Location Awareness Programming Guide

User Experience

2010-05-20

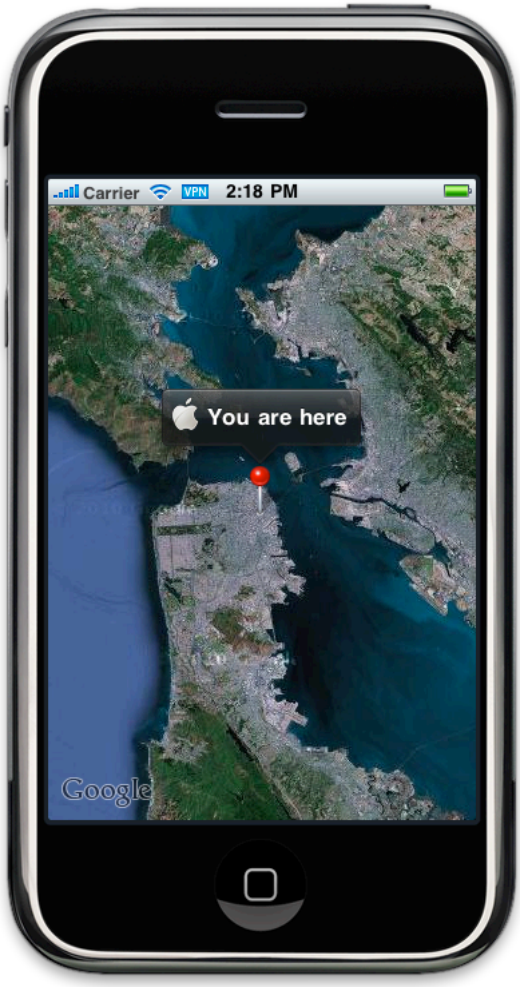
Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures, Tables, and Listings

Making Your Application Location-Aware

Using location-based information in your applications is a 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 improve the overall user experience.



Location-based information in iOS comprises two pieces: location services and maps. Location services are provided by the Core Location framework, which provides Objective-C interfaces for obtaining information about the user's location and heading. Maps are provided by the Map Kit framework, which supports both the display and annotation of maps similar to those found in the Maps application.

At a Glance

Map and location services provide a way for you to enhance user interactions. By incorporating geographic data into your applications, you can orient the user to the surrounding environment or help the user stay connected to other people nearby.

Location Services Provide a Geographical Context for Applications

Location services is all about mobility and the fact that your application is running on a device that can go anywhere. Knowing the user's geographic location can help you improve the quality of the information you offer, and in some cases it might even be at the heart of your application. Applications that offer navigation features use location services to monitor the user's position and generate updates. And many other types of applications use location as a way of connecting nearby users socially.

Relevant Chapters: "Getting the User's Location" (page 11), "Geocoding Location Data" (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 for implementing a compass but this technology is also used to support augmented reality, games, and navigational applications. And even on devices that do not have a magnetometer—the hardware used to get precise heading information—information about the user's course and speed are still available for applications that need it.

Relevant Chapters: "Getting Direction-Related Events" (page 19)

Maps Support Navigation and the Display of Geographically Relevant Content

Maps are a way to visualize geographical data in a way that is easy to understand. The Map Kit framework provides standard views that you can incorporate into your application and use to display information tied to specific geographic points. In addition, this framework provides the means to layer custom information on top of the map and have it scroll along with the rest of the map content.

Relevant Chapters: "Displaying Maps" (page 27), "Annotating Maps" (page 33)

How to Use This Document

You do not have to read this entire document to use each of the technologies. The services provided by the Core Location and Map Kit frameworks are separate and can be used independent of other services. Therefore, the beginning of each chapter introduces the terminology and information you need to understand the

corresponding technology followed by examples and task-related steps on how to use it. The only exception is the "Annotating Maps" (page 33) chapter, which builds on the information presented in the "Displaying Maps" (page 27) 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 *Map Kit Framework Reference*.

Getting the User's Location

Applications use location data for a wide variety of purposes, ranging from social networking to turn-by-turn navigation services. They get location data by using 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 of changes to that location. (iOS 4.0 and later).
- The standard location service offers a more configurable way to get the current location.
- Region monitoring lets you monitor boundary crossings for a defined area. (iOS 4.0 and later).

To use the features of the Core Location framework, you must link your application 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 Order to Run

If your application relies on location services to function properly, you should include the `UIRequiredDeviceCapabilities` key in the application's `Info.plist` file. You use this key to specify the location services that must be present in order for your application to run. The App Store uses the information in this key from preventing users from downloading applications to devices that do not contain the listed features.

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

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

Important: If your application uses location services but is able to operate successfully without them, do not 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 application. The framework uses information obtained from the built-in cellular, Wi-Fi, or GPS hardware to triangulate a location fix for the device. It reports that location to your code and, depending on how you configure the service, also provides periodic updates as it receives new or improved data.

There are two different services you can use to get the user's current location:

- The **standard location service** is a configurable, general-purpose solution and is supported in all versions of iOS.
- The **significant-change location service** offers a low-power location service for devices with cellular radios. This service is available only in iOS 4.0 and later and can also wake up an application that is suspended or not running.

Gathering location data is a power-intensive operation. It involves powering up the onboard radios and querying the available cell towers, Wi-Fi hotspots, or GPS satellites, which can take several seconds. Leaving the standard location service running for extended periods can drain the device's battery. (The significant-change location service drastically reduces battery drain by monitoring only cell tower changes, but the service works only on devices with cellular radios.) For most applications, it is usually sufficient to establish an initial position fix and then acquire updates only periodically after that. If you are sure you need regular position updates, you should use the significant-change location service where you can; otherwise, you should configure the parameters of the standard location service in a way that minimizes its impact on battery life.

Determining Whether Location Services Are Available

Every iOS-based device is capable of supporting location services in some form but there are still situations where location services may not be available:

- The user can disable location services in the Settings application.
- The user can deny location services for a specific application.
- The device might be in Airplane mode and unable to power up the necessary hardware.

For these reasons, it is recommended that you always call the `locationServicesEnabled` class method of `CLLocationManager` before attempting to start either the standard or significant-change location services. (In iOS 3.x and earlier, check the value of the `locationServicesEnabled` property instead.) If this class method returns `YES`, you can start location services as planned. If it returns `NO` and you attempt to start location services anyway, the system prompts the user to confirm whether location services should be reenabled. Given that location services are very likely to be disabled on purpose, the user might not welcome this prompt.

