Backgrounding
Evolve Advanced Track, Chapter 9

Xamarin Inc.

Overview

Background processing or *backgrounding* is the process of letting applications perform tasks in the background while another application is running in the foreground.

Backgrounding in mobile applications is fundamentally different than the traditional concept of multitasking on the desktop. Desktop machines have a variety of resources available to an application, including screen real estate, power, and memory. Applications are able to run side-by-side and remain performant and usable. On a mobile device, resources are much more limited. It is difficult to show more than one application on a small screen, and running several applications at full speed would drain the battery. Backgrounding is a constant compromise between giving applications the resources to run the background tasks they require to perform well, and keeping the foregrounded application and the device responsive. Both iOS and Android have provisions for backgrounding, but they handle it in very different ways.

In iOS, backgrounding is recognized as an application state, and apps are moved in and out of the background state depending on the behavior of the app and the user. iOS also offers several options for wiring an app to run in the background, including asking the OS for time to complete an important task, or operating as a type of known background-necessary application that gets special backgrounding privileges.

In Android, Activities are moved in and out of the backgrounded state as part of the Activity Lifecycle. The OS will send inactive Activities into the background in order to conserve system resources. These Activities cannot perform background processing, and are usually stopped soon after entering the background. To solve the problem of running tasks in the background, Android introduces another application component called a *Service*. Services and are used to create, start, and stop tasks that run in the background, and to provide a programmatic interface for callers to interact with in the background.

In this chapter, we are going to learn to perform application tasks in the background in both iOS and Android. We will cover key concepts and best practices, and then walk through creating a real world app that receives location updates in the background.

Cross-Platform Considerations

Despite the differences between iOS and Android with regards to backgrounding, there are certain guidelines developers should keep in mind while writing applications that perform tasks in the background. A mobile device has limited processing power and resources, and these should be rationed wisely. Otherwise, an application could cause the battery to drain quickly, or cause other apps to feel sluggish and non-performant.

iOS is very strict about the amount of processing a backgrounded application can do, and will terminate any application that doesn’t comply with iOS rules. Android, on the other hand, places no hard limits on what sorts of tasks can run in the background, although the OS will kill off lower-priority Activities when resources are needed.

To ensure that the device remains responsive and that the application runs smoothly, it is a good idea to apply the same sanity checks and best practices to all applications, regardless of the device they are running on. We will cover some of these best practices in this chapter. For an overview of iOS backgrounding rules and guidelines, refer to the Being a Responsible Background App section of the Apple App States and Multitasking guide. While this document specifically covers iOS guidelines, many of these can be applied to Android and Windows development as well.

Let's learn about how backgrounding works in iOS, and then we'll look at Android.

Introduction to Backgrounding in iOS

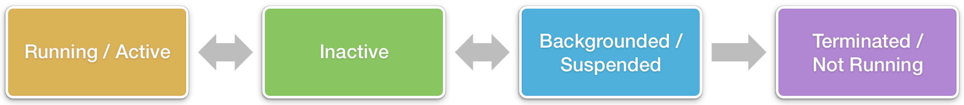
iOS regulates background processing very tightly, and offers three different approaches to implement it:

- **Register a Long-Running Task:** If an app needs to perform non-UI tasks that can't be interrupted, or needs extra time to wrap them up before being put in the background, it can ask iOS for time to complete them. For example, an application might need to finish uploading a file to a server, or save user data.
- **Register as a Known Background Application:** An app can register as a specific type of application that has known, specific backgrounding requirements. These apps include, *Audio*, *VoIP*, *External Accessory*, *Newsstand*, and *Location* applications. They are allowed continuous background processing privileges as long as they are performing tasks that are within the parameters of the registered application type.
- **Use Notifications:** Notifications are not a form of backgrounding per se; however, they give applications the ability to present users with an opportunity to pull an application out of a backgrounded state. Notifications are covered in Chapter 1.

Application States and Application Delegate Methods

Before we dive into the code for background processing in iOS, it's important to understand the lifecycle of an iOS application, and how backgrounding affects it.

An application can be in one of four states, as illustrated by the following diagram:



1. **Running/Active**: The app is in the foreground.
2. **Inactive**: The app is interrupted by an incoming phone call or text.

3. **Background/Suspended:** The user accepts an interruption, and the app continues operating in the background.
4. **Terminated/Not Running**: The application is shut down. Usually only happens in low-memory situations, or if the user decides to manually kill the app.

An application will move back and forth between states, depending on the behavior of the user and the resource requirements of the OS.

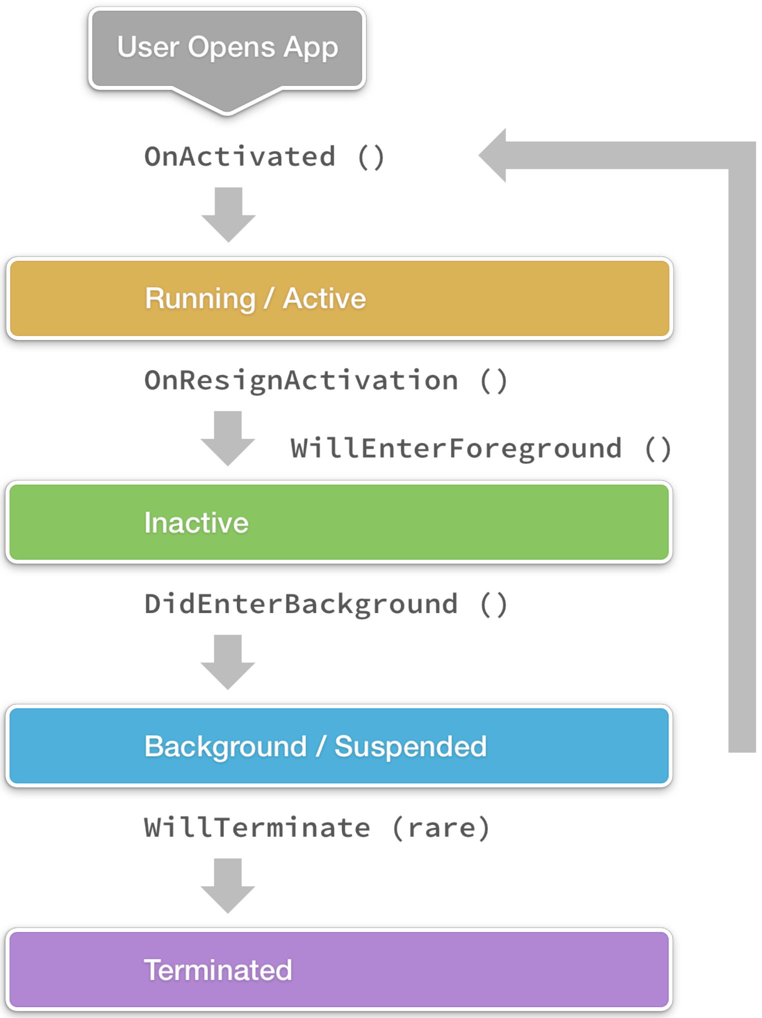
> **Note:** Since the introduction of multitasking support, iOS rarely terminates applications, preferring to keep them around in memory so that they may be re-launched faster. Applications will be terminated if they behave badly, or if resources have become extremely limited.

Application Lifecycle Methods

When an app changes state, iOS notifies the application through event methods in the `AppDelegate` class:

- `OnActivated`: This is called the first time the application is launched, and subsequently every time the app comes back into the foreground. This is the place to put any code that needs to run every single time the app is opened.
- `OnResignActivation`: If the user receives an interruption such as a text or phone call, this method gets called and the app is temporarily inactivated. Should the user accept the phone call, the app would be sent to the background.
- `DidEnterBackground`: Called when the app enters the backgrounded state, this method gives an application about five seconds to prepare for possible termination. Use this time to save user data and tasks, and remove sensitive information from the screen. Any attempt to update the UI from the background will result in iOS terminating the app.
- `WillEnterForeground`: If iOS does not terminate the app after it has moved to the background, the app will move back along the pipeline into the inactive state, and this method will get called. This is the time to prepare the app to take the foreground by rehydrating any state saved during `DidEnterBackground`. `OnActivated` will be called immediately after this method completes.
- `WillTerminate`: This only gets called if memory becomes constrained, or if the user manually kills the application.

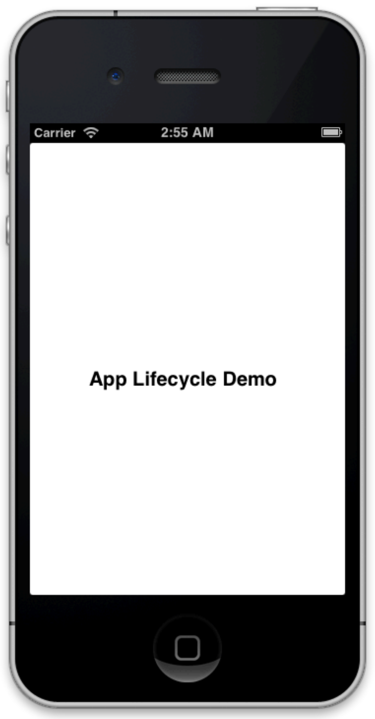
The following diagram illustrates how the application states and lifecycle methods fit together:



We will see the Application Lifecycle in action in the next section.

Application Lifecycle Demo

In this section, we are going to examine an application that demonstrates the four Application States, and the role of the `AppDelegate` methods in notifying the application of when states get changed. The application will print updates to the console whenever the app changes state:

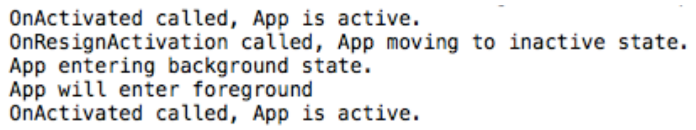


```
OnActivated called, App is active.
OnResignActivation called, App moving to inactive state.
App entering background state.
App will enter foreground
OnActivated called, App is active.
```

1. Open the *Lifecycle.iOS* project in the *LifecycleDemo* solution.
2. Open up the `AppDelegate` class. Note that we have added logging to the lifecycle methods to let us know when the application has changed state:

```
public override void OnActivated(UIApplication application)
{
        Console.WriteLine("OnActivated called, App is active.");
}
public override void WillEnterForeground(UIApplication application)
{
        Console.WriteLine("App will enter foreground");
}
public override void OnResignActivation(UIApplication application)
{
        Console.WriteLine("OnResignActivation called,
                App moving to inactive state.");
}
public override void DidEnterBackground(UIApplication application)
{
        Console.WriteLine("App entering background state.");
}
// [not guaranteed that this will run]
public override void WillTerminate(UIApplication application)
{
        Console.WriteLine("App is terminating.");
}
```

3. Launch the application in the simulator or on the device. `OnActivated` will be called when the app launches. The application is now in the Active state.
4. Hit the **Home** button on the simulator or device to bring the application to the background. `OnResignActivation` and `DidEnterBackground` will be called as the app transitions from Active to Inactive and into the Backgrounded state.
5. Navigate back to the app to bring it back into the foreground. `WillEnterForeground` and `OnActivated` will both be called:



```
OnActivated called, App is active.
OnResignActivation called, App moving to inactive state.
App entering background state.
App will enter foreground
OnActivated called, App is active.
```

Now that we understand iOS application states and transitions, let's look at the different options available for backgrounding in iOS.

iOS Backgrounding Techniques

Let's examine the different ways to perform background processing on iOS in more detail. First, we'll take a look at registering a long-running task to run in the background. Then, we will cover how to register an entire application for backgrounding privileges. Finally, we will walk through building and registering a Location application that gets continuous location updates in the background.

