29 Where Am I? Introducing Core Location

Core Location is a framework that allows applications to retrieve the location and heading of the device they are running on. To do this, Core Location can use a combination of a compass for heading, and either GPS, cellular radio, or WiFi technologies for location. Cellular radio and WiFi-based location is less accurate than GPS.

Applications cannot specify which method will be used, but they can specify a desired level of accuracy. Depending on the desired level of accuracy, Core Location tries to use the GPS hardware, cellular radio, or WiFi in that order.

This framework is not included in any of the standard iOS application templates. To use this framework in your code, you will need to add it manually to your project. You can do this from the Project Settings page in Xcode. Select the project node in the project navigator to display the settings page. On the settings page, switch to the Build Phases tab and click the + button under the Link Binary With Libraries category. Select CoreLocation.framework from the list of available frameworks (see Figure 29-1).

Core Location defines a manager class called `CLLocationManager` that you can use to interact with the framework. It allows you to specify the desired frequency and accuracy of location information. To receive location updates in an application, you need to create an instance of the `CLLocationManager` class and provide a delegate object to receive location updates and errors. This delegate object must implement the `CLLocationManagerDelegate` protocol.

The delegate object is often a view controller class but could also be any other class in your application. Using location hardware can have a significant drain on the device's batteries, and hence applications need to turn on and turn off receiving location updates.