Starting the Standard Location Service

The standard location service is the most common way to get the user's current location because it is available on all devices and in all versions of iOS. 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 which radios to enable and then proceeds to report location events to your application.

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.

Listing 1-1 shows a sample method that configures a location manager for use. This 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 application does not need precise location data, it configures the location service to report the general area of the user and notify it only when the user moves a significant distance, which in this case is 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;

    [locationManager startUpdatingLocation];
}
```

The code for receiving location updates from this service is shown in "Receiving Location Data from a Service" (page 14).

Starting the Significant-Change Location Service

In iOS 4.0 and later, you can use the significant-change location service to receive location events. This service offers a significant power savings and provides accuracy that is good enough for most applications. It uses the device's cellular radio 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. This service is also capable of waking up an application 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 14).

If you leave this service running and your application is subsequently suspended or terminated, the service automatically wakes up your application when new location data arrives. At wake-up time, your application is put into the background and given a small amount of time to process the location data. Because your application is in the background, it should 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 application may be terminated.

Receiving Location Data from a Service

Whether you use the standard location service or the significant-change location service to get location events, the way you receive those events is the same. Whenever a new event is available, the location manager reports it to the `locationManager:didUpdateToLocation:fromLocation:` method of its delegate. 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 is 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 fairly recent events are likely to be good enough. If you were implementing a navigation application, 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
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    // If it's a relatively recent event, turn off updates to save power
    NSDate* eventDate = newLocation.timestamp;
    NSTimeInterval howRecent = [eventDate timeIntervalSinceNow];
```

```
    if (abs(howRecent) < 15.0)
    {
        NSLog(@"latitude %+.6f, longitude %+.6f\n",
                newLocation.coordinate.latitude,
                newLocation.coordinate.longitude);
    }
    // else skip the event and process the next one.
}
```

In addition to a location object's timestamp, you can also use the accuracy reported by that object as a means of determining 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 application wastes less time on events that cannot be used effectively anyway.

Monitoring Shape-Based Regions

In iOS 4.0 and later, applications can use region monitoring to be notified when the user crosses geographic boundaries. You can use this capability to generate alerts when the user gets close to a specific location. For example, upon approaching a specific dry cleaners, an application could notify the user to pick up any clothes that had been dropped off and are now ready.

Regions associated with your application are tracked at all times, including when your application is not running. If a region boundary is crossed while an application is not running, that application is relaunched into the background to handle the event. Similarly, if the application is suspended when the event occurs, it is woken up and given a short amount of time to handle the event.

Determining the Availability of Region Monitoring

Before attempting to monitor any regions, your application should check to see if region monitoring is supported on the current device. There are several reasons why region monitoring might not be available:

- The device may not have the hardware needed to support region monitoring.
- The user may have disabled location services in the Settings application.
- The device might be in Airplane mode and unable to power up the necessary hardware.

For these reasons, it is recommended that you always call the `regionMonitoringAvailable` and `regionMonitoringEnabled` class methods of `CLLocationManager` before attempting to monitor regions. The `regionMonitoringAvailable` method lets you know whether the underlying hardware supports region monitoring. If it returns `NO`, there is no chance that your application will ever be able to use region monitoring on the device. If region monitoring is available, the `regionMonitoringEnabled` method reports whether the feature is currently enabled. If region monitoring is available but not enabled when you attempt to monitor a region, the system prompts the user to confirm whether region monitoring should be reenabled. Given that the feature is likely to be disabled on purpose, the user might not welcome this prompt.

Defining a Region to Be Monitored

To begin monitoring a region, you must define the region and register it with the system. Regions are defined using the `CLRegion` class, which currently supports the creation of circular regions. Each region you create must include both the data that defines the desired geographic area and a unique identifier string. (The identifier string is required and is the only guaranteed way for your application to identify regions later.) To register a region, you call the `startMonitoringForRegion:desiredAccuracy:` method of your `CLLocationManager` object.

Listing 1-4 shows a sample method that creates a new region based on a circular Map Kit overlay. 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. After registering the region, the region object itself can be released. Core Location stores the data associated with a region but does not typically store the region object itself.

Listing 1-4 Creating and registering a region based on a Map Kit overlay

```
- (BOOL)registerRegionWithCircularOverlay:(MyCircle*)overlay
andIdentifier:(NSString*)identifier
{
   // Do not create regions if support is unavailable or disabled.
   if ( ![CLLocationManager regionMonitoringAvailable] ||
        ![CLLocationManager regionMonitoringEnabled] )
      return NO;

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

   // Create the region and start monitoring it.
   CLRegion* region = [[CLRegion alloc]
initCircularRegionWithCenter:overlay.coordinate
                      radius:radius identifier:identifier];
   [self.locManager startMonitoringForRegion:region
                  desiredAccuracy:kCLLocationAccuracyHundredMeters];

   [region release];
   return YES;
}
```

Monitoring of a region begins immediately after registration. However, do not expect to receive an event right away. Only boundary crossings can generate an event. Thus, if at registration time the user's location is already inside the region, the location manager does not generate an event. Instead, you must wait for the user to cross the region boundary before an event is generated and sent to the delegate.

You should always 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 the number of regions that may be simultaneously monitored by a single application. To work around these limits, you should 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 Region

Every time the user's current location crosses a boundary region, the system generates an appropriate region event for your application. If your application is already running, these events go directly to the delegates of any current location manager objects. If your application is not running, the system launches it in the background so that it can respond. Applications can implement the following methods to handle boundary crossings:

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

The system does not report boundary crossings until the boundary plus a designated cushion distance is exceeded. You specify the desired cushion distance for a region when you register it using the `startMonitoringForRegion:desiredAccuracy:` method. 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 application is running in the background, you can use local notifications to alert the user; otherwise, you can simply post an alert.

Getting Location Events in the Background

If your application needs location updates delivered whether the application is in the foreground or background, there are multiple options for doing so. The preferred option is to use the significant location change service to wake your application at appropriate times to handle new events. However, if your application needs to use the standard location service, you can declare your application as needing background location services.

An application should request background location services only if the absence of those services would impair its ability to operate. In addition, any application that requests background location services should use those services to provide a tangible benefit to the user. For example, a turn-by-turn navigation application would be a likely candidate for background location services because of its need to track the user's position and report when it is time to make the next turn.