Registering a Long-Running Task

Some applications contain tasks that can't be interrupted by the iOS should the application change state. One way to protect these tasks from being interrupted is to register them with iOS as long-running tasks. For example, a great candidate for this pattern would be tasks such as uploading or downloading a file.

The following code snippet from the *Task.iOS* project in *BackgroundingDemo* demonstrates registering a task to run in the background:

```
int taskID =
UIApplication.SharedApplication.BeginBackgroundTask( () => {});

// perform your task on a new thread
new Task( { () = {
       DoWork();
       // tell the iOS you're done
       UIApplication.SharedApplication.EndBackgroundTask(taskID);
}).Start();
```

The registration process pairs a task with a unique identifier, `taskID`. To generate the identifier, we make a call to the `BeginBackgroundTask` method on the `UIApplication` object, and then start the long-running task on a new thread. When the task is complete, we call `EndBackgroundTask` and pass in the same identifier. This is important because iOS will terminate the application if a `BeginBackgroundTask` call does not have a matching `EndBackgroundTask`.

Avoiding App Termination With Expiration Handlers

If an application with registered tasks gets moved to the background, the registered tasks will get about ten minutes to run. If a long-running task exceeds this time before it calls `EndBackgroundTask`, iOS will terminate the app.

However, iOS provides a graceful way to handle background time expiration through an **Expiration Handler**. This is an optional block of code that will get executed when the time allotted for a task is about to expire. Code in the Expiration Handler should call `EndBackgroundTask` and pass the task ID, which will indicate that the app is behaving well and prevent iOS from terminating the app even if the task runs out of time.

The following is an example of a call to `BeginBackgroundTask` with an expiration handler:

```
int taskID =
UIApplication.SharedApplication.BeginBackgroundTask( () => {
       // do cleanup or stop task
       // call EndBackgroundTask and exit gracefully
       UIApplication.SharedApplication.EndBackgroundTask(taskID);

});
```

Accessing Task Time Remaining

We can check how much time the task has to complete using the static `BackgroundTimeRemaining` property of the `UIApplication` class. The following code will give us the time, in seconds, that our background task has left:

```
int timeRemaining =
UIApplication.SharedApplication.BackgroundTimeRemaining;
```

Note: Most applications will continue to exist in the backgrounded state even after all background tasks have completed. It's important to understand that while the application process will not be terminated after the 600 seconds allotted for a task to complete, the application still can't do any processing outside of the task.

Performing Tasks During DidEnterBackground

In addition to making a long-running task background-safe, registration can also be used to kick off tasks as an application is being put in the background. For example, tasks such as saving user data can be done here. iOS provides an event method in the AppDelegate class called `DidEnterBackground` that can be used for this purpose. An application has approximately five seconds to return from this method or it will get terminated. Therefore, tasks that might take more than 5 seconds to complete can be registered and invoked on a separate thread from inside the `DidEnterBackground` method.

The process is nearly identical to that of registering a long-running task. The following code snippet illustrates this in action:

```
public override void DidEnterBackground (UIApplication
application)
{
        int taskID =
        UIApplication.SharedApplication.BeginBackgroundTask( () =>
        {});

        new Task ( () => {
                DoWork();
                UIApplication.SharedApplication
                        .EndBackgroundTask(taskID);
        }).Start();
}
```

We begin by overriding the `DidEnterBackground` method in the `AppDelegate`, where we register our task via `BeginBackgroundTask`, as we did in the previous example. Next, we spawn a new thread and perform our long-running task. Note that the `EndBackgroundTask` call is now made from inside the long-running task.

Registering as a Background Necessary Application

Registering individual tasks for background privileges works for some applications, but what happens if an application is constantly called upon to perform important, long-running tasks, such as getting directions for the user via GPS? Applications such as these should instead be registered as a known background necessary application.

Registering an app signals iOS that the application should be given special privileges necessary to perform certain tasks in the background.

Registered apps can fall into several categories:

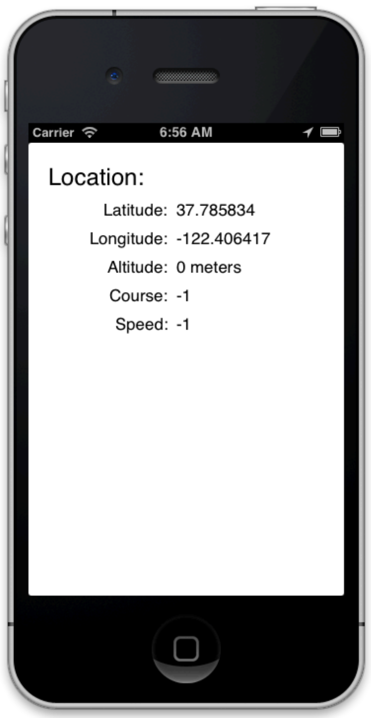
- Audio: Music players and other applications that work with audio content may be registered to continue playing audio even when the app is no longer in the foreground. If an app in this category attempts to do anything other than play audio or download while in the background, iOS will terminate it.
- VoIP: Voice Over Internet Protocol (VoIP) applications get of the same privileges granted to audio applications to keep processing audio in the background. They are also allowed to respond as needed to the VoIP services that power them in order to keep their connections alive.
- External Accessories and Bluetooth: Reserved for applications that need to communicate with Bluetooth devices and other external hardware accessories, registration under these categories allows the app to stay connected to the hardware.
- Newsstand: A Newsstand application can continue to sync content in the background.

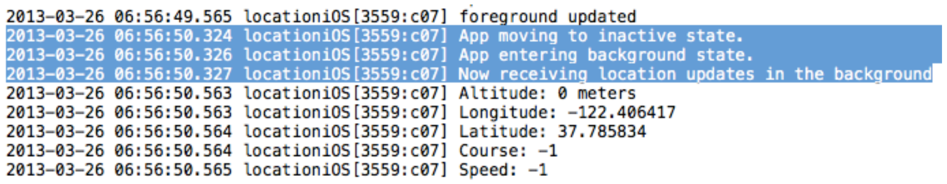
- **Location:** Applications that make use of GPS or network location data can send and receive location updates in the background.

In the next section, we will look at an example that registers a Location application, and runs location updates in the background.

iOS Location Application Walkthrough

In this example, we are going to build an iOS Location application that prints information about our current location: latitude, longitude, and other parameters to the screen. This application will demonstrate how to properly perform location updates while the application is either Active or Backgrounded:





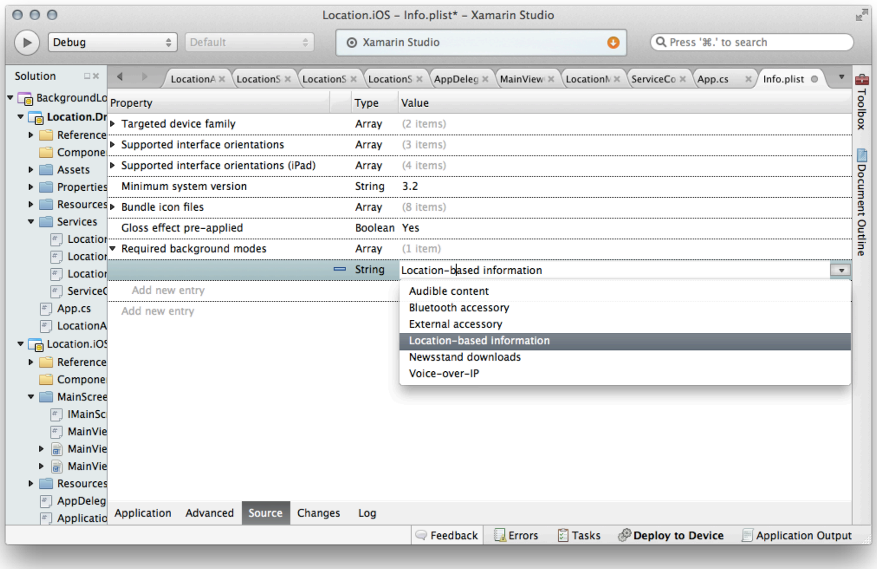
```
2013-03-26 06:56:49.565 locationiOS[3559:c07] foreground updated
2013-03-26 06:56:50.324 locationiOS[3559:c07] App moving to inactive state.
2013-03-26 06:56:50.326 locationiOS[3559:c07] App entering background state.
2013-03-26 06:56:50.327 locationiOS[3559:c07] Now receiving location updates in the background
2013-03-26 06:56:50.563 locationiOS[3559:c07] Altitude: 0 meters
2013-03-26 06:56:50.563 locationiOS[3559:c07] Longitude: -122.406417
2013-03-26 06:56:50.564 locationiOS[3559:c07] Latitude: 37.785834
2013-03-26 06:56:50.564 locationiOS[3559:c07] Course: -1
2013-03-26 06:56:50.565 locationiOS[3559:c07] Speed: -1
```

During the walkthrough, we will touch on some key backgrounding concepts, including registering an app as a background-necessary application, suspending UI updates when the app is backgrounded, and working with the `WillEnterBackground` and `WillEnterForeground` `AppDelegate` methods:

1. First, create a new iOS project. We're going to call it *Location.iOS*.
2. A location application qualifies as a background-necessary application in iOS. We can register the application as a Location application by editing the `Info.plist` file for our project.

Under **Solution Explorer**, double click on the **Info.plist** file to open it, and navigate to the **Source** panel. Then click **Add new entry**. From the **Property** dropdown, select **Required Background Modes**. From the **Value** dropdown, select **Location-based information**.

In Xamarin Studio, it will look like something like this:



Property	Type	Value
Targeted device family	Array	(2 items)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)
Minimum system version	String	3.2
Bundle icon files	Array	(8 items)
Gloss effect pre-applied	Boolean	Yes
Required background modes	Array	(1 item)
	String	Location-based information

In Visual Studio, **Info.plist** needs to be updated manually by adding the following key/value pair:

```
<key>UIBackgroundModes</key>
<array>
        <string>location</string>
</array>
```

3. Now that the application is registered, we need to get location data from the device to our application. In iOS, we can do this with the help of a `CLLocationManager`. The `CLLocationManager` provides a key class of Core Location and will raise events giving us location updates.