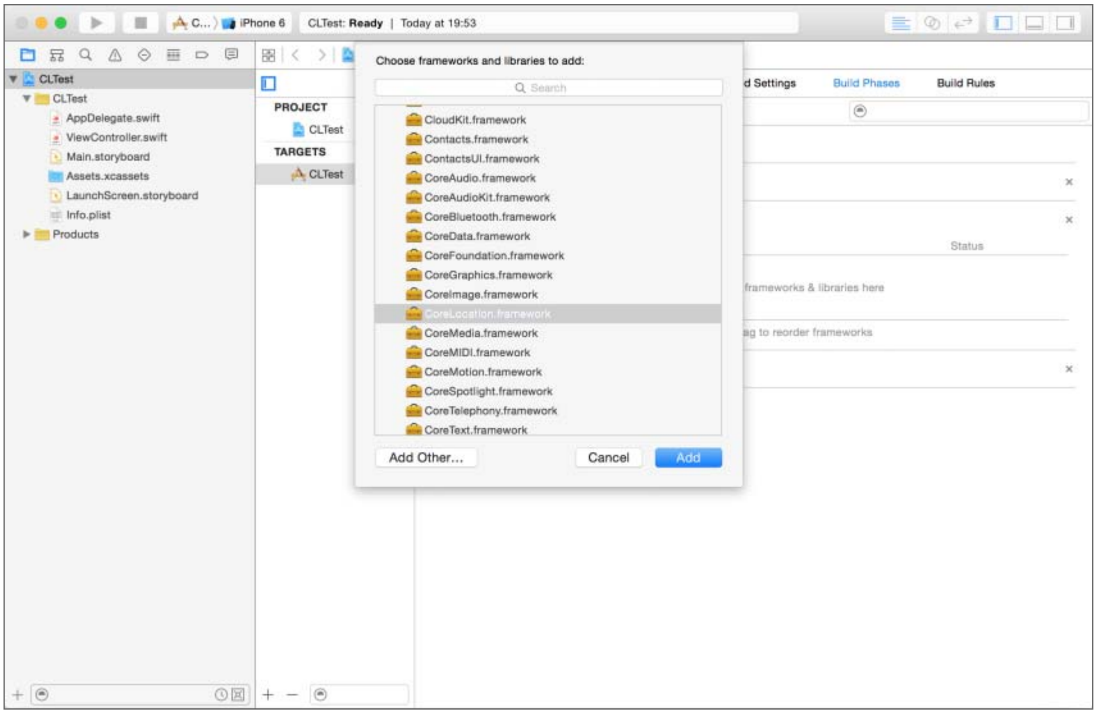


FIGURE 29-1

PERMISSIONS

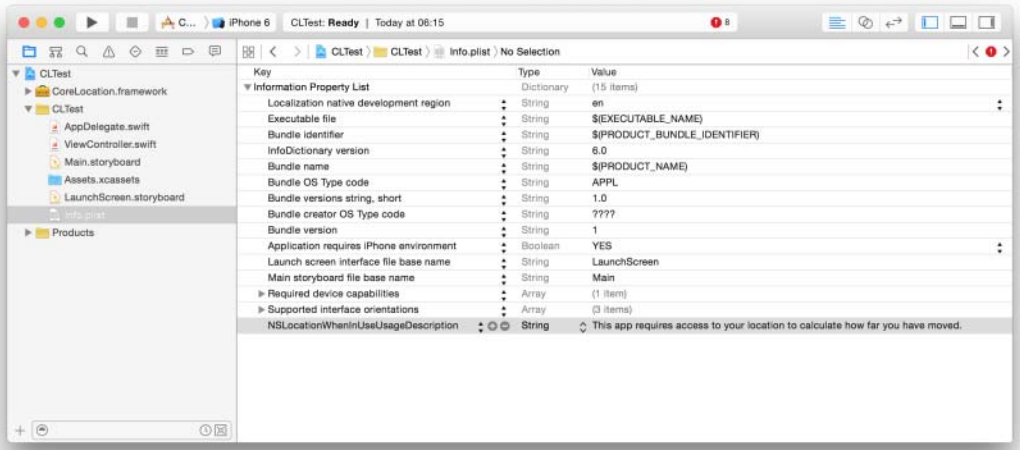
From iOS 8 onward, Apple requires that applications ask the user for permission before attempting to access location information. There are two types of permissions available:

- **Always Authorization:** For apps that need to access location information while in both the foreground and background modes
- **When In Use Authorization:** For apps that need only access location information in the foreground mode

You need to ask for either type of permission, but not both. The process of asking for permission has two parts. The first part involves adding a key to the `Info.plist` file that contains some text that will be presented to the user while asking for permission. This text should describe the reason for application requiring access to location data.

Depending on the type of permission you wish to ask for, you need to add either of the following keys to the `Info.plist` file (see Figure 29-2).

- `NSLocationAlwaysUsageDescription`
- `NSLocationWhenInUseUsageDescription`



Key	Type	Value
Information Property List	Dictionary	(15 items)
Localization native development region	String	en
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
NSLocationWhenInUseUsageDescription	String	This app requires access to your location to calculate how far you have moved.

FIGURE 29-2

The second part requires that you make a call to either the `requestWhenInUseAuthorization()` or `requestAlwaysAuthorization()` class methods of `CLLocationManager` and find out the user's decision in a delegate method:

```
func locationManager(manager: CLLocationManager,
    didChangeAuthorizationStatus status: CLAuthorizationStatus)
```

The user's decision is returned in the second parameter of this delegate method and can be one of the following:

- `Denied`
- `AuthorizedAlways`
- `AuthorizedWhenInUse`

The following code assumes that you wish to request When-in-use authorization, and demonstrates the basic setup required to receive location updates:

```
let locationManager = CLLocationManager()
locationManager.delegate = self
locationManager.desiredAccuracy = kCLLocationAccuracyBestForNavigation
locationManager.requestWhenInUseAuthorization()

...
...
...

func locationManager(manager: CLLocationManager,
    didChangeAuthorizationStatus status: CLAuthorizationStatus)
{
        var shouldAllow = false

        switch status {
            case CLAuthorizationStatus.AuthorizedWhenInUse:
                shouldAllow = true
            case CLAuthorizationStatus.AuthorizedAlways:
                shouldAllow = true
            default:
                 shouldAllow = false
        }

        if shouldAllow == true {
            manager.startUpdatingLocation()
        }

    }
```

If your application just cannot function without location services, add the `UIRequiredDeviceCapabilities` key to the `Info.plist` file. This key is a dictionary and can contain a list of strings each of which describe a single capability required by your application. The App Store examines the information in this key when users try to download your app and will prevent users from downloading your application to devices that don't contain the listed features.

The values to include for location service hardware are:

- `location-services:` Your application requires location services in general.
- `gps:` Your application requires the accuracy offered only by GPS hardware.

ACCURACY

An application can set up the `desiredAccuracy` property of the `CLLocationManager` instance to specify a desired accuracy. Core Location will try its best to achieve the desired accuracy. The more accurate a reading required, the more battery power is needed.