The process for configuring a background location application is described in "Executing Code in the Background" in *iOS Application Programming Guide*.

Tips for Conserving Battery Power

Receiving and transmitting data using the radios of an iOS-based device require more power than any other operation on the device. Because Core Location relies on these radios to determine the user's location, you should use location services judiciously in your applications. Most applications do not need location services to be running all the time, and so turning off those services is the simplest way to save power.

- **Turn off location services when you are not using them.** This may seem obvious but it is worth repeating. With the exception of navigation applications that offer turn-by-turn directions, most applications do not 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 should not 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 applications that need to track changes in the user's location but do not need the higher precision offered by the standard location services.

- **Use lower-resolution values for the desired accuracy unless doing so would impair your application.** Requesting a higher accuracy than you need causes Core Location to power up additional hardware and waste power for precision you are not using. Unless your application really needs to know the user's position within a few meters, do not put the values `kCLLocationAccuracyBest` or `kCLLocationAccuracyNearestTenMeters` in the `desiredAccuracy` property. And remember that specifying a value of `kCLLocationAccuracyThreeKilometers` does not 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 using Wi-FI and cellular signals.

- **Turn off location events if the accuracy does not improve over a period of time.** If your application is not 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 is not 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 application from wasting power.

Getting Direction-Related Events

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**.

Remember that heading and course information do not 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 does not take into account the device orientation. Depending on your application, you might prefer one over the other or use a combination of the two. For example, a navigation application 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 application requires some form of direction-related information in order to function properly, you should include the `UIRequiredDeviceCapabilities` key in the application's `Info.plist` file. This key contains an array of strings indicating the features that your application requires of the underlying iOS-based device. The App Store uses this information to prevent users from installing applications on a device without the minimum required hardware.

For direction-related events, there are two relevant strings you can associate with this key:

- `magnetometer`—Include this string if your application requires the presence of heading information.
- `gps`—Include this string if your application requires the presence of course-related information.

Important: If your application uses heading or course events but is able to operate successfully without them, do not include the corresponding string value with the `UIRequiredDeviceCapabilities` key.

In both cases, you should 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 applications 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.

The steps for receiving heading events are as follows:

1. Create a `CLLocationManager` object.
2. Determine whether heading events are available by calling the `headingAvailable` class method. (In iOS 3.x and earlier, check the value of the `headingAvailable` property instead.)
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 2-1 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. This code runs in iOS 4.0 and later

Listing 2-1 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 application. Upon receiving a new event, you should check the `headingAccuracy` property to ensure that the data you just received is valid, as shown in Listing 2-2. In addition, if you are using the true heading value, you should also check to see if it contains a valid value before using it.

Listing 2-2 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 indicating the device's current course and speed. Course information is used to indicate the direction in which the device is moving and does not necessarily reflect the orientation of the device itself. As a result, it is primarily intended for applications that provide navigation information while the user is moving.

The actual course and speed information is returned to your application 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 is 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 11)

Geocoding Location Data

Location data is usually returned as a pair of numerical values representing 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 are not 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 obtain 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. Many geocoding services allow you to perform conversions in either direction but iOS currently supports only **reverse geocoding**, which converts a latitude and longitude into a placemark. If you want to convert placemark information into a latitude and longitude—a process known as **forward geocoding**—you would need to find an appropriate third-party service to use.

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 cannot connect to the service it needs and must therefore return an appropriate error.

Each application has a limited amount of geocoding capacity, so it is to your advantage to use geocoding requests sparingly. Here are some rules of thumb to apply when creating your requests:

- Send at most one reverse-geocoding request for any one user action.
- If the user performs multiple actions that involve reverse-geocoding the same location, reuse the results from the initial reverse-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 reverse-geocoding request only when the user's location has moved a significant distance and after a reasonable amount of time has passed. For example, in a typical situation, you should not send more than one reverse-geocode request per minute.
- Do not start a reverse-geocoding request at a time when the user will not see the results immediately. For example, do not start a request if your application recently resigned the active state (possibly because of an interruption such as a phone call) and is waiting to become active again.

Getting Placemark Information from the Reverse Geocoder

Reverse geocoding is done using the `MKReverseGeocoder` class of the Map Kit framework. This class uses a delegate-based approach for geocoding a single location. This means that you can use a single instance of the `MKReverseGeocoder` class only once. In addition, the Google terms of service require that the `MKReverseGeocoder` class be used in conjunction with a Google map.

To initiate a reverse geocoding request, create an instance of the `MKReverseGeocoder` class, assign an appropriate object to the `delegate` property, and call the `start` method. If the query completes successfully, your delegate's `reverseGeocoder:didFindPlacemark:` method is called and passed an `MKPlacemark` object with the results. If there is a problem reverse geocoding the location, the `reverseGeocoder:didFailWithError:` method is called instead.

Listing 3-1 shows the code required to use a reverse geocoder. Upon successful completion of the geocoding operation, the code adds a button to the annotation view's callout so that it can display the placemark information. Because the annotation is not automatically available to the delegate, the custom `annotationForCoordinate:` method is included to find the appropriate annotation object from the map view.

Listing 3-1 Geocoding a location using `MKReverseGeocoder`

```
@implementation MyGeocoderViewController (CustomGeocodingAdditions)
- (void)geocodeLocation:(CLLocation*)location
forAnnotation:(MapLocation*)annotation
{
    MKReverseGeocoder* theGeocoder = [[MKReverseGeocoder alloc]
initWithCoordinate:location.coordinate];

    theGeocoder.delegate = self;
    [theGeocoder start];
}

// Delegate methods
- (void)reverseGeocoder:(MKReverseGeocoder*)geocoder
didFindPlacemark:(MKPlacemark*)place
{
    MapLocation*    theAnnotation = [map
annotationForCoordinate:place.coordinate];
    if (!theAnnotation)
        return;

   // Associate the placemark with the annotation.
    theAnnotation.placemark = place;

    // 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];
    }
}
```

```
- (void)reverseGeocoder:(MKReverseGeocoder*)geocoder
didFailWithError:(NSError*)error
{
    NSLog(@"Could not retrieve the specified place information.\n");
}
@end

@implementation MKMapView (GeocoderAdditions)

- (MapLocation*)annotationForCoordinate:(CLLocationCoordinate2D)coord
{
    // Iterate through the map view's list of coordinates
    // and return the first one whose coordinate matches
    // the specified value exactly.
    id<MKAnnotation> theObj = nil;

    for (id obj in [self annotations])
    {
        if (([obj isKindOfClass:[MapLocation class]]))
        {
            MapLocation* anObj = (MapLocation*)obj;

            if ((anObj.coordinate.latitude == coord.latitude) &&
                (anObj.coordinate.longitude == coord.longitude))
            {
                theObj = anObj;
                break;
            }
        }
    }

    return theObj;
}
@end
```