   In our code, let's create a new class called **LocationManager** that provides a single place for various screens and code to subscribe to location updates. In the **LocationManager** class, make an instance of the `CLLocationManager` called `LocMgr`:

```
public class LocationManager
{
        public LocationManager ()
        {
                this.locMgr = new CLLocationManager();
        } protected CLLocationManager locMgr;

        public CLLocationManager LocMgr
        {
                get { return this.locMgr; }
        }
}
```

4. Inside the `LocationManager` class, create a method called `StartLocationUpdates` with the following code. We will use this to set the update parameters and start receiving location updates from the `CLLocationManager`:

   ```
   public void StartLocationUpdates()
   {
           // we need the user’s permission to use GPS, so we check to make
           // sure they’ve accepted
           if (CLLocationManager.LocationServicesEnabled) {
                   //set the desired accuracy, in meters
                   LocMgr.DesiredAccuracy = 1;
                   LocMgr.StartUpdatingLocation();
           } else {

                  Console.WriteLine ("Location services not enabled");
           }
   }
   ```

   There are several important things happening in this method. First, we perform a check to see if the application has access to location data on the device. We verify this by calling `LocationServicesEnabled` on the `CLLocationManager`. This method will return false if the user has denied the application access to location information.

   Next, we will tell the location manager how often to update. `CLLocationManager` provides many options for filtering and configuring location data, including the frequency of updates. In this example, we set the `DesiredAccuracy` to update whenever the location changes by a meter. For more information on configuring location update frequency and other preferences, refer to the CLLocationManager Class Reference in the Apple documentation.

   Finally, call `StartUpdatingLocation` on the `CLLocationManager` instance. This tells the location manager to get an initial fix on the current location, and to start sending updates

5. So far, we’ve created a location manager, specified the kinds of data we want to receive, and gotten an initial fix on our location. Now we need a way to get the location data to our View. We can do this with a custom event that takes a `CLLocation` as an argument:

   ```
   // event for the location changing
   public event EventHandler<LocationUpdatedEventArgs> LocationUpdated =
           delegate { };
   ```

   The next step is to subscribe to location updates from the `CLLocationManager`, and raise the custom `LocationUpdated` event when new location data becomes available, passing in the location as an argument.

   This is complicated by a recent change in the `CLLocationManager` API: iOS 6 deprecated the `UpdatedLocation` event and introduced a new one called `LocationsUpdated`. To make the application compatible with different versions of the OS, we need to wire up the event inside a system version check, as illustrated by the example below:

```
if (CLLocationManager.LocationServicesEnabled) {

        //set the desired accuracy, in meters
        LocMgr.DesiredAccuracy = 1;

        if (UIDevice.CurrentDevice.CheckSystemVersion (6, 0)) {
                LocMgr.LocationsUpdated += (object sender,
                CLLocationsUpdatedEventArgs e) =>
                {
                        // fire our custom Location Updated event
                        LocationUpdated (this, new LocationUpdatedEventArgs
                        (e.Locations [e.Locations.Length - 1]));
                };
        } else {
                // this will be called pre-iOS 6
                LocMgr.UpdatedLocation += (object sender,
                CLLocationUpdatedEventArgs e) =>
                {
                        this.LocationUpdated (this,
                        new LocationUpdatedEventArgs (e.NewLocation));
                };

        }

        LocMgr.StartUpdatingLocation();
}
```

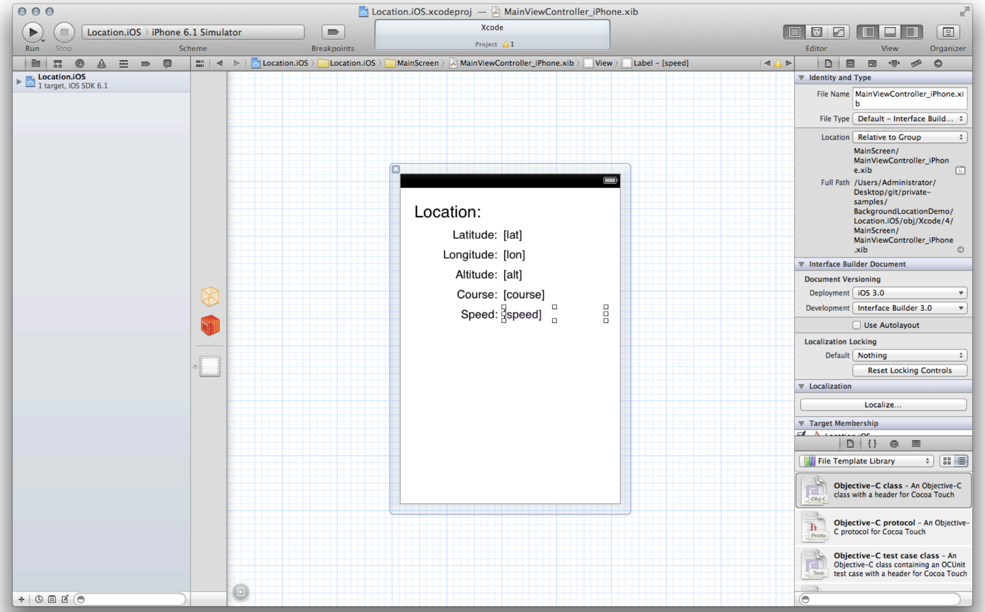
Note that `LocMgr.UpdatedLocation` will trigger a warning at compile time when building with iOS 6 and above, telling us that the call is obsolete/deprecated, however, we can safely ignore it.

6. Now we're ready to build our screen that will display our location. Create a new iPhone View Controller called `MainViewController`, and open the acoompanying `.xib` in Interface Builder.

   In Interface Builder, drag several labels onto the screen to act as placeholders the location information we want to display. In our example, we've wired up labels for latitude, longitude, altitude, course, and speed using the following outlets:

Latitude	`LblLatitude`
Longitude	`LblLongitude`
Altitude	`LblAltitude`
Course	`LblCourse`
Speed	`LblSpeed`

The layout should resemble the following:


Location.iOS.xcodeproj — MainViewController_iPhone.xib
Run
Stop
Location.iOS › iPhone 6.1 Simulator
Scheme
Breakpoints
Xcode
Project ⚠1
Editor
View
Organizer
Location.iOS › Location.iOS › MainScreen › MainViewController_iPhone.xib › View › Label - [speed]
Location.iOS
1 target, iOS SDK 6.1
Location:
Latitude: [lat]
Longitude: [lon]
Altitude: [alt]
Course: [course]
Speed: [speed]
Identity and Type
Interface Builder Document
Document Versioning
Localization Locking
Reset Locking Controls
Localization
Localize...
Target Membership
File Template Library
Objective-C class - An Objective-C class with a header for Cocoa Touch
Objective-C protocol - An Objective-C protocol for Cocoa Touch
Objective-C test case class - An Objective-C class containing an OCUnit test case with a header for Cocoa Touch

File Name: MainViewController_iPhone.xib
File Type: Default - Interface Build...
Location: Relative to Group
MainScreen/MainViewController_iPhone.xib
Full Path: /Users/Administrator/Desktop/git/private-samples/BackgroundLocationDemo/Location.iOS/obj/Xcode/4/MainScreen/MainViewController_iPhone.xib
Deployment: iOS 3.0
Development: Interface Builder 3.0
☐ Use Autolayout
Default: Nothing

Note: The sample application uses an interface called `IMainScreen` for the `UILabel` data. This interface allows us to use a different `.xib` for iPad and iPhone with the same code.

7. Inside the `FinishedLaunching` method in the **AppDelegate**, create a new instance of the **LocationManager** and call `StartLocationUpdates` on it:

```
public override bool FinishedLaunching (UIApplication app,
        NSDictionary options)
{
        ...
        LocationManager Manager = new LocationManager();
        Manager.StartLocationUpdates();
        ...
}
```

This will start the location updates on application start-up, although no data will be displayed. We'll address this in the next step.

8. Now that we've got location updates going, we can begin updating the screen with location data. The following method gets the location from our `LocationUpdated` event and prints it to the screen:

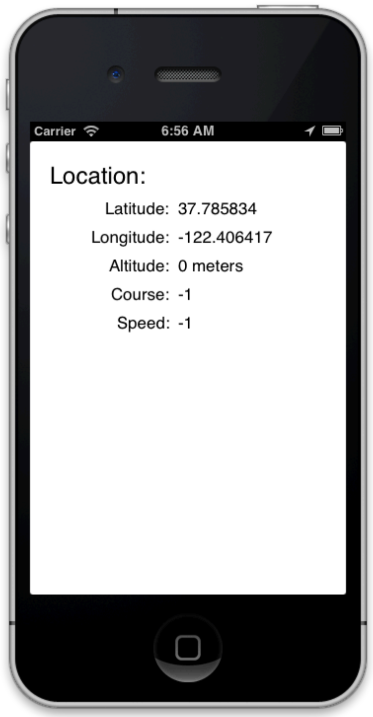
```
public void HandleLocationChanged (object sender,
        LocationUpdatedEventArgs e)
{
        // handle foreground updates

        CLLocation location = e.Location;
        IMainScreen ms = mainScreen;

        ms.LblAltitude.Text = location.Altitude + " meters";
        ms.LblLongitude.Text = location.Coordinate.Longitude.ToString ();
```

```
            ms.LblLatitude.Text = location.Coordinate.Latitude.ToString ();
            ms.LblCourse.Text = location.Course.ToString ();
            ms.LblSpeed.Text = location.Speed.ToString ();

            Console.WriteLine ("foreground updated");
      }
```

We still need to subscribe to the `LocationUpdated` event in our AppDelegate, and call the new method to update the UI. We can do this in `ViewDidLoad`, right after the `StartLocationUpdates` call:

```
        Manager.LocationUpdated += HandleLocationChanged;
```

Now, if we run it, the application should look something like this:


Carrier 6:56 AM

Location:

```
    Latitude: 37.785834
   Longitude: -122.406417
    Altitude: 0 meters
      Course: -1
       Speed: -1
```

9. The application is printing location updates while it is Active; it's time to see what happens when the app enters the background. Let's start by overriding the `AppDelegate` methods that track application state changes, so that the application can notify us when it transitions between the foreground and the background:

```
public override void DidEnterBackground (UIApplication application)
{
        Console.WriteLine ("App entering background state.");
}

public override void WillEnterForeground (UIApplication application)
{
        Console.WriteLine ("App will enter foreground");
}
```

We will need to verify that location updates continue when the app is in the background. Add the following code in the LocationManager to continuously print updated location data to the application output:

```
public class LocationManager
{
        public LocationManager ()
        {
                ...
                LocationUpdated += PrintLocation;
        }
        ...
        //This will keep going in the background and the foreground
        public void PrintLocation (object sender,
                LocationUpdatedEventArgs e)
        {
                CLLocation location = e.Location;
                Console.WriteLine ("Altitude: " + location.Altitude
                                      + " meters");
                Console.WriteLine ("Longitude: "
                                      + location.Coordinate.Longitude);
                Console.WriteLine ("Latitude: "
                                      + location.Coordinate.Latitude);
                Console.WriteLine ("Course: " + location.Course);
                Console.WriteLine ("Speed: " + location.Speed);
        }
}
```

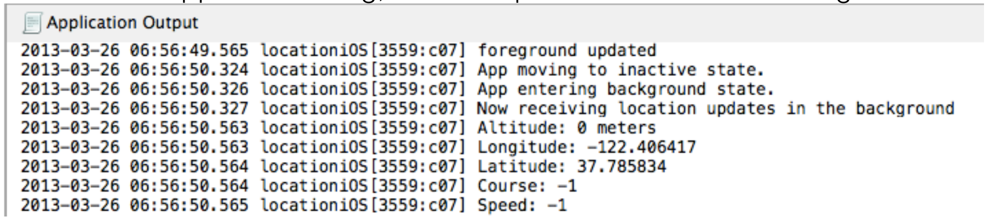
10. Our app works pretty well, but there's one major issue. If we attempt to update the UI when the app is backgrounded, iOS will terminate it. So we need to make sure that when the app goes into the background, we unsubscribe to location updates in our screen and stop updating it.

    iOS provides us notifications when the app is about to transition to a different application states. In this case, we can subscribe to the `ObserveDidEnterBackground` Notification.

