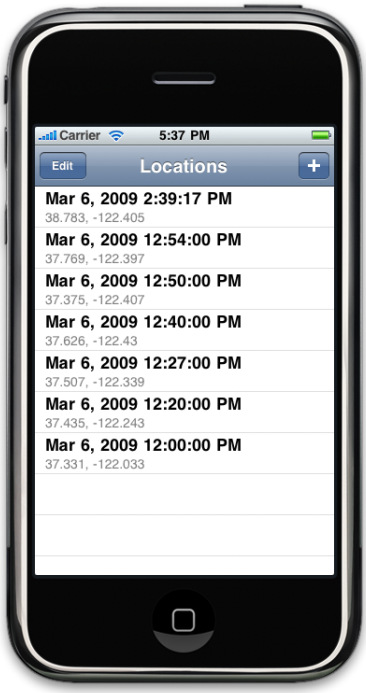
このチュートリアルの目的は、Core Dataフレームワークの実践的な入門書を提供し、その使いかたを説明することです。

この文書のねらいは、洗練されたアプリケーションを作成することではなく、Core Dataを基盤としたあらゆるプログラムで使用する基本的なクラス、ツール、および手法を示すことです。この文書では、このフレームワークのすべての機能の詳しい説明はしませんが、さらに理解を深めるために読むべきほかの文書を参考文献として挙げます。

少しだけ面白くするために、このチュートリアルではCore Locationフレームワークも利用します。Core Locationマネージャは非常に単純なオブジェクトであり、このプロジェクトのためにこのオブジェクトの詳細を理解する必要はありません。

これから作成するアプリケーションは概念的には単純です。このアプリケーションを利用すると、いつでも自分の位置を“イベント”として記録でき、記録したすべてのイベントの時刻、緯度、および経度をTable Viewを使用して表示できます。このアプリケーションには、新規イベントを追加するための「Add」ボタンと、リストからイベントを削除するための「Edit」ボタンがあります。

このチュートリアルでは、Core Dataを使用して主として`Event`オブジェクトを表現し、アプリケーションの起動時にそれらを表示できるように外部ファイルに保存します。



注： 表記規則として、このチュートリアルの中で、読者が実行しなければならない手順を含む段落の先頭には>> が付いています（その後に、箇条書きリストが続く場合もあります）。
コードのリストでは、Xcodeのテンプレートファイルに含まれているコメントは示しません。

プロジェクトの作成

この章で実行するただ1つの手順は、プロジェクトそのものを作成して、Core Locationフレームワークをリンクすることです。

>> Xcodeで、「iPhone OS」セクションの「Window-based Application」テンプレートを使用して新規プロジェクトを作成します。*「オプション(Options)」セクションで、ストレージにCore Dataを使用するスイッチ（「Use Core Data for storage」）を選択します。*このプロジェクトを「Locations」という名前にします。

後でこのチュートリアルで必要となるコードをコピーアンドペーストできるように、プロジェクトを「Locations」という名前にすることは重要です。

>> このプロジェクトをCore Locationフレームワークにリンクします（アプリケーションのターゲットの「情報(Info)」ウインドウの「一般(General)」ペインを使用します）。

Core Dataを基盤とするプロジェクトについて

このテンプレートには、ほかのさまざまなサポートファイルと一緒に、以下のものが含まれています。

- アプリケーションデリゲートクラス
- MainWindowのインターフェイス(`.xib`)ファイル
- Core Dataのモデル(`.xcdatamodel`)ファイル（通常、**管理オブジェクトモデル**と呼ばれる）

このアプリケーションは、Core Dataフレームワークにもリンクしています。

これらのリソースのうち、最初の2つについてはよく知っているはずです。ただし、デリゲートクラスの詳細については知らないかもしれません。モデルファイルについては、後で「管理オブジェクトとモデル」（21 ページ）で説明します。ここでは、アプリケーションデリゲートクラスのヘッダファイルを詳しく見てみましょう。このファイルには、標準のウインドウおよびView Controllerのほかに、4つのプロパティと1つの新規メソッドが含まれています。

```
- (IBAction)saveAction:sender;

@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSManagedObjectContext
*managedObjectContext;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

@property (nonatomic, readonly) NSString *applicationDocumentsDirectory;
```

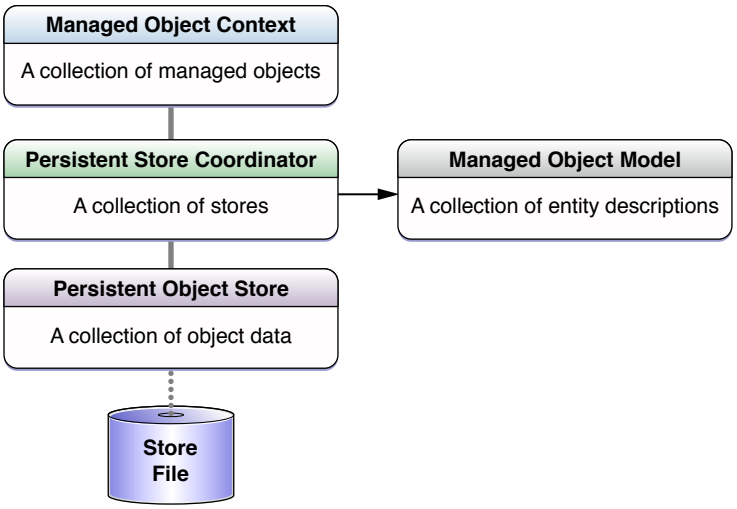
applicationDocumentsDirectoryプロパティは、その名のとおり、アプリケーションのドキュメントディレクトリのパスを返すだけです。このディレクトリには、アプリケーションのデータを含むファイルが置かれます。同様に、saveActionメソッドはアプリケーションのデータをディスクに保存します。保存については、この文書全体を通してさらに詳しく説明します。残りのプロパティは、Core Dataスタックと呼ばれるものへのアクセスを提供します。

Core Dataスタック

スタックは、モデル化されたオブジェクトを**永続ストア**（データの保存先となるファイル）から取得したりそこに保存したりするために、連携して動作するCore Dataフレームワークオブジェクトのコレクションを表すために使われる用語です。概念的には、永続ストアはテーブルとレコードを持つデータベースに似ています（Core Dataと一緒に使用できるストアタイプの1つにSQLiteがあります。ただし、ストアが実際にデータベースである必要はありません）。

図 1-1 （11 ページ）に、このスタックの最も単純な（最も一般的な）構成を示します。

図 1-1 単純なCore Dataスタック



通常、デベロッパが直接取り扱うオブジェクトは、このスタックの一番上にある管理オブジェクトコンテキストとそれに含まれる管理オブジェクトです。

管理オブジェクトと管理オブジェクトコンテキスト

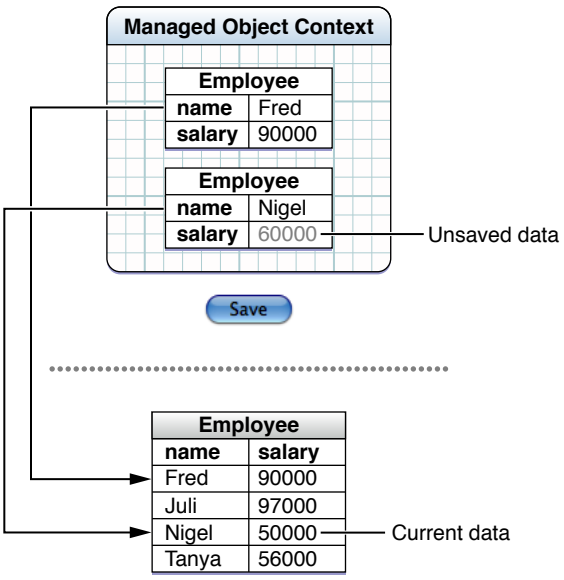
管理オブジェクトは、NSManagedObjectまたはNSManagedObjectのサブクラスのインスタンスです。概念的には、データベースのテーブル内のレコードのオブジェクト表現です。つまり、Core Dataによって管理される（Mode-View-Controllerデザインパターンにおける）Modelオブジェクトです。管理オブジェクトはアプリケーション内で操作されるデータを表します。たとえば、人材アプリケーションにおける部署と社員、描画アプリケーションにおける図形、テキスト領域、およびグループ、楽曲管理アプリケーションにおけるアルバム、アーティスト、およびトラックなどです。管理オブジェクトは、必ず管理オブジェクトコンテキストに関連付けられています。

