Location and Maps Programming Guide

Developer

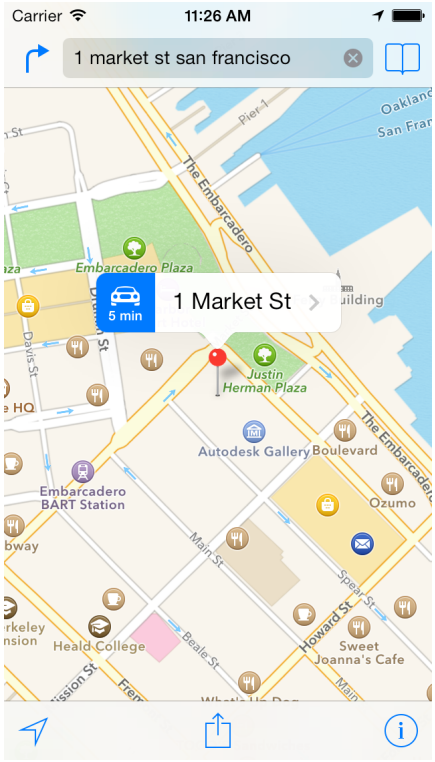
Contents

Figures, Tables, and Listings

Objective-CSwift

About Location Services and Maps

Using location-based information in your app is a great way to keep the user connected to the surrounding world. Whether you use this information for practical purposes (such as navigation) or for entertainment, location-based information can enhance the overall user experience.



Location-based information consists of two pieces: location services and maps. Location services are provided by the Core Location framework, which defines Objective-C interfaces for obtaining information about the user's location and heading (the direction in which a device is pointing). Maps are provided by the Map Kit framework, which supports both the display and annotation of maps similar to those found in the Maps app. (To use the features of the Map Kit framework, you must turn on the Maps capability in your Xcode project.) Location services and maps are available on both iOS and OS X.

At a Glance

By incorporating geographic data into your apps, you can orient users to the surrounding environment and keep them connected to people nearby.

Because maps and location services are available in both iOS and OS X, location-based apps use very similar code on both platforms. The differences are in user interface code (for example, using `UIView` in iOS and `NSView` in OS X) and in the few features that are supported in iOS only (such as heading service).

Location Services Provide a Geographical Context for Apps

Knowing the user's geographic location can improve the quality of the information you offer, and might even be at the heart of your app. Apps with navigation features use location services to monitor the user's position and generate updates. Other types of apps use location services to enable social connections among nearby users.

Relevant Chapters: Getting the User's Location (page 12), Region Monitoring and iBeacon (page 23), Geocoding Location Data (page 38)

iBeacon Transmitters Enhance the User's Experience of a Location

iBeacon transmitters provide a way to create and monitor beacons that advertise certain identifying information using Bluetooth low-energy wireless technology. Bluetooth low-energy beacons that advertise the same universally unique identifier (UUID) form a beacon region that your app can monitor through the Core Location region monitoring support. Beacons with the same UUID can be distinguished by the additional information they advertise. While a beacon is in range of a user's device, apps can also monitor for the relative distance to the beacon.

You can use the information advertised by beacons to enhance the user's experience of a particular location. For example, a museum app can monitor for beacons placed near the museum's important exhibits. As a user approaches a particular exhibit, the app can use the relative distance of the beacon as a cue to provide more information about that exhibit rather than another.

Because beacons advertise information using Bluetooth low-energy technology, you can turn any iOS device that supports Bluetooth low-energy data sharing into a beacon.

Relevant Chapter: Region Monitoring and iBeacon (page 23)

Heading Information Indicates the User's Current Orientation

Heading services complement the basic location services by providing more precise information about which way a device is pointed. The most obvious use for this technology is a compass, but you can also use it to support augmented reality, games, and navigational apps. Even on devices that don't have a magnetometer—the hardware used to get precise heading information—information about the user's course and speed are available for apps that need it.

Relevant Chapter: Getting the Heading and Course of a Device (page 34)

Maps Support Navigation and the Display of Geographically Relevant Content

Maps help users visualize geographical data in a way that is easy to understand. For example, a map can show satellite data for an area, or pitch a map view at an angle and display three-dimensional buildings for a 3D perspective on the area. You can incorporate Map Kit framework standard views into your app and display information tied to specific geographic points. In addition, this framework lets you layer custom information on top of a map, scroll it along with the rest of the map content, and take static snapshots of a map for printing.

Relevant Chapters: Displaying Maps (page 42), Annotating Maps (page 54)

Routing Apps Provide Directions to the User

A routing app can receive coordinates from the Maps app and use those coordinates to provide point-to-point directions to users. An app that provides navigation capabilities can declare itself a routing app with minimal additional effort. In addition to driving and walking directions, routing apps can support several other modes of transport, including taxi, airplane, and various public transportation options.

Relevant Chapter: Providing Directions (page 78)

Local Search

Users often want to find locations based on descriptive information, such as a name, address, or business type. Using the Map Kit local search API, you can perform searches that are based on this type of user input and show the results on your map.

Relevant Chapter: Enabling Search (page 88)

How to Use This Document

You don't have to read this entire document to use each technology. Core Location and Map Kit framework services are independent of other services. The beginning of each chapter introduces the terminology and information you need to understand the corresponding technology, followed by examples and task-related steps. The only exception is the Annotating Maps (page 54) chapter, which builds on the information presented in the Displaying Maps (page 42) chapter.

See Also

For information about the classes of the Core Location framework, see *Core Location Framework Reference*.

For information about the classes of the Map Kit framework, see *MapKit Framework Reference*.

To learn how to enable the Maps service in your project, see "Configuring Maps" in *App Distribution Guide*.

Getting the User's Location

Objective-CSwift

Apps use location data for many purposes, ranging from social networking to turn-by-turn navigation services. They get location data through the classes of the Core Location framework. This framework provides several services that you can use to get and monitor the device's current location:

- The significant-change location service provides a low-power way to get the current location and be notified when significant changes occur.
- The standard location service offers a highly configurable way to get the current location and track changes.
- Region monitoring lets you monitor boundary crossings for defined geographical regions and Bluetooth low-energy beacon regions. (Beacon region monitoring is available in iOS only.)

To use the features of the Core Location framework, you must link your app to `CoreLocation.framework` in your Xcode project. To access the classes and headers of the framework, include an `#import <CoreLocation/CoreLocation.h>` statement at the top of any relevant source files.

For general information about the classes of the Core Location framework, see *Core Location Framework Reference* .

Requiring the Presence of Location Services in an iOS App

If your iOS app requires location services to function properly, include the `UIRequiredDeviceCapabilities` key in the app's `Info.plist` file. The App Store uses the information in this key to prevent users from downloading apps to devices that don't contain the listed features.

The value for the `UIRequiredDeviceCapabilities` is an array of strings indicating the features that your app requires. Two strings are relevant to location services:

- Include the `location-services` string if you require location services in general.
- Include the `gps` string if your app requires the accuracy offered only by GPS hardware.

Important: If your iOS app uses location services but is able to operate successfully without them, don't include the corresponding strings in the `UIRequiredDeviceCapabilities` key.

For more information about the `UIRequiredDeviceCapabilities` key, see *Information Property List Key Reference*.

Getting the User's Current Location

The Core Location framework lets you locate the current position of the device and use that information in your app. The framework reports the device's location to your code and, depending on how you configure the service, also provides periodic updates as it receives new or improved data.

Two services can give you the user's current location:

- The *standard location service* is a configurable, general-purpose solution for getting location data and tracking location changes for the specified level of accuracy.
- The *significant-change location service* delivers updates only when there has been a significant change in the device's location, such as 500 meters or more.

Gathering location data is a power-intensive operation. For most apps, it's usually sufficient to establish an initial position fix and then acquire updates only periodically after that. Regardless of the importance of location data in your app, you should choose the appropriate location service and use it wisely to avoid draining a device's battery. For example:

- If your iOS app must keep monitoring location even while it's in the background, use the standard location service and specify the `location` value of the `UIBackgroundModes` key to continue running in the background and receiving location updates. (In this situation, you should also make sure the location manager's `pausesLocationUpdatesAutomatically` property is set to `YES` to help conserve power.) Examples of apps that might need this type of location updating are fitness or turn-by-turn navigation apps.
- If GPS-level accuracy isn't critical for your app and you don't need continuous tracking, use the signicant-change location service. This service can save power by not tracking continuously and only waking the device and launching your app when significant changes occur.

Determining Whether Location Services Are Available

There are situations where location services may not be available. For example:

- The user disables location services in the Settings app or System Preferences.

- The user denies location services for a specific app.
- The device is in Airplane mode and unable to power up the necessary hardware.

For these reasons, it's recommended that you always call the `locationServicesEnabled` class method of `CLLocationManager` before attempting to start either the standard or significant-change location services. If it returns `NO` and you attempt to start location services anyway, the system prompts the user to confirm whether location services should be re-enabled. Because the user probably disabled location services on purpose, the prompt is likely to be unwelcome.

Starting the Standard Location Service

The standard location service is the most common way to get a user's current location because it's available on all devices and in both iOS and OS X. Before using this service, you configure it by specifying the desired accuracy of the location data and the distance that must be traveled before reporting a new location. When you start the service, it uses the specified parameters to determine the hardware to enable and then proceeds to report location events to your app. Because this service takes into account these parameters, it's most appropriate for apps that need more fine-grained control over the delivery of location events. The precision of the standard location service is needed by navigation apps or any app that requires high-precision location data or a regular stream of updates. Because this service typically requires the location-tracking hardware to be enabled for longer periods of time, higher power usage can result.

To use the standard location service, create an instance of the `CLLocationManager` class and configure its `desiredAccuracy` and `distanceFilter` properties. To begin receiving location notifications, assign a delegate to the object and call the `startUpdatingLocation` method. As location data becomes available, the location manager notifies its assigned delegate object. If a location update has already been delivered, you can also get the most recent location data directly from the `CLLocationManager` object without waiting for a new event to be delivered. To stop the delivery of location updates, call the `stopUpdatingLocation` method of the location manager object.

Listing 1-1 shows a sample method that configures a location manager for use. The sample method is part of a class that caches its location manager object in a member variable for later use. (The class also conforms to the `CLLocationManagerDelegate` protocol and so acts as the delegate for the location manager.) Because the app doesn't need precise location data, it configures the location service to report the user's general area and send notifications only when the user moves at least half a kilometer.

Listing 1-1 Starting the standard location service

```
- (void)startStandardUpdates
{
    // Create the location manager if this object does not
```

```
    // already have one.
    if (nil == locationManager)
        locationManager = [[CLLocationManager alloc] init];

    locationManager.delegate = self;
    locationManager.desiredAccuracy = kCLLocationAccuracyKilometer;

    // Set a movement threshold for new events.
    locationManager.distanceFilter = 500; // meters

    [locationManager startUpdatingLocation];
}
```

The code for receiving location updates from this service is shown in Receiving Location Data from a Service (page 16).

Starting the Significant-Change Location Service

The significant-change location service provides accuracy that's good enough for most apps and represents a power-saving alternative to the standard location service. The service uses Wi-Fi to determine the user's location and report changes in that location, allowing the system to manage power usage much more aggressively than it could otherwise. The significant-change location service can also wake up an iOS app that is currently suspended or not running in order to deliver new location data.

To use the significant-change location service, create an instance of the `CLLocationManager` class, assign a delegate to it, and call the `startMonitoringSignificantLocationChanges` method as shown in Listing 1-2. As location data becomes available, the location manager notifies its assigned delegate object. If a location update has already been delivered, you can also get the most recent location data directly from the `CLLocationManager` object without waiting for a new event to be delivered.

Listing 1-2 Starting the significant-change location service

```
- (void)startSignificantChangeUpdates
{
    // Create the location manager if this object does not
    // already have one.
    if (nil == locationManager)
        locationManager = [[CLLocationManager alloc] init];
```

```
    locationManager.delegate = self;
    [locationManager startMonitoringSignificantLocationChanges];
}
```

As with the standard location service, location data is delivered to the delegate object as described in Receiving Location Data from a Service (page 16). To stop the significant change location service, call the `stopMonitoringSignificantLocationChanges` method.

If you leave the significant-change location service running and your iOS app is subsequently suspended or terminated, the service automatically wakes up your app when new location data arrives. At wake-up time, the app is put into the background and you are given a small amount of time (around 10 seconds) to manually restart location services and process the location data. (You must manually restart location services in the background before any pending location updates can be delivered, as described in Knowing When to Start Location Services (page 17).) Because your app is in the background, it must do minimal work and avoid any tasks (such as querying the network) that might prevent it from returning before the allocated time expires. If it does not, your app will be terminated. If an iOS app needs more time to process the location data, it can request more background execution time using the `beginBackgroundTaskWithName:expirationHandler:` method of the `UIApplication` class.

> **Note:** When a user disables the Background App Refresh setting either globally or for your app, the significant-change location service doesn't relaunch your app. Further, while Background App Refresh is off an app doesn't receive significant-change or region monitoring events even when it's in the foreground.

Receiving Location Data from a Service

The way you receive location events is the same whether you use the standard or the significant-change location service to get them. Beginning in OS X v10.9 and iOS 6, the location manager reports events to the `locationManager:didUpdateLocations:` method of its delegate when they become available. (In earlier versions of both operating systems, the location manager reports events to the `locationManager:didUpdateToLocation:fromLocation:` method.) If there is an error retrieving an event, the location manager calls the `locationManager:didFailWithError:` method of its delegate instead.

Listing 1-3 shows the delegate method for receiving location events. Because the location manager object sometimes returns cached events, it's recommended that you check the timestamp of any location events you receive. (It can take several seconds to obtain a rough location fix, so the old data simply serves as a way to

reflect the last known location.) In this example, the method throws away any events that are more than fifteen seconds old under the assumption that events up to that age are likely to be good enough. If you are implementing a navigation app, you might want to lower the threshold.

Listing 1-3 Processing an incoming location event

```
// Delegate method from the CLLocationManagerDelegate protocol.
- (void)locationManager:(CLLocationManager *)manager
      didUpdateLocations:(NSArray *)locations {
    // If it's a relatively recent event, turn off updates to save power.
   CLLocation* location = [locations lastObject];
   NSDate* eventDate = location.timestamp;
   NSTimeInterval howRecent = [eventDate timeIntervalSinceNow];
   if (abs(howRecent) < 15.0) {
      // If the event is recent, do something with it.
      NSLog(@"latitude %+.6f, longitude %+.6f\n",
              location.coordinate.latitude,
              location.coordinate.longitude);
   }
}
```

In addition to a location object's timestamp, you can also use the accuracy reported by that object to determine whether you want to accept an event. As it receives more accurate data, the location service may return additional events, with the accuracy values reflecting the improvements accordingly. Throwing away less accurate events means your app wastes less time on events that can't be used effectively anyway.

Knowing When to Start Location Services

Apps that use location services should not start those services until they're needed. With a few exceptions, avoid starting location services immediately at launch time or before such services might reasonably be used. Otherwise you might raise questions in the user's head about how your app uses location data. The user knows when your app starts location services because the first time your app starts the service, the system prompts the user for permission to use it. Waiting until the user performs a task that actually requires those services helps build trust that your app is using them appropriately. Another way to build trust is to include the `NSLocationUsageDescription` key in your app's `Info.plist` file and set the value of that key to a string that describes how your app intends to use location data.

If you are monitoring regions or using the significant-change location service in your app, there are situations where you must start location services at launch time. Apps using those services can be terminated and subsequently relaunched when new location events arrive. Although the app itself is relaunched, location services are not started automatically. When an app is relaunched because of a location update, the launch options dictionary passed to your `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:` method contains the `UIApplicationLaunchOptionsLocationKey` key. The presence of that key signals that new location data is waiting to be delivered to your app. To obtain that data, you must create a new `CLLocationManager` object and restart the location services that you had running prior to your app's termination. When you restart those services, the location manager delivers all pending location updates to its delegate.

Getting Location Events in the Background (iOS Only)

iOS supports the delivery of location events to apps that are suspended or no longer running. The delivery of location events in the background supports apps whose functionality would be impaired without them, so configure your app to receive background events only when doing so provides a tangible benefit to the user. For example, a turn-by-turn navigation app needs to track the user's position at all times and notify the user when it's time to make the next turn. If your app can make do with alternate means, such as region monitoring, it should do so.

You have multiple options for obtaining background location events, and each has advantages and disadvantages with regard to power consumption and location accuracy. Whenever possible, apps should use the significant-change location service (described in Starting the Significant-Change Location Service (page 15)), which can use Wi-Fi to determine the user's position and consumes the least amount of power. But if your app requires higher precision location data, you can configure it as a background location app and use the standard location service.

Important: A user can explicitly disable background capabilities for any app. If a user disables Background App Refresh in the Settings app—either globally for all apps or for your app in particular—your app is prevented from using any location services in the background. You can determine whether your app can process location updates in the background by checking the value of the `backgroundRefreshStatus` property of the `UIApplication` class.