    The following code snippet shows how to use a Notification to let the View know when to halt UI updates. This will go in ViewDidLoad:

    ```
    UIApplication.Notifications.ObserveDidEnterBackground ((sender, args) => {
            Manager.LocationUpdated -= HandleLocationChanged;
    });
    ```

    When the app is running, the output will look something like this:



Application Output

```
2013-03-26 06:56:49.565 locationiOS[3559:c07] foreground updated
2013-03-26 06:56:50.324 locationiOS[3559:c07] App moving to inactive state.
2013-03-26 06:56:50.326 locationiOS[3559:c07] App entering background state.
2013-03-26 06:56:50.327 locationiOS[3559:c07] Now receiving location updates in the background
2013-03-26 06:56:50.563 locationiOS[3559:c07] Altitude: 0 meters
2013-03-26 06:56:50.563 locationiOS[3559:c07] Longitude: -122.406417
2013-03-26 06:56:50.564 locationiOS[3559:c07] Latitude: 37.785834
2013-03-26 06:56:50.564 locationiOS[3559:c07] Course: -1
2013-03-26 06:56:50.565 locationiOS[3559:c07] Speed: -1
```

11. The application prints location updates to the screen when operating in the foreground, and continues to print data to the application output window while operating in the background.

    Only one outstanding issue remains: our screen starts UI updates when the app is first loaded, but it has no way of knowing when the app has re-entered the foreground. If the backgrounded application is brought back into the foreground, UI updates won't resume.

To fix this, we nest our call to start UI updates inside another Notification, which will fire when the application enters the Active state:

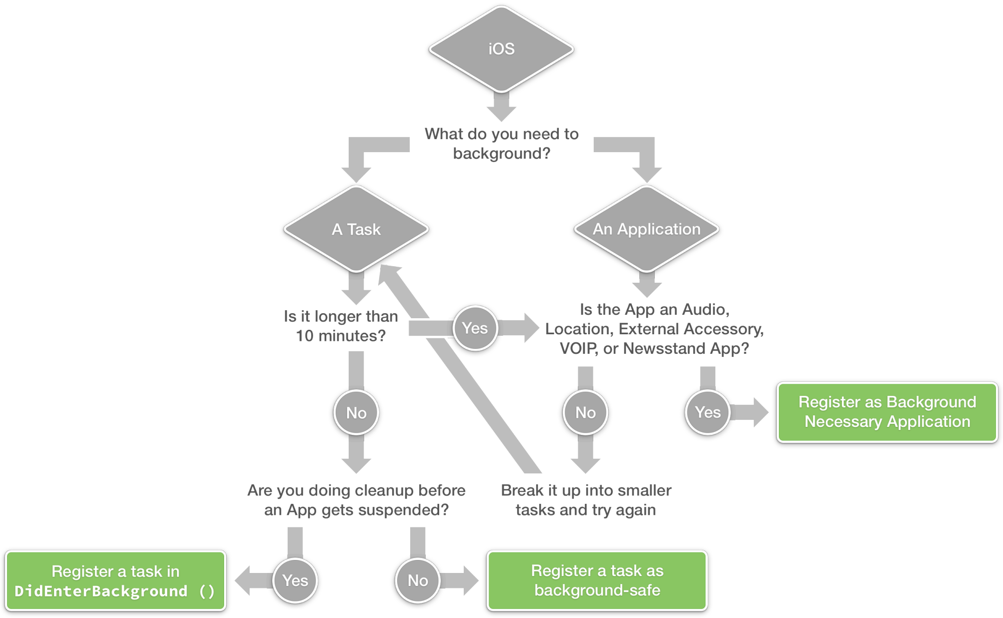
```
UIApplication.Notifications.ObserveDidBecomeActive ((sender, args) => {
        Manager.LocationUpdated += HandleLocationChanged;
});
```

Now the UI will begin updating when the application is first started, and resume updating any time the app comes back into the foreground.

In this section, we have built a well-behaved, background-aware iOS application that prints location data to both the screen and the application output window.

iOS Backgrounding Guidance

Refer to the following diagram for choosing which backgrounding technique to use in iOS:


iOS
What do you need to background?
A Task
An Application
Is it longer than 10 minutes?
Yes
Is the App an Audio, Location, External Accessory, VOIP, or Newsstand App?
No
No
Yes
Register as Background Necessary Application
Are you doing cleanup before an App gets suspended?
Break it up into smaller tasks and try again
Register a task in DidEnterBackground ()
Yes
No
Register a task as background-safe

Now let's switch gears and take a look at backgrounding in Android.

Introduction to Backgrounding in Android

Android Activities can be in different states depending on the behavior of the user and the needs of the OS. The Activity Lifecycle is a response to the changes in state that an Activity goes through: Activities are created, moved in and out of the background, and destroyed as necessary. The notion of an Activity being in a backgrounded state is misleading, however: Activities can't do any actual processing while in the background. For this reason, background processes in Android are typically handled via application components known as Services, which are UI-less classes that can run independent of the Activity lifecycle.

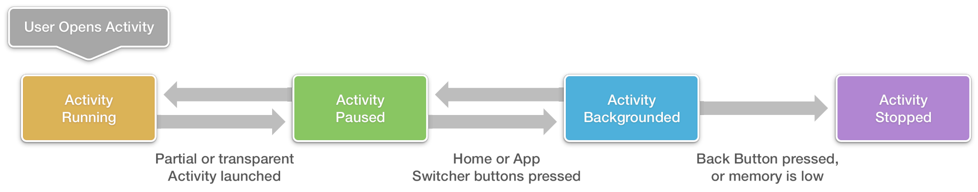
Additionally, Activities have a set of responsibilities during their lifecycles that, when adhered to, result in well-behaved application that don't monopolize resources or cause application crashes. Let's do a quick review of the Activity Lifecycle as it relates to application state changes and backgrounding.

Activity Lifecycle

The Android Activity lifecycle comprises a collection of methods exposed within the Activity class that provide the developer with a resource management framework. This framework allows developers to meet the unique state management requirements of each Activity within an application and properly handle resource management.

Activity States

The Android OS arbitrates Activities based on their state. This helps Android identify activities that are no longer in use, allowing the OS to reclaim memory and resources. The following diagram illustrates the states an Activity can go through, during its lifetime:


User Opens Activity
Activity Running
Activity Paused
Activity Backgrounded
Activity Stopped
Partial or transparent Activity launched
Home or App Switcher buttons pressed
Back Button pressed, or memory is low

These states can be broken into 4 main groups as follows:

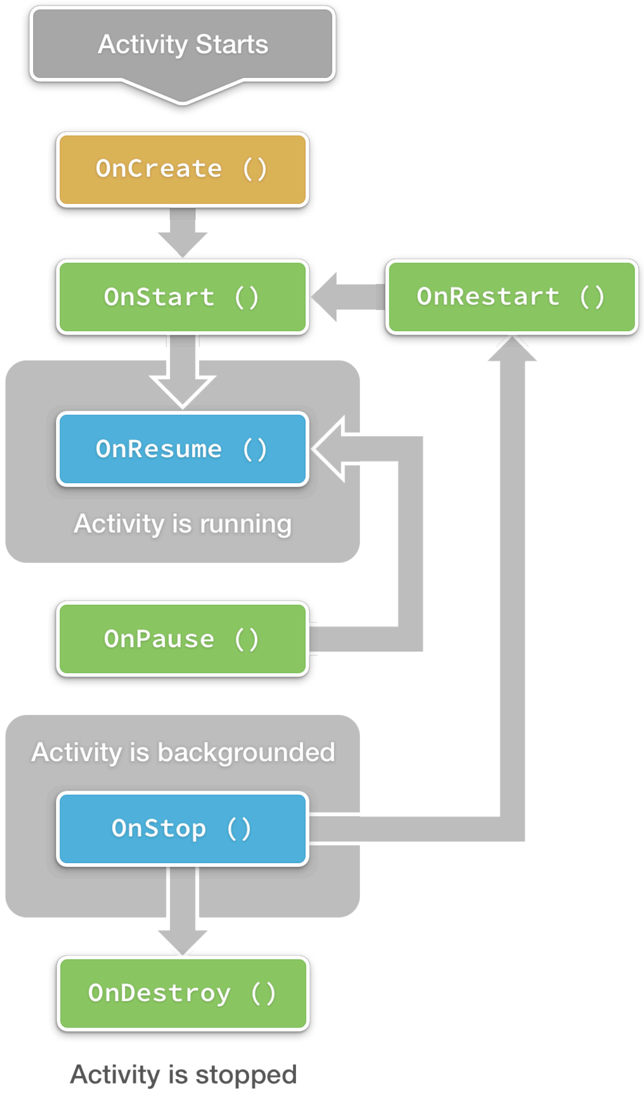
- Active or Running - Activities are considered active or running if they are in the foreground, also known as the top of the Activity stack. This is considered the highest priority Activity in Android, and as such will only be killed by the OS in extreme situations, such as if the Activity tries to use more memory than is available on the device as this could cause the UI to become unresponsive.
- Paused - When the device goes to sleep, or an Activity is still visible but partially hidden by a new, non-full-sized or transparent Activity, the Activity is considered paused. Paused activities are still alive: that is, they maintain all state and member information and remain attached to the window manager. This is considered to be the second highest priority Activity in Android and, as such, will only be killed by the OS if killing this Activity will satisfy the resource requirements needed to keep the Active/Running Activity stable and responsive.
- Stopped/Backgrounded - Activities that are completely obscured by another Activity are considered stopped or in the background. Stopped activities still try to retain their state and member information for as long as possible, but stopped activities are considered to be the lowest priority of the three states and, as such, the OS will kill activities in this state first to satisfy the resource requirements of higher priority activities.

- Restarted – It is possible that an Activity that went from paused to stopped was dropped from memory by Android. If the user navigates back to the Activity it must be restarted, restored to its previously saved state, and then displayed to the user.

Activity Lifecycle Methods

The Android SDK and, by extension, the Xamarin.Android framework provide a powerful model for managing the state of activities within an application. When an Activity's state is changing, the Activity is notified by the OS, which calls specific methods on that Activity.

The following diagram illustrates these methods in relationship to the Activity Lifecycle:



As a developer, you can handle state changes by overriding these methods within an Activity. It's important to note, however, that *all* lifecycle methods are called on the UI thread and will block the OS from performing the next piece of UI work, such as hiding the current Activity, displaying a new Activity, etc. As such, code in them should be as brief as possible to make an application feel performant. Any long-running tasks should be executed on a background thread.