管理オブジェクトコンテキストはNSManagedObjectContextのインスタンスです。コンテキストは、アプリケーション内の1つのオブジェクト空間（スクラッチパッド）を表します。その主な仕事は、管理オブジェクトのコレクションを管理することです。これらのオブジェクトは、関連するModelオブジェクトのグループを形成し、1つ以上の永続ストアに関して整合性のある内部ビューを表します。コンテキストは、アプリケーションで中心的な役割を果たす強力なオブジェクトで、ライフサイクル管理から、妥当性検証、関係の管理、アンドゥ／リドゥまでを担当します。

新規の管理オブジェクトを作成したときは、それをコンテキストに挿入します。データベース内の既存のレコードをフェッチしたら、それを管理オブジェクトとしてコンテキストに挿入します（フェッチについては、「イベントのフェッチ」（33 ページ）で詳しく説明します）。すべての変更（オブジェクト全体の挿入や削除、プロパティ値の操作など）は、コンテキストを保存することによって実際にストアにコミットするまでは、メモリ内に保持されます。

図 1-2（12 ページ）に、2つの管理オブジェクトを含む管理オブジェクトコンテキストと、それに対応する、外部データベース内の2つのレコードを示します。これらのオブジェクトの1つは、プロパティ値がメモリ内では変更されていますが、その変更はデータベースにコミットされていません。

図 1-2　コンテキスト内の管理オブジェクトと永続ストア内のテーブル

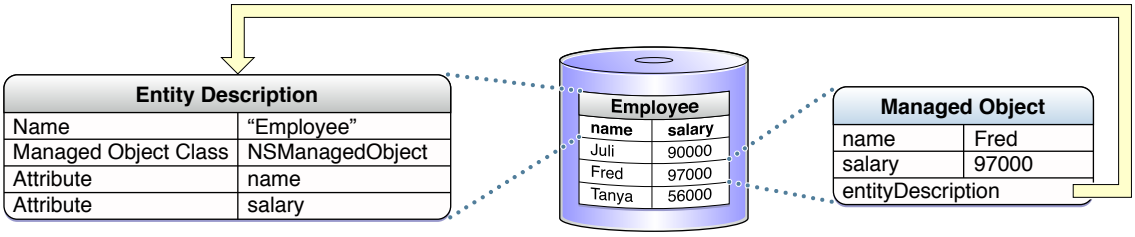


管理オブジェクトモデル

管理オブジェクトモデルはNSManagedObjectModelのインスタンスです。これは、データベースを定義するスキーマのオブジェクト表現です。つまり、アプリケーションで使用する管理オブジェクトです。モデルはエンティティ記述オブジェクト（NSEntityDescriptionのインスタンス）のコレクションです。エンティティ記述では、エンティティの名前、アプリケーション内でそのエンティティを表すために使用するクラスの名前、およびそのエンティティが持つプロパティ（属性と関係）によってエンティティ（データベース内のテーブル）を定義します。

図 1-3（13 ページ）に、1つのモデル内のエンティティ記述、データベース内のテーブル、およびそのテーブル内の1つのレコードに対応する管理オブジェクトの関係を示します。

図 1-3 エンティティ記述、データベース内のテーブル、および管理オブジェクト



どの管理オブジェクトも、インスタンス元になったエンティティへの参照を持っています。

Core Dataは、このモデルを使用して、アプリケーション内の管理オブジェクトとデータベース内のレコードのマッピングを行います。アプリケーションのスキーマを変更すると、Core Dataはそれまでのモデルを使用して作成されたストアを読めなくなることに注意してください（これは、多くの永続化メカニズムに共通です。ただし、Core Dataでは、このような変更を管理するためのインフラストラクチャを提供しています。詳細については『*Core Data Model Versioning and Data Migration Programming Guide*』を参照してください）。

永続ストアコーディネータ

永続ストアコーディネータは、Core Dataがデータを管理する際に中心的な役割を果たします。ただし、フレームワークを使用するときに、コーディネータと直接やり取りをすることはほとんどありません。このセクションでは、永続ストアコーディネータの詳細について説明しますが、このセクションをスキップして、必要に応じて後から参照してもかまいません（永続ストアコーディネータについては、『*Core Data Programming Guide*』の「Core Data Basics」でも説明しています）。

永続ストアコーディネータは`NSPersistentStoreCoordinator`のインスタンスです。これは、**永続オブジェクトストア**のコレクションを管理します。永続オブジェクトストアは、永続化されたデータの外部ストア（ファイル）を表します。これは、アプリケーション内のオブジェクトとデータベース内のレコードを実際にマッピングするオブジェクトです。Core Dataがサポートするさまざまなファイルタイプに対応して、さまざまな永続オブジェクトストアクラスがあります。独自のファイルタイプをサポートしたい場合は、独自のクラスを実装することもできます（『*Atomic Store Programming Topics*』を参照）。永続ストアとそのさまざまなタイプの詳細については、『*Core Data Programming Guide*』の「Persistent Store Features」を参照してください。

iPhoneアプリケーションでは、通常は1つのストアしか持ちませんが、複雑なデスクトップアプリケーションでは、さまざまなエンティティを含むストアが複数存在する場合もあります。永続ストアコーディネータの役割は、これらのストアを管理して、管理オブジェクトコンテキストに、1つの統一されたストアのファサードを提供することです。レコードをフェッチすると、（関心のあるストアを指定しない限り）Core Dataはすべてのストアから結果を取得します。

どのアプリケーションでも、複数の管理オブジェクトコンテキストを持つことができます。管理オブジェクトの集合とそれらのオブジェクトに対する編集を個別に管理することもできます。あるいは、あるコンテキストを使用してバックグラウンドの操作を実行しながら、別のコンテキストではユーザにオブジェクトとのやり取りを許可することができます。これらのコンテキストはいずれも、同じコーディネータに接続しています。

図 1-4 （14 ページ）に、コーディネータが果たす役割を示します。通常、スタックはこれほど複雑ではありません。

図 1-4 複雑なCore Dataスタック

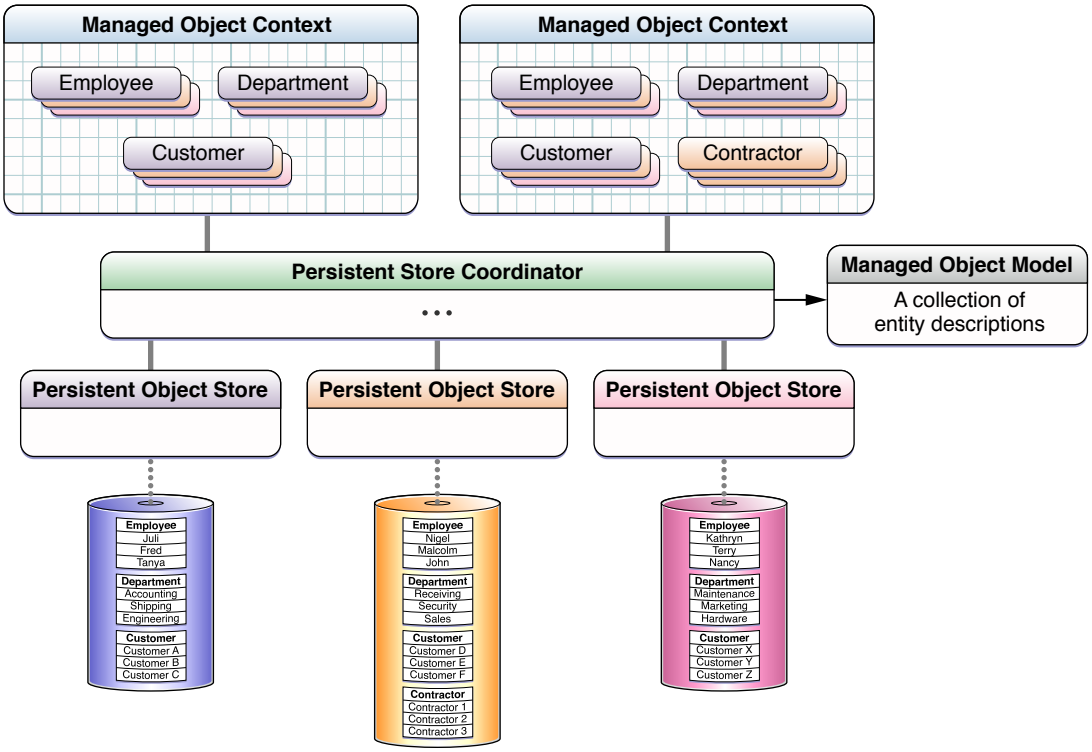


Table View Controller

この章の目標は、Table View Controllerの最初の実装を作成し、そのTable View Controllerのインスタンスを作成して設定するようにアプリケーションデリゲートを更新することです。