Using the Standard Location Service in the Background

iOS apps can use the standard location service in the background if they provide services that require continuous location updates. As a result, this capability is most appropriate for apps that assist the user in navigation- and fitness-related activities. To enable this capability in your app, enable the Background Modes capability in your Xcode project (located in the Capabilities tab of your project) and enable the Location updates mode. The code you write to start and stop the standard location services is unchanged.

The system delivers location updates to background location apps when the app is in the foreground, is running in the background, or is suspended. In the case of a suspended app, the system wakes up the app, delivers the update to the location manager's delegate, and then returns the app to the suspended state as soon as possible. While it is running in the background, your app should do as little work as possible to process the new location data.

When location services are enabled, iOS must keep the location hardware powered up so that it can gather new data. Keeping the location hardware running degrades battery life, and you should always stop location services in your app whenever you do not need the resulting location data.

If you must use location services in the background, you can help Core Location maintain good battery life for the user's device by taking additional steps when you configure your location manager object:

- Make sure the location manager's `pausesLocationUpdatesAutomatically` property is set to `YES`. When this property is set to `YES`, Core Location pauses location updates (and powers down the location hardware) whenever it makes sense to do so, such as when the user is unlikely to be moving anyway. (Core Location also pauses updates when it can't obtain a location fix.)
- Assign an appropriate value to the location manager's `activityType` property. The value in this property helps the location manager determine when it is safe to pause location updates. For an app that provides turn-by-turn automobile navigation, setting the property to `CLActivityTypeAutomotiveNavigation` causes the location manager to pause events only when the user does not move a significant distance over a period of time.
- Call the `allowDeferredLocationUpdatesUntilTraveled:timeout:` method whenever possible to defer the delivery of updates until a later time, as described in Deferring Location Updates While Your App Is in the Background (page 20).

When the location manager pauses location updates, it notifies its delegate object by calling its `locationManagerDidPauseLocationUpdates:` method. When the location manager resumes updates, it calls the delegate's `locationManagerDidResumeLocationUpdates:` method. You can use these delegate methods to perform tasks or adjust the behavior of your app. For example, when location updates are paused, you might use the delegate notification to save data to disk or stop location updates altogether. A navigation app in the middle of turn-by-turn directions might prompt the user and ask whether navigation should be disabled temporarily.

> **Note:** If your app is terminated either by a user or by the system, the system doesn't automatically restart your app when new location updates arrive. A user must explicitly relaunch your app before the delivery of location updates resumes. The only way to have your app relaunched automatically is to use region monitoring or the significant-change location service.
>
> However, when a user disables the Background App Refresh setting either globally or specifically for your app, the system doesn't relaunch your app for *any* location events, including significant-change or region monitoring events. Further, while Background App Refresh is off your app won't receive significant-change or region monitoring events even when it's in the foreground. When a user reenables Background App Refresh for your app, Core Location restores all background services, including any previously registered regions.

Deferring Location Updates While Your App Is in the Background

In iOS 6 and later, you can defer the delivery of location updates when your app is in the background. It's recommended that you use this feature when your app could process the location data later without any problems. For example, a fitness app that tracks the user's location on a hiking trail could defer updates until the user hikes a certain distance or until a certain amount of time has elapsed and then process the updates all at once. Deferring updates saves power by letting your app sleep for longer periods of time. Because deferring location updates requires the presence of GPS hardware on the target device, be sure to call the `deferredLocationUpdatesAvailable` class method of the `CLLocationManager` class to determine whether the device supports deferred location updates.

Call the `allowDeferredLocationUpdatesUntilTraveled:timeout:` method of the `CLLocationManager` class to begin deferring location updates. As shown in Listing 1-4, the usual place to call this method is in the `locationManager:didUpdateLocations:` method of your location manager delegate object. This method shows how a sample hiking app might defer location updates until the user hikes a minimum distance. To prevent itself from calling the `allowDeferredLocationUpdatesUntilTraveled:timeout:` method multiple times, the delegate uses an internal property to track whether updates have already been deferred. It sets the property to `YES` in this method and sets it back to `NO` when deferred updates end.

Listing 1-4 Deferring location updates

```
// Delegate method from the CLLocationManagerDelegate protocol.
- (void)locationManager:(CLLocationManager *)manager
      didUpdateLocations:(NSArray *)locations {
   // Add the new locations to the hike
   [self.hike addLocations:locations];

   // Defer updates until the user hikes a certain distance
   // or when a certain amount of time has passed.
   if (!self.deferringUpdates) {
      CLLocationDistance distance = self.hike.goal - self.hike.distance;
      NSTimeInterval time = [self.nextAudible timeIntervalSinceNow];
      [locationManager allowDeferredLocationUpdatesUntilTraveled:distance
                                                         timeout:time];
   self.deferringUpdates = YES;
   }
}
```

When a condition you specified in the `allowDeferredLocationUpdatesUntilTraveled:timeout:` method is met, the location manager calls the `locationManager:didFinishDeferredUpdatesWithError:` method of its delegate object to let you know that it has stopped deferring the delivery of location updates. The location manager calls this delegate method exactly once for each time your app calls the `allowDeferredLocationUpdatesUntilTraveled:timeout:` method. After deferred updates end, the location manager proceeds to deliver any location updates to your delegate's `locationManager:didUpdateLocations:` method.

You can stop the deferral of location updates explicitly by calling the `disallowDeferredLocationUpdates` method of the `CLLocationManager` class. When you call this method, or stop location updates altogether using the `stopUpdatingLocation` method, the location manager calls your delegate's `locationManager:didFinishDeferredUpdatesWithError:` method to let you know that deferred updates have indeed stopped.

If the location manager encounters an error and can't defer location updates, you can access the cause of the error when you implement the `locationManager:didFinishDeferredUpdatesWithError:` delegate method. For a list of the possible errors that may be returned—and what, if anything, you can to resolve them—see the `CLError` constants in *Core Location Constants Reference*.

Tips for Conserving Battery Power

Getting location data in an iOS-based device can require a lot of power. Because most apps don't need location services to be running all the time, turning off location services when they're not needed is the simplest way to save power.

- **Turn off location services when you aren't using them.** This guideline may seem obvious but it's worth repeating. With the exception of navigation apps that offer turn-by-turn directions, most apps don't need location services to be on all the time. Turn location services on just long enough to get a location fix and then turn them off. Unless the user is in a moving vehicle, the current location shouldn't change frequently enough to be an issue. And you can always start location services again later if needed.
- **Use the significant-change location service instead of the standard location service whenever possible.** The significant-change location service provides significant power savings while still allowing you to leave location services running. This is highly recommended for apps that need to track changes in the user's location but don't need the higher precision offered by the standard location services.
- **Use lower-resolution values for the desired accuracy unless doing so would impair your app.** Requesting a higher accuracy than you need causes Core Location to power up additional hardware and waste power for precision you don't need. Unless your app really needs to know the user's position within a few meters, don't put the values `kCLLocationAccuracyBest` or `kCLLocationAccuracyNearestTenMeters` in the `desiredAccuracy` property. And remember that specifying a value of `kCLLocationAccuracyThreeKilometers` doesn't prevent the location service from returning better data. Most of the time, Core Location can return location data with an accuracy within a hundred meters or so.
- **Turn off location events if the accuracy doesn't improve over a period of time.** If your app isn't receiving events with the desired level of accuracy, you should look at the accuracy of events you do receive and see if it is improving or staying about the same over time. If accuracy isn't improving, it could be because the desired accuracy is simply not available at the moment. Turning off location services and trying again later prevents your app from wasting power.
- *Specify an activity type for your app when using the standard location service to process location data in the background.* Letting Core Location know what type of activity is associated with your app (for example, whether it's an automobile navigation app or a fitness app) helps the location manager determine the most appropriate time to pause location updates when your app is in the background. Helping the location manager to determine when to pause location updates in your app can help improve battery life on the user's device.
- *Allow the location manager to defer the delivery of location updates when your app is in the background.* When your app can't do anything useful with the location updates it receives from the location manager—other than log the information and go back to sleep—allow the location manager to defer those updates until they're more meaningful to your app.

Region Monitoring and iBeacon

Objective-CSwift

The Core Location framework provides two ways to detect a user's entry and exit into specific regions: geographical region monitoring (iOS 4.0 and later and OS X v10.8 and later) and beacon region monitoring (iOS 7.0 and later). A *geographical region* is an area defined by a circle of a specified radius around a known point on the Earth's surface. In contrast, a *beacon region* is an area defined by the device's proximity to Bluetooth low-energy beacons. Beacons themselves are simply devices that advertise a particular Bluetooth low-energy payload—you can even turn your iOS device into a beacon with some assistance from the Core Bluetooth framework.

Apps can use region monitoring to be notified when a user crosses geographic boundaries or when a user enters or exits the vicinity of a beacon. While a beacon is in range of an iOS device, apps can also monitor for the relative distance to the beacon. You can use these capabilities to develop many types of innovative location-based apps. Because geographical regions and beacon regions differ, the type of region monitoring you decide to use will likely depend on the use case of your app.

In iOS, regions associated with your app are tracked at all times, including when the app isn't running. If a region boundary is crossed while an app isn't running, that app is relaunched into the background to handle the event. Similarly, if the app is suspended when the event occurs, it's woken up and given a short amount of time (around 10 seconds) to handle the event. When necessary, an app can request more background execution time using the `beginBackgroundTaskWithExpirationHandler:` method of the `UIApplication` class.

In OS X, region monitoring works only while the app is running (either in the foreground or background) and the user's system is awake. As a result, the system doesn't launch apps to deliver region-related notifications.

Determining the Availability of Region Monitoring

Before attempting to monitor any regions, your app should check whether region monitoring is supported on the current device. Here are some reasons why region monitoring might not be available:

- The device doesn't have the necessary hardware to support region monitoring.
- The user denied the app the authorization to use region monitoring.
- The user disabled location services in the Settings app.
- The user disabled Background App Refresh in the Settings app, either for the device or for your app.

- The device is in Airplane mode and can't power up the necessary hardware.

In iOS 7.0 and later, always call the `isMonitoringAvailableForClass:` and `authorizationStatus` class methods of `CLLocationManager` before attempting to monitor regions. (In OS X v10.8 and later and in previous versions of iOS, use the `regionMonitoringAvailable` class instead.) The `isMonitoringAvailableForClass:` method tells you whether the underlying hardware supports region monitoring for the specified class at all. If that method returns `NO`, your app can't use region monitoring on the device. If it returns `YES`, call the `authorizationStatus` method to determine whether the app is currently authorized to use location services. If the authorization status is `kCLAuthorizationStatusAuthorized`, your app can receive boundary crossing notifications for any regions it registered. If the authorization status is set to any other value, the app doesn't receive those notifications.

> **Note:** Even when an app isn't authorized to use region monitoring, it can still register regions for use later. If the user subsequently grants authorization to the app, monitoring for those regions will begin and will generate subsequent boundary crossing notifications. If you don't want regions to remain installed while your app is not authorized, you can use the `locationManager:didChangeAuthorizationStatus:` delegate method to detect changes in your app's status and remove regions as appropriate.

Finally, if your app needs to process location updates in the background, be sure to check the `backgroundRefreshStatus` property of the `UIApplication` class. You can use the value of this property to determine if doing so is possible and to warn the user if it is not. Note that the system doesn't wake your app for region notifications when the Background App Refresh setting is disabled globally or specifically for your app.

Monitoring Geographical Regions

Geographical region monitoring uses location services to detect entry and exit into known geographical locations (learn more about location services in Getting the User's Location (page 12)). You can use this capability to generate alerts when the user gets close to a specific location or to provide other relevant information. For example, upon approaching a specific dry cleaners, an app could notify the user to pick up any clothes that are now ready.

Defining a Geographical Region to Be Monitored

To begin monitoring a geographical region, you must define the region and register it with the system. In iOS 7.0 and later, you define geographical regions using the `CLCircularRegion` class. (In OS X v10.8 and later and in previous versions of iOS, you use the `CLRegion` class instead.) Each region you create must include both the data that defines the desired geographic area and a unique identifier string. The identifier string is the only guaranteed way for your app to identify a region later. To register a region, call the `startMonitoringForRegion:` method of your `CLLocationManager` object.

Listing 2-1 shows a sample method that creates a new geographic region based on a circular overlay region. The overlay's center point and radius form the boundary for the region, although if the radius is too large to be monitored, it is reduced automatically. You don't need to save strong references to the regions you create but might want to store the region's identifier if you plan to access the region information later.

Listing 2-1 Creating and registering a geographical region based on a Map Kit overlay

```
- (void)registerRegionWithCircularOverlay:(MKCircle*)overlay
andIdentifier:(NSString*)identifier {


   // If the overlay's radius is too large, registration fails automatically,
   // so clamp the radius to the max value.
   CLLocationDistance radius = overlay.radius;
   if (radius > self.locManager.maximumRegionMonitoringDistance) {
      radius = self.locManager.maximumRegionMonitoringDistance;
   }


   // Create the geographic region to be monitored.
   CLCircularRegion *geoRegion = [[CLCircularRegion alloc]
      initWithCenter:overlay.coordinate
              radius:radius
          identifier:identifier];
   [self.locManager startMonitoringForRegion:geoRegion];
}
```

Monitoring of a geographical region begins immediately after registration for authorized apps. However, don't expect to receive an event right away, because only boundary crossings generate an event. In particular, if the user's location is already inside the region at registration time, the location manager doesn't automatically

generate an event. Instead, your app must wait for the user to cross the region boundary before an event is generated and sent to the delegate. To check whether the user is already inside the boundary of a region, use the `requestStateForRegion:` method of the `CLLocationManager` class.

Be judicious when specifying the set of regions to monitor. Regions are a shared system resource, and the total number of regions available systemwide is limited. For this reason, Core Location limits to 20 the number of regions that may be simultaneously monitored by a single app. To work around this limit, consider registering only those regions in the user's immediate vicinity. As the user's location changes, you can remove regions that are now farther way and add regions coming up on the user's path. If you attempt to register a region and space is unavailable, the location manager calls the `locationManager:monitoringDidFailForRegion:withError:` method of its delegate with the `kCLErrorRegionMonitoringFailure` error code.

Handling Boundary-Crossing Events for a Geographical Region

By default, every time a user's current location crosses a boundary region, the system generates an appropriate region event for your app. Apps can implement the following methods to handle boundary crossings:

- `locationManager:didEnterRegion:`
- `locationManager:didExitRegion:`

You can customize which boundary-crossing events notify your app by explicitly setting the `notifyOnEntry` and `notifyOnExit` properties of the `CLRegion` class when you define and register a region. (The default value of both properties is `YES`.) For example, if you want to be notified only when the user exits the boundary of a region, you can set the value of the region's `notifyOnEntry` property to `NO`.

The system doesn't report boundary crossings until the boundary plus a system-defined cushion distance is exceeded. This cushion value prevents the system from generating numerous entered and exited events in quick succession while the user is traveling close the edge of the boundary.

When a region boundary is crossed, the most likely response is to alert the user of the proximity to the target item. If your app is running in the background, you can use local notifications to alert the user; otherwise, you can simply post an alert.

Monitoring Beacon Regions

Beacon region monitoring uses an iOS device's onboard radio to detect when the user is in the vicinity of Bluetooth low-energy devices that are advertising iBeacon information. As with geographical region monitoring, you can use this capability to generate alerts or to provide other relevant information when the user enters or

exits a beacon region. Rather than being identified by fixed geographical coordinates, however, a beacon region is identified by the device's proximity to Bluetooth low-energy beacons that advertise a combination of the following values:

- A *proximity UUID* (universally unique identifier), which is a 128-bit value that uniquely identifies one or more beacons as a certain type or from a certain organization
- A *major* value, which is a 16-bit unsigned integer that can be used to group related beacons that have the same proximity UUID
- A *minor* value, which is a 16-bit unsigned integer that differentiates beacons with the same proximity UUID and major value

Because a single beacon region can represent multiple beacons, beacon region monitoring supports several interesting use cases. For example, an app dedicated to enhancing the experience of customers at a particular department store can use the same proximity UUID to monitor all stores in the department store chain. When the user approaches a store, the app detects the store's beacons and uses the major and minor values of those beacons to determine additional information, such as which specific store was encountered or which section of the store the user is in. (Note that although every beacon must advertise a proximity UUID, major and minor values are optional.)

Defining a Beacon Region to Be Monitored

To begin monitoring a beacon region, define the region and register it with the system. You define a beacon region with the appropriate initialization method of the `CLBeaconRegion` class. When you create a `CLBeaconRegion` object, you specify the `proximityUUID`, `major`, and `minor` properties of the beacons you want to monitor (the proximity UUID is required; the major and minor values are optional). You must also supply a string that uniquely identifies the region so that you can refer to it in your code. Note that a region's identifier is unrelated to the identifying information that a beacon advertises.