Displaying Maps

Introduced in iOS 3.0, the Map Kit framework lets you embed a fully functional map interface into your application. The map support provided by this framework includes many of the features normally found in the Maps application. You can display standard street-level map information, satellite imagery, or a combination of the two. You can zoom and pan the map programmatically, and the framework provides automatic support for the touch events that let users zoom and pan the map. You can also annotate the map with custom information.

To use the features of the Map Kit framework, you must link your application to `MapKit.framework` in your Xcode project. To access the classes and headers of the framework, include an `#import <MapKit/MapKit.h>` statement at the top of any relevant source files. For general information about the classes of the Map Kit framework, see *Map Kit Framework Reference*.

Important: 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.

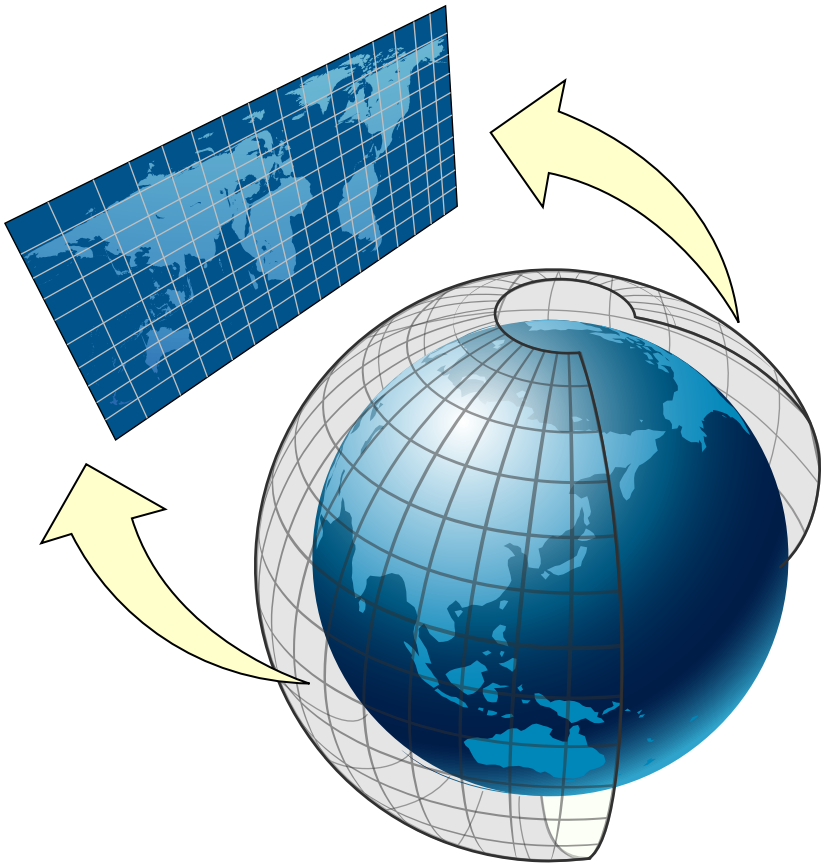
Understanding Map Geometry

A map view contains a flattened representation of a spherical object, namely the Earth. In order to use maps effectively, you need to understand a little bit about 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

In order 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 4-1 shows how the surface of the Earth can be mapped to a two-dimensional surface.

Figure 4-1 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 4-1 (page 28). 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 application, 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 `UIView` 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 application, 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 normally 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 typically use map points when specifying the shape of overlays. Table 4-1 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 4-1 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:` (`MKOverlayView`) `rectForMapRect:` (`MKOverlayView`)
Points	Map coordinates	`convertPoint: toCoordinateFromView:` (`MKMapView`) `convertRect: toRegionFromView:` (`MKMapView`)
Points	Map points	`mapPointForPoint:` (`MKOverlayView`) `mapRectForRect:` (`MKOverlayView`)

Adding a Map View to Your User Interface

The `MKMapView` class is a self-contained interface for presenting map data in your application. This class provides support for displaying map data, for managing user interactions, and for hosting custom content provided by your application. You should never subclass `MKMapView` but should only embed it as-is into your application's view hierarchy. You should also assign a delegate object to the map. The map view reports all relevant interactions to its delegate so that it has a chance to respond appropriately.

You can add a map view to your application 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 view hierarchy.

Because it is a view, you can manipulate a map view in the same ways you manipulate other views. You can change its size and position in your view hierarchy, you can configure its autoresizing behaviors, and you can add subviews to it. Unlike a view, you never handle touch events directly in a map view. 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 do not 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 33).

New maps are configured to accept user interactions and display map data only. 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 changing the value in the `mapType` property. If you want to limit user interactions, you can change the values in the `zoomEnabled` and `scrollEnabled` properties as well. If you want to respond to user interactions, you should do so using a delegate as described in "Responding to User Interactions with a Map" (page 32).

Configuring the Properties of a Map

The `MKMapView` class has a handful of properties that you can configure programmatically. These properties control important information such as which part of the map is currently visible 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 it is first created, a map's 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 at any time by assigning a new value to the `region` property. This property contains an `MKCoordinateRegion` structure, which has the following definition:

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

The interesting part of an `MKCoordinateRegion` structure is the span. The span is analogous to the width and height values of a rectangle but is 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 using meters, you can use the `MKCoordinateRegionMakeWithDistance` to create a region data structure using 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 cannot 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 adjust the `region` property accordingly. To find out the resulting region without actually changing the value in the `region` property, you can use the `regionThatFits:` method of the map view.

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), change the value in the `centerCoordinate` property of the map view or call the `setCenterCoordinate:animated:` method.
- 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.