この章では、Table Viewをセットアップします。そして、Navigation ControllerとTable View Controllerのインスタンスを作成してCore Locationマネージャを設定します。これが、このアプリケーションのアーキテクチャを提供します。次の章では、Core Dataを使用して実際のデータを管理します。

ここでは、読者がすでにView ControllerとTable Viewについてよく知っていることを前提としています。このため、この章では、このアプリケーションの各コンポーネントの役割を理解するために必要な情報以上の詳しい説明は行いません。ここでの内容が難しすぎる場合は、一旦立ち止まって、いくつかのアプリケーションを作成する練習をしてから先に進んでください。

アプリケーションデリゲートは、Navigation ControllerとTable View Controllerの作成、設定、および表示を担当します。

Table View Controllerはイベントオブジェクトの配列を表示します。これをサポートするために、このControllerでは、基本的なTable View Controllerに次の4つのプロパティを追加します。

- **可変配列。**Table View Controllerが表示するイベントオブジェクトのコレクションを格納します。この配列は、アプリケーションの起動時に永続ストアから読み込まれます。また、ユーザがイベントを追加したり削除したりすると更新されます。
- **管理オブジェクトコンテキスト。**Core Dataスタックへのゲートウェイとしての役割を果たします。
- **Core Locationマネージャ。**位置情報をアプリケーションに提供します。Core Locationマネージャが有効になっている場合にのみ、ユーザは新規イベントを追加できます（iPhone Simulatorは動きをシミュレートするので、テストのためにアプリケーションをデバイスにインストールする必要はありません）。
- **バーボタン項目。**ユーザがイベントを追加するために必要になります。Core Locationマネージャの状態の変化に対応して、ボタンを有効または無効にできるように、このボタンへの参照が必要になります。

RootViewControllerクラスの作成と定義

まず、新規クラスのファイルを作成します。

>> Xcodeで、新規の`UITableViewController`サブクラスを作成し、それを`RootViewController`という名前にします。

次に、イベント配列、管理オブジェクトコンテキスト、Core Locationマネージャ、および「Add」ボタンに対応する4つのプロパティを追加します。このルートView ControllerはCore Locationマネージャのデリゲートとしての役割を果たします。したがって、`CLLocationManagerDelegate`プロトコルを採用しなければなりません。

>> RootViewControllerのヘッダファイルの内容を次のコードに置き換えます。

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

@interface RootViewController :UITableViewController <CLLocationManagerDelegate>
 {

    NSMutableArray *eventsArray;
    NSManagedObjectContext *managedObjectContext;

    CLLocationManager *locationManager;
    UIBarButtonItem *addButton;
}

@property (nonatomic, retain) NSMutableArray *eventsArray;
@property (nonatomic, retain) NSManagedObjectContext *managedObjectContext;

@property (nonatomic, retain) CLLocationManager *locationManager;
@property (nonatomic, retain) UIBarButtonItem *addButton;

@end
```

RootViewControllerクラスの実装

最初の実装は、いくつかの部分に分かれています。以下の作業を行う必要があります。

- 宣言したプロパティを合成します。
- Core Locationマネージャと、「Add」ボタンおよび「Edit」ボタンをセットアップするように`viewDidLoad`を実装します。
- Core Locationマネージャのアクセサメソッドを記述して、そのデリゲートメソッドのうちの2つを実装します。
- メモリ管理のためのメソッドを実装します。

以降の各セクションで説明するすべてのコードを`RootViewController`クラスの`@implementation`ブロックに挿入し、テンプレートによって提供された実装を必要に応じて置き換えます。

プロパティの合成

>> 次のコードを追加します。

```
@synthesize eventsArray;
@synthesize managedObjectContext;
@synthesize addButton;
@synthesize locationManager;
```

Core Locationマネージャのアクセサメソッドの記述

>> 要求に応じてCore Locationマネージャを動的に作成するアクセサメソッドを作成します。

```
- (CLLocationManager *)locationManager {

    if (locationManager != nil) {
        return locationManager;
    }

    locationManager = [[CLLocationManager alloc] init];
    locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters;
    locationManager.delegate = self;

    return locationManager;
}
```

次に、状況に応じて「Add」ボタンを有効または無効にする2つのデリゲートメソッドを実装します。Core Locationマネージャが更新情報を生成している場合は、このボタンを有効にします。Core Locationマネージャが動作していない場合は、このボタンを無効にします。

>> Core Locationマネージャの次の2つのデリゲートメソッドを追加します。

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
           fromLocation:(CLLocation *)oldLocation {
    addButton.enabled = YES;
}

- (void)locationManager:(CLLocationManager *)manager
       didFailWithError:(NSError *)error {
    addButton.enabled = NO;
}
```

viewDidLoadの実装

viewDidLoadメソッドでは、Core Locationマネージャと、「Add」ボタンおよび「Edit」ボタンをセットアップする必要があります。

>> viewDidLoadの実装を次のコードで置き換えます。

```
- (void)viewDidLoad {

    [super viewDidLoad];

    // タイトルを設定する。
    self.title = @"Locations";

    // ボタンをセットアップする。
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
    target:self action:@selector(addEvent)];
    addButton.enabled = NO;
    self.navigationItem.rightBarButtonItem = addButton;

    // ロケーションマネージャを起動する。
    [[self locationManager] startUpdatingLocation];
}
```

メモリ管理用のメソッドの実装

>>viewDidUnloadとdeallocの既存の実装を置き換えます。viewDidUnloadの実装では、viewDidLoadで作成したオブジェクトのうち、再作成可能なものの所有権を放棄します。

```
- (void)viewDidUnload {
    self.eventsArray = nil;
    self.locationManager = nil;
    self.addButton = nil;
}

- (void)dealloc {
    [managedObjectContext release];
    [eventsArray release];
    [locationManager release];
    [addButton release];
    [super dealloc];
}
```

アプリケーションデリゲートの設定

アプリケーションデリゲートは、ルートView Controllerとそれを含むNavigation Controllerを作成して設定する仕事を担当します。

Navigation Controllerプロパティの追加

Navigation Controllerに対応するプロパティを追加する必要があります。

>> アプリケーションデリゲートのヘッダファイル(LocationsAppDelegate.h)に、次のインスタンス変数を追加します。

```
UINavigationController *navigationController;
```

>> 次のプロパティ宣言を追加します。

```
@property (nonatomic, retain) UINavigationController *navigationController;
```

アプリケーションデリゲートの実装

アプリケーションデリゲートの実装ファイル(LocationsAppDelegate.m)では、次の作業を行う必要があります。

- RootViewControllerのヘッダファイルをインポートします。
- navigationControllerプロパティを合成します。
- applicationDidFinishLaunching:メソッド内で、RootViewControllerとそれを含むNavigation Controllerのインスタンスを作成します。

また、アプリケーションの管理オブジェクトコンテキストをこのルートView Controllerに渡す必要があります。

>>アプリケーションデリゲートクラスの@implementationブロックの前で、RootViewControllerクラスのヘッダファイルをインポートします。

```
#import "RootViewController.h"
```