Let's examine each of these lifecycle methods and their use:

OnCreate

This is the first method to be called when an Activity is created. `OnCreate` is always overridden to perform any startup initializations that may be required by an Activity such as:

- Creating views
- Initializing variables
- Binding static data to lists

`OnCreate` takes a `Bundle` parameter, which is a dictionary for storing and passing state information and objects between activities or the state. If the bundle is not `null`, this indicates the Activity is restarting and it should restore its state from the previous instance. The following code illustrates how to retrieve values from the bundle:

```
protected override void OnCreate(Bundle bundle)
{
   base.OnCreate(bundle);

   string extraString;
   bool extraBool;

   if (bundle != null)
   {
      intentString = bundle.GetString("myString");
      intentBool = bundle.GetBoolean("myBool");
   }

   // Set our view from the "main" layout resource
   SetContentView(Resource.Layout.Main);
}
```

Once `OnCreate` has finished, Android will call `OnStart`.

OnStart

This method is always called by the system after `OnCreate` is finished. Activities may override this method if they need to perform any specific tasks right before an Activity becomes visible such as refreshing current values of views within the Activity. Android will call `OnResume` immediately after this method.

OnResume

The system calls this method when the Activity is ready to start interacting with the user. Activities should override this method to perform tasks such as:

- Ramping up frame rates (a common task in game building)

- Starting animations
- Listening for GPS updates
- Display any relevant alerts or dialogs
- Wire up external event handlers.

As an example, the following code snippet shows how to initialize the camera:

```
public void OnResume()
{
    base.OnResume(); // Always call the superclass first.

    if (_camera==null)
    {
        // Do camera initializations here
    }
}
```

`OnResume` is important, because *any* operation that is done in `OnPause` should be un-done in `OnResume`, since it's the only lifecycle method that is guaranteed to be executed after `OnPause`, in the case of bringing the Activity back to life.

OnPause

This method is called when the system is about to put the Activity into the background or when the Activity becomes partially obscured. Activities should override this method if they need to:

- Commit unsaved changes to persistent data
- Destroy or clean up other objects consuming resources
- Ramp down frame rates and pausing animations
- Unregister external event handlers or notification handlers (i.e. those that are tied to a Service). This must be done to prevent Activity memory leaks.
- Likewise, if the Activity has displayed any dialogs or alerts, they must be cleaned up with the `.Dismiss()` method.

As an example, the following code snippet will release the camera, as the Activity cannot make use of it while Paused:

```
public void OnPause()
{
    base.OnPause(); // Always call the superclass first

    // Release the camera as other activities might need it
    if (_camera != null)
    {
        _camera.Release();
        _camera = null;
    }
}
```

There are two possible lifecycle methods that will be called after `OnPause`:

- `OnResume` will be called if the Activity is to be returned to the foreground

- `OnStop` will be called if the Activity is being placed in the background.

OnStop

This method is called when the Activity is no longer visible to the user. This happens when one of the following occurs:

- A new Activity is being started and is covering up this Activity.
- An existing Activity is being brought to the foreground.
- The Activity is being destroyed.

`OnStop` may never be called in low-memory situations, when the system doesn't have the resources to put an Activity in the background before terminating it. For this reason, it is best not to rely on `OnStop` getting called when preparing an Activity for destruction.

The next lifecycle methods that may be called after this one will be `OnDestroy` if the Activity is going away, or `OnRestart` if the Activity is coming back to interact with the user.

OnDestroy

This is the final method that is called on an Activity instance before it's destroyed and completely removed from memory. In extreme situations Android may kill the application process that is hosting the Activity, which will result in `OnDestroy` not being invoked. Most Activities will not implement this method because most clean up and shut down has been done in the `OnPause` method. This method is typically overridden to clean up long running resources that might leak resources. An example of this might be background threads that were started in `OnCreate`.

There will be no lifecycle methods called after the Activity has been destroyed.

OnRestart

This method is called after your Activity has been stopped, prior to it being started again. A good example of this would be when the user presses the home button while on an Activity in the application. When this happens `OnPause` and then `OnStop` methods are called, and the Activity is moved to the background but is not destroyed. If the user were then to restore the application by using the task manager or a similar application, Android will call the `OnRestart` method of the Activity.

There are no compelling reason to put code in `OnRestart`, because `OnStart` is always invoked regardless of the Activity is being created or being restarted. Any resources required by the Activity should be initialized in `OnStart`, rather than `OnRestart`.

The next lifecycle method called after `OnRestart` will be `OnStart`.

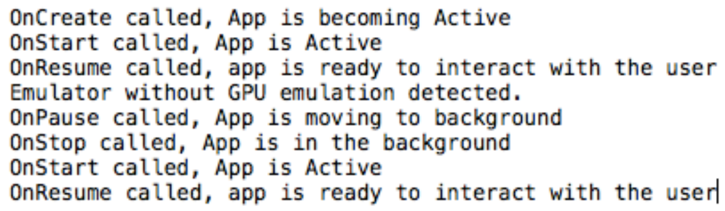
Let's take a look at what happens in the Activity Lifecycle when an Activity is in the background.

> **Note:** There is a subtle difference between the Android **Back** and **Home** buttons that determines whether an Activity gets backgrounded or

destroyed. When a user clicks the Home button, the Activity is placed into the background. In contrast, when the user clicks the Back button, Android will destroy the Activity.

Android Activity Lifecycle Demo: Exploring OnPause and OnStop

In this section, we are going to examine an application that notifies us of changing Activity states by printing information to the application output. We will expand this example by launching a second Activity that consumes a large amount of memory quickly, causing `OnStop` to not be called in the first Activity as the Activity is destroyed by the OS.

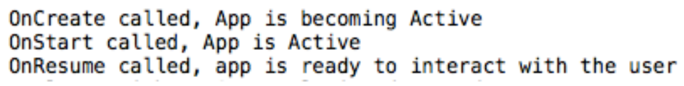


```
OnCreate called, App is becoming Active
OnStart called, App is Active
OnResume called, app is ready to interact with the user
Emulator without GPU emulation detected.
OnPause called, App is moving to background
OnStop called, App is in the background
OnStart called, App is Active
OnResume called, app is ready to interact with the user
```

1. Open up the *Lifecycle.Droid* project in the *LifecycleDemo* solution.
2. Open *MainActivity* and note that we have added logging to the lifecycle methods so we can track when the Activity changes states:

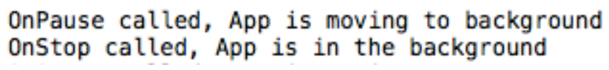
```
protected override void OnCreate()
{
        Log.Debug (logTag, "OnCreate called, Activity is becoming Active");
        base.OnCreate();

        SetContentView(Resource.Layout.Main);
        ...
}
protected override void OnStart()
{
        Log.Debug (logTag, "OnStart called, App is Active");
        base.OnStart();
}
protected override void OnResume()
{
        Log.Debug (logTag, "OnResume called, app is
        ready to interact with the user");
        base.OnResume();
}
protected override void OnPause()
{
        Log.Debug (logTag, "OnPause called, App is moving to background");
        base.OnPause();
}
protected override void OnStop()
{
        Log.Debug (logTag, "OnStop called, App is in the background");
        base.OnStop();
}
protected override void OnDestroy ()
{
        base.OnDestroy ();
        Log.Debug (logTag, "OnDestroy called, App is Terminating");
}
```

3. Launch the application in an emulator or on a device. Notice that `OnCreate`, `OnStart`, and `OnResume` are all called when the application is started. The application output should display something similar to:



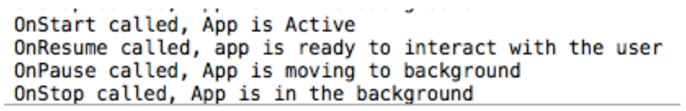
```
OnCreate called, App is becoming Active
OnStart called, App is Active
OnResume called, app is ready to interact with the user
```

4. Hit the **Home** button and observe that `OnPause` and `OnStop` are called, and the app is moved to the background. The application output should read:



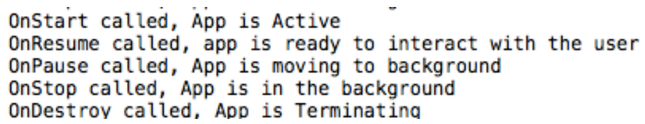
```
OnPause called, App is moving to background
OnStop called, App is in the background
```

5. Navigate back to the application and hit the **Application Switcher** button and observe that `OnPause` and `OnStop` are called here as well, and the app is moved to the background.



```
OnStart called, App is Active
OnResume called, app is ready to interact with the user
OnPause called, App is moving to background
OnStop called, App is in the background
```

6. Finally, navigate back to the app and hit the **Back** button. Notice that `OnPause` and `OnStop` are called, followed by `OnDestroy`. The application is destroyed when the back button is hit.



```
OnStart called, App is Active
OnResume called, app is ready to interact with the user
OnPause called, App is moving to background
OnStop called, App is in the background
OnDestroy called, App is Terminating
```

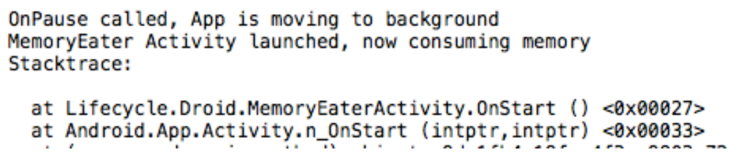
7. We're going to explore a scenario where `OnPause` is called without being followed by `OnStop`. This can happen in low-memory situations, when Android needs to destroy the Activity quickly.

   Open up the Activity called *MemoryEaterActivity*. Here, we have added a memory-consuming operation to `OnStart`:

```
protected override void OnCreate (Bundle bundle)
{
        base.OnCreate (bundle);
        Log.Debug("MemoryEater", "MemoryEater Activity launched,
                now consuming memory");
}

protected override void OnStart()
{
        base.OnStart();

        // consume memory
```

```
            int[] Big = new int[1000000000];
        }
```

8. Run the application in the emulator or on a device, and hit the "Consume Memory" button. You should see `OnPause` getting called in the *MainActivity*, and `OnCreate` getting called in the *MemoryEaterActivity* before the application starts experiencing memory problems. Notice that `OnStop` is never called:



```
OnPause called, App is moving to background
MemoryEater Activity launched, now consuming memory
Stacktrace:

  at Lifecycle.Droid.MemoryEaterActivity.OnStart () <0x00027>
  at Android.App.Activity.n_OnStart (intptr,intptr) <0x00033>
```

We've demonstrated the Activity Lifecycle, and shown that an Activity is very limited in what it can do while in the background. Next, let's take a look at a unique Android component useful for background processing – an Android *Service*.

Android Services

In the previous section, we learned how Android backgrounds Activities, and that Activities cannot perform any operations while in that state. To enable background processing, Android provides application components called *Services*. Services are generally still created as part of the application, but they run in their own lifecycle, separate from the lifecycle of any given Activity. This means that a Service can continue to perform processing after the user has put the application in the background, even if the OS stops or even destroys all the application's Activities.