If you only want to pan the map, you should only do so by modifying 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, to pan the map to the left by half the current map width, you could use the following code to find the coordinate at the left edge of the map and use that as the new center point, as shown here:

```
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. In other words 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, you should return `nil` from that method instead.

Responding to User Interactions with a Map

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. You provide this object in situations where you want to 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 33).

Annotating Maps

The `MKMapView` class implements an opaque view hierarchy for displaying a scrollable map. Although the map itself is scrollable, any subviews you add to a map view remain fixed in place and do not scroll. If you want to affix content to the map itself, and thus have that content scroll along with the rest of the map, you must use annotations and overlays.

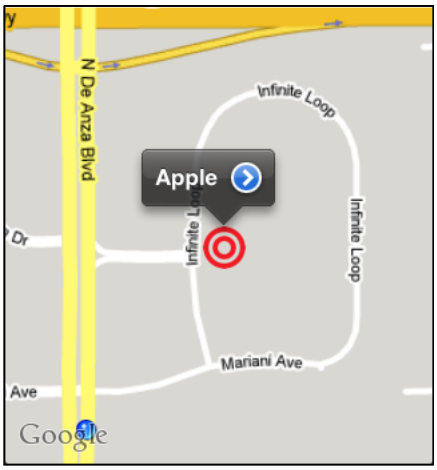
Annotations are used to display content that can be defined by a single coordinate point. By contrast, **overlays** are used to display content that is defined by any number of points and may constitute one or more contiguous or noncontiguous shapes. For example, you use annotations to represent information 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 traffic information, the boundaries of parks, lakes, cities, states, countries, or other bounded areas.

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 and means that 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 specific 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 5-1 shows an annotation that uses a custom image to highlight a particular location.

Figure 5-1 Displaying an annotation in a map



In order to display an annotation on a map, your application must provide two distinct objects:

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

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 application into an annotation object. In practice, it is 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 and you can also define custom annotation views if you want. However, you do not add annotation views directly to the map surface. Instead, you provide an annotation view when asked for it and let the map view incorporate that view into its opaque view hierarchy. You provide the annotation view using your map view delegate object.

The annotations you create are typically anchored to a single map coordinate that does not change. However, you can change the coordinate for an annotation programmatically as needed and can implement support to allow the user to drag annotations around the map. In iOS 4.0 and later, support for dragging annotations is even incorporated into the Map Kit classes; implementing this support in prior versions of the operating system requires some additional custom code on your part.

Checklist for Adding an Annotation to the Map

The steps for implementing and using annotations in your map-based application are shown below. These steps assume that your application 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 onscreen callout bubble.
    - Define a custom object that conforms to the `MKAnnotation` protocol, as described in "Defining a Custom Annotation Object" (page 35). This type of annotation can store any type of data you want.
2. Define an annotation view to present the data on screen. How you define your annotation view depends on your needs and may be one of the following:
    - If the annotation can be represented by a static image, create an instance of the `MKAnnotationView` class and assign the image to its `image` property; see "Using the Standard Annotation Views" (page 36).
    - If you want to use a standard pin annotation, create an instance of the `MKPinAnnotationView` class; see "Using the Standard Annotation Views" (page 36).
    - 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 37).

3. Implement the `mapView:viewForAnnotation:` method in your map view delegate.

   Your implementation of this method should dequeue an existing annotation view if one exists or create a new one. If your application supports multiple types of annotations, you must include logic in this method to create a view of the appropriate type for the provided annotation object. For more information about implementing this method, see "Creating Annotation Views from Your Delegate Object" (page 38).

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, this could result in your annotation views overlapping 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 application might display information only for major cities when the map is zoomed out to show the entire state. As the user zooms in, the application 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 "Managing the Map's Annotation Objects" (page 39).

Defining a Custom Annotation Object

The most important part of an annotation is the annotation object, which is an object that conforms to the `MKAnnotation` protocol. 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 need to represent additional information with the annotation, you need to define a custom annotation object.

A custom annotation object consists of a map coordinate and whatever other data you want to associate with the annotation. Listing 5-1 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 `coordinate` property, since it is read-only. Your own declaration would likely also include methods and properties defining the additional annotation data.

Listing 5-1 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
```

The implementation for your custom class must provide an implementation for the coordinate property and a way to set its value. Because `coordinate` is a declared property, you can synthesize the code needed to implement it easily enough using the `@synthesize` keyword. All that remains is to implement the code for the custom `initWithLocation:` method, which is shown in Listing 5-2.

Listing 5-2 Implementing the MyCustomAnnotation class

```
@implementation MyCustomAnnotation
@synthesize coordinate;

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

Important: When you implement the `coordinate` property in your class, it is recommended that you synthesize its creation. If you choose to implement the methods for this property yourself, or if you manually modify the variable underlying that property in other parts of your class after the annotation has been added to the map, be sure to send out key-value observing (KVO) notifications when you do. Map Kit uses KVO notifications to detect changes to the `coordinate`, `title`, and `subtitle` properties of your annotations and make any needed changes to the map display. If you do not 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 an example of an annotation object that is based on a Core Data object, see the sample code project *WeatherMap*.

Using the Standard Annotation Views

Using one of the standard annotation views is the easiest way to present annotations on your map. The `MKAnnotationView` class is a concrete view that defines the basic behavior for all annotation views. The `MKPinAnnotationView` class is a subclass of `MKAnnotationView` that displays one of the standard system pin images at the associated annotation's coordinate point.

The `MKAnnotationView` class is perfect for situations where you have a static image that you want to display for an annotation. After creating an instance of this class, assign your custom image to the `image` property of the object. When the annotation is displayed, the image is displayed centered over the target map coordinate. If you do not want the image to be centered on the map coordinate, you can use the `centerOffset` property to move the center point horizontally and vertically in any direction. Listing 5-3 shows an example of how to create an annotation view with a custom image and offset.

Listing 5-3 Creating a standard annotation view

```
MKAnnotationView* aView = [[[MKAnnotationView alloc] initWithAnnotation:annotation
                                    reuseIdentifier:@"MyCustomAnnotation"]
autorelease];
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 38).

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 two ways. You can continue to use the `image` property of `MKAnnotationView` but change the image at regular intervals, or you can also override the view's `drawRect:` method and draw your content dynamically every time. As with any custom drawing you do in a view, you should always consider performance before choosing an approach. Custom drawing gives you the most flexibility but if most of your content is fixed, using images may still be faster.