>>アプリケーションデリゲートクラスの@implementationブロック内で、Navigation Controllerプロパティを合成します。

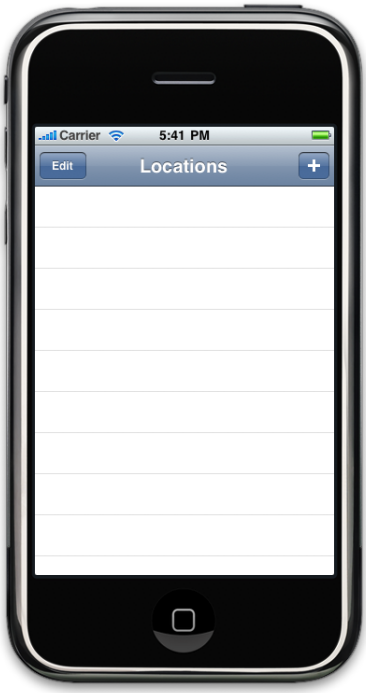
```
@synthesize navigationController;
```

>>アプリケーションデリゲートのapplicationDidFinishLaunching:メソッドを次の実装で置き換えます。

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // ウインドウを設定して表示する

    RootViewController *rootViewController = [[RootViewController alloc]
    initWithStyle:UITableViewStylePlain];

    NSManagedObjectContext *context = [self managedObjectContext];
    if (!context) {
        // エラーを処理する。
    }
    // 管理オブジェクトコンテキストをView Controllerに渡す。
    rootViewController.managedObjectContext = context;

    UINavigationController *aNavigationController = [[UINavigationController
alloc]
    initWithRootViewController:rootViewController];
    self.navigationController = aNavigationController;

    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];

    [rootViewController release];
    [aNavigationController release];
}
```

ビルドとテスト

この段階で、プロジェクトをビルドしてテストし、すべてが動作することを確認します。Navigation Barを持つ空のテーブルビューが表示されます。このNavigation Barには、「Edit」ボタンと「Add」ボタンが含まれています。



「Add」ボタンは最初は無効になっていますが、数秒後には（ロケーションマネージャがイベントを送信し始めると）有効になります。`addEvent`イベントをまだ実装していないため、このボタンをタップすると、もちろんアプリケーションはクラッシュします。ただ、イベントを追加する前に、Eventエンティティを定義する必要があります。次の章ではこの作業を行います。

管理オブジェクトとモデル

この章の目標は、ユーザが「Add」ボタンをタップしたときに、新規イベントが作成されるようにすることです。それには、管理オブジェクトモデルにEventエンティティを定義して、それに対応するクラスを実装し、Addメソッド内でこのクラスのインスタンスを作成する必要があります。

データのモデリング

「管理オブジェクトモデル」（12 ページ）で説明したように、モデルとは、エンティティとプロパティで記述されたオブジェクトのコレクションです。これらのオブジェクトが、アプリケーション内の管理オブジェクトに関する情報をCore Dataに伝えます。モデルは、プログラムで作成することもできますし、Interface Builderを使用してユーザインターフェイスを作成するのと同様に、Xcodeのモデリングツールを使用してグラフィカルに作成するこもできます。

実際には、モデルの構成要素を編集する方法は複数ありますが、ここでは、そのうちの手順を1つだけ説明します。モデリングツールおよびその他のモデル編集方法の詳細については、『*Xcode Tools for Core Data*』を参照してください。

このアプリケーションはエンティティ(Event)を1つだけ持ちます。このエンティティには3つの属性（作成日、緯度、経度）があります。

エンティティの追加

まず、Eventエンティティを追加します。

>> Xcodeで、「Resources」グループからモデルファイル(`Locations.xcdatamodel`)を選択して、モデルエディタを表示します。

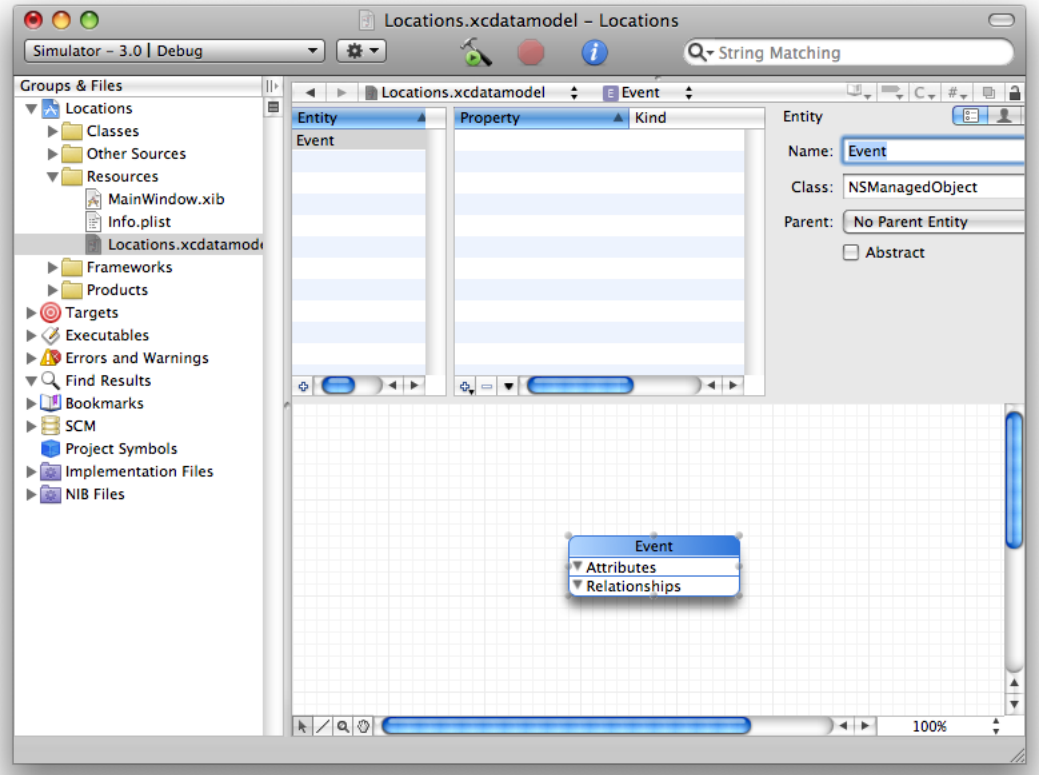
>> 「設計(Design)」＞「データモデル(Data Model)」＞「エンティティを追加(Add Entity)」を選び、新規エンティティをモデルに追加します。

エンティティペインの左下にある追加ボタン(+)を使用したり、モデルエディタのダイアグラムビュー内でショートカットメニューを使用することもできます。

ドキュメントエディタの左上のエンティティペインに、このエンティティに対応する新規エンティティ（「Entity」）が表示されます。また、ダイアグラムビューには、このエンティティのグラフック表現（角が丸い矩形）が表示されます。次に、この新規エンティティに名前を設定します。

>>エンティティペインで、この新規エンティティが選択されていることを確認します。右側の詳細ペインにはこのエンティティに関する情報が表示されます。エンティティの名前を`Event`に変更します（クラス名を変更してはいけません）。

モデルは次のように表示されます。



エンティティの名前と、このエンティティのインスタンスを表すために使用するObjective-Cクラスの名前には重要な違いがあります。Core Dataはエンティティ記述を使用して管理対象のデータオブジェクトを見つけます。したがって、クラス名はエンティティ名と同じである必要はありません。それどころか、複数のエンティティを同じクラス(`NSManagedObject`)で表す場合もあります。Core Dataは、対応するエンティティ記述に基づいてインスタンスを識別できます。

属性の追加

まず、作成日に対応する属性を追加します。

>>エンティティペインでEventが選択されていることを確認して、「設計(Design)」>「データモデル(Data Model)」>「属性を追加(Add Attribute)」を選びます。

プロパティペインに新規の属性(`newAttribute`)が表示されます。その名前とデータ型を設定する必要があります。

>>プロパティペインでこの新規属性が選択されていることを確認します。次に、詳細ペインで属性の名前を`creationDate`に変更し、「データ型(Type)」ポップアップメニューから「日付(Date)」を選択します。

その他の値を設定する必要はありません。

次に、緯度と経度に対応する属性を追加します。

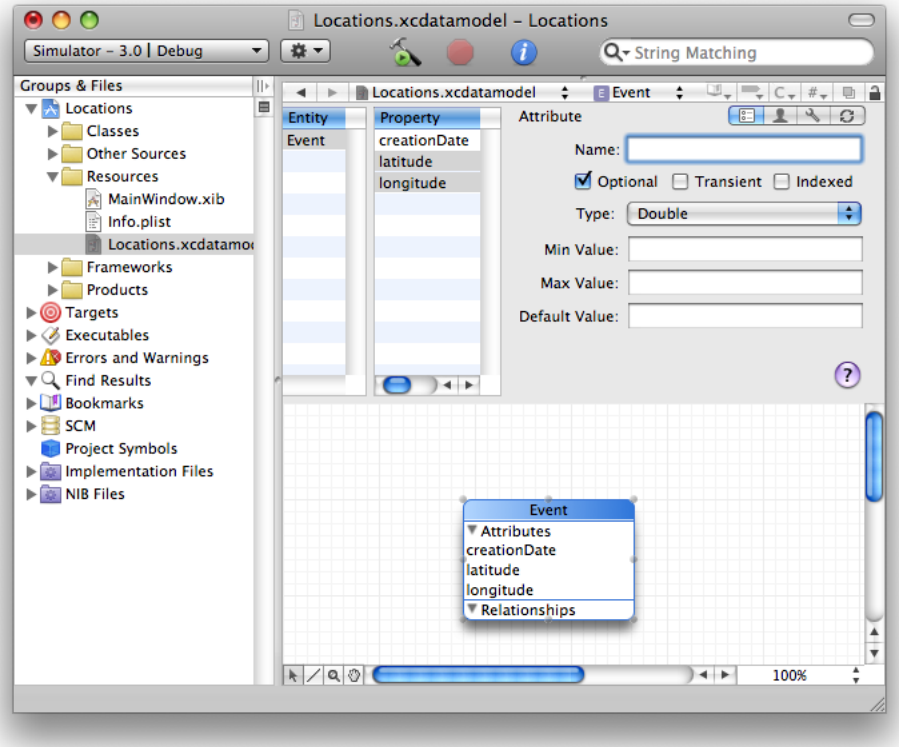
>>エンティティペインでEventが選択されていることを確認して、「設計(Design)」＞「データモデル(Data Model)」＞「属性を追加(Add Attribute)」を選びます（2つの属性を追加するために、これを2回実行します）。

>> プロパティペインで*両方の*新規属性を選択します。次に、詳細ペインで「データ型(Type)」ポップアップメニューから「倍精度(Double)」を選択します。

>> プロパティペインで最初の新規属性だけを選択して、詳細ペインで属性の名前を`latitude`に変更します。

>> プロパティペインで2番目の新規属性だけを選択して、詳細ペインで属性の名前を`longitude`に変更します。

モデルは次のように表示されます。



カスタム管理オブジェクトクラス

次に、Xcodeを使用して、Eventエンティティを表すカスタムクラスのファイルを生成します。

>>Xcodeで、モデルから*Eventエンティティを選択します*（この選択は、これから作成するサブクラスに対応する要素を示すために使われます）。

>>「ファイル(File)」＞「新規ファイル(New File)」を選びます。「新規ファイル(New File)」ダイアログで、「管理オブジェクトクラス(Managed Object Class)」を選択します。

使用中のXcodeのバージョンによっては、「管理オブジェクトクラス(Managed Object Class)」が「Cocoa Touch Classes」の下の「iPhone OS」セクションにある場合もあります。あるいは、「Cocoa」の下の「Mac OS X」セクションからテンプレートを選ぶ必要がある場合もあります。どちらも正しく動作します。

>>「次へ(Next)」をクリックします。適切な場所とターゲットが選択されているはずです。「次へ(Next)」をクリックして、それを受け入れます。

Entity選択ペインが表示され、Eventエンティティが選択されているはずです。「アクセサを生成(Generate accessors)」オプションと「Obj-C 2.0プロパティを生成(Generate Objective-C 2.0 properties)」オプションも選択されています。

>>「完了(Finish)」をクリックしてファイルを生成します。

Eventクラスのインターフェイスファイルと実装ファイルが作成されてプロジェクトに追加されます。次の点に注目してください。

- インターフェイスファイル(`Event.h`)では、すべての属性がオブジェクト値で表現されています。

  緯度と経度の属性タイプを倍精度(Double)に指定しましたが、実行時のプロパティ値は`NSNumber`のインスタンスです。Core Dataでは、オブジェクトを使用して値を表します。
- 実装ファイル(`Event.m`)では、プロパティは`dynamic`として実装されています。

  通常は、`synthesized`が含まれていますが、Core Dataは実行時にアクセサメソッドを生成します。
- 実装ファイル(`Event.m`)には、`dealloc`メソッドはありません。

  通常は、インスタンス変数を解放するための`dealloc`メソッドが含まれていますが、管理オブジェクトのすべての*モデル化された*プロパティのライフサイクルはCore Dataが管理します（管理オブジェクト内に対応するプロパティが存在しない独自のインスタンス変数を追加した場合は、通常どおり自分でこれらを管理する必要があります）。
- モデルも更新されています（Eventエンティティは`Event`クラスで表現されています）。

  モデルが変更されたので、それを保存する必要があります。

>> モデルファイルを保存します。

最後に、Table View Controllerはこの新規クラスを利用するので、Table View Controllerの実装ファイルでこのクラスのヘッダファイルをインポートします。

>>Table View Controllerの実装ファイル(`RootViewController.m`)で、最初のimportステートメントの後に、次のコードを追加します。