Service Lifecycle

The lifecycle of a Service - that is, when it starts and when it stops - is governed by *how* it's started and *who* is connected to it. Generally, Services are referred to either as a Started Service or a Bound Service:

- Started Service: A *Started Service* is a Service that is explicitly started by either an object in the same application, or is started at device boot (if the Service is configured to do so). Generally, a Started Service will run until explicitly stopped by the caller, the OS, or from inside the Service itself.
- Bound Service: A *Bound Service* provides a direct link to a particular Service within an application. Bound Services use a *Binder* that provides a reference to the Service so that an application can access members on the Service directly. Bound Services are typically started when a client within the same application (usually an Activity), *Binds* to that Service, which causes Android to start the Service. Android will keep Bound Services running as long as clients are connected or *bound* to it. When all clients disconnect, or *unbind*, Android will stop that Service.

- **Hybrid Service:** The lifecycles of both Bound and Started Services can be combined by explicitly starting a Service and then binding to it. This offers the advantage of starting a Service independently of the rest of the application to provide it with an opportunity to do work, and then to also be able to connect to it directly in order to access it.

  The hybrid approach is very powerful and common way to use Services. For example, let's say we are writing an app that tracks location and movement while hiking. We might want the app to log locations continuously for later use, but to only display location information when a particular screen is showing. In this case, the Service might be started when the app launched, but it would only have an active client when a particular Activity was in the foreground. The Activity would bind to the Service so that it could get location information, but the Service should still run when the Activity unbinds from it.

  A hybrid Service will run until it is explicitly stopped *and* no clients are bound to it, or until the system shuts it down in the event of a crash or memory pressure.

Starting a Service

There are three ways to start a Service:

- **Starting a Service from the Application:** A Service can be started directly from the application by calling `StartService` on the Service. This Service will implement methods from the Started Service lifecycle, most notably overriding `OnStartCommand`.
- **Using a Binder to Start a Service:** If a Service is not explicitly started with a `BroadcastReceiver` or a `StartService` call, it will start when a client binds to it. This Service will follow the Bound Service lifecycle.

Communicating with a Service

The way another component communicates with a Service depends on where the Service is running. Android provides three options:

- **Service Binding:** If a Service is part of the same application, a client can communicate with the Service directly by binding to it. A Service that binds to a client will override Bound Service lifecycle methods, and communicate with the client using a `Binder` and a `ServiceConnection`.
- **Service Messengers**: Sometimes a client needs to communicate with a Service in another application. Because applications are unable to access each other's memory, a direct connection is not possible in this case. Instead, data needs to be marshaled across the process boundary in what's known as *Inter-Process Communication (IPC)*. In this case, a **Service Messenger** can be used to communicate via messages across process boundaries, allowing the client to bind to the remote Service, and call methods on it.

- **Android Interface Definition Language (AIDL)**: Another way to do IPC in Android is with the AIDL. AIDL is a low-level interface for marshaling objects into primitive data types across process boundaries, and is what Service Messengers use under the hood. Since the introduction of Service Messengers, AIDL has largely been superseded, because Service Messengers are much simpler to use and easier to implement. However, because messages sent via a Service Messenger are handled serially instead of concurrently, applications needing to make multiple concurrent calls to a Service across a process boundary should still use AIDL. Fortunately, this is not a requirement in most mobile applications.

This chapter will cover how to start and communicate with local Services. To learn more about communicating across processes with Service Messengers or AIDL, refer to the Xamarin Creating Services guide.

Now that we understand how to start and communicate with Services, let's create a Service. We are going to take a look at the *Service.Droid* project in the *BackgroundingDemo* solution.

The Service Class

The first step in creating a Service is to create a subclass of the `Service` class. The `Service` class is the base class for any Service. As with Activites, Services need to be decorated with a `ServiceAttribute` custom attribute:

```
[Service]
public class MyService : Service
{
        ...
}
```

The `ServiceAttribute` automatically registers the new Service with the `AndroidManifest.xml` (refer to the Xamarin Working with Android Manifest guide for more on the Android Manifest file). Now any component that inherits from `Context`, such as an `Activity`, can start the Service.

Let's take a more in-depth look at Started Services.

Starting a Started Service

A Started Service is initialized with a call to the `StartService` method on a Context (such as an Activity). If the Service is being started from an Activity, the call can be made using `this.StartService`. Otherwise, `Android.App.Application.Context` can be used to obtain the current Context.

To start a Service, pass an Intent with the current Context and the type of Service that needs to be started. For example, the following code in an Activity will start a Service of type `MyService`:

```
this.StartService (new Intent (this, typeof(MyService)));
```

Calling `StartService` will cause Android to call the `OnStartCommand` method on the Service. The implementation of `OnStartCommand` must return a

`StartCommandResult` enumeration value, which tells Android whether or not the Service should be restarted if it gets stopped, and can be one of four options:

- **Sticky** – A sticky Service will be restarted, and a null intent will be delivered to `OnStartCommand` at restart. Used when the Service is continuously performing a long-running operation, such as updating a stock feed.
- **RedeliverIntent** – This option is used when the Intent that started the Service contains extra information critical to the Service performing normally. The Service is restarted, and the last Intent that was delivered to `OnStartCommand` before the Service was stopped by the system is redelivered.
- **NotSticky** – The Service is not automatically restarted.

For example, the following code snippet shows an `OnStartCommand` implementation that returns `StartResultCommand.Sticky`, which will restart the Service automatically:

```
public override StartCommandResult OnStartCommand (Intent intent,
StartCommandFlags flags, int startId)
{
        // start a task here
        new Task (() => {
                // long running code
                DoWork();
        })Start();
        return StartCommandResult.Sticky;
}
```

Notice that we use a Task object to perform Service initialization work. This is because a Service runs on the UI thread, so any long-running task would hold up UI rendering, making the application unresponsive.

Stopping a Started Service

Unless the task is started with the intention of running indefinitely, a Started Service should call `StopSelf` after any long-running work is done. This is important because the service runs independently of the component that called `StartService`, and will continue drawing on system resources until it is explicitly stopped or shut down by the OS.

The following code calls the `StopSelf` method after it completes its work:

```
public void DoWork ()
{
        new Task (() => {
                Thread.Sleep (5000);
                StopSelf ();
        }).Start();
}
```

To avoid the possibility of a Service continuing indefinitely, the caller can also request that the Service be stopped by calling the `StopService` method, as shown below:

```
StopService (new Intent (this, typeof(MyService)));
```

When a Service is stopped, the `OnDestroy` method will be called on the Service. This is the part of the process where any cleanup of Service-wide resources should be done. To implement, simply override `OnDestroy` as follows:

```
public override void OnDestroy ()
{
        base.OnDestroy ();
        // cleanup code
}
```

Bound Services

A Bound Service is a Service that can be directly referenced from the application. As long as a Service is running locally, an application can call methods on the Service just by binding to it.

Connecting to a Bound Service requires implementing two Interfaces:

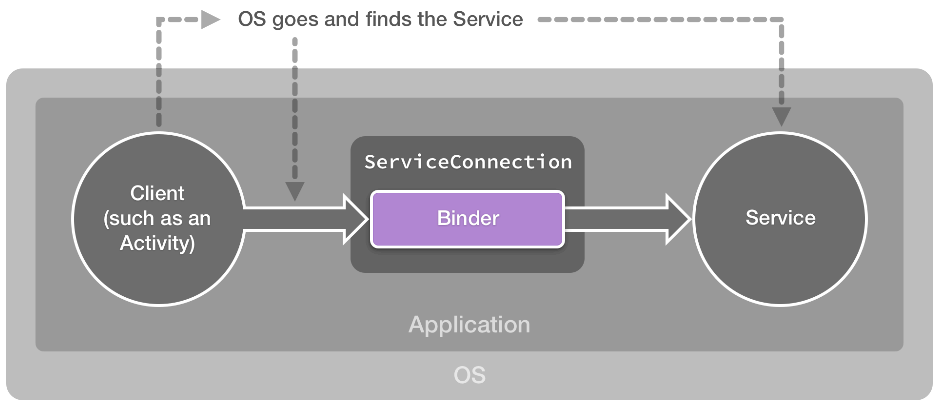
- *IBinder* – An `IBinder` is a very simple interface that contains a reference to the actual running Service.
- *IServiceConnection* – An `IServiceConnnection` interface is used by Android to actually locate the running Service and attach a binder to it.

The process of connecting to a Bound Service is as follows:

1. Create an object that implements `IServiceConnection` and wire up the `ServiceConnected` event on it.
2. Create an `Intent` that contains the `Context` of the client (such as an Activity), and the `Type` of the Service to be connected to.
3. Call *BindService* on a `Context` and pass the Intent.

Android will then locate the Service, create it, and call `OnBind` on it.

The following diagram illustrates how the caller uses a `Binder` and `ServiceConnection` to communicate with a Service running locally:



The Service must override the *OnBind* method and return an `IBinder` object that contains a reference to the Service, e.g.:

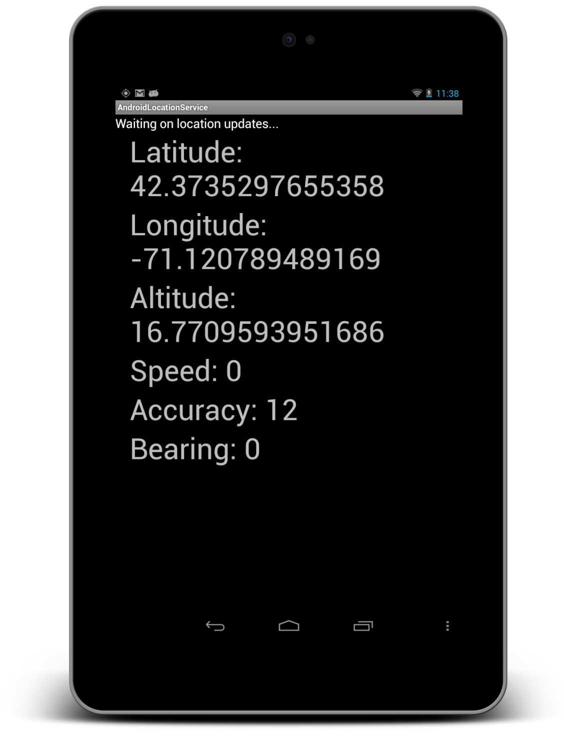
```
public override IBinder OnBind (Intent intent)
{
        binder = new MyServiceBinder (this);
        return binder;
}
```

Once the `OnBind` method returns, Android will raise the `ServiceConnected` event on the `IServiceConnection` object and the client can then reference the Service via the `Binder` that is owned by the `IServiceConnnection`.

In the next section, we'll see this in action.

Android Location Application Walkthrough

Let's walk through creating an application that uses a hybrid Service and raises location update events. The completed version of this application is called *Location.Droid* and can be found in the *BackgroundLocationDemo* solution.



11:38

AndroidLocationService

Waiting on location updates...