To register a beacon region, call the `startMonitoringForRegion:` method of your `CLLocationManager` object. Listing 2-2 shows a sample method that creates and registers a beacon region.

Listing 2-2 Creating and registering a beacon region

```
- (void)registerBeaconRegionWithUUID:(NSUUID *)proximityUUID
andIdentifier:(NSString*)identifier {


   // Create the beacon region to be monitored.
   CLBeaconRegion *beaconRegion = [[CLBeaconRegion alloc]
      initWithProximityUUID:proximityUUID
                 identifier:identifier];
```

```
    // Register the beacon region with the location manager.
    [self.locManager startMonitoringForRegion:beaconRegion];
}
```

As with geographical region monitoring, monitoring of a beacon region begins immediately after registration for authorized apps. When a user's device detects a beacon that is advertising the identifying information defined by the registered beacon region (proximity UUID, major value, and minor value), the system generates an appropriate region event for your app.

> **Note:** It's not unusual to configure a beacon region using only a UUID value. Doing so yields a region that is activated when the device comes within range of any beacon with the specified UUID. Upon entering the beacon region, you would then begin ranging for beacons to obtain detailed information about the specific beacons that are nearby. For more information about beacon ranging, see Determining the Proximity of a Beacon Using Ranging (page 29).

Handling Boundary-Crossing Events for a Beacon Region

When a user enters the registered beacon region, the location manager calls the `locationManager:didEnterRegion:` of its delegate object. Similarly, when a user is no longer in range of any beacons in the registered beacon region, the location manager calls the `locationManager:didExitRegion:` of its delegate object. Note that a user must cross the region's boundary to trigger one of these calls; in particular, the location manager doesn't call `locationManager:didEnterRegion:` if the user is already within the region. You can implement these delegate methods to alert the user appropriately or to present location-specific UI.

You can specify which boundary-crossing events should notify your app by setting the `notifyOnEntry` and `notifyOnExit` properties of the beacon region. (The default value of both properties is `YES`.) For example, if you want to be notified only when the user exits the boundary of a region, you can set the value of the region's `notifyOnEntry` property to `NO`.

You can also postpone notifying a user upon entering a beacon region until the user turns on the device's display. To do so, simply set the value of the beacon region's `notifyEntryStateOnDisplay` property value to `YES` and set the region's `notifyOnEntry` property to `NO` when you register the beacon region. To prevent redundant notifications from being delivered to the user, post a local notification only once per region entry.

Determining the Proximity of a Beacon Using Ranging

While a user's device is inside a registered beacon region, apps can use the `startRangingBeaconsInRegion:` method of the `CLLocationManager` class to determine the relative proximity of one or more beacons in the region and to be notified when that distance changes. (Always call the `isRangingAvailable` class method of the `CLLocationManager` class before attempting to range beacons in a beacon region.) Knowing the relative distance to a beacon can be useful for many apps. For example, imagine a museum that places a beacon at each exhibit. A museum-specific app could use a particular exhibit's proximity as a cue to provide information about that exhibit rather than another.

The location manager calls the `locationManager:didRangeBeacons:inRegion:` of its delegate object whenever beacons in the specified beacon region come within range, go out of range, or their proximity changes. This delegate method provides an array of `CLBeacon` objects that represent the beacons currently in range. The array of beacons is ordered by approximate distance from the device, with the closest beacon at the beginning of the array. You can use the information in these objects to determine the proximity of the user to each beacon. The value in the `proximity` property of the `CLBeacon` object gives a general sense of the relative distance to a beacon.

> **Note:** Beacon ranging depends on detecting the strength of Bluetooth low-energy radio signals, and the accuracy of those signals is attenuated (or lessened) by walls, doors, and other physical objects. The signals are also affected by water, which means the human body itself will affect the signals. It is important to be aware of these factors when planning your iBeacon deployment because they will impact the `proximity` value reported by each beacon. If needed, use the `accuracy` and `rssi` values reported by each `CLBeacon` object to adjust the placement of your beacons during deployment.

Inspired by the sample museum app described earlier in this section, Listing 2-3 shows how to use a beacon's `proximity` property to determine its relative distance from the user's device. The code presents a UI that provides more information about a particular museum exhibit when the proximity of the closest beacon in the array is relatively close to the user (as defined by the `CLProximityNear` constant).

Listing 2-3 Determining the relative distance between a beacon and a device

```
// Delegate method from the CLLocationManagerDelegate protocol.
- (void)locationManager:(CLLocationManager *)manager
        didRangeBeacons:(NSArray *)beacons
               inRegion:(CLBeaconRegion *)region {

   if ([beacons count] > 0) {
```

```
        CLBeacon *nearestExhibit = [beacons firstObject];

        // Present the exhibit-specific UI only when
        // the user is relatively close to the exhibit.
        if (CLProximityNear == nearestExhibit.proximity) {
          [self presentExhibitInfoWithMajorValue:nearestExhibit.major.integerValue];
        } else {
           [self dismissExhibitInfo];
    }
}
```

To promote consistent results in your app, use beacon ranging only while your app is in the foreground. If your app is in the foreground, it is likely that the device is in the user's hand and that the device's view to the target beacon has fewer obstructions. Running in the foreground also promotes better battery life by processing incoming beacon signals only while the user is actively using the device.

> **Note:** If multiple beacon devices are advertising the same combination of proximity UUID, major values, and minor values, they may be reported by the `locationManager:didRangeBeacons:inRegion:` method as having different proximities and accuracies. It's recommended that each beacon device be uniquely identified.
>
> Additionally, if you are ranging an iOS device that has been configured as a beacon, there may be a brief period in which the `locationManager:didRangeBeacons:inRegion:` method reports two devices with the same proximity UUID, major, and minor values instead of just one. This behavior occurs because the Bluetooth identifier of an iOS device changes periodically out of privacy concerns. The `proximity` property based on the original Bluetooth identifier reports a value of `CLProximityUnknown` within 2 seconds of the identifier change. Within 10 seconds, the identifiers resolve and only one beacon region is reported.

Turning an iOS Device into an iBeacon

Any iOS device that supports sharing data using Bluetooth low energy can be used as an iBeacon. Because the app you write must run in the foreground, iBeacon support on iOS devices is intended for testing purposes and for apps that always run in the foreground anyway, such as point-of sale apps. For other types of iBeacon implementations, you need to acquire dedicated beacon hardware from third-party manufacturers.

Because turning your iOS device into a beacon requires the use of the Core Bluetooth framework, be sure to link your app to `CoreBluetooth.framework` in your Xcode project. To access the classes and headers of the framework, include an `#import <CoreBluetooth/CoreBluetooth.h>` statement at the top of any relevant source files.

Creating and Advertising a Beacon Region

To use your iOS device as a beacon, you first generate a 128-bit UUID that will be your beacon region's proximity UUID. Open Terminal and type `uuidgen` on the command line. You receive a unique 128-bit value in an ASCII string that is punctuated by hyphens, as in this example.

```
$ uuidgen
39ED98FF-2900-441A-802F-9C398FC199D2
```

Next, create a beacon region with the UUID you generated for the beacon's proximity UUID, defining the major and minor values as needed. Be sure to also use a unique string identifier for the new region. This code shows how to create a new beacon region using the example UUID above.

```
NSUUID *proximityUUID = [[NSUUID alloc]
   initWithUUIDString:@"39ED98FF-2900-441A-802F-9C398FC199D2"];

// Create the beacon region.
CLBeaconRegion *beaconRegion = [[CLBeaconRegion alloc]
   initWithProximityUUID:proximityUUID
              identifier:@"com.mycompany.myregion"
```

Now that you have created a beacon region, you need to advertise your beacon's proximity UUID (and any major or minor value you specified) using the `CBPeripheralManager` class of the Core Bluetooth framework. In Core Bluetooth, a *peripheral* is a device that advertises and shares data using Bluetooth low energy. Advertising your beacon's data is the only way other devices can detect and range your beacon.

To advertise peripheral data in Core Bluetooth, you call the `startAdvertising:` method of the `CBPeripheralManager` class on an instance of a `CBPeripheralManager` object. This method expects a dictionary (an instance of `NSDictionary`) of advertisement data. As the next example shows, use the `peripheralDataWithMeasuredPower:` method of the `CLBeaconRegion` class to receive a dictionary that encodes your beacon's identifying information along with other information needed for Core Bluetooth to advertise the beacon as a peripheral.

```
// Create a dictionary of advertisement data.
NSDictionary *beaconPeripheralData =
   [beaconRegion peripheralDataWithMeasuredPower:nil];
```

Note: The code above passes in a value of `nil` to the `measuredValue` parameter, which represents the received signal strength indicator (RSSI) value (measured in decibels) of the beacon from one meter away. This value is used during ranging. Specifying `nil` uses the default value for the device. Optionally, specify an RSSI value if you need to further calibrate the device for better ranging performance in certain environments.

Next, create an instance of the `CBPeripheralManager` class, and ask it to advertise your beacon for other devices to detect, as shown in the code below.

```
// Create the peripheral manager.
CBPeripheralManager *peripheralManager = [[CBPeripheralManager alloc]
   initWithDelegate:selfqueue:nil options:nil];

// Start advertising your beacon's data.
[peripheralManager startAdvertising:beaconPeripheralData];
```

Important: When you create a peripheral manager object, the peripheral manager calls the `peripheralManagerDidUpdateState:` method of its delegate object. You must implement this delegate method to ensure that Bluetooth low energy is supported and available to use on the local peripheral device. For more information about this delegate method, see *CBPeripheralManagerDelegate Protocol Reference*.

After advertising your app as a beacon, your app must continue running in the foreground to broadcast the needed Bluetooth signals. If the user quits the app, the system stops advertising your device as a peripheral.

For more information about how to use a peripheral manager to advertise data using Bluetooth low energy, see *CBPeripheralManager Class Reference* and *Core Bluetooth Programming Guide*.

Testing an iOS App's Region Monitoring Support

When testing your region monitoring code in iOS Simulator or on a device, realize that region events may not happen immediately after a region boundary is crossed. To prevent spurious notifications, iOS doesn't deliver region notifications until certain threshold conditions are met. Specifically, the user's location must cross the region boundary, move away from the boundary by a minimum distance, and remain at that minimum distance for at least 20 seconds before the notifications are reported.

The specific threshold distances are determined by the hardware and the location technologies that are currently available. For example, if Wi-Fi is disabled, region monitoring is significantly less accurate. However, for testing purposes, you can assume that the minimum distance is approximately 200 meters.

Getting the Heading and Course of a Device

Core Location supports two different ways to get direction-related information:

- Devices with a magnetometer can report the direction in which a device is pointing, also known as its *heading*.
- Devices with GPS hardware can report the direction in which a device is moving, also known as its *course*.

Heading and course information don't represent the same information. The heading of a device reflects the actual orientation of the device relative to true north or magnetic north. The course of the device represents the direction of travel and doesn't take into account the device orientation. Depending on your app, you might prefer one type of information over the other or use a combination of the two. For example, a navigation app might toggle between course and heading information depending on the user's current speed. At walking speeds, heading information would be more useful for orienting the user to the current environment, whereas in a car, course information provides the general direction of the car's movement.

Adding a Requirement for Direction-Related Events

If your iOS app requires direction-related information in order to function properly, include the `UIRequiredDeviceCapabilities` key in the app's `Info.plist` file. This key contains an array of strings indicating the features that your app requires of the underlying iOS-based device. The App Store uses this information to prevent users from installing apps on a device without the minimum required hardware.

For direction-related events, you can associate two relevant strings with the `UIRequiredDeviceCapabilities` key:

- `magnetometer`—Include this string if your app requires heading information.
- `gps`—Include this string if your app requires course-related information.

> **Important:** If your iOS app uses heading or course events but is able to operate successfully without them, don't include the corresponding string value with the `UIRequiredDeviceCapabilities` key.

In both cases, also include the `location-services` string in the array. For more information about the `UIRequiredDeviceCapabilities` key, see *Information Property List Key Reference*.

Getting Heading-Related Events

Heading events are available to apps running on a device that contains a magnetometer. A magnetometer measures nearby magnetic fields emanating from the Earth and uses them to determine the precise orientation of the device. Although a magnetometer can be affected by local magnetic fields, such as those emanating from fixed magnets found in audio speakers, motors, and many other types of electronic devices, Core Location is smart enough to filter out fields that move with the device.

Heading values can be reported relative either to magnetic north or true north on the map. Magnetic north represents the point on the Earth's surface from which the planet's magnetic field emanates. This location is not the same as the North Pole, which represents true north. Depending on the location of the device, magnetic north may be good enough for many purposes, but the closer to the poles you get, the less useful this value becomes.

To get heading events:

1. Create a `CLLocationManager` object.
2. Determine whether heading events are available by calling the `headingAvailable` class method.
3. Assign a delegate to the location manager object.
4. If you want true north values, start location services.
5. Call the `startUpdatingHeading` method to begin the delivery of heading events.

Listing 3-1 shows a custom method that configures a location manager and starts the delivery of heading events. In this case, the object is a view controller that displays the current heading to the user. Because the view controller displays the true north heading value, it starts location updates in addition to heading updates.

Listing 3-1 Initiating the delivery of heading events

```
- (void)startHeadingEvents {
    if (!self.locManager) {
      CLLocationManager* theManager = [[[CLLocationManager alloc] init] autorelease];

      // Retain the object in a property.
      self.locManager = theManager;
      locManager.delegate = self;
    }

    // Start location services to get the true heading.
    locManager.distanceFilter = 1000;
```

```
    locManager.desiredAccuracy = kCLLocationAccuracyKilometer;
    [locManager startUpdatingLocation];

    // Start heading updates.
    if ([CLLocationManager headingAvailable]) {
        locManager.headingFilter = 5;
        [locManager startUpdatingHeading];
    }
}
```

The object you assign to the delegate property must conform to the `CLLocationManagerDelegate` protocol. When a new heading event arrives, the location manager object calls the `locationManager:didUpdateHeading:` method to deliver that event to your app. Upon receiving a new event, check the `headingAccuracy` property to ensure that the data you just received is valid, as shown in Listing 3-2. In addition, if you are using the true heading value, also check whether it contains a valid value before using it.

Listing 3-2 Processing heading events

```
- (void)locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading
*)newHeading {
    if (newHeading.headingAccuracy < 0)
        return;

    // Use the true heading if it is valid.
    CLLocationDirection  theHeading = ((newHeading.trueHeading > 0) ?
            newHeading.trueHeading : newHeading.magneticHeading);

    self.currentHeading = theHeading;
    [self updateHeadingDisplays];
}
```

Getting Course Information While the User Is Moving

Devices that include GPS hardware can generate information that represents the device's current course and speed. Course information indicates the direction in which the device is moving and doesn't necessarily reflect the orientation of the device itself. As a result, course information is primarily intended for apps that provide navigation information while the user is moving.

The actual course and speed information is returned to your app in the same `CLLocation` objects you use to get the user's position. When you start location updates, Core Location automatically provides course and speed information when it's available. The framework uses the incoming location events to compute the current direction of motion. For more information on how to start location updates, see Getting the User's Location (page 12).

Geocoding Location Data

Location data is usually returned as a pair of numerical values that represent the latitude and longitude of the corresponding point on the globe. These coordinates offer a precise and easy way to specify location data in your code but they aren't very intuitive for users. Instead of global coordinates, users are more likely to understand a location that is specified using information they are more familiar with such as street, city, state, and country information. For situations where you want to display a user friendly version of a location, you can use a geocoder object to get that information.

About Geocoder Objects

A *geocoder object* uses a network service to convert between latitude and longitude values and a user-friendly *placemark*, which is a collection of data such as the street, city, state, and country information. *Reverse geocoding* is the process of converting a latitude and longitude into a placemark; *forward geocoding* is the process of converting place name information into latitude and longitude values. Reverse geocoding is supported in all versions of iOS, but forward geocoding is supported only in iOS 5.0 and later. Both reverse and forward geocoding are supported in OS X v10.8 and later.

> **Note:** To let users search for map locations by name, address, or type of establishment, use the `MKLocalSearch` API. To incorporate local searches into your app, see Enabling Search (page 88).

Because geocoders rely on a network service, a live network connection must be present in order for a geocoding request to succeed. If a device is in Airplane mode or the network is currently not configured, the geocoder can't connect to the service it needs and must therefore return an appropriate error. Here are some rules of thumb for creating geocoding requests:

- Send at most one geocoding request for any one user action.
- If the user performs multiple actions that involve geocoding the same location, reuse the results from the initial geocoding request instead of starting individual requests for each action.
- When you want to update the location automatically (such as when the user is moving), reissue the geocoding request only when the user's location has moved a significant distance and after a reasonable amount of time has passed. In general, you shouldn't send more than one geocoding request per minute.
- Don't start a geocoding request if the user won't see the results immediately. For example, don't start a request if your app is in the background or was interrupted and is currently in the inactive state.