```
#import "Event.h"
```

Core Dataのまとめ

Xcodeのデータモデリングツールを使用して、新規エンティティを作成しました。次に、そのエンティティを表すカスタムクラスを作成しました。次の章では、このエンティティのインスタンスを作成します。

モデリングツールの詳細については、『*Xcode Tools for Core Data*』を参照してください。

イベントの追加

この章の目標は、ユーザが新規のイベントオブジェクトを作成してユーザインターフェイスに表示できるように、アプリケーションのロジックを作成することです。

addEventメソッドの実装

addEventメソッド内で新規のEventオブジェクトを作成します。このメソッドは、ユーザが「Add」ボタンをタップしたときに呼び出されることを思い出してください（「viewDidLoadの実装」（17 ページ）を参照）。このメソッドは、いくつかの部分に分かれます。次の処理を実行しなければなりません。

- 現在位置を取得します。
- Eventオブジェクトを作成し、現在位置の情報を使用してそれを設定します。
- Eventオブジェクトを保存します。
- イベント配列とユーザインターフェイスを更新します。

ただし、先にaddEventメソッドを宣言します。

>> addEventメソッドの宣言をRootViewControllerヘッダファイルに追加します。

```
- (void)addEvent;
```

現在位置の取得

新規のEventオブジェクトを作成したら、その位置を設定する必要があります。位置は、ロケーションマネージャから取得します。位置を取得できない場合は、処理を継続しません。

>> RootViewController実装ファイルに次のコードを追加します。

```
- (void)addEvent {

    CLLocation *location = [locationManager location];
    if (!location) {
        return;
    }
}
```

Eventオブジェクトの作成と設定

通常は、NSEntityDescriptionの簡易メソッド(insertNewObjectForEntityForName:inManagedObjectContext:)を使用して管理オブジェクトを作成します。このメソッドは、指定したエンティティに対応するクラスを適切に初期化したインスタンスを返し、それを管理オブジェクトコンテキストに挿入します（この初期化処理の詳細については、『*Core Data Programming Guide*』の「Managed Objects」を参照してください）。オブジェクトを作成したら、ほかのオブジェクトと同様にアクセサメソッドを使用してプロパティ値を設定できます。

位置情報から緯度と経度をスカラー値として取得します。したがって、これらの値をEventオブジェクト用のNSNumberオブジェクトに変換する必要があります。位置情報からタイムスタンプを取得することもできます。ただし、iPhone Simulatorではこの値は固定の値です。その代わりに、ここではNSDateのdateメソッドを使用して、現在の日時を表す日付オブジェクトを取得できます。

>> addEventの現在の実装の最後に次のコードを追加します。