Applications should, in general, try to use the least accuracy possible to satisfy their requirements. The property can have the following values, listed in decreasing order of accuracy:

- `kCLLocationAccuracyBestForNavigation`
- `kCLLocationAccuracyBest`
- `kCLLocationAccuracyNearestTenMeters`
- `kCLLocationAccuracyHundredMeters`
- `kCLLocationAccuracyKilometer`
- `kCLLocationAccuracyThreeKilometers`

An application can also set up the `distanceFilter` property of the `CLLocationManager` instance to specify the minimum distance in meters a device must move before an update is provided to the application.

The default value of this property is `kCLDistanceFilterNone`, which specifies the application wants to know of all movements.

RECEIVING LOCATION UPDATES

To start receiving location updates, you must call the `startUpdatingLocation()` method on the `CLLocationManager` instance.

```
locationManager.startUpdatingLocation()
```

When your application does not want to receive location updates, it must call the `stopUpdating Location` method of the `CLLocationManager` instance:

```
locationManager.stopUpdatingLocation()
```

The `CLLocationManagerDelegate` protocol defines two methods that are used by an application to handle a location update:

```
func locationManager(manager: CLLocationManager,
     didUpdateLocations locations: [AnyObject])

func locationManager(manager: CLLocationManager,
     didFailWithError error: NSError)
```

A typical implementation of the `locationManager(manager: CLLocationManager, didUpdate-Locations locations: [AnyObject])` would resemble the following:

```
func locationManager(manager: CLLocationManager,
     didUpdateLocations locations: [AnyObject])
{
    let locationArray = locations as NSArray
    for newLocation in locationArray
    {
        // lat/lon values should only be considered if
        // horizontalAccuracy is not negative.
        if newLocation.horizontalAccuracy >= 0
        {
            let currentLatitude:CLLocationDegrees =
               newLocation.coordinate.latitude;

            let currentLongitude:CLLocationDegrees =
               newLocation.coordinate.longitude;

            // do something with currentLatitude and currentLongitude.
        }

        // altitude values should only be considered if
        // verticalAccuracy is not negative.
        if (newLocation.verticalAccuracy >= 0)
        {
            let currentAltitude:CLLocationDegrees = newLocation.altitude;

            // do something with currentAltitude
        }

    }

}
```

The `locationManager(manager: CLLocationManager, didUpdateLocations locations: [AnyObject])` method's arguments are the `CLLocationManager` instance, and an array of location updates in chronological order. Each element in this array is an instance of `CLLocation`.

A `CLLocation` object encapsulates a location. It contains a `coordinate` property that is a structure containing a `latitude` and `longitude` member, each expressed as `CLLocationDegrees` values. `CLLocationDegrees` is an alias for a floating-point (decimal) value.

The location object also has the `horizonalAccuracy` property that signifies the radius of a circle centered at the coordinate property. The device can be anywhere within this circle. A larger `horizontalAccuracy` implies a larger circle, and thus a less accurate measurement. If the `horizontalAccuracy` property is negative, the reading should be discarded as being inaccurate.

The `CLLocation` object also provides altitude information using two properties: `altitude` and `verticalAccuracy`. A positive `altitude` value is a height above sea level, and a negative altitude is below sea level. A positive `verticalAccuracy` implies that the altitude measurement is off that amount; a negative value implies an invalid altitude measurement.

NOTE *Although the location updates are served to your delegate in chronological order, the* `horizontalAccuracy` *and* `verticalAccuracy` *values may vary across the updates. In general, if you wait for more updates, the accuracy of the readings will increase. When using the* `startUpdatingLocation()` *method, you need to provide custom logic in your application to pick the reading with the best accuracy and this can be a tradeoff between taking an earlier but somewhat inaccurate location, or waiting until you get a sufficiently accurate reading.*

Starting with iOS9, Core Location has a new method called `requestLocation()`, which can be used to get a single location reading. When you call `requestLocation()`, behind the scenes Core Location will collect a number of readings and provide you one that it feels is reasonably accurate.

`requestLocation()` is mutually exclusive with `startUpdatingLocation()` with the latter taking precedence. Thus, if you call `startUpdatingLocation()` while a previous call to `requestLocation()` hasn't completed, the call to `requestLocation()` will automatically be cancelled.