If you choose to define a custom annotation view, you subclass like any other view. The only difference is that instead of subclassing `UIView`, you subclass `MKAnnotationView` as shown in Listing 5-4.

Listing 5-4 Declaring a custom annotation view

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

@interface MyCustomAnnotationView : MKAnnotationView
{
   // Custom data members
}
// Custom properties and methods.
@end
```

When drawing content using the `drawRect:` method, you must always remember to specify a nonzero frame size for your annotation view shortly after initialization. The default initialization method for annotation views does not take a frame rectangle as a parameter. Instead, it uses the image you specify in the `image` property to set that frame size later. If you do not set an image, though, you must set the `frame` property of the view explicitly in order for your rendered content to be visible, as shown in Listing 5-5. Because the view draws in only part of its frame, it also sets its opaque property to `NO` so that the remaining map content shows through. If you do not do this, the drawing system fills your view with the current background color before calling your `drawRect:` method.

Listing 5-5 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 all other 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 as needed 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, you need to set up a timer to fire at periodic intervals and update your view.

For information on how to set up timers, see *Timer Programming Topics*. For information about how views draw content in iOS, see *View Programming Guide for iOS*.

Creating Annotation Views from Your Delegate Object

When is needs an annotation view, the map view calls the `mapView:viewForAnnotation:` method of its delegate object. If you do not 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 the method and create your views there.

Before trying to create a new view in your `mapView:viewForAnnotation:` method, you should always check to see if a similar annotation view already exists. Like table views, the map view has the option of caching unused annotation views that it is not using. When it does this, it makes the unused views available from the `dequeueReusableAnnotationViewWithIdentifier:` method. If this method returns a value other than `nil`, you should update the view's attributes and return it. If the method returns `nil`, just create a new instance of the appropriate annotation view class. In both cases, it is your responsibility to take the annotation passed to this method and assign it to your annotation view. You should also use this method to update the view before returning it.

Listing 5-6 shows a sample implementation of the `mapView:viewForAnnotation:` method. This method 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, setting up the basic properties of the view and configuring an accessory view for the annotation's callout. 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.

Listing 5-6 Creating annotation views

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView
                    viewForAnnotation:(id <MKAnnotation>)annotation
{
    // If it's 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:@"CustomPinAnnotation"]
                            autorelease];
            pinView.pinColor = MKPinAnnotationColorRed;
            pinView.animatesDrop = YES;
            pinView.canShowCallout = YES;

            // Add a detail disclosure button to the callout.
            UIButton* rightButton = [UIButton buttonWithType:
                                  UIButtonTypeDetailDisclosure];
            [rightButton addTarget:self action:@selector(myShowDetailsMethod:)
                                  forControlEvents:UIControlEventTouchUpInside];
            pinView.rightCalloutAccessoryView = rightButton;
        }
        else
            pinView.annotation = annotation;

        return pinView;
    }

    return nil;
}
```

Managing the Map's Annotation Objects

If your application works with more than a few annotations, you might need to think about how you manage those objects. The map view does not make any distinction between active and inactive annotations; it considers all annotation objects it knows about to be active. 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. And if your map includes hundreds of annotations, zooming out far enough could lead to a visually unappealing mass of annotation views. Even worse, the views may be so close together that the user cannot 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.

In iOS 4.0 and later, Map Kit includes numerous functions to 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 absolute distances 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 *Map Kit Functions Reference*.

Marking Your Annotation View as Draggable

In iOS 4.0 and later, annotation views provide built-in dragging support. This support 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, you must do the following:

- 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.

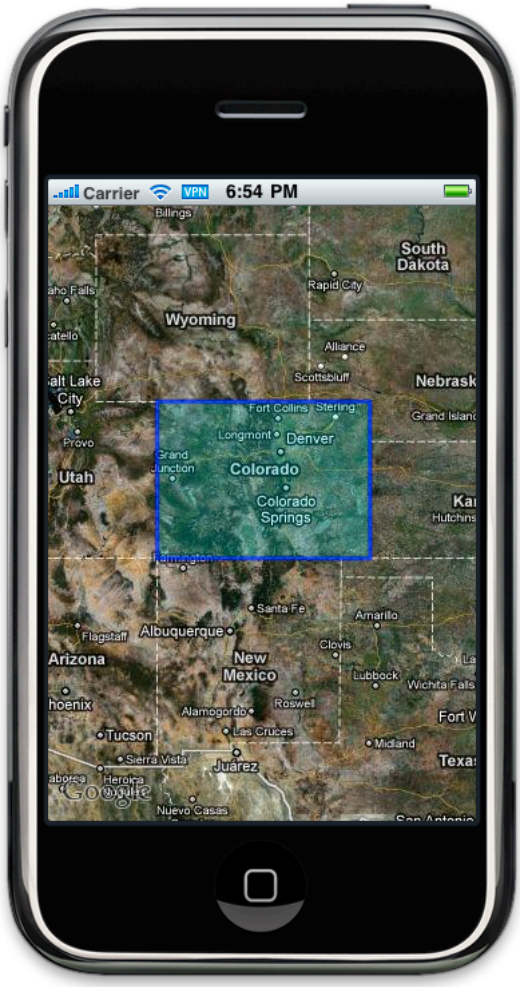
If you want to animate your view during a drag operation, you can do that by implementing a custom `dragState` method in your annotation view. As the map view processes drag-related touch events, it updates the `dragState` property of the affected annotation view. Implementing a custom `dragState` method gives you a chance to intercept these changes and perform additional actions, such as animate 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.

If you need to support draggable annotations in earlier versions of iOS, you must implement the support for it yourself. For information and sample code showing how to do it, see "Legacy Map Techniques" (page 49).

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 5-2 shows a filled and stroked overlay covering the state of Colorado.

Figure 5-2 Displaying an overlay on a map



In order to display an overlay on a map, your application must provide two distinct objects:

- An object that conforms to the `MKOverlay` protocol and manages the data points for the overlay. (This object is the **overlay object**.)
- A view (derived from the `MKOverlayView` class) used to draw the visual representation of the overlay on the map surface. (This is the **overlay view**.)

Overlay objects are typically 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 application 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 views that are capable of drawing any shapes represented by the concrete overlay objects. Like annotations, you do not add overlay views directly to the map surface. Instead, you provide an overlay view when asked for it and let the map view incorporate that view into its opaque view hierarchy. You provide the annotation view using your map view delegate object.