```
// Eventエンティティの新規インスタンスを作成して設定する
Event *event = (Event *)[NSEntityDescription
insertNewObjectForEntityForName:@"Event"
inManagedObjectContext:managedObjectContext];

CLLocationCoordinate2D coordinate = [location coordinate];
[event setLatitude:[NSNumber numberWithDouble:coordinate.latitude]];
[event setLongitude:[NSNumber numberWithDouble:coordinate.longitude]];
[event setCreationDate:[NSDate date]];
```

新規イベントの保存

管理オブジェクトコンテキストはスクラッチパッドのような役割を果たすことを思い出してください（「管理オブジェクトと管理オブジェクトコンテキスト」 （11 ページ）を参照）。プロパティ値の編集であれ、オブジェクト全体の追加や削除であれ、どんな変更もコンテキストを保存するまでは実際には永続ストア（ファイル）にコミットされません。通常、iPhoneアプリケーションでは、ユーザが変更を行うとすぐにその変更を保存します。

>> addEventの現在の実装の最後に次のコードを追加します。

```
NSError *error;
if (![managedObjectContext save:&error]) {
    // エラーを処理する。
}
```

いくつかのCore Dataメソッドと同様に、NSManagedObjectContext save:メソッドはエラーパラメータを受け取って、成功か失敗かを表すブール値を返します。この状況は、実際のところほかのアプリケーションとまったく同じです。save:メソッドの戻り値とエラーパラメータは、問題が発生している可能性があれば、それをより強調するというだけのことです。

エラー処理

Core Dataエラーの処理の仕方はデベロッパに任されています。

「新規イベントの保存」（28 ページ）で説明したものと同じシナリオ（想定される変更が、1つのオブジェクトの追加だけの場合）では、データが保存できない場合は、復旧が困難または不可能な致命的なエラーの可能性があります。このような場合は、ユーザにアプリケーションの再起動を指示する警告シートを表示するだけという対応が考えられます。

さらに複雑なシナリオの場合は、ユーザがプロパティ値を変更したり、管理オブジェクトを追加したり削除したりした結果、特定のオブジェクトの整合がとれていない状態（妥当性検証エラー）になるか、オブジェクトグラフ全体の整合がとれていない状態になる可能性もあります。複数の管理オブジェクトコンテキストを持つ場合は、別のコンテキストでの変更がコミットされて永続ストアが更新されているために、現在のコンテキスト内のオブジェクトが、ストア内の対応するレコードと整合がとれなくなる可能性もあります。

一般に、どこが悪いのかをエラーオブジェクトに問い合わせることができます。

エラーが発生しているときのユーザ体験をどのようにするべきかについても注意深く検討する必要があります。どのような情報をユーザに表示するべきか。問題から復旧するために、どのような選択肢を提示できるか。これらは、Core Dataが答える質問ではありません。

イベント配列とTable Viewの更新

最後に、新規の`Event`オブジェクトをイベント配列に追加して、Table Viewを更新する必要があります。最新のイベントがリストの一番上になるようにイベントを表示するため、この新規オブジェクトをイベント配列の先頭に追加します。また、それに対応する行をTable Viewの一番上に追加して、新規の行が表示されるようにTable Viewをスクロールします。

>> `addEvent`の現在の実装の最後に次のコードを追加します。

```
[eventsArray insertObject:event atIndex:0];
NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];
[self.tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
    withRowAnimation:UITableViewRowAnimationFade];
[self.tableView scrollToRowAtIndexPath:[NSIndexPath indexPathForRow:0 inSection:0]
 atScrollPosition:UITableViewScrollPositionTop animated:YES];
```

次の作業は、イベントを表示するために、Table Viewのデータソース用メソッドの実装を完成させることです。

Table Viewでのイベントの表示

イベントを表示するには、Table Viewの2つのデータソース用メソッドを更新する必要があります。

まず、表示するイベント数をTable Viewに伝えます。

>> イベント配列内のオブジェクト数を返すように`tableView:numberOfRowsInSection:`の実装を更新します（1つのセクションしか存在しないため、セクション番号をテストする必要はありません）。

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [eventsArray count];
}
```

次に、各イベントに関する情報を表示するために、Table View Cellを設定する必要があります。かなりの量のコードがありますが、そのほとんどは、データ管理ではなくユーザインターフェイスと表示に関連するものです。

>>tableView:(UITableView *)tableView cellForRowAtIndexPath:の実装を次のコードで置き換えます。

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

    // タイムスタンプ用の日付フォーマッタ
    static NSDateFormatter *dateFormatter = nil;
    if (dateFormatter == nil) {
        dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setTimeStyle:NSDateFormatterMediumStyle];
        [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
    }

    // 緯度と経度用の数値フォーマッタ
    static NSNumberFormatter *numberFormatter = nil;
    if (numberFormatter == nil) {
        numberFormatter = [[NSNumberFormatter alloc] init];
        [numberFormatter setNumberStyle:NSNumberFormatterDecimalStyle];
        [numberFormatter setMaximumFractionDigits:3];
    }

    static NSString *CellIdentifier = @"Cell";

    // 新規セルをデキューまたは作成する
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:CellIdentifier]
autorelease];
    }

    Event *event = (Event *)[eventsArray objectAtIndex:indexPath.row];

    cell.textLabel.text = [dateFormatter stringFromDate:[event creationDate]];

    NSString *string = [NSString stringWithFormat:@"%@, %@",
                        [numberFormatter stringFromNumber:[event latitude]],
                       [numberFormatter stringFromNumber:[event longitude]]];
    cell.detailTextLabel.text = string;

    return cell;
}
```

ビルドとテスト

プロジェクトをビルドすると、エラーなくコンパイルされるはずです。アプリケーションも、「Add」ボタンをタップしない限りは適切に起動されて動作します。「Add」ボタンをタップした時点でクラッシュします。これは、イベント配列がまだ作成されていないからです。

>> *テストのためだけに*、`viewDidLoad`の`RootViewController`オブジェクトの実装の最後に次の行を追加します。