You can measure the distance between two locations using the `distanceFromLocation()` method of the `CLLocation` class. The distance in meters is expressed as a `CLLocationDistance` value, which is also an alias for a floating-point value:

```
Let distanceTravelled = oldLocation.distanceFromLocation(newLocation)
```

To compute the distance of a location update from a fixed point, you can instantiate a `CLLocation` object that represents the fixed point and use the `distanceFromLocation()` method as normal. For example, if you want to find out the distance of a location update from the center of London (lat = 51.5001524, lon = –0.1262362), you can use code similar to the following:

```
let londonLocation = CLLocation(latitude: 51.5001524, longitude: -0.1262362)

let distanceTravelled = londonLocation.distanceFromLocation(newLocation as!
                        CLLocation)
```

HANDLING ERRORS AND CHECKING HARDWARE AVAILABILITY

If Core Location is unable to get a location fix, your delegate's `locationManager(manager: CLLocationManager, didFailWithError error: NSError)` method will be called. The error argument is of type `NSError`. Its `code` property can be examined to determine the reason for failure:

- `kCLErrorDenied`: The user has denied access to location data.
- `kCLErrorLocationUnknown`: Core Location has tried, but could not get a location fix.
- `kCLErrorNetwork`: There is no means for Core Location to get a location fix.

If the user has denied access to Core Location, then the `CLLocationManager` will not try to get a location fix again, and in such a case, it is best to call the `stopUpdatingLocation()` method to the instance.

Some location services require the presence of specific hardware on the device. In general, you must check whether the desired service is available before attempting to use it. Table 29-1 lists some of the methods provided by the `CLLocationManager` class to test service availability.

TABLE 29-1: CLLocationManager Service Availability Methods

METHOD	DESCRIPTION
`func locationServicesEnabled() -> Bool`	Returns True if location services are enabled on the device. The user can disable location services from device settings.
`func isMonitoringAvailableForClass(regionClass: AnyClass) -> Bool`	Returns True if region monitoring is supported on the current device for the specific type of region.
`isRangingAvailable() -> Bool`	Returns True if ranging is supported on the current device.
`func headingAvailable() -> Bool`	Returns True if the location manager is able to generate heading-related events.
`func significantLocationChangeMonitoringAvailable() -> Bool`	Returns True if significant location change monitoring is available on the current device.
`func authorizationStatus() -> CLAuthorizationStatus`	Returns a value indicating whether an application is authorized to use location services.

NOTE *The iOS Simulator can simulate either a device at a fixed location or a device that is moving along one of three preset routes. These features can be accessed from the Debug ➪ Location menu of the iOS Simulator.*

GEOCODING AND REVERSE GEOCODING

Geocoding involves converting between a latitude/longitude coordinate pair and an address. Core Location provides the `CLGeocoder` class that provides methods to perform both forward and reverse geocoding. Forward-geocoding involves converting from an address to a latitude/longitude value. Reverse-geocoding involves converting a latitude/longitude value into an address. The result of a geocoding request is represented by a `CLPlacemark` object. A forward-geocoding request returns an array of `CLPlacemark` objects because multiple results may be returned.

You should try to use one geocoding request per action and avoid making the same geocoding request multiple times. To perform a forward-geocoding request from an address string, you can call the `geocodeAddressString(addressString: String, completionHandler: CLGeocodeCompletionHandler)` method on a geocoder instance. This message requires you to specify a String object that contains an address string and a block handler that is called when the geocoding operation is complete. The following code snippet converts an address string into a latitude/longitude coordinate pair:

```
let localGeocoder = CLGeocoder()
let addressString = "170 Bilton Road, Perivale, UB6 7HL, United Kingdom"

localGeocoder.geocodeAddressString(addressString) { (placemarks:[CLPlacemark]?,
                                   error:NSError?) -> Void in

    if placemarks != nil
    {
        let firstPlacemark = placemarks!.first
        let latValue = firstPlacemark!.location.coordinate.latitude;
        let lonValue = firstPlacemark!.location.coordinate.longitude;
    }
}
```

You can send the geocoder a reverse-geocoding request by calling the `reverseGeocodeLocation (location: CLLocation, completionHandler: CLGeocodeCompletionHandler)` method, as shown in the following snippet:

```
let localGeocoder = CLGeocoder()
let londonLocation = CLLocation(latitude: 51.5001524, longitude: -0.1262362)

localGeocoder.reverseGeocodeLocation(londonLocation) {
              (placemarks:[CLPlacemark]?,
               error:NSError?) -> Void in

if placemarks != nil {

            let firstPlacemark = placemarks!.first

            let countryCode = firstPlacemark!.ISOcountryCode
            let countryName = firstPlacemark!.country
            let adminArea = firstPlacemark!.administrativeArea
            let city = firstPlacemark!.locality
            let postCode = firstPlacemark!.postalCode
            let streetAddress1 = firstPlacemark!.thoroughfare
        }
};
```