Getting Placemark Information Using CLGeocoder

To initiate a reverse-geocoding request using the `CLGeocoder` class, create an instance of the class and call the `reverseGeocodeLocation:completionHandler:` method. The geocoder object initiates the reverse geocoding request asynchronously and delivers the results to the block object you provide. The block object is executed whether the request succeeds or fails. In the event of a failure, an error object is passed to the block indicating the reason for the failure.

> **Note:** The same `CLGeocoder` object can be used to initiate any number of geocoding requests but only one request at a time may be active for a given geocoder.

Listing 4-1 shows an example of how to reverse geocode a point on the map. The only code specific to the geocoding request are the first few lines, which allocate the geocoder object as needed and call the `reverseGeocodeLocation:completionHandler:` method to start the reverse-geocoding operation. (The `geocoder` variable represents a member variable used to store the geocoder object.) The rest of the code is specific to the sample app itself. In this case, the sample app stores the placemark with a custom annotation object (defined by the `MapLocation` class) and adds a button to the callout of the corresponding annotation view.

Listing 4-1 Geocoding a location using `CLGeocoder`

```
@implementation MyGeocoderViewController (CustomGeocodingAdditions)
- (void)geocodeLocation:(CLLocation*)location forAnnotation:(MapLocation*)annotation
{
    if (!geocoder)
        geocoder = [[CLGeocoder alloc] init];

    [geocoder reverseGeocodeLocation:location completionHandler:
        ^(NSArray* placemarks, NSError* error){
            if ([placemarks count] > 0)
            {
                annotation.placemark = [placemarks objectAtIndex:0];

                // Add a More Info button to the annotation's view.
                MKPinAnnotationView* view = (MKPinAnnotationView*)[map
viewForAnnotation:annotation];
                if (view && (view.rightCalloutAccessoryView == nil))
                {
```

```
                    view.canShowCallout = YES;
                    view.rightCalloutAccessoryView = [UIButton
buttonWithType:UIButtonTypeDetailDisclosure];
                }
            }
    }];
}
@end
```

The advantage of using a block object in a sample like this is that information (such as the annotation object) can be easily captured and used as part of the completion handler. Without blocks, the process of wrangling data variables becomes much more complicated.

Converting Place Names Into Coordinates

Use the `CLGeocoder` class with a dictionary of Address Book information or a simple string to initiate forward-geocoding requests. There is no designated format for string-based requests: Delimiter characters are welcome, but not required, and the geocoder server treats the string as case-insensitive. For example, any of the following strings would yield results:

- "Apple Inc"
- "1 Infinite Loop"
- "1 Infinite Loop, Cupertino, CA USA"

The more information you can provide to the forward geocoder, the better the results returned to you. The geocoder object parses the information you give it and if it finds a match, returns some number of placemark objects. The number of returned placemark objects depends greatly on the specificity of the information you provide. For this reason, providing street, city, province, and country information is much more likely to return a single address than providing only street and city information. The completion handler block you pass to the geocoder should be prepared to handle multiple placemarks, as shown below.

```
[geocoder geocodeAddressString:@"1 Infinite Loop"
     completionHandler:^(NSArray* placemarks, NSError* error){
         for (CLPlacemark* aPlacemark in placemarks)
         {
             // Process the placemark.
```

```
        }
}];
```

Displaying Maps

Objective-CSwift

The Map Kit framework lets you embed a fully functional map interface into your app. The map support provided by this framework includes many features of the Maps app in both iOS and OS X. You can display standard street-level map information, satellite imagery, or a combination of the two. You can zoom, pan, and pitch the map programmatically, display 3D buildings, and annotate the map with custom information. The Map Kit framework also provides automatic support for the touch events that let users zoom and pan the map.

To use the features of the Map Kit framework, turn on the Maps capability in your Xcode project (doing so also adds the appropriate entitlement to your App ID). Note that the only way to distribute a maps-based app is through the iOS App Store or Mac App Store. If you're unfamiliar with entitlements, code signing, and provisioning, start learning about them in *App Distribution Quick Start*. For general information about the classes of the Map Kit framework, see *MapKit Framework Reference*.

> **Important:** In iOS 5.1 or earlier, the Map Kit framework uses Google services to provide map data. Use of the framework and its associated interfaces binds you to the Google Maps/Google Earth API terms of service. You can find these terms of service at http://code.google.com/apis/maps/iphone/terms.html.

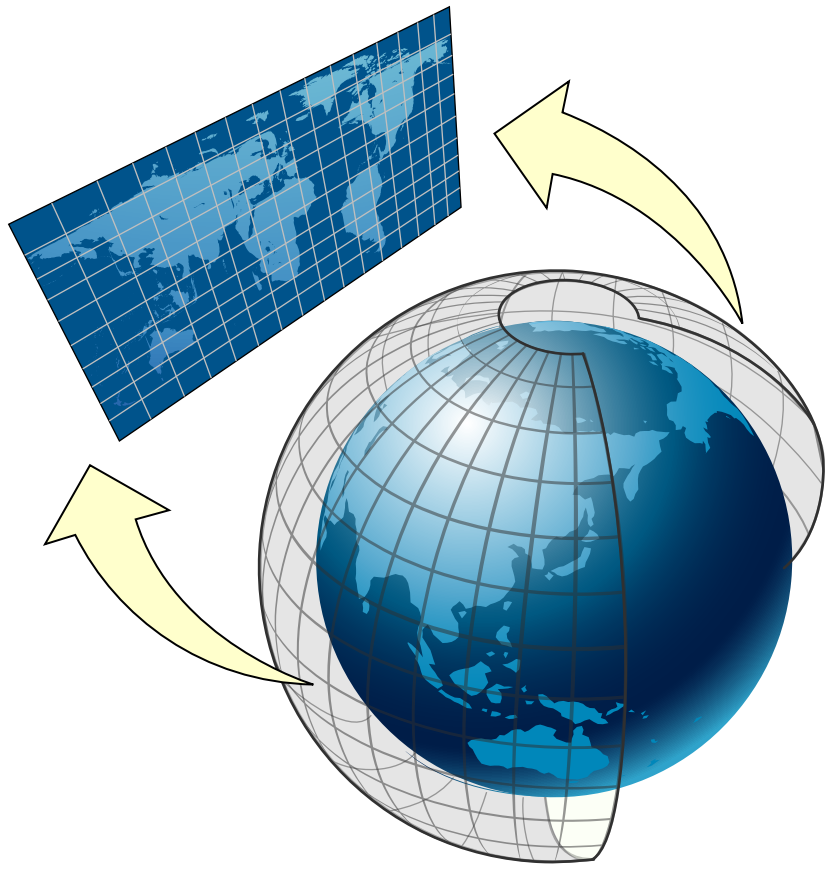
Understanding Map Geometry

A map view contains a flattened representation of a spherical object, namely the Earth. To use maps effectively, you need to understand how to specify points in a map view and how those points translate to points on the Earth's surface. Understanding map coordinate systems is especially important if you plan to place custom content, such as overlays, on top of the map.

Map Coordinate Systems

To understand the coordinate systems used by Map Kit, it helps to understand how the three-dimensional surface of the Earth is mapped to a two-dimensional map. Figure 5-1 shows how the surface of the Earth can be mapped to a two-dimensional surface.

Figure 5-1 Mapping spherical data to a flat surface



Map Kit uses a Mercator map projection, which is a specific type of cylindrical map projection like the one shown in Figure 5-1. In a cylindrical map projection, the coordinates of a sphere are mapped onto the surface of a cylinder, which is then unwrapped to generate a flat map. In such a projection, the longitude lines that normally converge at the poles become parallel instead, causing land masses to be distorted as you move away from the equator. The advantage of a Mercator projection is that the map content is scaled in a way that benefits general navigation. Specifically, on a Mercator map projection, a straight line drawn between any two points on the map yields a course heading that can be used in actual navigation on the surface of the Earth. The projection used by Map Kit uses the Prime Meridian as its central meridian.

How you specify data points on a map depends on how you intend to use them. Map Kit supports three basic coordinate systems for specifying map data points:

- A *map coordinate* is a latitude and longitude on the spherical representation of the Earth. Map coordinates are the primary way of specifying locations on the globe. You specify individual map coordinate values using the `CLLocationCoordinate2D` structure. You can specify areas using the `MKCoordinateSpan` and `MKCoordinateRegion` structures.
- A *map point* is an x and y value on the Mercator map projection. Map points are used for many map-related calculations instead of map coordinates because they simplify the mathematics involved in the calculations. In your app, you use map points primarily when specifying the shape and position of custom map overlays. You specify individual map points using the `MKMapPoint` structure. You can specify areas using the `MKMapSize` and `MKMapRect` structures.
- A *point* is a graphical unit associated with the coordinate system of a view object. Map points and map coordinates must be mapped to points before drawing custom content in a view. You specify individual points using the `CGPoint` structure. You can specify areas using the `CGSize` and `CGRect` structures.

In most situations, the coordinate system you should use is predetermined by the Map Kit interfaces you are using. When it comes to storing actual data in files or inside your app, map coordinates are precise, portable, and the best option for storing location data. Core Location also uses map coordinates when specifying location values.

Converting Between Coordinate Systems

Although you typically specify points on the map using latitude and longitude values, there may be times when you need to convert to and from other coordinate systems. For example, you usually use map points when specifying the shape of overlays. Table 5-1 lists the conversion routines you use to convert from one coordinate system to another. Most of these conversions require a view object because they involve converting to or from points.

Table 5-1 Map coordinate system conversion routines

Convert from	Convert to	Conversion routines
Map coordinates	Points	`convertCoordinate: toPointToView:` (`MKMapView`) `convertRegion: toRectToView:` (`MKMapView`)
Map coordinates	Map points	`MKMapPointForCoordinate`
Map points	Map coordinates	`MKCoordinateForMapPoint` `MKCoordinateRegionForMapRect`
Map points	Points	`pointForMapPoint:` (`MKOverlayRenderer`) `rectForMapRect:` (`MKOverlayRenderer`)

Convert from	Convert to	Conversion routines
Points	Map coordinates	`convertPoint: toCoordinateFromView:` (`MKMapView`) `convertRect: toRegionFromView:` (`MKMapView`)
Points	Map points	`mapPointForPoint:` (`MKOverlayRenderer`) `mapRectForRect:` (`MKOverlayRenderer`)

Adding a Map View to Your User Interface

The `MKMapView` class is a self-contained interface for presenting map data in your app: It provides support for displaying map data, managing user interactions, and hosting custom content provided by your app. Never subclass `MKMapView`. Instead, embed it as-is into your app's view hierarchy.

Also assign a delegate object to the map. The map view reports all relevant interactions to its delegate so that the delegate has a chance to respond appropriately.

You can add a map view to your app programmatically or using Interface Builder:

- To add a map using Interface Builder, drag a Map view object to the appropriate view or window.
- To add a map programmatically, create an instance of the `MKMapView` class, initialize it using the `initWithFrame:` method, and then add it as a subview to your window or view hierarchy.

Because a map view is a view, you can manipulate it the same way you manipulate other views. You can change its size and position in your view hierarchy, configure its autoresizing behaviors, and add subviews to it. The map view itself is an opaque container for a complex view hierarchy that handles the display of map-related data and all interactions with that data. Any subviews you add to the map view retain the position specified by their `frame` property and don't scroll with the map contents. If you want content to remain fixed relative to a specific map coordinate (and thus scroll with the map itself), you must use annotations or overlays as described in Annotating Maps (page 54). It's best to avoid any modification of a map view's hierarchy.

New maps are configured to accept user interactions and display map data only. By default, a standard map uses a 3D perspective by enabling pitch, which tilts the map, and rotation, which lets the map display a heading. You can specify pitch and rotation by creating an `MKMapCamera` object. You can configure the map to display satellite imagery or a mixture of satellite and map data by changing the Type attribute of the map in Interface Builder or by changing the value in the `mapType` property. If you want to limit user interactions, you can change the values in the `rotateEnabled`, `pitchEnabled`, `zoomEnabled`, and `scrollEnabled` properties as well. If you want to respond to user interactions, use a delegate as described in Using the Delegate to Respond to User Interactions (page 52).

Configuring the Properties of a Map

The `MKMapView` class has several properties that you can configure programmatically. These properties control important information such as which part of the map is currently visible, whether the content is displayed in 3D, and what user interactions are allowed.

Setting the Visible Portion of the Map

The `region` property of the `MKMapView` class controls the currently visible portion of the map. When a map is first created, its visible region is typically set to the entire world. In other words, the region encompasses the area that shows as much of the map as possible. You can change this region by assigning a new value to the `region` property. This property contains an `MKCoordinateRegion` structure, which has the definition shown below.

```
typedef struct {
   CLLocationCoordinate2D center;
   MKCoordinateSpan span;
} MKCoordinateRegion;
```

The interesting part of an `MKCoordinateRegion` structure is the span. The **span** defines how much of the map at a given point should be visible. Although the span is analogous to the width and height values of a rectangle, it's specified using map coordinates and thus is measured in degrees, minutes, and seconds. One degree of latitude is equivalent to approximately 111 kilometers, but longitudinal distances vary with the latitude. At the equator, one degree of longitude is equivalent to approximately 111 kilometers, but at the poles this value is zero. If you prefer to specify the span in meters, use the `MKCoordinateRegionMakeWithDistance` to create a region data structure with meter values instead of degrees.

The value you assign to the `region` property (or set using the `setRegion:animated:` method) is usually not the same value that is eventually stored by that property. Setting the span of a region nominally defines the rectangle you want to view but also implicitly sets the zoom level for the map view itself. The map view can't display arbitrary zoom levels and must adjust any regions you specify to match the zoom levels it supports. It chooses the zoom level that allows your entire region to be visible while still filling as much of the screen as possible. It then adjusts the `region` property accordingly. To find out the resulting region without actually changing the value in the `region` property, use the `regionThatFits:` method of the map view.

> **Note:** In iOS 7 and OS X v10.9 and later, the `region` and `visibleMapRect` properties can refer to an area that spans the 180th meridian. If you use the value of either property in calculations, be prepared to receive a structure that includes a center (or origin) of 0 latitude and 180 longitude.

Displaying a 3D Map

A 3D map is a standard 2D map viewed at an angle from a vantage point above the map's plane. The point's altitude, combined with the angle from which the map is viewed, determine the span and tilt (also known as **pitch**) of the 2D map surface. Users can adjust the pitch and rotation of a map and, in iOS 7 and OS X v10.9 and later, you can use the `MKMapCamera` class to programmatically adjust a 3D map.

> **Note:** Always check the value of the `pitchEnabled` property to determine whether a map can support 3D.

A camera object uses the following properties to define the appearance of a 3D map:

- Altitude. The camera's height (in meters) above the surface of the map.
- Pitch. The angle at which the camera tilts, relative to the ground. (Note that a pitch of 0 produces a standard 2D map because the camera is looking straight down.)
- Heading. The cardinal direction in which the camera is facing.
- Center. The point on the map surface that appears in the center of the screen or window.

In iOS 7 and OS X v10.9 and later, maps are 3D by default, which can affect your app in the following ways:

- Because a pitched map can expose the sky, users can see areas that are beyond the boundaries of the map. Be sure to check the validity of values returned by the map view's conversion methods (such as `convertPoint:toCoordinateFromView:`) so your app doesn't try to place annotations in the sky.
- If Map Kit detects a nonsensical pitch value, such as 180 degrees (that is, looking straight up at the sky), it clamps the pitch to a reasonable value.
- Often, the visible area of a 3D map is not rectangular when it's viewed in two dimensions. In this scenario, the `region` and `visibleMapRect` properties specify a rectangular area that contains a 2D approximation of the pitched map's visible area.
- Annotations automatically maintain their size and orientation even as the map rotates or pitches, so your artwork won't get distorted or resized. (Learn more about working with annotations in Annotating Maps (page 54).)

For the most part, 3D maps work the same in iOS and OS X apps. The few differences between the platforms are primarily in the user interface and in some underlying object types: A 3D map in an OS X app displays compass and zoom controls and a map data attribution label; a 3D map in an iOS app doesn't display these items. In OS X, Map Kit defines objects that inherit from `NSView` and `NSImage`; in iOS, the analogous objects inherit from `UIView` and `UIImage`.

In iOS 7 and OS X v10.9 and later, the `MKMapView` class includes a `camera` property you can use to create and access a 3D map, save and restore map state, and programmatically zoom and pan. For example, you can easily create a 3D map of a location by specifying a position and altitude from which to view the location, asking Map Kit to create an appropriate camera object, and assigning the object to your map view's `camera` property. Listing 5-1 shows how to do this.