```
eventsArray = [[NSMutableArray alloc] init];
```

ここでビルドして実行すると、「Add」ボタンをタップしたときに新規イベントがテーブルビューに表示されます。ただし、アプリケーションを終了して再起動すると、起動時にはイベントのリストは表示されません。これを改良するには、起動時にイベント配列に既存の`Event`オブジェクトを読み込む必要があります。この作業は次の章で行います。その前に、プロジェクトをテスト前の状態に復元します。

>> テスト用に追加した行を削除します。

Core Dataのまとめ

この章ではたくさんのコードが出てきましたが、Core Dataに直接関連するコードはあまりありませんでした。重要な点は次のとおりです。

- 新規の管理オブジェクトは、通常、`NSEntityDescription`の簡易メソッド(`insertNewObjectForEntityForName:inManagedObjectContext:`)を使用して作成します。

  このメソッドを利用すると、指定したエンティティを表すクラスのインスタンスを適切に初期化したものを確実に取得できます。
- 変更を永続ストアにコミットするには、管理オブジェクトコンテキストを保存する必要があります。

  コンテキストはスクラッチパッドのような役割を果たします。オブジェクトを追加したり変更したりすると、その変更は`save:`が呼び出されるまではメモリ内に保持されます。保存操作中に発生するエラーの処理の仕方はデベロッパに任されています。
- ほかのオブジェクトと同様に、アクセサメソッドを使用して管理オブジェクトのプロパティ値を取得したり設定したりします。

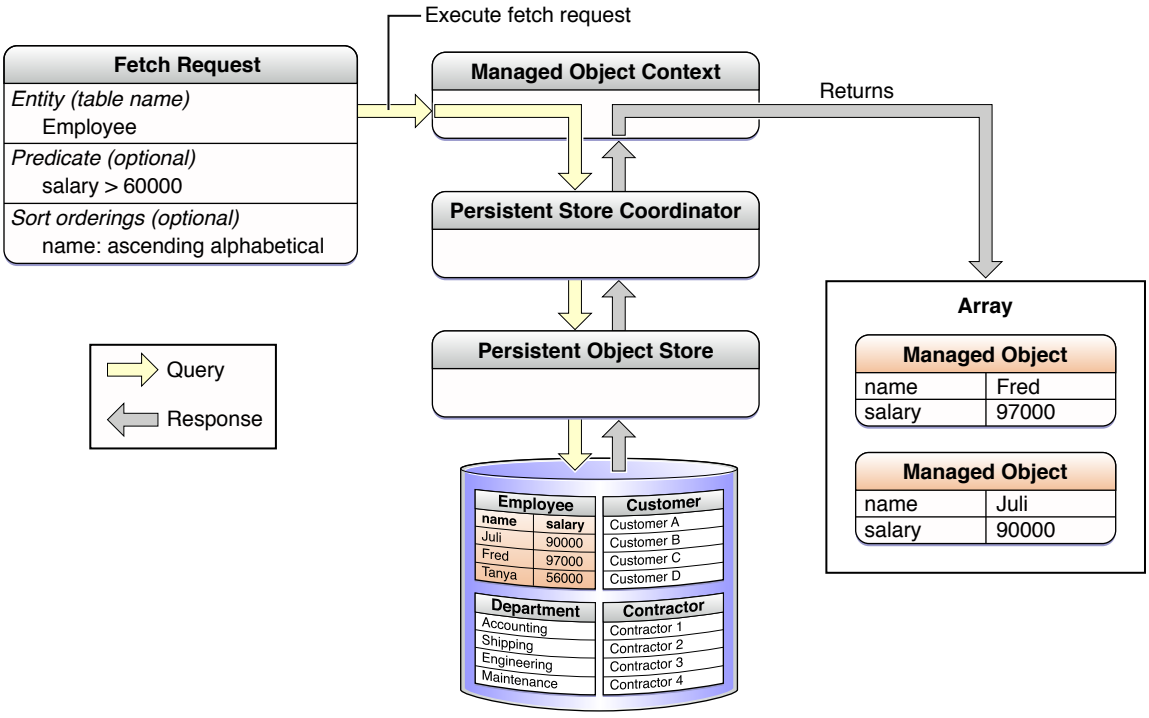
  また、ほかのオブジェクトと同様に、キー値コーディングを使用することもできます。ただし、アクセサメソッドを使用する方がはるかに効率的です（『*Core Data Programming Guide*』の「Using Managed Objects」を参照）。

イベントのフェッチ

この章の目標は、アプリケーションの起動時に既存のEventオブジェクトをフェッチすることです。

管理オブジェクトのフェッチ

永続ストアからオブジェクトをフェッチするには、管理オブジェクトコンテキストと**フェッチ要求**が必要になります。フェッチ要求はNSFetchRequestのインスタンスです。フェッチ要求では、最低限、関心のあるエンティティを指定します。オブジェクトが持つ値に関する制限を指定したり、オブジェクトを返す順番を指定したりすることもできます。たとえば、社内情報アプリケーションで、一定額を超える給与をもらっているEmployeeオブジェクトを名前順に取得するためのフェッチ要求を作成することができます。制限は述語（NSPredicateのインスタンス）で表します（述語の詳細については、『*Predicate Programming Guide*』を参照してください）。ソートの順番はNSSortOrderingオブジェクトの配列で表します。



特定のエンティティのすべてのオブジェクトが本当に必要な場合以外は、述語を使用して、返されるオブジェクトの数を本当に関心があるオブジェクトに限定するべきです（Table Viewにオブジェクトを表示する場合は、フェッチ結果コントローラ(NSFetchedResultsController)を使用して、結果セットを管理することもできます。このコントローラは、メモリ内に保持するデータができる限り少なくなるように動作します）。

オブジェクトを取得するために、必ずしもフェッチを実行する必要はありません。Core Dataは、必要であれば、自動的に関係先のオブジェクトを取得します。たとえば、Employeeオブジェクトを取得するためにフェッチを実行した後に、それに関連するDepartmentを要求すると、それがまだフェッチされていない場合は、Core Dataが自動的にDepartmentをフェッチします。

フェッチ要求の作成と実行

Table View Controllerは、ビューをロードするときに、Eventオブジェクトをフェッチして、後で表示できるようにそれらをイベント配列に保持する必要があります。ユーザはイベントを追加したり削除したりできるため、このイベント配列は可変でなければなりません。

フェッチ要求の作成

フェッチ要求を作成してエンティティを設定します。

>> viewDidLoadの現在の実装の最後に次のコードを追加します。

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
inManagedObjectContext:managedObjectContext];
[request setEntity:entity];
```

ここで注目すべきメソッドは、NSEntityDescriptionのentityForName:inManagedObjectContext:です。要求するエンティティの名前と現在扱っている管理オブジェクトコンテキストを渡すと、このメソッドは（管理オブジェクト）コンテキストの（永続ストア）コーディネータの（管理オブジェクト）モデルに問い合わせて、指定された名前のエンティティを取得します（この様子を示した図については、前述の「Core Dataスタック」（11 ページ）を参照してください）。概念的にはそれほど難しくありません（単純にスタックを下に移動するだけです）。したがって、自分でもこの処理を簡単に実装することができます。ただし、このクラスメソッドを使用した方がはるかに簡単です。

ソート記述子の設定

ソート記述子を指定しないと、フェッチから返されるオブジェクトの順番は未定義になります。Eventオブジェクトを時間順に取得するには、フェッチに対してソート記述子を指定する必要があります。複数のソート順を指定する場合もあるので（たとえば、部署、姓、および名で社員をソートするような場合）、ソート記述子を配列に入れる必要があります。

>> viewDidLoadの実装に最後で、Eventオブジェクトを（最新の日付が先頭になるように）作成日順に並べるためのソート記述子と可変配列を作成します。このソート記述子をこの配列に追加し、この配列をフェッチ要求のsortDescriptors配列に設定します。

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
initWithKey:@"creationDate" ascending:NO];
NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor,
nil];
[request setSortDescriptors:sortDescriptors];
[sortDescriptors release];
[sortDescriptor release];
```

（後でソート記述子を追加する場合に備えて、通常、NSArrayのinitWithObjects:メソッドを使用するのが便利です。）

フェッチ要求の実行

フェッチ要求を作成したら、それを実行します。イベント配列は可変でなければなりません。したがって、結果の可変コピーを作成します。

>> viewDidLoadの現在の実装の最後に次のコードを追加します。

```
NSError *error;
NSMutableArray *mutableFetchResults = [[managedObjectContext
executeFetchRequest:request error:&error] mutableCopy];
if (mutableFetchResults == nil) {
    // エラーを処理する。
}
```

前述のとおり、この例ではエラーの処理の仕方はデベロッパに任されています（「エラー処理」（28 ページ）を参照）。

終了処理

最後の手順は、View Controllerのイベント配列インスタンス変数を設定して、割り当て済みのオブジェクトを解放することです。

>> viewDidLoadの現在の実装の最後に次のコードを追加します。