The message requires you to provide a `CLLocation` object that represents a latitude/longitude coordinate pair and block handler that is called with the results of the reverse-geocoding operation. The `CLLocation` instance in this example is created with a fixed set of coordinates (lat=51.5001524, lon=–0.1262362) but could have just as well been obtained from a location update.

The actual geocoding operation is performed asynchronously. The results are supplied as an array of `CLPlacemark` objects, but in this case, the array will contain just one element. If an error occurred, the array is `nil` and the error variable contains more information on the error.

A `CLPlacemark` object contains several properties that encapsulate information on an address associated with a specific coordinate. Some of the properties are:

- `location`: A `CLLocation` object that provides the coordinate pair associated with the placemark
- `ISOcountryCode`: An `NSString` object that contains the abbreviated country code
- `country`: An `NSString` object that contains the name of country
- `postalCode`: An `NSString` object that contains the postal code
- `administrativeArea`: An `NSString` object that contains the state/province
- `locality`: An `NSString` object that contains the city
- `thoroughfare`: An `NSString` object that contains the street address
- `subThoroughfare`: An `NSString` object that contains additional street address information

If the coordinates lie over an inland water body, or an ocean, this information can be accessed through the `inlandWater` and `ocean` properties, respectively, both of which are `String` objects.

OBTAINING COMPASS HEADINGS

You can determine if a compass is available on a device by calling the `headingAvailable()` method of the location manager. If a compass is available on the device, you can use the location manager to receive heading updates. Heading updates work much like location updates. Once you have set up the `CLLocationManager` instance, you can call the `startUpdatingHeading()` and `stopUpdatingHeading()` methods to begin receiving heading updates.

The `CLLocationManagerDelegate` protocol defines two methods that are related to heading updates:

```
func locationManager(manager: CLLocationManager,
      didUpdateHeading newHeading: CLHeading)

func locationManagerShouldDisplayHeadingCalibration(manager: CLLocationManager)
     -> Bool
```

Heading data is supplied as a `CLHeading` object to the `locationManager(manager: CLLocationManager, didUpdateHeading newHeading: CLHeading)` delegate method. The `CLHeading` class encapsulates the magnetic heading, the true heading, and an accuracy measure in its `magneticHeading`, `trueHeading`, and `headingAccuracy` properties, respectively.

The earth's geographic North Pole is different from the magnetic north pole. The geographic North Pole is fixed at the North Pole, whereas the magnetic north pole is a few hundred miles away. Make sure you know the difference between geographic north and magnetic north when you build any application that uses the compass feature.

The geographic North Pole heading is contained in the `trueHeading` member of the `CLHeading` instance. Data in this member is available only if you enable both heading updates and location updates.

The `locationManagerShouldDisplayHeadingCalibration(manager: CLLocationManager)` is called on the delegate object when the location manager wants to display a calibration prompt to the user. If you find this prompt annoying, you can implement this method to return `NO`. If you were to do so, the compass would try to calibrate itself automatically but the results of the calibration process might not be accurate.

NOTE *The iOS Simulator cannot simulate compass headings. You need to test applications that require this feature on an actual device.*

TRY IT

In this Try It, you create a simple iPhone application based on the Single View Application template called `CLTest` that displays the current location and the distance traveled since the last location reading was obtained.

Lesson Requirements

- Launch Xcode.
- Create a new iPhone project based on the Single View Application template.
- Add a few `UILabel` elements that will display the location readings. Create outlets for these in the view controller class.
- Add a `UIButton` that will be used to stop/start receiving location updates. Create an appropriate outlet and action.
- Initialize Core Location when the button is pressed. Stop receiving location updates when the button is pressed a second time.
- Implement `CLLocationManagerDelegate` methods.

REFERENCE *The code for this Try It is available at* `www.wrox.com/go/swiftios`.