Latitude:
42.3735297655358

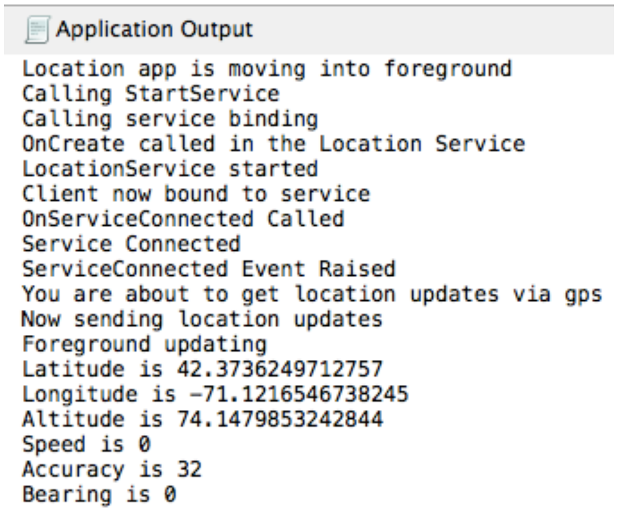
Longitude:
-71.120789489169

Altitude:
16.7709593951686

Speed: 0

Accuracy: 12

Bearing: 0

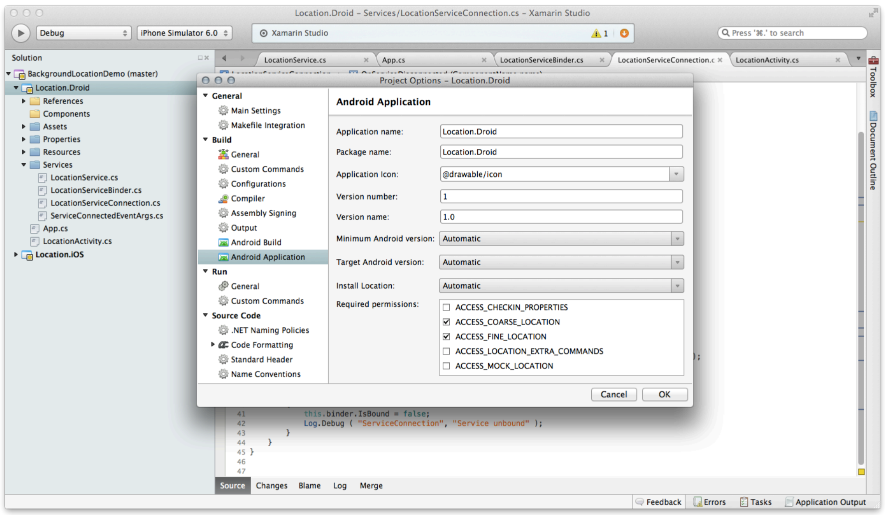


```
Location app is moving into foreground
Calling StartService
Calling service binding
OnCreate called in the Location Service
LocationService started
Client now bound to service
OnServiceConnected Called
Service Connected
ServiceConnected Event Raised
You are about to get location updates via gps
Now sending location updates
Foreground updating
Latitude is 42.3736249712757
Longitude is -71.1216546738245
Altitude is 74.1479853242844
Speed is 0
Accuracy is 32
Bearing is 0
```

Location data will be displayed on the screen when the app is running in the foreground, and in the application output when the app is operating in the background.

1. First, create a new Android application called *Location.Droid*.
2. Next, give the application permission to use the device's location data. In Xamarin Studio, double click on the project and select the **Android Application** panel. If you have not already created an Android Manifest, you can add one now.

   Under **Required Permissions**, check ACCESS_FINE_LOCATION and ACCESS_COURSE_LOCATION. This gives the application access to GPS and Network data.

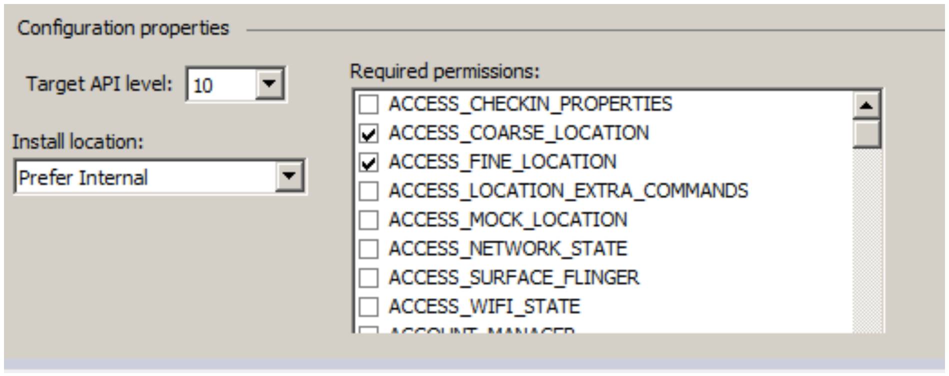


Android Application

Application name: Location.Droid
Package name: Location.Droid
Application Icon: @drawable/icon
Version number: 1
Version name: 1.0
Minimum Android version: Automatic
Target Android version: Automatic
Install Location: Automatic
Required permissions:
- ☐ ACCESS_CHECKIN_PROPERTIES
- ☑ ACCESS_COARSE_LOCATION
- ☑ ACCESS_FINE_LOCATION
- ☐ ACCESS_LOCATION_EXTRA_COMMANDS
- ☐ ACCESS_MOCK_LOCATION

```
            this.binder.IsBound = false;
            Log.Debug ( "ServiceConnection", "Service unbound" );
        }
    }
}
```

In Visual Studio, this is available under **Properties** > **Android Manifest**:



Target API level: 10

Install location: Prefer Internal

- [ ] ACCESS_CHECKIN_PROPERTIES
- [✓] ACCESS_COARSE_LOCATION
- [✓] ACCESS_FINE_LOCATION
- [ ] ACCESS_LOCATION_EXTRA_COMMANDS
- [ ] ACCESS_MOCK_LOCATION
- [ ] ACCESS_NETWORK_STATE
- [ ] ACCESS_SURFACE_FLINGER
- [ ] ACCESS_WIFI_STATE

3. Recall that in Android, communication with the Service is done through a `Binder`. In the project, create a new subclass of `Binder` called LocationServiceBinder:

```
public class LocationServiceBinder : Binder
{
        public LocationService Service
        {
               get { return this.service; }
        } protected LocationService service;

        public bool IsBound { get; set; }

        public LocationServiceBinder (LocationService service)
        {
                this.service = service;
        }
}
```

   Clients use the `LocationServiceBinder` class to obtain a reference to the Service, which is our case is going to be the `LocationService`. The client will use the `Binder` to call the `LocationService`'s methods. We'll see how this works in the next step.

4. Let's make the Service that contains our location methods. Begin by making a new Service subclass called `LocationService`, and decorate the Service with the `ServiceAttribute` so that it gets registered with the `AndroidManifest.xml`. Then, override the `OnBind` lifecycle method to return an instance of the `LocationServiceBinder` we created in the last step:

```
[Service]
public class LocationService : Service, ILocationListener
{
        IBinder binder;
        public override IBinder OnBind (Intent intent)
        {
               binder = new LocationServiceBinder (this);
               return binder;
        }


}
```

5. Inside the **LocationService** class, override `OnStartCommand` to return `StartComandResult.Sticky`. A Sticky Service will get restarted with a null intent if the OS ever shuts it down due to memory pressure:

   ```
   public override StartCommandResult OnStartCommand (Intent intent,
           StartCommandFlags flags, int startId)
           {
                   return StartCommandResult.Sticky;
           }
   ```

6. To get location data from the system to the Service, we need to use a `LocationManager`. The location manager provides a way for the application to interact with the system location Service.

   In the `LocationService`, create a location manager called `LocMgr`:

   ```
   protected LocationManager locMgr =
           Android.App.Application.Context.GetSystemService ("location")
           as LocationManager;
   ```

   To initialize the location manager, use the current Context to get the location Service provided by the system. This will be our **LocationManager**.

   Next, we update the `LocationService` class to implement the `ILocationListener` interface. This allows `LocationService` to subscribe to notifications from the `LocationManager` about changes in location. `ILocationListener` methods will be covered in the next step.

   We are now ready to use the location manager to generate location data. We start by creating new `Criteria` and setting the desired *Accuracy*, *Power* usage, and *Provider* for the location updates.

   The following code shows an example of setting the criteria:

   ```
   [Service]
   public class LocationService : Service, ILocationListener
   {
           // we implemented OnBind in a previous step
           IBinder binder;
           public override IBinder OnBind (Intent intent)
           {
                   binder = new LocationServiceBinder (this);
                   return binder;
           }

           public void StartLocationUpdates ()
           {
                    var locationCriteria = new Criteria();

                    locationCriteria.Accuracy = Accuracy.NoRequirement;
                    locationCriteria.PowerRequirement = Power.NoRequirement;

                    var locationProvider = LocMgr.GetBestProvider(
                           locationCriteria, true);

                   LocMgr.RequestLocationUpdates(locationProvider, 2000,
                           0, this);
           }
   ```

```
        ...
}
```

The last step is to call `RequestLocationUpdates` on the location manager, and register for location updates from the system. We will pass in the location provider, the minimum time and distance required for a location update, and a `LocationListener`, which in this case is the LocationService.

For more information on using the Android location Service, refer to the Xamarin Maps and Locations guide.

7. Once the location manager has requested location updates, the application can listen to changes in location by implementing the `ILocationListener` interface. `ILocationListener` makes four methods available to track changes reported by the `LocationManager`:

    - OnLocationChanged: called when new location data is available.
    - OnProviderEnabled: called when the user enables a provider, such as GPS or network.
    - OnProviderDisabled: called when the user disables the provider.
    - OnStatusChanged: called when the availability of a provider changes.

    Because our Service implements the `ILocationListener` interface, it already subscribes to these methods. However, we need a way of letting the client using the Service know when the methods get called. The following code snippet shows how to use an event to track when `OnLocationChanged` gets called, and outputs the updated location information to the console:

```
public event EventHandler<LocationChangedEventArgs> LocationChanged =
        delegate { };

public void OnLocationChanged (Android.Locations.Location location)
{
        this.LocationChanged (this,
                new LocationChangedEventArgs (location));
                Log.Debug (logTag, String.Format ("Latitude is {0}",
                        location.Latitude));
                Log.Debug (logTag, String.Format ("Longitude is {0}",
                        location.Longitude));
                Log.Debug (logTag, String.Format ("Altitude is {0}",
                        location.Altitude));
                Log.Debug (logTag, String.Format ("Speed is {0}",
                        location.Speed));
                Log.Debug (logTag, String.Format ("Accuracy is {0}",
                        location.Accuracy));
                Log.Debug (logTag, String.Format ("Bearing is {0}",
                        location.Bearing));
}
```

Follow this pattern to override `OnProviderEnabled`, `OnProviderDisabled`, and `OnStatusChanged`.

8. The next step is to give the caller a reference to the `LocationServiceBinder` we created. This is done with a `ServiceConnection`. A `ServiceConnection` is a class that provides a calling interface between the client and the Service, and gives the caller feedback about the state of the Service.