Listing 5-1 Creating a 3D map

```
// Create a coordinate structure for the location.
CLLocationCoordinate2D ground = CLLocationCoordinate2DMake(myLatitude, myLongitude);


// Create a coordinate structure for the point on the ground from which to view
the location.
CLLocationCoordinate2D eye = CLLocationCoordinate2DMake(eyeLatitude, eyeLongitude);


// Ask Map Kit for a camera that looks at the location from an altitude of 100
meters above the eye coordinates.
MKMapCamera *myCamera = [MKMapCamera cameraLookingAtCenterCoordinate:ground
fromEyeCoordinate:eye eyeAltitude:100];


// Assign the camera to your map view.
mapView.camera = myCamera;
```

Because a camera object fully defines the appearance of a map, it's a good idea to use it to save and restore your map's state. The `MKMapCamera` class conforms to the `NSSecureCoding` protocol, so you can use a camera object with an archiver or, in an iOS app, the UIKit state restoration APIs. Listing 5-2 shows an example of saving and restoring a map's state.

Listing 5-2 Archiving and unarchiving a map

```
MKMapCamera *camera = [map camera]; // Get the map's current camera.
[NSKeyedArchiver archiveRootObject:camera toFile:stateFile]; // Archive the camera.
…
```

```
MKMapCamera *camera = [NSKeyedUnarchiver unarchiveObjectWithFile:stateFile]; //
Unarchive the camera.
[map setCamera:camera]; // Restore the map.
```

Zooming and Panning the Map Content

Zooming and panning allow you to change the visible portion of the map at any time:

- To pan the map (but keep the same zoom level, pitch, and rotation), change the value in the `centerCoordinate` property of the map view or the camera, or call the map view's `setCenterCoordinate:animated:` or `setCamera:animated:` methods.
- To change the zoom level (and optionally pan the map), change the value in the `region` property of the map view or call the `setRegion:animated:` method. You can also vary the altitude of the camera in a 3D map (doubling or halving the altitude is approximately the same as zooming in or out by one level).

If you only want to pan the map, you should do so by modifying only the `centerCoordinate` property. Attempting to pan the map by changing the `region` property usually causes a change in the zoom level as well, because changing any part of the region causes the map view to evaluate the zoom level needed to display that region appropriately. Changes to the current latitude almost always cause the zoom level to change and other changes might cause a different zoom level to be chosen as well. Using the `centerCoordinate` property (or the `setCenterCoordinate:animated:` method) lets the map view know that it should leave the zoom level unchanged and update the span as needed. For example, this code pans the map to the left by half the current map width by finding the coordinate at the left edge of the map and using it as the new center point.

```
CLLocationCoordinate2D mapCenter = myMapView.centerCoordinate;
mapCenter = [myMapView convertPoint:
               CGPointMake(1, (myMapView.frame.size.height/2.0))
               toCoordinateFromView:myMapView];
[myMapView setCenterCoordinate:mapCenter animated:YES];
```

To zoom the map, modify the span of the visible map region. To zoom in, assign a smaller value to the span. To zoom out, assign a larger value. For example, as shown here, if the current span is one degree, specifying a span of two degrees zooms out by a factor of two.

```
MKCoordinateRegion theRegion = myMapView.region;


// Zoom out
```

```
theRegion.span.longitudeDelta *= 2.0;
theRegion.span.latitudeDelta *= 2.0;
[myMapView setRegion:theRegion animated:YES];
```

Displaying the User's Current Location on the Map

Map Kit includes built-in support for displaying the user's current location on the map. To show this location, set the `showsUserLocation` property of your map view object to `YES`. Doing so causes the map view to use Core Location to find the user's location and add an annotation of type `MKUserLocation` to the map.

The addition of the `MKUserLocation` annotation object to the map is reported by the delegate in the same way that custom annotations are. If you want to associate a custom annotation view with the user's location, you should return that view from your delegate object's `mapView:viewForAnnotation:` method. If you want to use the default annotation view, return `nil` from that method. To learn more about adding annotations to a map, see Annotating Maps (page 54).

Creating a Snapshot of a Map

In some cases, it doesn't make sense to add a fully interactive map view to your app. For example, if your app lets users choose a location from a scrolling list of map images, enabling interaction with each map is unnecessary and may impair scrolling performance. Another reason to create a static image of a map is to implement a printing feature. In both situations, you can use a `MKMapSnapshotter` object to create a static map image asynchronously. The resulting snapshot contains an image view, to which you can apply all the effects that you apply to other images in your app.

> **Note:** If you're currently using `renderInContext:` to create a snapshot of a map, use the `MKMapSnapshotter` API instead.

In general, follow these steps to create a map snapshot:

1. Make sure you have a network connection and that your app is in the foreground.
2. Create and configure an `MKMapSnapshotOptions` object, which specifies the appearance of the map and the size of the output. (An iOS app can also specify a scale for the output.)
3. Create an `MKMapSnapshotter` object and initialize it with the options you specified in step 1.
4. Call `startWithCompletionHandler:` to start the asynchronous snapshot-creation task.
5. When the task completes, retrieve the map snapshot from your completion handler block and draw any overlays or annotations that should appear in the final image.

> **Note:** Make sure your app responds appropriately if snapshot generation fails, perhaps by displaying a placeholder image.

In an iOS app, you often use `drawContentForPageAtIndex:inRect:` to implement printing. Because this method expects to receive the printable content synchronously, you have to modify the snapshot-creation steps to include the use of a dispatch semaphore and queue that help you block on the completion of the snapshot. (To learn more about dispatch semaphores and queues, see Using Dispatch Semaphores to Regulate the Use of Finite Resources.)

To create a map snapshot for printing in an iOS app:

1. Create and configure an `MKMapSnapshotOptions` object. (Note that iOS apps generally specify a scale of 2 because most printers are high-resolution.)
2. Create an `MKMapSnapshotter` object and initialize it with the options you specified in step 1.
3. Create a dispatch semaphore that allows you to wait for a resource—in this case, the snapshot—to become available.
4. Choose a dispatch queue on which to receive a callback when the snapshot is ready.
5. Create variables to hold the results of the snapshot-creation task.
6. Call `startWithQueue:completionHandler:` to start generating the snapshot asynchronously.
7. When the task completes, pass the snapshot's image to `drawContentForPageAtIndex:inRect:` for printing.

The code in Listing 5-3 implements most of the steps for printing a map snapshot. The code doesn't show the creation of the `MKMapSnapshotOptions` object.

Listing 5-3 Creating a map snapshot for printing

```
// Initialize the semaphore to 0 because there are no resources yet.
dispatch_semaphore_t snapshotSem = dispatch_semaphore_create(0);


// Get a global queue (it doesn't matter which one).
dispatch_queue_t queue = dispatch_get_global_queue(myQueuePriorityLevel, 0);


// Create variables to hold return values. Use the __block modifier because these
 variables will be modified inside a block.
__block MKMapSnapshot *mapSnapshot = nil;
__block NSError *error = nil;
```

```
// Start the asynchronous snapshot-creation task.
[snapshotter startWithQueue:queue
          completionHandler:^(MKMapSnapshot *snapshot, NSError *e) {
    mapSnapshot = snapshot;
error = e;
    // The dispatch_semaphore_signal function tells the semaphore that the async
task is finished, which unblocks the main thread.
    dispatch_semaphore_signal(snapshotSem);
}];

// On the main thread, use dispatch_semaphore_wait to wait for the snapshot task
to complete.
dispatch_semaphore_wait(snapshotSem, DISPATCH_TIME_FOREVER);
if (error) { // Handle error. }

// Get the image from the newly created snapshot.
UIImage *image = mapSnapshot.image;
// Optionally, draw annotations on the image before displaying it.
```

Using the Delegate to Respond to User Interactions

The `MKMapView` class reports significant map-related events to its associated delegate object. The delegate object is an object that conforms to the `MKMapViewDelegate` protocol. Providing a delegate object helps you respond to the following types of events:

- Changes to the visible region of the map
- The loading of map tiles from the network
- Changes in the user's location
- Changes associated with annotations and overlays

For information about handling changes associated with annotations and overlays, see Annotating Maps (page 54).

Launching the Maps App

If you would prefer to display map information in the Maps app as opposed to your own app, you can launch Maps programmatically using one of two techniques:

- In iOS 6 and and OS X v10.9 and later, use an `MKMapItem` object to open Maps.
- In iOS 5 and earlier, create and open a specially formatted map URL as described in *Apple URL Scheme Reference*.

The preferred way to open the Maps app is to use the `MKMapItem` class. This class offers both the `openMapsWithItems:launchOptions:` class method and the `openInMapsWithLaunchOptions:` instance method for opening the app and displaying locations or directions.

For an example showing how to open the Maps app, see Asking the Maps App to Display Directions (page 78).

Annotating Maps

Objective-CSwift

Annotations display content that can be defined by a single coordinate point; *overlays* display content that is defined by any number of points and may constitute one or more contiguous or noncontiguous shapes. For example, you use annotations to represent information such as the user's current location, a specific address, or a single point of interest. You use overlays to present more complex information such as routes or traffic information, or the boundaries of areas such as parks, lakes, cities, states, or countries.

Unlike generic subviews, annotations and overlays remain fixed to the map so that they move appropriately when the user zooms, pans, or scrolls. In a 3D map, annotations maintain their position, size, and orientation regardless of how the map rotates or pitches.

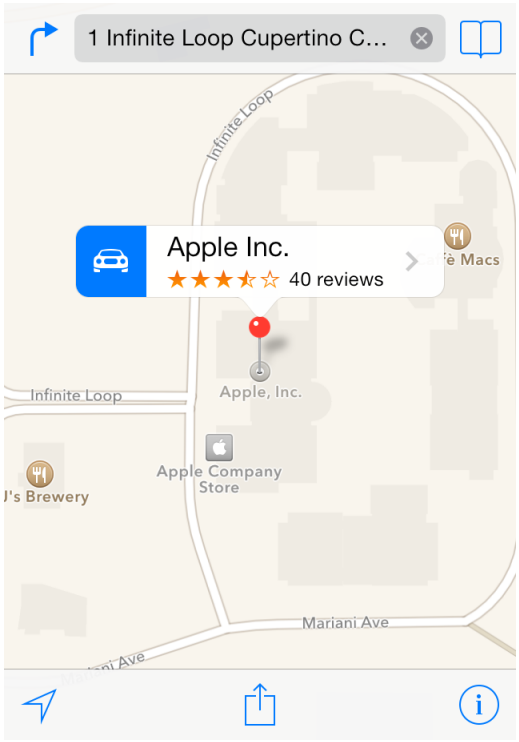
Map Kit separates the data associated with an annotation or overlay from its visual presentation on the map. This separation allows the map to manage visible annotations and overlays much more efficiently: You can add hundreds of annotations and overlays to a map and still expect reasonable performance.

Adding Annotations to a Map

Annotations offer a way to highlight specific coordinates on the map and provide additional information about them. You can use annotations to call out particular addresses, points of interest, and other types of destinations. When displayed on a map, annotations typically have some sort of image to identify their location and may

also have a callout bubble providing information and links to more content. Figure 6-1 shows an annotation that uses a standard pin annotation view to mark a particular location and provides a callout bubble that displays additional information including a disclosure indicator that leads to more details.

Figure 6-1 Displaying an annotation in a map



To display an annotation on a map, your app must provide two distinct objects:

- An *annotation object*, which is an object that conforms to the `MKAnnotation` protocol and manages the data for the annotation.
- An *annotation view*, which is a view (derived from the `MKAnnotationView` class) used to draw the visual representation of the annotation on the map surface.

Annotation objects are typically small data objects that store the map coordinate data and any other relevant information about the annotation, such as a title string. Because annotations are defined using a protocol, you can turn any class in your app into an annotation object. In practice, it's good to keep annotation objects lightweight, especially if you intend to add large numbers of them to the map. The map view keeps a reference to the annotation objects you add to it and uses the data in those objects to determine when to display the corresponding view.

Map Kit provides some standard annotation views, such as the pin annotation, and you can also define custom annotation views. Whether you use standard or custom annotation views, you don't add them directly to the map surface. Instead, the map delegate provides an annotation view when the map view asks for one and the map view incorporates the annotation into its view hierarchy.

The annotations you create are typically anchored to a single map coordinate that doesn't change, but you can modify an annotation's coordinate programmatically if needed. In addition, you can implement support to allow users to drag annotations around the map.

Steps for Adding an Annotation to the Map

The steps for implementing and using annotations in your map-based app are shown below. It's assumed that your app incorporates an `MKMapView` object somewhere in its interface.

1. Define an appropriate annotation object using one of the following options:
   - Use the `MKPointAnnotation` class to implement a simple annotation. This type of annotation contains properties for specifying the title and subtitle strings to display in the annotation's callout bubble.
   - Define a custom object that conforms to the `MKAnnotation` protocol, as described in Defining a Custom Annotation Object (page 57). A custom annotation can store any type of data you want.
2. Define an annotation view to present the annotation data on screen. How you define your annotation view depends on your needs and may be one of the following:
   - If you want to use a standard pin annotation, create an instance of the `MKPinAnnotationView` class; see Using the Standard Annotation Views (page 58).
   - If the annotation can be represented by a custom static image, create an instance of the `MKAnnotationView` class and assign the image to its `image` property; see Using the Standard Annotation Views (page 58).
   - If a static image is insufficient for representing your annotation, subclass `MKAnnotationView` and implement the custom drawing code needed to present it. For information about how to implement custom annotation views, see Defining a Custom Annotation View (page 59).
3. Implement the `mapView:viewForAnnotation:` method in your map view delegate.

   In your implementation of this method, dequeue an existing annotation view if one exists; if not, create a new annotation view. If your app supports multiple types of annotations, include logic in this method to create a view of the appropriate type for the provided annotation object. For more information about implementing the `mapView:viewForAnnotation:` method, see Creating Annotation Views from Your Delegate Object (page 60).
4. Add your annotation object to the map view using the `addAnnotation:` (or `addAnnotations:`) method.

When you add an annotation to a map view, the map view displays the corresponding annotation view whenever the coordinate for the annotation is in the visible map rectangle. If you want to hide annotations selectively, you must manually remove them from the map view yourself. You can add and remove annotations at any time.

All annotations are drawn at the same scale every time, regardless of the map's current zoom level. If your map contains many annotations, your annotation views could overlap each other as the user zooms out. To counter this behavior, you can add and remove annotations based on the map's current zoom level. For example, a weather app might display information only for major cities when the map is zoomed out to show the entire state. As the user zooms in, the app could then add new annotations containing weather information for smaller cities and regions. Implementing the logic necessary to add and remove annotations is your responsibility.

For more information about how to manage the annotations of a map view effectively, see Displaying Multiple Annotation Objects (page 66).

Defining a Custom Annotation Object

If all you want to do is associate a title with a map coordinate, you can use the `MKPointAnnotation` class for your annotation object. However, if you want to represent additional information with the annotation, you need to define a custom annotation object. All annotation objects must conform to the `MKAnnotation` protocol.

A custom annotation object consists of a map coordinate and whatever other data you want to associate with the annotation. Listing 6-1 shows the minimal code needed to declare a custom annotation class. The `coordinate` property declaration is from the `MKAnnotation` protocol and must be included in all annotation classes. Because this is a simple annotation, it also includes an initializer method, which is used to set the value of the read-only `coordinate` property. Your own declaration would likely also include methods and properties that define additional annotation data.

Listing 6-1 Creating a simple annotation object

```
@interface MyCustomAnnotation : NSObject <MKAnnotation> {
    CLLocationCoordinate2D coordinate;
}
@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
- (id)initWithLocation:(CLLocationCoordinate2D)coord;

// Other methods and properties.
@end
```

Your custom class must implement the coordinate property and a way to set its value. (It's recommended that you synthesize `coordinate` because it ensures that Map Kit can automatically update the map based on changes to the property.) All that remains is to implement the custom `initWithLocation:` method, which is shown in Listing 6-2.

Listing 6-2 Implementing the MyCustomAnnotation class

```
@implementation MyCustomAnnotation
@synthesize coordinate;

- (id)initWithLocation:(CLLocationCoordinate2D)coord {
    self = [super init];
    if (self) {
        coordinate = coord;
    }
    return self;
}
@end
```

Important: If you choose to implement the methods for this property yourself, or if you manually modify the variable underlying the property in other parts of your class after the annotation has been added to the map, be sure to send out notifications when you do. Map Kit uses key-value observing (KVO) notifications to detect changes to the `coordinate`, `title`, and `subtitle` properties of your annotations and to make any needed changes to the map display. If you don't send out KVO notifications, the position of your annotations may not be updated properly on the map.

For more information about how to implement KVO-compliant accessor methods, see *Key-Value Observing Programming Guide*.

For more examples of annotation objects, see the sample code project *MapCallouts*.

Using the Standard Annotation Views

The standard annotation views make it easy to present annotations on your map. The `MKAnnotationView` class defines the basic behavior for all annotation views. For example, the `MKPinAnnotationView` subclass of `MKAnnotationView` displays one of the standard system pin images at the associated annotation's coordinate point. You can also use `MKAnnotationView` to display a custom static image without subclassing.