Once defined, the position of an overlay on the map typically never changes. Although it is possible to create draggable overlays, doing so is rare and you would need to implement the code to track the dragging operation and update the overlay coordinate points yourself.

Checklist for Adding an Overlay to the Map

Here are the steps for implementing and using overlays in your map-based application. These steps assume that your application 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.
    - Subclass `MKShape` or `MKMultiPoint` to create overlays that provide application-specific behaviors or use custom shapes.
    - Use an existing class from your application and make it conform to the `MKOverlay` protocol.
2. Define an overlay view to present on the screen using one of the following options:
    - For standard shapes, use the `MKCircleView`, `MKPolygonView`, or `MKPolylineView` to represent the annotation. You can customize many of the drawing attributes of the final shape using these classes.
    - For custom shapes descended from `MKShape`, define an appropriate subclass of `MKOverlayPathView` to render the shape.
    - For all other custom shapes and overlays, subclass `MKOverlayView` and implement your custom drawing code.
3. Implement the `mapView:viewForOverlay:` 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. In addition, you can rearrange their Z-ordering of overlays in a map to ensure that specific overlays are always displayed on top of others.

Using the Standard Overlay Objects and Views

If all you want to do is highlight a specified map region, the standard overlay classes are 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 `MKCircleView`, `MKPolygonView`, or `MKPolylineView` classes, which handle the rendering of that shape on the map surface.

Listing 5-7 shows an example of how you would create the rectangular polygon shown in Figure 5-2 (page 41). 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 5-7 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];
```

In order for an overlay can be shown on the map, the `mapView:viewForOverlay:` method of your map view delegate needs to provide an appropriate overlay view. For the standard overlay shapes, you can do this by creating a view that matches the type of shape you want to display. Listing 5-8 shows an implementation of this method that creates the polygon view 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 5-8 Creating a polygon view for rendering a shape

```
- (MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(id
<MKOverlay>)overlay
{
    if ([overlay isKindOfClass:[MKPolygon class]])
    {
        MKPolygonView*    aView = [[[MKPolygonView alloc]
initWithPolygon:(MKPolygon*)overlay] autorelease];

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

        return aView;
    }

    return nil;
}
```

It is important to remember that the standard overlay views are there to simply fill and stroke the shape represented by the overlay. If you want to display additional information, you need to create a custom overlay view 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.

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 a couple of options for defining custom overlays. You can subclass `MKShape` or `MKMultiPoint` to define new types of shape-based overlays or you can adopt the `MKOverlay` protocol

into one of your application's existing classes. The choice between the two techniques depends mostly on whether you already have a class with the data you want. If you do, you should incorporate the protocol into your existing class; otherwise, subclass `MKShape` or `MKMultiPoint` to create a custom shape subclass.

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

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

Of the two pieces of information, the bounding rectangle is the one that is most important 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 view 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 real work involved with displaying an overlay is incurred by the corresponding overlay view object. The overlay object simply defines where on the map the overlay should be placed, whereas the overlay view defines the final appearance of the overlay, including what information (if any) is displayed for the overlay. The creation of custom overlay views is described further in "Defining a Custom Overlay View" (page 44).

Defining a Custom Overlay View

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 view. Custom overlays give you the opportunity to draw any content you want. For example, if you are drawing a traffic overlay, you could use a custom overlay view 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 view, you must subclass `MKOverlayView`. (If you simply want to modify the drawing behavior of an existing shape-based overlay, you can subclass `MKOverlayPathView` instead.) In your custom implementation, 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 view 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. Unlike drawing in a normal view, drawing in an overlay view involves some special considerations, including the following:

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

  Also, you typically do not 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, you must 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 are done 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 5-9 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 5-3 (page 46).

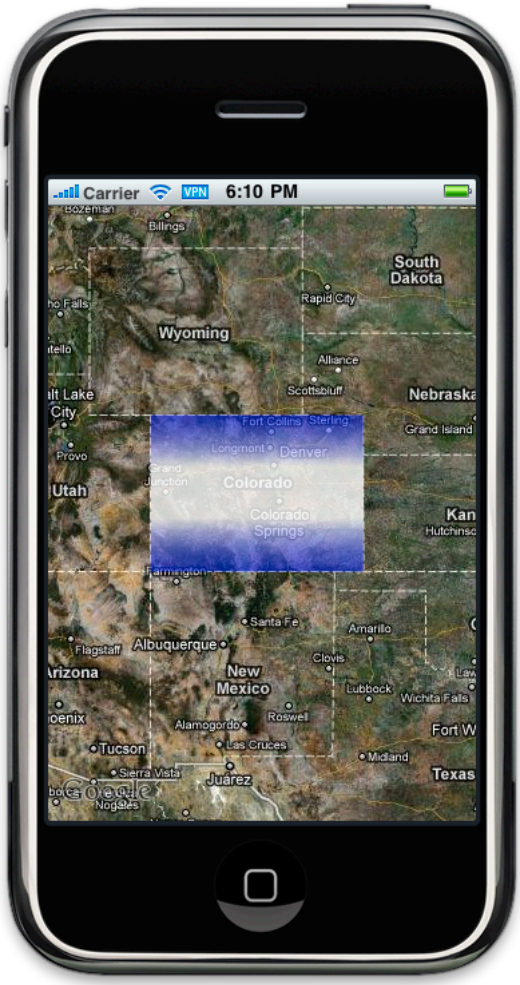
Listing 5-9 Drawing a gradient in a custom overlay view

```
- (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 5-3 shows the results of drawing custom content over the overlay for the state of Colorado. In this case, the overlay view fills its content with a custom gradient.

Figure 5-3 Using a custom overlay view to draw



Creating Overlay Views from Your Delegate Object

When it needs an overlay view, the map view calls the `mapView:viewForOverlay:` method of its delegate object. If you do not implement this method, or if you implement it and always return `nil`, the map view does not display anything for the specified overlay. Therefore, you must implement this method and return a valid overlay view for any overlays you want displayed on the map.

For the most part, every overlay is different. Although you should always create your overlay views in your `mapView:viewForOverlay:` method, you may need to be a little more creative in how you configure those views. If all of your views share the same drawing attributes, you can implement this method in a way similar to the one shown in Listing 5-8 (page 43). 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 this method.

Because overlays are typically different from one another, the map view does not recycle those views when they are removed from the map. Instead of dequeueing an existing overlay view, you must create a new overlay view every time.