```
[self setEventsArray:mutableFetchResults];
[mutableFetchResults release];
[request release];
```

ビルドとテスト

アプリケーションをビルドして実行すると、適切にコンパイルされて、アプリケーションの起動時に既存のEventオブジェクトが表示されます。

Core Dataのまとめ

この章の重要な点は、次のとおりです。

- 管理オブジェクトをフェッチするには、フェッチ要求を作成します。

  フェッチ要求では、最低限、エンティティを指定する必要があります。エンティティは、NSEntityDescriptionのentityForName:inManagedObjectContext:簡易メソッドを使用して取得します。述語とソート順の配列を指定することもできます。

必要以上に多くのオブジェクトを取得しないようにして、メモリ使用量を抑えるために、通常は、述語を使用してできる限り要求を限定するべきです。

- 管理対象オブジェクトは、必ずしも明示的にフェッチする必要はありません。

  このチュートリアルにはオブジェクト間の関係が出てこないため、これを直接コードで扱うことはありませんでした。以前にも説明しましたが、Core Dataは、必要であれば自動的に関係先のオブジェクトを取得します。たとえば、Employeeオブジェクトを取得するためにフェッチを実行してからそれに関連するDepartmentを要求すると、それがまだフェッチされていない場合、Core Dataは自動的にDepartmentをフェッチします。

イベントの削除

この章の目標は、ユーザがリストからイベントを削除できるようにすることです。

管理オブジェクトの削除

新規の管理オブジェクトを作成したときにわかったとおり、データベース内のレコードの存続期間と特定の管理オブジェクトの存続期間は関連付けられていません。管理オブジェクトを作成しても、データベース内のそのオブジェクトに自動的にレコードが作成されるわけではありません。それには、コンテキストを保存する必要があります。同様に、オブジェクトを解放したからといって、それに対応するレコードそのものが破棄されるわけではありません。

レコードを削除するには、`NSManagedObjectContext`の`deleteObject:`メソッドを使用して、オブジェクトに削除マークを付けるように管理オブジェクトコンテキストに指示します。次に、実際にレコードを破棄するには、`save:`を使用してこのアクションをコミットします。

イベントの削除

削除を処理するには、Table Viewのデータソースメソッド`tableView:commitEditingStyle:forRowAtIndexPath:`を実装します。それには、次の3つの手順を実行する必要があります。

1. 選択されているオブジェクトを削除します。
2. Table Viewを更新します。
3. 変更を保存します。

アクションが削除の場合にのみ、この処理を実行します。

>>`RootViewController`の実装ファイルで、`tableView:commitEditingStyle:forRowAtIndexPath:`メソッドを次のように実装します。

```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete) {

        // 指定のインデックスパスにある管理オブジェクトを削除する。
        NSManagedObject *eventToDelete = [eventsArray
objectAtIndex:indexPath.row];
        [managedObjectContext deleteObject:eventToDelete];
```

```
        // 配列とTable Viewを更新する。
        [eventsArray removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
withRowAnimation:YES];

        // 変更をコミットする。
        NSError *error;
        if (![managedObjectContext save:&error]) {
            // エラーを処理する。
        }
    }
}
```

ビルドとテスト

アプリケーションをビルドしてテストします。アプリケーションはエラーなくコンパイルされて実行されるはずです。「Edit」をタップすると、Table Viewが編集モードになります。行を削除すると、その行がTable Viewから適切に削除されます。アプリケーションを終了して再起動すると、削除した行はもう表示されません。

これで、このチュートリアルを完結しました。ここからは、Core Dataについての知識と理解を深めるための研究を開始できます。次の章では、そのための提案をいくつか示します。

Core Dataのまとめ

ここでは、Core Dataを使用するために慣れておく必要がある基本的な作業を実行しました。

- 管理オブジェクトモデルにエンティティを作成しました。そのエンティティを表すカスタムクラスも作成しました。
- 管理オブジェクトのインスタンスを作成しました。また、そのプロパティ値をいくつか変更しました。
- 管理対象オブジェクトをフェッチしました。
- 管理オブジェクトを削除しました。

これらの作業を通して、Core Dataスタック（「Core Dataスタック」（11 ページ）を参照）内のオブジェクトのうち、デベロッパが直接やり取りをしなければならないのは、管理オブジェクトコンテキストだけだということに気付いたと思います。スタック内のその他のオブジェクトにアクセスすることもできますが、通常はそれらを直接使用する必要はありません。Xcodeのテンプレートがそれらのセットアップを行ってくれます。あるいは、簡易クラスメソッドを使用して、それらにアクセスする必要がある特定のタスクを実行します。

次のステップ

この章の目標は、Core Dataの理解を深めるために取るべき次のステップと、今後のアプリケーションでのCore Dataの使いかたを提案することです。

次のステップ

ここでは、Core Dataについての理解を深める方法と、Core Dataをアプリケーションに組み込む方法についての提案をいくつか示します。探求する中で、助けが必要な場合は『*Core Data Programming Guide*』を参照してください。

覚えておくべき重要な点の1つは、管理オブジェクトモデルのスキーマを変更した場合は、通常、アプリケーションは以前のバージョンを使用して作成されたファイルを開けなくなるということです。経験を積むにつれて、この問題にも対処しなければなりません（『*Core Data Model Versioning and Data Migration Programming Guide*』を参照）。もっとも、最初はもっと簡単なステップから始めます。

Core Data Utility Tutorial

少しの間iPhoneを離れて、『*Core Data Utility Tutorial*』を使って学習することも価値があります。これは、多くの点でこのチュートリアルに似ていますが、ユーザインターフェイスに惑わされることからは解放されます。また、管理オブジェクトのライフサイクルに関する2つの新しい概念が紹介されています。また、プログラムで管理オブジェクトモデルを作成することによって、管理オブジェクトモデルが単なるオブジェクトのコレクションであるということがより明確になります。

フェッチ結果コントローラの使用

iPhoneに話を戻して、`NSFetchedResultsController`オブジェクトを使用するようにLocationsアプリケーションを更新してみます。フェッチ結果コントローラは、主に、大量のオブジェクトのフェッチを大幅に効率化することを目的としていますが、小規模なデータセットを使用してその使い方を練習することは価値があります。参考のために、「CoreDataBooks」サンプルを参照してください。

Xcodeを使用した管理オブジェクトモデルの作成

チュートリアル『*Creating a Managed Object Model with Xcode*』を使って学習します。Core Data用のXcodeツールと、特にエンティティ間の関係の作成方法についてさらに詳しく学習します。これは、「ドリルダウン型のインターフェイス」（40 ページ）で提案するような、相互に関連するエンティティを含むアプリケーションを作成する場合には不可欠です。

ドリルダウン型のインターフェイス

ユーザがイベントを詳しく調査できる（コメントを編集したり、写真を追加できる）ようにするために、ドリルダウン型のインターフェイスを提供するようにLocationsアプリケーションを拡張します。Eventエンティティにプロパティを追加する必要があります。また、エンティティをもう1つ追加する必要があるかもしれません。*TaggedLocations*サンプルは、対多関係を持つもう1つのエンティティを使用する例を提供しています。

写真を追加する場合は、ストアから取得するすべてのEventと一緒に写真をフェッチすることによるメモリ管理への影響を考慮します。Core Dataには**フォールティング**という機能があります（『*Core Data Programming Guide*』の「Managed Objects」を参照）。この機能を利用すると、オブジェクトグラフを完成させる必要がなくなります。通常は、写真を別のエンティティとしてモデル化し、Eventエンティティから Photoエンティティへの関係（写真からイベントへの相互関係）を定義します。`Event`オブジェクトだけを取得した場合は、写真との関係はフォールトによって表現されます。イベントに写真を要求すると、Core Dataは自動的にこのフォールトを解決して、それに対応するデータを取得します。例については*PhotoLocations*を参照してください。

「Add」シートの追加

「Add」シートを利用すると、イベントを作成するときにより多くの情報を入力できます。「Add」シートコントローラに渡す情報について検討します。「Add」シートでの`Event`オブジェクトの編集と、アプリケーションのそれ以外の部分での編集を独立に維持する方法についても検討します（ヒント：2つの管理オブジェクトコンテキストの使用を検討します。「CoreDataBooks」サンプルを参照）。

書類の改訂履歴

この表は「*iPhone OS Core Data*チュートリアル」の改訂履歴です。

日付	メモ
2009-09-09	サンプルアプリケーションへのリンクを修正しました。
2009-06-04	誤植を修正しました。
2009-03-19	applicationDidFinishLaunching:の実装に欠けていたコード行を追加しました。
2009-03-15	Core Dataを使用したiPhoneアプリケーション開発を紹介するチュートリアルの初版。

書類の改訂履歴