To display a custom image as an annotation, create an instance of `MKAnnotationView` and assign the custom image to the object's `image` property. When the annotation is displayed, your custom image appears, centered over the target map coordinate. If you don't want the image to be centered on the map coordinate, you can use the `centerOffset` property to move the center point in any direction. Listing 6-3 shows an example of creating an annotation view with a custom image that should be offset to the right and above the annotation coordinate.

Listing 6-3 Creating a standard annotation view with a custom image

```
MKAnnotationView* aView = [[MKAnnotationView alloc] initWithAnnotation:annotation
                                    reuseIdentifier:@"MyCustomAnnotation"];
aView.image = [UIImage imageNamed:@"myimage.png"];
aView.centerOffset = CGPointMake(10, -20);
```

You create the standard annotation views in your delegate's `mapView:viewForAnnotation:` method. For more information about how to implement this method, see Creating Annotation Views from Your Delegate Object (page 60).

Defining a Custom Annotation View

If a static image is insufficient for representing your annotation, you can subclass `MKAnnotationView` and draw content dynamically in one of the following two ways:

- Continue to use the `image` property of `MKAnnotationView`, but change the image at regular intervals.
- Override the annotation view's `drawRect:` method and draw your content dynamically every time.

As with any custom drawing you do in a view, always consider performance before choosing an approach. Although custom drawing gives you the most flexibility, using images can be faster, especially if most of your content is fixed.

When you use the `drawRect:` method to draw content, specify a nonzero frame size for the annotation view shortly after initialization to ensure that your rendered content becomes visible. Specifying a nonzero frame size is necessary because the default initialization method for annotation views uses the image specified in the `image` property to set the frame size later. If you draw the image instead of setting the property, you must set the `frame` property of the view explicitly or your rendered content won't be visible. And because the view draws in only part of its frame, set its `opaque` property to `NO` so that the remaining map content shows through. If you don't set the `opaque` property, the drawing system fills your view with the current background color before calling the `drawRect:` method. Listing 6-4 shows an example initialization method that sets the frame size and opacity of a custom annotation view.

Listing 6-4 Initializing a custom annotation view

```
- (id)initWithAnnotation:(id <MKAnnotation>)annotation reuseIdentifier:(NSString
*)reuseIdentifier
{
    self = [super initWithAnnotation:annotation reuseIdentifier:reuseIdentifier];
    if (self)
    {
        // Set the frame size to the appropriate values.
        CGRect  myFrame = self.frame;
        myFrame.size.width = 40;
        myFrame.size.height = 40;
        self.frame = myFrame;

        // The opaque property is YES by default. Setting it to
       // NO allows map content to show through any unrendered parts of your view.
        self.opaque = NO;
    }
    return self;
}
```

In most respects, drawing custom content in an annotation view is the same as it is in any view. The system calls your view's `drawRect:` method to redraw portions of the view that need it, and you can force a redraw operation by calling the `setNeedsDisplay` or `setNeedsDisplayInRect:` method of your view at any time. If you want to animate the contents of your view, set up a timer to fire at periodic intervals and update your view. To set up timers, see *Timer Programming Topics*. To learn more about how views draw content, see *View Programming Guide for iOS* or *Cocoa Drawing Guide* (for OS X information).

Creating Annotation Views from Your Delegate Object

When the map view needs an annotation view, it calls the `mapView:viewForAnnotation:` method of its delegate object. If you don't implement this method—or if you implement it and always return `nil`—the map view uses a default annotation view, which is typically a pin annotation view. If you want to return annotation views other than the default ones, you need to override `mapView:viewForAnnotation:` and create your views there.

Before creating a new view in the `mapView:viewForAnnotation:` method, always check to see if a similar annotation view already exists. The map view has the option of caching unused annotation views that it isn't using, so an unused view may be available from the `dequeueReusableAnnotationViewWithIdentifier:` method. If the dequeuing method returns a value other than `nil`, update the view's attributes and return it; if the method returns `nil`, create a new instance of the appropriate annotation view class. In both cases, it's your responsibility to take the annotation passed to this method and assign it to your annotation view. Also use `mapView:viewForAnnotation:` to update the view before returning it.

Listing 6-5 shows an example implementation of the `mapView:viewForAnnotation:` method that provides pin annotation views for custom annotation objects. If an existing pin annotation view already exists, this method associates the annotation object with that view. If no view is in the reuse queue, this method creates a new one and sets up its basic properties. If the map is currently showing the user's location, this method returns `nil` for any `MKUserLocation` objects so that the map uses the default annotation view. (You would also customize the callout in the `mapView:viewForAnnotation:` method; see Creating Callouts (page 62).)

Listing 6-5 Creating annotation views

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView
                      viewForAnnotation:(id <MKAnnotation>)annotation
{
    // If the annotation is the user location, just return nil.
    if ([annotation isKindOfClass:[MKUserLocation class]])
        return nil;

    // Handle any custom annotations.
    if ([annotation isKindOfClass:[MyCustomAnnotation class]])
    {
        // Try to dequeue an existing pin view first.
        MKPinAnnotationView*    pinView = (MKPinAnnotationView*)[mapView
        dequeueReusableAnnotationViewWithIdentifier:@"CustomPinAnnotationView"];

        if (!pinView)
        {
            // If an existing pin view was not available, create one.
            pinView = [[MKPinAnnotationView alloc] initWithAnnotation:annotation
                       reuseIdentifier:@"CustomPinAnnotationView"];
            pinView.pinColor = MKPinAnnotationColorRed;
            pinView.animatesDrop = YES;
```

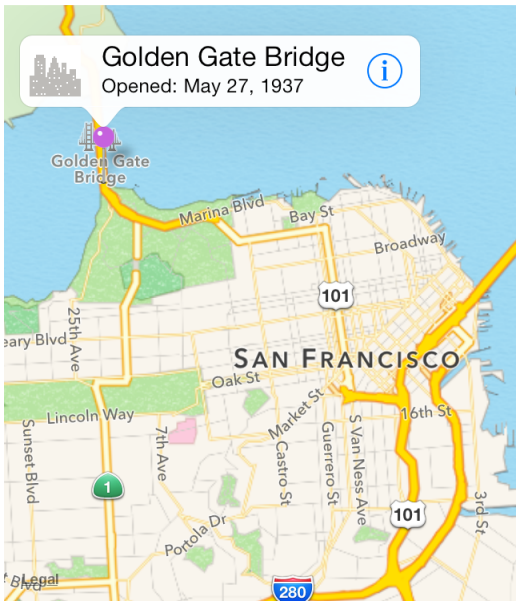
```
            pinView.canShowCallout = YES;

            // If appropriate, customize the callout by adding accessory views
(code not shown).
        }
        else
            pinView.annotation = annotation;

        return pinView;
    }

    return nil;
}
```

Creating Callouts

A callout is a standard or custom view that can appear with an annotation view. A standard callout displays the annotation's title, and it can display additional content such as a subtitle, images, and a control. If you want to display a custom view that behaves like a callout, add a custom subview to the annotation view and update the annotation view's hit testing method to handle user interaction with the subview.

Using a standard callout is the easiest way to display custom content without creating a custom subview. For example, Figure 6-2 shows a standard callout that's been modified to display a custom image and a detail disclosure button (Listing 6-6 shows the code that creates this modified callout).

Figure 6-2 A standard callout modified to include a custom image and disclosure button



Listing 6-6 Customizing a standard callout

```
// This code snippet assumes that an annotation for the Golden Gate Bridge has
already been added to the map view.


- (MKAnnotationView *)mapView:(MKMapView *)theMapView viewForAnnotation:(id
<MKAnnotation>)annotation
{
// Try to dequeue an existing pin view first (code not shown).


// If no pin view already exists, create a new one.
MKPinAnnotationView *customPinView = [[MKPinAnnotationView alloc]
                                        initWithAnnotation:annotation
reuseIdentifier:BridgeAnnotationIdentifier];
customPinView.pinColor = MKPinAnnotationColorPurple;
customPinView.animatesDrop = YES;
customPinView.canShowCallout = YES;
```

```
// Because this is an iOS app, add the detail disclosure button to display details
 about the annotation in another view.
UIButton *rightButton = [UIButton buttonWithType:UIButtonTypeDetailDisclosure];
[rightButton addTarget:nil action:nil forControlEvents:UIControlEventTouchUpInside];
customPinView.rightCalloutAccessoryView = rightButton;


      // Add a custom image to the left side of the callout.
UIImageView *myCustomImage = [[UIImageView alloc] initWithImage:[UIImage
imageNamed:@"MyCustomImage.png"]];
customPinView.leftCalloutAccessoryView = myCustomImage;


return customPinView;
}
```

In an iOS app, it's good practice to use the `mapView:annotationView:calloutAccessoryControlTapped:` delegate method to respond when users tap a callout view's control (as long as the control is a descendant of `UIControl`). In your implementation of this method you can discover the identity of the callout view's annotation view so that you know which annotation the user tapped. In a Mac app, the callout view's view controller can implement an action method that responds when a user clicks the control in a callout view.

When you use a custom view instead of a standard callout, you need to do extra work to make sure your callout shows and hides appropriately when users interact with it. The steps below outline the process for creating a custom callout that contains a button:

1. Design an `NSView` or `UIView` subclass that represents the custom callout. It's likely that the subclass needs to implement the `drawRect:` method to draw your custom content.
2. Create a view controller that initializes the callout view and performs the action related to the button.
3. In the annotation view, implement `hitTest:` to respond to hits that are outside the annotation view's bounds but inside the callout view's bounds, as shown in Listing 6-7 (page 65).
4. In the annotation view, implement `setSelected:animated:` to add your callout view as a subview of the annotation view when the user clicks or taps it. If the callout view is already visible when the user selects it, the `setSelected:` method should remove the callout subview from the annotation view (see Listing 6-8 (page 65)).
5. In the annotation view's `initWithAnnotation:` method, set the `canShowCallout` property to `NO` to prevent the map from displaying the standard callout when the user selects the annotation.

Listing 6-7 shows an example of implementing `hitTest:` to handle hits in the callout view that might be outside the bounds of the annotation view.

Listing 6-7 Responding to hits within a custom callout

```
- (NSView *)hitTest:(NSPoint)point
{
    NSView *hitView = [super hitTest:point];
    if (hitView == nil && self.selected) {
        NSPoint pointInAnnotationView = [self.superview convertPoint:point
toView:self];
        NSView *calloutView = self.calloutViewController.view;
        hitView = [calloutView hitTest:pointInAnnotationView];
    }
    return hitView;
}
```

Listing 6-8 shows an example of implementing `setSelected:animated:` to animate the arrival and dismissal of a custom callout view when the user selects the annotation view.

Listing 6-8 Adding and removing a custom callout view

```
- (void)setSelected:(BOOL)selected
{
    [super setSelected:selected];

    // Get the custom callout view.
    NSView *calloutView = self.calloutViewController.view;
    if (selected) {
        NSRect annotationViewBounds = self.bounds;
        NSRect calloutViewFrame = calloutView.frame;
      // Center the callout view above and to the right of the annotation view.
        calloutViewFrame.origin.x = -(NSWidth(calloutViewFrame) -
NSWidth(annotationViewBounds)) * 0.5;
        calloutViewFrame.origin.y = -NSHeight(calloutViewFrame) + 15.0;
        calloutView.frame = calloutViewFrame;

        [self addSubview:calloutView];
```

```
    } else {
        [calloutView.animator removeFromSuperview];
    }
}
```

Displaying Multiple Annotation Objects

If your app works with more than a few annotations, you might need to think about how you display them. The map view considers all annotation objects it knows about to be active and as a result, it always tries to display a corresponding annotation view when the given coordinate point is on the screen. If the coordinates for two annotations are close together, this could lead to overlap between the corresponding annotation views. If your map includes hundreds of annotations, zooming out far enough could lead to a visually unappealing mass of annotation views. Even worse, the annotations may be so close together that the user can't access some of them.

The only way to eliminate annotation overcrowding is to remove some of the annotation objects from the map view. This typically involves implementing the `mapView:regionWillChangeAnimated:` and `mapView:regionDidChangeAnimated:` methods to detect changes in the map zoom level. During a zoom change, you can add or remove annotations as needed based on their proximity to one another. You might also consider other criteria (such as the user's current location) to eliminate some annotations.

Map Kit includes numerous functions that make determining the proximity of map points easier. If you convert the map coordinate of your annotation to the map point coordinate space, you can use the `MKMetersBetweenMapPoints` method to get the absolute distance between two points. You can also use each coordinate as the center of a map rectangle and use the `MKMapRectIntersectsRect` function to find any intersections. For a complete list of functions, see *MapKit Functions Reference*.

Marking Your Annotation View as Draggable

Annotation views provide built-in dragging support, which makes it very easy to drag annotations around the map and to ensure that the annotation data is updated accordingly. To implement minimal support for dragging:

- In your annotation objects, implement the `setCoordinate:` method to allow the map view to update the annotation's coordinate point.
- When creating your annotation view, set its `draggable` property to `YES`.

When the user touches and holds a draggable annotation view, the map view begins a drag operation for it. As the drag operation progresses, the map view calls the `mapView:annotationView:didChangeDragState:fromOldState:` method of its delegate to notify it of changes to the drag state of your view. You can use this method to affect or respond to the drag operation.

To animate your view during a drag operation, implement a custom `dragState` method in your annotation view. As the map view processes drag-related touch events, it updates the `dragState` property of the affected annotation view. Implementing a custom `dragState` method gives you a chance to intercept these changes and perform additional actions, such as animating the appearance of your view. For example, the `MKPinAnnotationView` class raises the pin off the map when a drag operation starts and drops the pin back down on the map when it ends.

Displaying Overlays on a Map

Overlays offer a way to layer content over an arbitrary region of the map. Whereas annotations are always defined by a single map coordinate, overlays are typically defined by multiple coordinates. You can use these coordinates to create contiguous or noncontiguous sets of lines, rectangles, circles, and other shapes, which

can then be filled or stroked with color. For example, you might use overlays to layer traffic information on top of roadways, highlight the boundaries of a park, or show city, state, and national borders. Figure 6-3 shows a filled and stroked overlay covering the state of Colorado.

Figure 6-3 Displaying an overlay on a map

To display an overlay on a map, your app must provide two distinct objects:

- An *overlay object*, which is an object that conforms to the `MKOverlay` protocol and manages the data points for the overlay.
- An *overlay renderer*, which is a class (derived from `MKOverlayRenderer`) used to draw the visual representation of the overlay on the map surface.

> **Note:** In iOS 7.0 and later, `MKOverlayRenderer` replaces `MKOverlayView`. Renderers provide the same functionality as views, but they are more lightweight and efficient.

Overlay objects are generally small data objects that store the points that define the overlay and any other relevant information, such as a title string. Because overlays are defined using a protocol, you can turn any class in your app into an overlay object. In addition, Map Kit defines several concrete overlay objects for specifying different types of standard shapes. The map view keeps a reference to the overlay objects you add to it and uses the data in those objects to determine when to display a corresponding view.

Map Kit provides standard overlay renderers that can draw any shapes represented by the concrete overlay objects. As with annotations, you don't add overlay renderers directly to the map surface. Instead, the delegate object provides an overlay renderer when the map view asks for one, and the map view incorporates the overlay into its opaque view hierarchy.

Typically, the coordinates of an overlay on the map never change. Although it's possible to create draggable overlays, doing so is rare. You would need to implement the code to track the dragging operation, and you must update the overlay coordinate points yourself.

Steps for Adding an Overlay to the Map

Here are the steps for implementing and using overlays in your map-based app. It's assumed that your app incorporates an `MKMapView` object somewhere in its interface.

1. Define an appropriate overlay data object using one of the following options:
    - Use the `MKCircle`, `MKPolygon`, or `MKPolyline` class as-is.
    - Use the `MKTileOverlay` class if your overlay is represented by collection of custom bitmap tiles.
    - Subclass `MKShape` or `MKMultiPoint` to create overlays that provide app-specific behaviors or use custom shapes.
    - Use an existing class from your app and make it conform to the `MKOverlay` protocol.
2. Define an overlay renderer to present the overlay onscreen using one of the following options:
    - For standard shapes, use the `MKCircleRenderer`, `MKPolygonRenderer`, or `MKPolylineRenderer` class to represent the overlay. You can customize many of the drawing attributes of the final shape using these classes.
    - For a tiled overlay, use the `MKTileOverlayRenderer` class to handle the drawing of tiles on the map.
    - For custom shapes descended from `MKShape`, define an appropriate subclass of `MKOverlayPathRenderer` to render the shape.

- For all other custom shapes and overlays, subclass `MKOverlayRenderer` and implement your custom drawing code.

3. Implement the `mapView:rendererForOverlay:` method in your map view delegate.
4. Add your overlay data object to the map view using the `addOverlay:` method or one of many others.

Unlike annotations, rendered overlays are automatically scaled to match the current zoom level of the map. Scaling the overlay is necessary because overlays generally highlight boundaries, roads, and other content that also scales during zooming.

You can rearrange the Z-ordering of overlays in a map to ensure that specific overlays are always displayed on top of others. And, to specify the level of an overlay with respect to other map content, such as roads, labels, and annotation views, use one of the `MKOverlayLevel` constants:

- `MKOverlayLevelAboveRoads`
- `MKOverlayLevelAboveLabels`

> **Note:** In a 3D map, an overlay that uses the `MKOverlayLevelAboveRoads` level can be occluded by 3D buildings. This appearance is appropriate for overlays that should look as though they're on the ground.

Overlays can supplement or replace system-provided map content. If your overlay is opaque, use the `canReplaceMapContent` property to prevent the system from unnecessarily loading and drawing map content that users won't see behind the overlay.

Using the Standard Overlay Objects and Views

If all you want to do is highlight a specific map region, using the standard overlay classes is the easiest way to do it. The standard overlay classes include `MKCircle`, `MKPolygon`, and `MKPolyline`. These classes define the basic shape of the overlay and are used in conjunction with the `MKCircleRenderer`, `MKPolygonRenderer`, or `MKPolylineRenderer` classes, which handle the rendering of that shape on the map surface.

Listing 6-9 shows an example of how you would create the rectangular polygon shown in Figure 6-3 (page 68). This polygon consists of four map coordinates that correspond to the four corners of the state of Colorado. After creating the polygon, all you have to do is add it to the map using the `addOverlay:` method.

Listing 6-9 Creating a polygon overlay object