Managing the Map's Overlay Objects

If your application 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 do not automatically overlap one another. This means, 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:didAddOverlayViews:` method of your map view delegate. When this method is called, you can use the `MKMapRectIntersectsRect` function to see if the added overlay intersects the bounds of any other overlays. If there is an overlap, use whatever custom logic is needed to choose which one should be placed on top in the rendering tree and exchange positions as needed. (Because the map view is an interface item, any modifications to the `overlays` array should be synchronized and performed on the application's main thread. The actual comparisons may occur on a different thread, though.)

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, you are responsible for managing that object in two places. If you want to display both an overlay view and annotation view for it, you must implement both the `mapView:viewForOverlay:` and `mapView:viewForAnnotation:` methods in your application delegate. It also means that you must add and remove the object from both the `overlays` and `annotations` arrays of your map.

Legacy Map Techniques

Creating Draggable Annotations in Earlier Versions of iOS

If you want to support draggable annotations in iOS 3.x, you must implement the code for tracking drag-related touches yourself. Although they live in a special layer above the map content, annotation views are full-fledged views capable of receiving touch events. You can use these events to detect hits within the view and initiate a drag operation.

Note: Because maps are displayed in a scrolling interface, there is typically a short delay between the time the user touches your custom view and the time corresponding events are delivered. This delay gives the underlying scroll view a chance to determine whether the touch event is part of a scrolling gesture.

The following sequence of code listings shows you how to implement a user-movable annotation view in iOS 3.x. In this example, the annotation view displays a bulls-eye image directly over the annotation's coordinate point and includes a custom accessory view for displaying details about the target.

Listing A-1 shows the definition of the `BullseyeAnnotationView` class. The class includes some additional member variables that it uses during tracking to move the view correctly. It also stores a pointer to the map view itself, the value for which is set by the code in the `mapView:viewForAnnotation:` method when it creates or reinitializes the annotation view. The map view object is needed when event tracking is finished to adjust the map coordinate of the annotation object.

Listing A-1 The `BullseyeAnnotationView` class

```
@interface BullseyeAnnotationView : MKAnnotationView
{
    BOOL isMoving;
    CGPoint startLocation;
    CGPoint originalCenter;

    MKMapView* map;
}

@property (assign,nonatomic) MKMapView* map;

- (id)initWithAnnotation:(id <MKAnnotation>)annotation;

@end

@implementation BullseyeAnnotationView
@synthesize map;
- (id)initWithAnnotation:(id <MKAnnotation>)annotation
{
    self = [super initWithAnnotation:annotation
                reuseIdentifier:@"BullseyeAnnotation"];
    if (self)
```

```
    {
        UIImage*    theImage = [UIImage imageNamed:@"bullseye32.png"];
        if (!theImage)
            return nil;

        self.image = theImage;
        self.canShowCallout = YES;
        self.multipleTouchEnabled = NO;
        map = nil;

        UIButton*    rightButton = [UIButton buttonWithType:
                      UIButtonTypeDetailDisclosure];
        [rightButton addTarget:self action:@selector(myShowAnnotationAddress:)
                      forControlEvents:UIControlEventTouchUpInside];
        self.rightCalloutAccessoryView = rightButton;
    }
    return self;
}
@end
```

When a touch event first arrives in a bulls-eye view, the `touchesBegan:withEvent:` method of that class records information about the initial touch event, as shown in Listing A-2. It uses this information later in its `touchesMoved:withEvent:` method to adjust the position of the view. All location information is stored in the coordinate space of the superview.

Listing A-2 Tracking the view's location

```
@implementation BullseyeAnnotationView (TouchBeginMethods)
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // The view is configured for single touches only.
    UITouch* aTouch = [touches anyObject];
    startLocation = [aTouch locationInView:[self superview]];
    originalCenter = self.center;

    [super touchesBegan:touches withEvent:event];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch* aTouch = [touches anyObject];
    CGPoint newLocation = [aTouch locationInView:[self superview]];
    CGPoint newCenter;

    // If the user's finger moved more than 5 pixels, begin the drag.
    if ( (abs(newLocation.x - startLocation.x) > 5.0) ||
         (abs(newLocation.y - startLocation.y) > 5.0) )
         isMoving = YES;

    // If dragging has begun, adjust the position of the view.
    if (isMoving)
    {
        newCenter.x = originalCenter.x + (newLocation.x - startLocation.x);
        newCenter.y = originalCenter.y + (newLocation.y - startLocation.y);
        self.center = newCenter;
    }
    else    // Let the parent class handle it.
        [super touchesMoved:touches withEvent:event];
```

```
}
@end
```

When the user stops dragging an annotation view, you need to adjust the coordinate of the original annotation to ensure the view remains in the new position. Listing A-3 shows the `touchesEnded:withEvent:` method for the `BullseyeAnnotationView` class. This method uses the map member variable to convert the pixel-based point into a map coordinate value. Because the `coordinate` property of an annotation is normally read-only, the annotation object in this case implements a custom `changeCoordinate` method to update the value it stores locally and reports using the `coordinate` property. If the touch event was canceled for some reason, the `touchesCancelled:withEvent:` method returns the annotation view to its original position.

Listing A-3 Handling the final touch events

```
@implementation BullseyeAnnotationView (TouchEndMethods)
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (isMoving)
    {
        // Update the map coordinate to reflect the new position.
        CGPoint newCenter = self.center;
        BullseyeAnnotation* theAnnotation = self.annotation;
        CLLocationCoordinate2D newCoordinate = [map convertPoint:newCenter
                            toCoordinateFromView:self.superview];

        [theAnnotation changeCoordinate:newCoordinate];

        // Clean up the state information.
        startLocation = CGPointZero;
        originalCenter = CGPointZero;
        isMoving = NO;
    }
    else
        [super touchesEnded:touches withEvent:event];
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (isMoving)
    {
        // Move the view back to its starting point.
        self.center = originalCenter;

        // Clean up the state information.
        startLocation = CGPointZero;
        originalCenter = CGPointZero;
        isMoving = NO;
    }
    else
        [super touchesCancelled:touches withEvent:event];
}
@end
```

Document Revision History

This table describes the changes to *Location Awareness Programming Guide*.

Date	Notes
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.
2010-03-24	New document describing how to use location and map services in an application.

Document Revision History