   To start, create a class called `LocationServiceConnection` that inherits from `ServiceConnection`:

```
public class LocationServiceConnection : Java.Lang.Object,
        IServiceConnection
{
        public LocationServiceConnection (LocationServiceBinder binder)
        {
                if (binder != null) {
                        this.binder = binder;
                }
        }
        public void OnServiceConnected (ComponentName name,
                IBinder service)
        {
                LocationServiceBinder serviceBinder = service as
                        LocationServiceBinder;

                if (serviceBinder != null) {
                        this.binder = serviceBinder;
                        this.binder.IsBound = true;

                        // raise the service bound event
                        this.ServiceConnected(this, new
                                ServiceConnectedEventArgs () {
                                Binder = service } );

                        // begin updating the location in the Service
                        serviceBinder.Service.StartLocationUpdates();
                }
        }

        public void OnServiceDisconnected (ComponentName name)
        {
                this.binder.IsBound = false;
        }
}
```

   The `ServiceConnection` calls two methods. `OnServiceConnected` is called when the client connects to the Service, and has two parameters: the name of the Service that's connecting, and the `IBinder` associated with the Service. We use `OnServiceConnected` to get a reference to the `LocationServiceBinder`. We also create an event called `ServiceConnected` that fires when the Service is bound. Lastly, we call `StartLocationUpdates` on the Service to start generating some location data.

   `OnServiceDisconnected` gets called if the Service gets unbound or terminates unexpectedly. This only happens in extreme situations such as application crashes. The following implementation of `OnServiceDisconnected` sets the `IsBound` property of the `LocationServiceBinder` instance to false if the Service ever crashes.

9. Let's create a singleton class called App for Application-wide objects. We will start and bind the Service from the App class. This way, our Service will start when the application starts, regardless of what is happening with the Activity or Activities that make up the app. We will start and bind to our Service in this class.

   Services run on the main thread of the application. Because of this, it is a good idea to start and bind a Service on a background thread, to ensure that we don't block the UI thread and stall the app.

   In the App class, start a new thread; then, call `StartService` on the LocationService to start the Service. Create an instance of the `LocationServiceConnection`. We can use this `ServiceConnection` to help us bind to the Service:

```
public event EventHandler<ServiceConnectedEventArgs>
        LocationServiceConnected = delegate {};

protected App ()
{
        new Task ( () => {

                Android.App.Application.Context.StartService (new Intent (
                        Android.App.Application.Context,
                        typeof(LocationService)))

                this.locationServiceConnection =
                        new LocationServiceConnection (null);
                this.locationServiceConnection.ServiceConnected += (
                        object sender, ServiceConnectedEventArgs e) => {
                        this.LocationServiceConnected ( this, e );
                };

                Intent locationServiceIntent = new Intent (
                        Android.App.Application.Context,
                        typeof(LocationService));
                Android.App.Application.Context.BindService (
                        locationServiceIntent,
                        locationServiceConnection, Bind.AutoCreate);
        } ).Start ();
}
```

   In the code above, we bind to the Service by calling `BindService`, passing an Intent and an instance of a `ServiceConnection`. `BindService` is an asynchronous call that will either return `false` if there is no Service to bind to, or cause a callback to be sent to the `OnServiceConnected` method of our `LocationServiceConnection` class when the connection is made. This will trigger the `ServiceConnected` event in the `LocationServiceConnection`, which in turn will cause our `LocationServiceConnected` event to fire, notifying our Activity that it's time to start updating the UI.

The above code uses the `Bind.AutoCreate` value in the `BindService` call to automatically create the Service if the binding exists.

10. Now that the Service is bound, it's time to let our Activity know what's happening in our Service. In **Activity1** in the MainActivity.cs file, subscribe to the `App.Current.LocationServiceConnected` event in `OnCreate` to let the Activity know when the Service has connected to the client.

    Now we can begin listening for changes in location by subscribing to the four events created in the **LocationService**, including the most important one, `LocationChanged`. Recall that these events listen for location updates from the `LocationManager` as reported by the `ILocationListener`, and that the `LocationManager` in turn tracks changes in the system location service. We've now created a pipeline for getting location information from the system to the Activity via a Service.

    The following code snippet demonstrates how all this fits together:

```
protected override void OnCreate (Bundle bundle)
{
        base.OnCreate (bundle);
        SetContentView (Resource.Layout.Main);

        // link our labels to the designer
        latText = FindViewById<TextView> (Resource.Id.lat);
        longText = FindViewById<TextView> (Resource.Id.longx);
        altText = FindViewById<TextView> (Resource.Id.alt);
        speedText = FindViewById<TextView> (Resource.Id.speed);
        bearText = FindViewById<TextView> (Resource.Id.bear);
        accText = FindViewById<TextView> (Resource.Id.acc);

        App.Current.LocationServiceConnected += (object sender,
                ServiceConnectedEventArgs e) => {
                Log.Debug (logTag, "ServiceConnected Event Raised");

                App.Current.LocationService.StartLocationUpdates();

                App.Current.LocationService.LocationChanged
                        += HandleLocationChanged;
                App.Current.LocationService.ProviderDisabled
                        += HandleProviderDisabled;
                App.Current.LocationService.ProviderEnabled
                        += HandleProviderEnabled;
                App.Current.LocationService.StatusChanged
                        += HandleStatusChanged;
        };
}
```

11. The final step is to get the location data to the screen. Recall that in the **App** class, we started our Service on a separate thread. This means that all location-related events, including `LocationChanged`, happen on a background thread. Any calls to update the UI will crash the app unless they are explicitly performed on the UI thread using `Activity.RunOnUiThread`.

The following method prints out latitude, longitude, and other coordinates to the screen when the app is in the foreground:

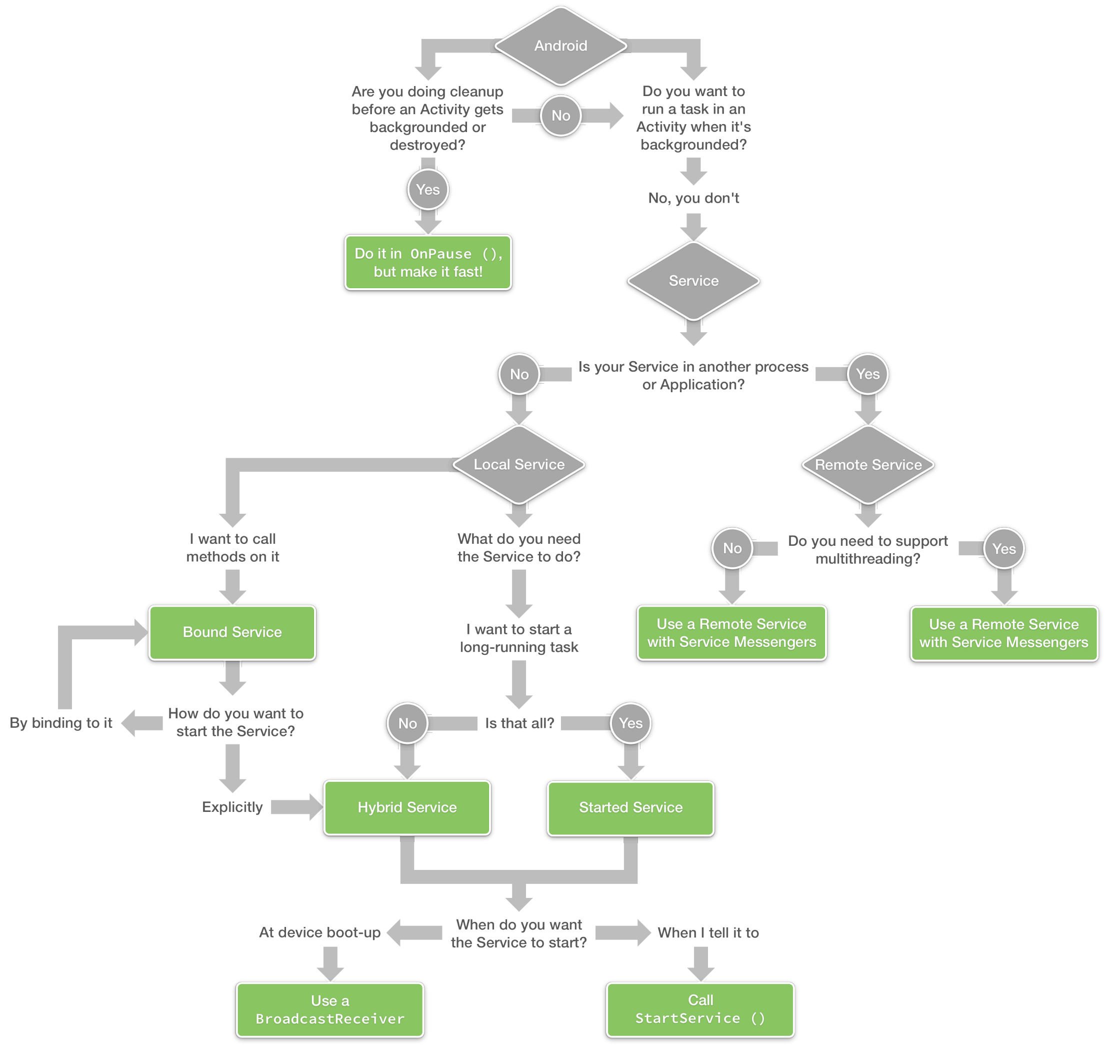
```
public void HandleLocationChanged(object sender,
        LocationChangedEventArgs e)
 {
        Android.Locations.Location location = e.Location;

        RunOnUiThread (() => {
                latText.Text = String.Format ("Latitude: {0}",
                        location.Latitude);
                longText.Text = String.Format ("Longitude: {0}",
                        location.Longitude);
                altText.Text = String.Format ("Altitude: {0}",
                        location.Altitude);
                speedText.Text = String.Format ("Speed: {0}",
                        location.Speed);
                accText.Text = String.Format ("Accuracy: {0}",
                        location.Accuracy);
                bearText.Text = String.Format ("Bearing: {0}",
                        location.Bearing);
        });
}
```

The application is now performing location updates both in the foreground and the background, using methods provided by the `LocationService`.

In this walkthrough, we learned to create and start a Service; to bind to the Service using a `Binder` and a `ServiceConnection`; and to take advantage of the Service lifecycle to continue receiving location updates even when the application is not running in the foreground.

Android Backgrounding Guidance

Refer to the following diagram for choosing which backgrounding technique to use in Android:


Android
Are you doing cleanup before an Activity gets backgrounded or destroyed?
No
Do you want to run a task in an Activity when it's backgrounded?
Yes
No, you don't
Do it in OnPause (), but make it fast!
Service
No
Is your Service in another process or Application?
Yes
Local Service
Remote Service
I want to call methods on it
What do you need the Service to do?
No
Do you need to support multithreading?
Yes
Bound Service
I want to start a long-running task
Use a Remote Service with Service Messengers
Use a Remote Service with Service Messengers
By binding to it
How do you want to start the Service?
No
Is that all?
Yes
Explicitly
Hybrid Service
Started Service
At device boot-up
When do you want the Service to start?
When I tell it to
Use a BroadcastReceiver
Call StartService ()

Summary

In this chapter, we introduced the different ways of doing background processing in iOS and Android. We covered iOS Application States and examined the role backgrounding plays in the iOS Application Lifecycle. On the Android side, we explored the Android Activity Lifecycle, and the limitations placed on backgrounded Activities. We learned how we could register individual tasks or entire applications to operate in the background in iOS, and saw that we could accomplish a similar effect in Android with Services, which facilitate background processing by running tasks outside the lifecycle of the component that called them. Finally, we reinforced our understanding of backgrounding on both platforms by building an application that performs location updates in the background.