```
// Define an overlay that covers Colorado.
CLLocationCoordinate2D  points[4];
```

```
    points[0] = CLLocationCoordinate2DMake(41.000512, -109.050116);
    points[1] = CLLocationCoordinate2DMake(41.002371, -102.052066);
    points[2] = CLLocationCoordinate2DMake(36.993076, -102.041981);
    points[3] = CLLocationCoordinate2DMake(36.99892, -109.045267);

    MKPolygon* poly = [MKPolygon polygonWithCoordinates:points count:4];
    poly.title = @"Colorado";

    [map addOverlay:poly];
```

For an overlay to be shown on the map, the `mapView:viewForOverlay:` method of your map view delegate needs to provide an appropriate overlay renderer. For the standard overlay shapes, you can do this by creating a renderer that matches the type of shape you want to display. Listing 6-10 shows an implementation of this method that creates the polygon renderer used to cover the state of Colorado. In this example, the method sets the colors to use for rendering the shape and the border width.

Listing 6-10 Creating a polygon renderer for rendering a shape

```
- (MKOverlayRenderer *)mapView:(MKMapView *)mapView viewForOverlay:(id
<MKOverlay>)overlay
{
    if ([overlay isKindOfClass:[MKPolygon class]])
    {
        MKPolygonRenderer*    aRenderer = [[MKPolygonRenderer alloc]
initWithPolygon:(MKPolygon*)overlay];

        aRenderer.fillColor = [[UIColor cyanColor] colorWithAlphaComponent:0.2];
       aRenderer.strokeColor = [[UIColor blueColor] colorWithAlphaComponent:0.7];
        aRenderer.lineWidth = 3;

        return aRenderer;
    }

    return nil;
}
```

It's important to remember that the standard overlay renderers simply fill and stroke the shape represented by the overlay. If you want to display additional information, you need to create a custom overlay renderer to do the necessary drawing. You should avoid adding subviews to an existing overlay in an attempt to render any extra content. Any subviews you add to an overlay are scaled along with the overlay itself and made to fit the zoom level of the map. Unless your subviews contain content that also scales well, the results would probably not look very good.

Working With Tiled Overlays

If you use custom bitmap tiles to provide an overlay, you can use `MKTileOverlay` to manage the tiles and `MKTileOverlayRenderer` to present them. A single `MKTileOverlay` object can handle the tiles for multiple zoom levels, because the class's URL template lets you specify a tile's map position, zoom level, and scale factor, in addition to its location within the app bundle or on a server.

By default, the upper-left corner of the map is the origin for the tile's x and y values. (You can use the `geometryFlipped` property to move the origin to the lower-left corner, but if you want to use a custom indexing scheme, you need to subclass `MKTileOverlay`.) Each zoom level increases or decreases the number of tiles by a power of two. For example, a zoom level of three means that there are eight tiles in both the x and y directions.

To create tiles that match the curvature of the map, use the EPSG:3857 spherical Mercator projection coordinate system.

Defining a Custom Overlay Object

The job of an overlay object is to manage the coordinate data and any additional information associated with the overlay. Map Kit provides the following options for defining custom overlays:

- Adopting the `MKOverlay` protocol in one of your app's existing classes
- Subclassing `MKShape` or `MKMultiPoint` to define new types of shape-based overlays
- Subclassing `MKTileOverlay` to manage tiles that use a custom indexing scheme

Whether you subclass or adopt the `MKOverlay` protocol, the work you have to do in a custom overlay object is the largely same. The main job of an overlay object is to vend two key pieces of information:

- A coordinate that defines the center point of the overlay
- A bounding rectangle that completely encompasses the overlay's content

The bounding rectangle is the most important piece of information to the overlay itself. The map view uses the bounding rectangle specified by an overlay object as its cue for when to add the corresponding overlay renderer to the map. (If you add the overlay to the map as an annotation as well, the coordinate value similarly determines when the corresponding annotation view should be added to the map.) The bounding rectangle itself must be specified using map points, not map coordinates. You can convert between the two coordinate systems using the Map Kit functions.

Most of the work involved with displaying an overlay is done by the corresponding overlay renderer object. The overlay object simply defines where on the map the overlay should be placed, whereas the overlay renderer defines the final appearance of the overlay, including what information (if any) is displayed for the overlay. The creation of custom overlay renderers is described further in Defining a Custom Overlay Renderer (page 73).

Defining a Custom Overlay Renderer

If you want to do more than draw the boundaries or fill the content of your overlay shape, you need to create a custom overlay renderer. For example, if you're drawing a traffic overlay, you could use a custom overlay renderer to color-code each roadway based on its conditions. You can also use custom drawing code to animate your overlay's appearance.

To create a custom overlay renderer, you must subclass `MKOverlayRenderer`. (If you simply want to modify the drawing behavior of an existing shape-based overlay, you can subclass `MKOverlayPathRenderer` instead.)

In your custom implementation of `MKOverlayRenderer`, you should implement the following methods:

- `drawMapRect:zoomScale:inContext:` to draw your custom content
- `canDrawMapRect:zoomScale:` if your drawing code depends on content that might not always be available

The `canDrawMapRect:zoomScale:` method is for situations where your content may not always be ready to draw. For example, a traffic overlay would need to download the needed traffic data from the network before it could draw. If you return `NO` from this method, the map view refrains from drawing your overlay until you signal that you are ready. You can do this by marking your view as dirty using either the `setNeedsDisplayInMapRect:` or `setNeedsDisplayInMapRect:zoomScale:` method.

When your view is ready to draw, the map view calls the `drawMapRect:zoomScale:inContext:` method to do the actual drawing. (Note that if you need to perform image processing on tiles as they are being drawn, you should subclass `MKTileOverlayRenderer`.) Unlike drawing in a normal view, drawing in an overlay view requires special considerations:

- In your drawing code, never use the view's bounds or frame as reference points for drawing. Instead, use the map points associated with the overlay object to define shapes. Immediately before drawing, the code should convert those map points to points (`CGPoint` and so on) using the conversion routines found in the `MKOverlayRenderer` class.

  Also, you typically don't apply the zoom scale value passed to this method directly to your content. Instead, you provide it only when a Map Kit function or method specifically requires it. As long as you specify content using map points and convert to points, your content should be scaled to the correct size automatically.
- If you use UIKit classes and functions to draw, explicitly set up and clean up the drawing environment. Before issuing any calls, call the `UIGraphicsPushContext` function to make the context passed to your method the current context. When you finish drawing, call `UIGraphicsPopContext` to remove that context.
- Remember that the map view may tile large overlays and render each tile on a separate thread. Your drawing code should therefore not attempt to modify variables or other data unless it can do so in a thread-safe manner.

Listing 6-11 shows the drawing code used to fill the bounding rectangle of an overlay using a gradient. When drawing gradients, it is especially important to contain the drawing operation by applying a clipping rectangle to the desired drawing area. The view's frame is actually larger than the overlay's bounding rectangle, so without a clipping rectangle, the gradient would render outside the expected area. Because the bounding rectangle of the overlay defines the actual shape in this case, this method simply clips to the bounding rectangle. For more complex overlays, you would want to clip to the path representing your overlay. The results of this drawing code are shown in Figure 6-4 (page 76).

Listing 6-11 Drawing a gradient in a custom overlay

```
- (void)drawMapRect:(MKMapRect)mapRect zoomScale:(MKZoomScale)zoomScale
inContext:(CGContextRef)context
{
   // Get the overlay bounding rectangle.
   MKMapRect  theMapRect = [self.overlay boundingMapRect];
   CGRect theRect = [self rectForMapRect:theMapRect];

   // Clip the context to the bounding rectangle.
   CGContextAddRect(context, theRect);
   CGContextClip(context);
```

```
    // Set up the gradient color and location information.
    CGColorSpaceRef myColorSpace = CGColorSpaceCreateDeviceRGB();
    CGFloat locations[4] = {0.0, 0.33, 0.66, 1.0};
    CGFloat components[16] = {0.0, 0.0, 1.0, 0.5,
                              1.0, 1.0, 1.0, 0.8,
                              1.0, 1.0, 1.0, 0.8,
                              0.0, 0.0, 1.0, 0.5};

    // Create the gradient.
    CGGradientRef myGradient = CGGradientCreateWithColorComponents(myColorSpace,
components, locations, 4);
    CGPoint start, end;
    start = CGPointMake(CGRectGetMidX(theRect), CGRectGetMinY(theRect));
    end = CGPointMake(CGRectGetMidX(theRect), CGRectGetMaxY(theRect));

    // Draw.
    CGContextDrawLinearGradient(context, myGradient, start, end, 0);

    // Clean up.
    CGColorSpaceRelease(myColorSpace);
    CGGradientRelease(myGradient);
}
```

Figure 6-4 shows the results of drawing custom content over the overlay for the state of Colorado. In this case, the overlay renderer fills its content with a custom gradient.

Figure 6-4 Using a custom overlay renderer to draw

Creating Overlay Renderers from Your Delegate Object

When it needs an overlay renderer, the map view calls the `mapView:rendererForOverlay:` method of its delegate object. If you don't implement this method—or if you implement it and always return `nil`—the map view doesn't display anything for the specified overlay. Therefore, you must implement this method and return a valid overlay renderer for any overlays you want displayed on the map.

For the most part, every overlay is different. Although you should always create overlay renderers in your `mapView:rendererForOverlay:` method, you may need to be a little more creative in how you configure those objects. If all of your renderers share the same drawing attributes, you can implement this method in a way similar to the one shown in Listing 6-10 (page 71). However, if each overlay uses different colors or drawing attributes, you should find a way to initialize that information using the annotation object, rather than having a large decision tree in `mapView:rendererForOverlay:`.

Because overlays are typically different from one another, the map view doesn't recycle those objects when they are removed from the map. Instead of dequeueing an existing overlay renderer, you must create a new one every time.

Displaying Multiple Overlay Objects

If your app works with more than one overlay, you might need to think about how to manage those objects. Like annotations, the overlays associated with a map are always displayed when any portion of the overlay intersects the visible portion of the map. Unlike annotations, overlays scale proportionally with the map and therefore don't automatically overlap one another. This means that you are less likely to have to remove overlays and add them later to prevent overcrowding. In cases where the bounding rectangles of two overlays do overlap, you can either remove one of the overlays or arrange their Z-order to control which one appears on top.

The `overlays` property of the `MKMapView` class stores the registered overlays in an ordered array. The order of the objects in this array matches the Z-order of the objects at render time, with the first object in the array representing the bottom of the Z-order. To place an overlay on top of all other overlays, you add it to the end of this array. You can also insert objects at different points in the array and exchange the position of two objects in the array using the map view's methods.

If you decide to implement some type of overlap-detection algorithm for overlays, one place to do so is in the `mapView:didAddOverlayRenderers:` method of your map view delegate. When this method is called, use the `MKMapRectIntersectsRect` function to see if the added overlay intersects the bounds of any other overlays. If there is an overlap, use custom logic to choose which one should be placed on top in the rendering tree and exchange their positions as needed. (The comparison logic can occur on any thread, but because the map view is an interface item, any modifications to the `overlays` array should be synchronized and performed on the app's main thread.)

Using Overlays as Annotations

The `MKOverlay` protocol conforms to the `MKAnnotation` protocol. As a result, all overlay objects are also annotation objects and can be treated as one or both in your code. If you opt to treat an overlay object as both an overlay and an annotation, you are responsible for managing that object in two places. If you want to display both an overlay renderer and annotation view for the object, you must implement both the `mapView:rendererForOverlay:` and `mapView:viewForAnnotation:` methods in your app delegate. You must also add and remove the object from both the `overlays` and `annotations` arrays of your map.

Providing Directions

Using Map Kit support for map-based directions, an app can:

- Ask the Maps app to display directions to the user
- Register as a routing app and provide point-to-point, turn-based directions to Maps and other apps on the user's device (iOS only)
- Get general-purpose directions information

In OS X v10.9 and iOS 7.0 and later, Map Kit includes the `MKDirections` API, which provides route-based directions data from Apple servers. An app can use the `MKDirections` API to enhance its route or directions information by adding step-by-step walking or driving directions, travel time estimates, alternate routes, and other information.

To use the features of the Map Kit framework, you must turn on the Maps capability in your Xcode project.

Asking the Maps App to Display Directions

Map Kit enables you to display specific map locations or turn-by-turn directions using the Maps app. This support allows apps to use the Maps app in situations where you might not want to display the map data yourself. For example, if your app doesn't have its own turn-by-turn navigation services, you can ask Maps to provide the information for you.

To generate a direction request, use the `openMapsWithItems:launchOptions:` or `openInMapsWithLaunchOptions:` method of `MKMapItem`. These methods let you send map items to the Maps app so that it can display them. If you include the `MKLaunchOptionsDirectionsModeKey` key with the launch options you pass to these methods, Maps displays directions between the start and end map items you provide.

Listing 7-1 shows a method that opens the specified region in the Maps app and centers the map on its location. You use the `launchOptions` parameter of the `openMapsWithItems:launchOptions:` method to specify the map region you want to display.

Listing 7-1 Displaying a location in the Maps app

```
- (void)displayRegionCenteredOnMapItem:(MKMapItem*)from {
```

```
    CLLocation* fromLocation = from.placemark.location;


    // Create a region centered on the starting point with a 10km span
    MKCoordinateRegion region =
MKCoordinateRegionMakeWithDistance(fromLocation.coordinate, 10000, 10000);