Hints

- You must add the `NSLocationWhenInUseUsageDescription` key to the `Info.plist` file in the project. The value of this key should be a string that describes what your application will do with the user's location data.
- Before calling `startUpdatingLocation()` on the `CLLocationManager` instance, you must check if the user has allowed your app to access location data.
- When creating a new project, you can use your website's domain name as the Company Identifier in the Project Options dialog box.
- You will need to add a reference to the Core Location framework to the project.
- To show the Object library, select View ➪ Utilities ➪ Show Object Library.
- To show the assistant editor, select View ➪ Assistant Editor ➪ Show Assistant Editor.

Step-by-Step

- Create a Single View Application in Xcode called `CLTest`.
  1. Launch Xcode and create a new application by selecting File ➪ New ➪ Project.
  2. Select the Single View Application template from the list of iOS project templates.
  3. In the project options screen use the following values:
     - **Product Name:** CLTest
     - **Organization Name:** your company
     - **Organization Identifier:** com.yourcompany
     - **Language:** Swift
     - **Devices:** iPhone
     - **Use Core Data:** Unchecked
     - **Include UI Tests:** Unchecked
     - **Include Unit Tests:** Unchecked
  4. Save the project onto your hard disk.
- Add a reference to the Core Location framework.
  1. In Xcode, make sure the project navigator is visible. To show it, select View ➪ Navigators ➪ Show Project Navigator.
  2. Click the root (project) node of the project navigator to display project settings.
  3. Select the Build Phases tab.
  4. Expand the Link Binary With Libraries group in this tab.

5. Click the + button at the bottom of this group and select `CoreLocation.framework` from the list of available frameworks.
6. Click the Add button.

➤ Add UI elements to the default scene.

1. Open the `Main.storyboard` file in the Interface Editor.
2. From the Object library, drag and drop six labels onto the scene and place to resemble Figure 29-3.

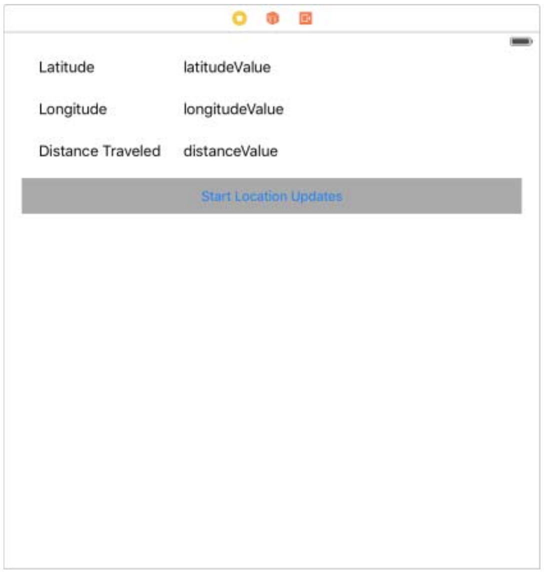


FIGURE 29-3

3. Double-click each label in turn and change its text to Latitude, Longitude, Distance Traveled, latitudeValue, longitudeValue, and distanceValue respectively.
4. Create layout constraints for each of the elements on the storyboard scene using the information in Table 29-2. When creating layout constraints using the pin constraints dialog box, ensure the Constrain to margins option is unchecked and Update Frames is set to Items of New Constraints.

TABLE 29-2: Layout Constraints

ELEMENT	LEFT	TOP	WIDTH	HEIGHT
Latitude label	39	8	63	21
Longitude label	39	26	77	21

ELEMENT	LEFT	TOP	WIDTH	HEIGHT
Distance Travelled label	39	26	141	21
latitudeValue label	99	8	99	21
longitudeValue label	85	26	114	21
distanceValue label	21	26	107	21

5. Using the assistant editor, create outlets for the `latitudeValue`, `longitudeValue`, and `distanceValue` labels. Call these outlets `latitudeValue`, `longitudeValue`, and `distanceValue`, respectively.

- Add a `UIButton` instance to start/stop receiving location updates.

  1. Ensure the Object library is visible. You can show it by selecting View ➪ Utilities ➪ Show Object Library.
  2. Use the Object library to add a `UIButton` instance and place it below the labels.
  3. Double-click the button and set its title to Start Location Updates.
  4. Select the button and display the pin constraints dialog box. Ensure the Constrain to margins options is unchecked and Update Frames is set to Items of New Constraints. Create the following layout constraints:
     - Left: 20
     - Top: 20
     - Right: 20
     - Height: 40
  5. Using the assistant editor, create an outlet called `toggleButton` in the `ViewController` class and connect it to the button.
  6. Using the assistant editor, create an action method in the view controller class and connect it to the Touch Up Inside event of the button. Call the new method `onButtonPressed`.