    // Open the item in Maps, specifying the map region to display.
    [MKMapItem openMapsWithItems:[NSArray arrayWithObject:from]
                launchOptions:[NSDictionary dictionaryWithObjectsAndKeys:
                    [NSValue valueWithMKCoordinate:region.center],
MKLaunchOptionsMapCenterKey,
                    [NSValue valueWithMKCoordinateSpan:region.span],
MKLaunchOptionsMapSpanKey, nil]];
}
```

For information about how to open the Maps app programmatically, see *MKMapItem Class Reference*.

Registering as a Routing App (iOS Only)

An app that is able to display point-to-point directions can register as a routing app and make those directions available to the Maps app and to all other apps on a user's device. Registering as a routing app improves the user experience by giving other apps a means to access routing information from your app, thus avoiding the need for every app to provide its own routing directions. It's also a great way to get your app in front of the user because Maps displays appropriate suggestions from the App Store as options for displaying directions.

To register your app as a directions provider:

- Configure your app to accept directions requests and declare a document type to handle incoming routing requests.
- Declare the map regions that your app supports using a geographic coverage file (specified using a GeoJSON file).
- Process direction request URLs when they are sent to your app.

Configuring Your App to Accept Directions Requests

To enable directions support for your app, configure the following support for your app target in Xcode:

- Enable routing requests and specify the types of directions your app is capable of providing.

- Configure a special document type to handle incoming routing requests.

iTunes Connect requires both items for all routing apps.

You enable routing request support by choosing the appropriate Routing checkboxes in the Maps section of your target's Capabilities pane in Xcode. (To learn more about the configuration process, see "Configuring an iOS Routing App" in *App Distribution Guide*.) Enabling directions support adds the `MKDirectionsApplicationSupportedModes` key to your app's `Info.plist` file and sets the value of that key to the transportation modes you select. You must also provide a GeoJSON file containing the regions for which your app is capable of providing directions. (For information about specifying the GeoJSON file, see Declaring the Supported Geographic Coverage for Directions (page 80).)

In addition to declaring the types of directions your app supports, you need to add to your project a document type whose purpose is to handle incoming routing requests. To add the document type, use the Xcode document type editor, located in the Info tab of your target. Create the new type and use the information in Table 7-1 to set the values for the type. You should use the values in the table exactly as they appear here.

Table 7-1 Keys and values for the directions request document type

Xcode field	Value	Description
Name	`MKDirectionsRequest`	The value in this field can be anything you want, but you are encouraged to use the suggested value.
Types	`com.apple.maps.directionsrequest`	The value of this key is a custom UTI type used to identify direction requests. The value of this key must be exactly as shown here.

The changes you make in the document types editor cause Xcode to add a corresponding entry to the `CFBundleDocumentTypes` key in your app's `Info.plist` file. For more information about your app's `Info.plist` file and the keys you put into it, see *Information Property List Key Reference*.

Declaring the Supported Geographic Coverage for Directions

Apps must declare the geographic regions for which they are capable of providing directions. Declaring the supported regions prevents the system from sending requests that your app is unable to handle. For example, if your app provides subway route information only for New York, you would not want it to receive requests to provide directions in London.

To declare your app's supported regions, create a GeoJSON file, which is a type of JavaScript Object Notation (JSON) file. This is your app's *geographic coverage file* and it contains a set of map coordinates that define the boundaries of one or more geographic regions. The format of this file is part of a public specification, the details of which you can obtain at http://geojson.org/geojson-spec.html.

Your geographic coverage file is used by the Maps app and the App Store, but not by your app. Uploading it to iTunes Connect makes it possible for Maps to look up apps on the App Store that are capable of providing directions in the target region. The geographic coverage file can then provide the user with that information in case the user wants to download one of those apps. Because you upload the geographic coverage file to iTunes Connect, you can also update the file at any time without submitting a new version of your app.

Specifying the Geographic Coverage File Contents

The geographic coverage file must contain a single `MultiPolygon` shape that defines the boundaries of the supported geographic regions. A single multi polygon may contain multiple child polygons, each of which defines the boundaries around a single geographic region. Each child polygon contains four or more coordinate values that define the boundaries of the polygon. Per the GeoJSON specification, every child polygon must represent a closed region—that is, the first and last coordinate values must always be identical. Therefore, you must specify at least four points to define a triangular region, which is the simplest possible shape. Of course, you use more points to define more complex polygons.

Listing 7-2 shows an example of a geographic coverage file that specifies two separate regions: one around San Francisco and one around Chicago. Note that the coordinates for each region are surrounded by what seem to be an extra set of brackets. In each case, the first open bracket marks the beginning of the overall polygon shape, the second open bracket defines the beginning of the exterior bounding polygon, and the third open bracket marks the beginning of the first coordinate.

Listing 7-2 A sample GeoJSON file

```
{ "type": "MultiPolygon",
  "coordinates": [
    [[[-122.7, 37.3], [-121.9, 37.3], [-121.9, 37.9], [-122.7, 37.9], [-122.7,
37.3]]],
    [[[-87.9, 41.5], [-87.3, 41.5], [-87.3, 42.1], [-87.9, 42.1], [-87.9, 41.5]]]
    ]
}
```

Use the following guidelines to help you define the regions for your app:

- Define as specific a region as possible. For example, if your app provides subway route information only for New York, you don't want it to receive requests to provide directions in London. The more specific the region you define, the more likely your app will appear in the search results.
- Keep your regions simple and don't try to trace precise routes around a given geographic area. It's recommended that each polygon contain no more than 20 points. Using more coordinates to define a region is possible but is inefficient and usually unnecessary. The system has to determine whether a given point is inside one of your polygons, and it's cheaper to make that determination against a rectangle or other simple shape than it is against a shape with hundreds of points.
- Limit the number of child polygons in your `MultiPolygon` shape to 20 or fewer.
- Use only `MultiPolygon` shapes in your app's coverage file. Although the GeoJSON specification supports other types of shapes, only `MultiPolygon` shapes are supported for routing apps.
- Don't include comments in your coverage file.
- Specify all coordinates as a longitude value followed by a latitude value.
- Don't include holes in the polygons in your coverage files. Although holes are supported by the GeoJSON specification, they are not supported in Apple's maps implementation.

Uploading Your GeoJSON File to iTunes Connect

When you upload your app to iTunes Connect, the server checks your app's `Info.plist` file for the presence of the `MKDirectionsApplicationSupportedModes` key. If that key is present, the iTunes Connect page for your app includes a space for uploading your geographic coverage file. You must upload the file before your app can be approved. See Uploading Icons, Screenshots, and Routing App Files for Your App in *iTunes Connect Developer Guide* for more information.

Debugging Your App's Geographic Coverage File

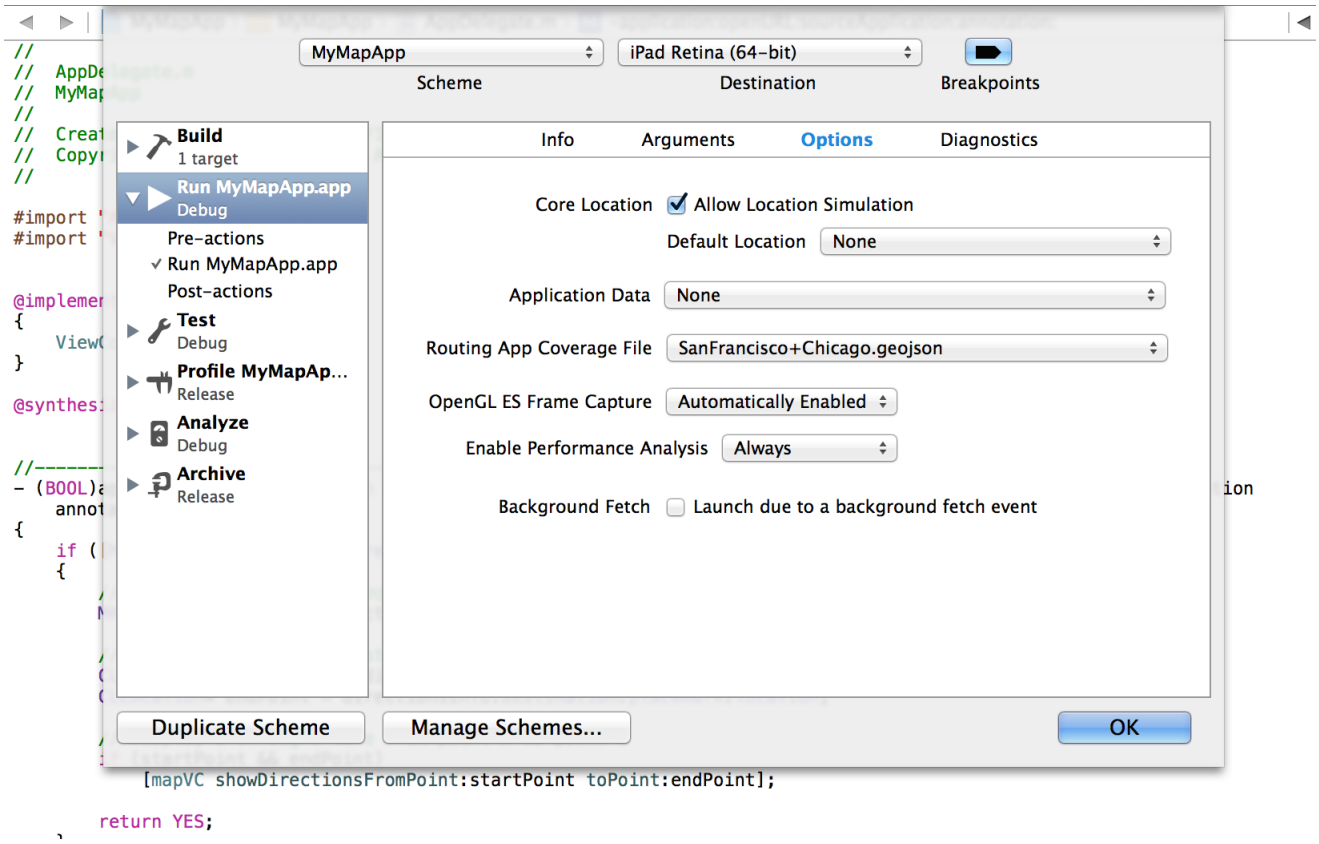
It's a good idea to test your GeoJSON file before uploading it to iTunes Connect. During development, use the scheme options of your Xcode project to specify a geographic coverage file to test. When you run your app in the simulator, Xcode transfers this file to the simulator or device so that the Maps app knows about the regions your app supports.

To specify the geographic coverage file for testing:

1. Open the scheme editor for your project.
2. Select the Run (Debug) scheme for your project.
3. In the Options tab, select your coverage file from the Routing App Coverage File popup menu.

The scheme editor should appear, similar to Figure 7-1.

Figure 7-1 Choosing a GeoJSON file in the scheme editor



After installing your app on the device or simulator, exit your app and launch the Maps app to specify the start and end points for directions. Your app should appear if its geographic coverage file is valid and contains the two specified points. If it doesn't appear, check the points for your geographical regions to make sure they are correct.

> **Note:** If your app is running in the simulator and Maps doesn't display it as one of the possible options, check the Console app for possible error messages. Maps reports any problems it encounters when reading your app's GeoJSON to the console output rather than to Xcode.

Handling URL Directions Requests

The system sends directions requests to a routing or other app using URLs. For example, when a user asks the Maps app for directions and chooses your app, Maps creates a URL with the start and end points and asks your app to open it. You handle the opening of this URL in your app's `application:openURL:sourceApplication:annotation:` method, just like you would for other URLs. But the steps you take to process the URL are a little more prescribed.

When your app receives a URL to open, perform the following steps in the `application:openURL:sourceApplication:annotation:` method:

1. Call the `isDirectionsRequestURL:` class method of `MKDirectionsRequest` to check whether the URL contains a directions request. If the URL contains a directions request, proceed with the remaining steps.
2. Create an `MKDirectionsRequest` object using the provided URL.
3. Obtain the start and end points of the route from the `MKDirectionsRequest` object.

   The start and end points are specified as `MKMapItem` objects, which contain the geographic coordinate of the location and possibly additional information. If the `MKMapItem` object represents the user's current location, use Core Location to retrieve the actual coordinate location data yourself.
4. Compute and display the route using your custom logic.

Because the preceding steps are the same for every app, you can include code in your `application:openURL:sourceApplication:annotation:` method similar to that shown in Listing 7-3. The most important addition you must make is to compute the route information and display it in your app. If your app handles other URL types, you would also include code for processing those URLs separately.

Listing 7-3 Handling a directions request URL

```
- (BOOL)application:(UIApplication *)application
           openURL:(NSURL *)url
           sourceApplication:(NSString *)sourceApplication
           annotation:(id)annotation {
   if ([MKDirectionsRequest isDirectionsRequestURL:url]) {
```

```
      MKDirectionsRequest* directionsInfo = [[MKDirectionsRequest alloc]
initWithContentsOfURL:url];
      // TO DO: Plot and display the route using the
      //   source and destination properties of directionsInfo.
      return YES;
   }
   else {
    // Handle other URL types...
   }
    return NO;
}
```

You also use the `MKDirectionsRequest` class to create requests for the `MKDirections` API. To learn more about how to use this API, see Getting General-Purpose Directions Information.

Getting General-Purpose Directions Information

Apps can use the `MKDirections` API to get information about a route, such as distance, expected travel time, localized advisory notices, and the set of individual steps that make up the route. For example, a routing app that displays train routes could request route information to display walking routes to and from train stations.

> **Note:** Directions requests made using the `MKDirections` API are server-based and require a network connection.
>
> There are no request limits per app or developer ID, so well-written apps that operate correctly should experience no problems. However, throttling may occur in a poorly written app that creates an extremely large number of requests.

To get general-purpose directions information:

1. Create an `MKDirectionsRequest` object and configure it with the `MKMapItem` objects that represent the start and end points of the route. If you need to specify a transportation type, such as walking or driving, set the request object's `transportType` property.
2. Create an `MKDirections` object and initialize it with the request object you created in step 1.
3. Start the route-finding process by calling `calculateDirectionsWithCompletionHandler:` and providing a block that handles the results.

4. Handle the route information contained in the `MKDirectionsResponse` object passed to your completion handler. (The response object holds an array of `MKRoute` objects, each of which represents one alternative route between the start and end points of the request.)

Listing 7-4 shows one way to create a directions request, calculate a single route, and handle the results.

Listing 7-4 Requesting directions

```
MKDirectionsRequest *walkingRouteRequest = [[MKDirectionsRequest alloc] init];
walkingRouteRequest.transportType = MKDirectionsTransportTypeWalking;
[walkingRouteRequest setSource:[startPoint mapItem]];
[walkingRouteRequest setDestination :[endPoint mapItem]];

MKDirections *walkingRouteDirections = [[MKDirections alloc]
initWithRequest:walkingRouteRequest];
[walkingRouteDirections
calculateDirectionsWithCompletionHandler:^(MKDirectionsResponse *
walkingRouteResponse, NSError *walkingRouteError) {
   if (walkingRouteError) {
      [self handleDirectionsError:walkingRouteError];
   } else {
      // The code doesn't request alternate routes, so add the single calculated
route to
      // a previously declared MKRoute property called walkingRoute.
      self.walkingRoute = walkingRouteResponse.routes[0];
   }
}];
```

A route object defines the geometry for one route and includes a polyline object that you can use as an overlay to show the route on a map. If you requested alternate routes, the response object that gets passed to your completion handler can contain multiple route objects in its `routes` array. Listing 7-5 shows one way to iterate over an array of routes and display the polyline associated with each alternate route.

Listing 7-5 Displaying alternate routes

```
- (void) myShowDirections:(MKDirectionsResponse *)response {
   self.response = response;
   for (MKRoute *route in self.response.routes) {
```

```
        [self.mapView addOverlay:route.polyline level:MKOverlayLevelAboveRoads];
    }
}
```

Enabling Search

To support searches based on user-entered queries, Map Kit provides the `MKLocalSearch` API. Apps can use this API to perform searches for locations that users describe by name, address, or type, such as coffee or theater.

Although local search and geocoding are similar, they support different use cases. Use geocoding when you want to convert between map coordinates and a structured address, such as an Address Book address. Use local search when you want to find a set of locations that match the user's input.

> **Note:** Search requests made using the `MKLocalSearch` API are server-based and require a network connection.
>
> There are no request limits per app or developer ID, so well-written apps that operate correctly should experience no problems. However, throttling may occur in a poorly written app that creates extremely large numbers of requests.

To search for a location that matches the query a user types into a search field:

1. Create an `MKLocalSearchRequest` object and specify a string that contains the user's natural-language query.

   (Optional) Define a geographical region to narrow the search results. It's recommended that you define a region to ensure that the user gets relevant results.
2. Create an `MKLocalSearch` object and initialize it with the search request you created in step 1.
3. Start the search by calling `startWithCompletionHandler:` and specifying a completion handler block to process the results.

Each `MKLocalSearch` object performs only one search; if you want to perform multiple searches, you have to define multiple search requests and use them to create multiple search objects. Because the search is performed asynchronously, you can create and begin multiple searches in parallel and handle the results as they arrive.

The results of a local search are provided as a set of `MKMapItem` objects. Each map item object contains the following attributes:

- The map coordinates of the location

- A structured address that represents the location
- If the location is a business, the name, phone number, and URL associated with the business might be available

Listing 8-1 shows one way to create a search request, initiate a local search, and display the results as annotations on a map.

Listing 8-1 Searching for locations that match the user's input

```
// Create and initialize a search request object.
MKLocalSearchRequest *request = [[MKLocalSearchRequest alloc] init];
request.naturalLanguageQuery = query;
request.region = self.map.region;

// Create and initialize a search object.
MKLocalSearch *search = [[MKLocalSearch alloc] initWithRequest:request];

// Start the search and display the results as annotations on the map.
[search startWithCompletionHandler:^(MKLocalSearchResponse *response, NSError
*error)
{
   NSMutableArray *placemarks = [NSMutableArray array];
   for (MKMapItem *item in response.mapItems) {
      [placemarks addObject:item.placemark];
   }
   [self.map removeAnnotations:[self.map annotations]];
   [self.map showAnnotations:placemarks animated:NO];
}];
```

Document Revision History

This table describes the changes to *Location and Maps Programming Guide*.

Date	Notes
2014-03-10	Described Map Kit features introduced in iOS 7 and OS X v10.9. Removed "Legacy Map Techniques" appendix.
2013-10-24	Corrected the types of devices that can be used as beacons.
2013-09-18	Added information about the new Core Location iBeacons. Formerly titled *Location and Maps Programming Guide*. Added information about using location services in the background. For details, see Getting Location Events in the Background (iOS Only) (page 18).
2012-09-19	Added information about how to support custom routing directions.
2011-10-12	Added information about the new Core Location geocoders.
2010-05-20	Added information about region monitoring. Added information about creating overlays. Expanded the existing information about maps and annotations. Updated the location-related sections to cover new technologies for obtaining the user's location.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010