- Add code to receive location updates to the View controller class.

  1. Open the `ViewController.swift` file in the project explorer.
  2. Import the Core Location framework into the view controller.
  3. Ensure the following import statements are located at the top of the `ViewController` class:

     ```
     import UIKit
     import CoreLocation
     ```

  4. Add the following variable declarations to the view controller class:

     ```
     var locationManager:CLLocationManager? = nil
     ```

```
var lastLocation:CLLocation? = nil
var isReceivingLocationUpdates:Bool = false
```

5. Ensure the View controller class implements the `CLLocationManager` delegate protocol by ensuring the class is declared as:

```
class ViewController: UIViewController, CLLocationManagerDelegate
```

6. Update the stub implementation of the `viewDidLoad` method to resemble the following:

```
override func viewDidLoad() {

    super.viewDidLoad()

    locationManager = CLLocationManager()
    locationManager!.delegate = self
    locationManager!.desiredAccuracy = kCLLocationAccuracyBestForNavigation

    lastLocation = CLLocation(latitude: 51.5001524, longitude: -0.1262362)

    toggleButton.titleLabel!.text = "Start location updates"
}
```

7. Update the empty implementation of the `onButtonPressed` method to resemble the following:

```
@IBAction func onButtonPressed(sender: AnyObject) {

    if isReceivingLocationUpdates == false
    {
        if CLLocationManager.authorizationStatus() !=
           CLAuthorizationStatus.AuthorizedWhenInUse
        {
            locationManager!.requestWhenInUseAuthorization()
        }
        else
        {
            isReceivingLocationUpdates = true
            toggleButton.titleLabel!.text = "Stop location updates"
            locationManager!.startUpdatingLocation()
        }
    }
    else
    {
        isReceivingLocationUpdates = false
        toggleButton.titleLabel!.text = "Start location updates"
        locationManager!.stopUpdatingLocation()
    }

}
```

8. Implement the `locationManager(manager: CLLocationManager, didChangeAuthorizationStatus status: CLAuthorizationStatus)` delegate method in the view controller class:

```
func locationManager(manager: CLLocationManager,
     didChangeAuthorizationStatus status: CLAuthorizationStatus)
{
    var shouldAllow = false

    switch status {
        case CLAuthorizationStatus.AuthorizedWhenInUse:
            shouldAllow = true
        case CLAuthorizationStatus.AuthorizedAlways:
            shouldAllow = true
        default:
            shouldAllow = false
    }

    if shouldAllow == true {
        isReceivingLocationUpdates = true
        toggleButton.titleLabel!.text = "Stop location updates"
        manager.startUpdatingLocation()
    }
}
```

9. Implement the `locationManager(manager: CLLocationManager, didUpdate Locations locations: [CLLocation])` delegate method in the view controller class:

```
func locationManager(manager: CLLocationManager,
     didUpdateLocations locations: [CLLocation])
{
    let locationArray = locations as NSArray
    for newLocation in locationArray
    {
        // lat/lon values should only be considered if
        // horizontalAccuracy is not negative.
        if newLocation.horizontalAccuracy >= 0
        {
            let currentLatitude:CLLocationDegrees =
                newLocation.coordinate.latitude;

            let currentLongitude:CLLocationDegrees =
                newLocation.coordinate.longitude;

            let distanceTravelled =
                newLocation.distanceFromLocation(lastLocation!)

            latitudeValue.text = "\(currentLatitude)"
            longitudeValue.text = "\(currentLongitude)"
            distanceValue.text = "\(distanceTravelled)"

            lastLocation = newLocation as? CLLocation
        }
    }
}
```

- Test your app in the iOS Simulator.
  1. Click the Run button in the Xcode toolbar. Alternatively you can select Project ➪ Run.
  2. Click the Start Location Updates button.
  3. Use the iOS Simulator's ability to simulate a device on the move by selecting Debug ➪ Location ➪ City Bicycle Ride.

REFERENCE *To see some of the examples from this lesson, watch the Lesson 29 video online at* `www.wrox.com/go/swiftiosvid`.