Advanced Android App Lifecycle

Evolve Advanced Track, Chapter 10

Xamarin Inc.

Overview

Android is a powerful mobile operating system that allows apps nearly full control of the device and the application possibilities are endless.

But the app lifecycle can be strange, crashes will destroy the application, but instead of launching the first Activity, it'll try launch the next to last (penultimate) Activity. Additionally, there's no built in way to provide a single point of application initialization, and even the lifecycle of an Activity can feel a little overwhelming on first examination.

In this chapter, we're going to cover some advanced Android app lifecycle, scenarios, including:

- Showing a loading screen while the application initializes
- Waiting for multiple, asynchronous items during initialization, such as waiting for services
- Handling application crashes in a sophisticated way

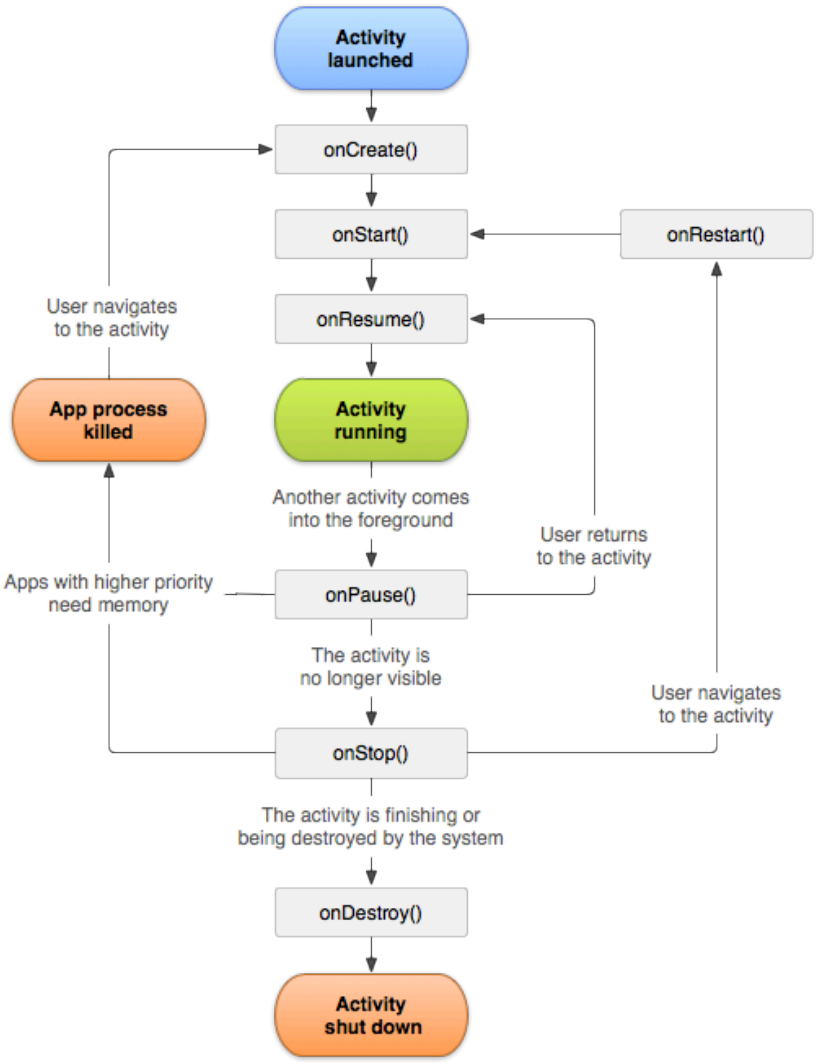
We're also going to take a look at some advanced Activity lifecycle concepts and responsibilities in the context of the app lifecycle, including:

- Memory management in the context of the Activity lifecycle
- Handling configuration changes such as device rotation

As well as the proper way to display dialogs and alerts so as to not leak memory and still persist their state during Activity state changes and tear-downs.

Activity Lifecycle

In the chapter on Backgrounding we covered the states that an Activity goes through during its lifecycle, as well as the event methods that are called during those state changes:



Most Important Lifecycle Methods

While there are quite a few in total, for the concerns of this chapter, in terms of backgrounding and configuration changes, we really only need concern ourselves with three: *OnCreate*, *OnResume*, and *OnPause*.

OnCreate

OnCreate is called only once in the lifetime of the Activity; when the Activity is instantiated. As such, this is the correct place to:

- Load or inflate your UI
- Instantiate any controls
- Resolve initial references to any controls, e.g.: `FindViewByID<Button>`
- Wire up any Activity event handlers, e.g.: `MyButton.Click += ...`

OnResume

OnResume is the only event guaranteed to run during all Activity is activated. As such, *anything* done in OnPause should be undone in OnResume. For example, OnResume is the place to:

- Rehydrate UI state
- Wire up an external event or notification handlers
- Start updating the UI

External event handlers need to be wired up here because it's important to remove them in OnPause, otherwise they'll case memory leaks.

OnPause

When an application is being backgrounded, paused, or even stopped, OnPause is the first method that will be called. As such it's the appropriate place to:

- Stop any UI updates
- Dismiss any dialogs
- Remove external event or notification handlers

Both dialogs and external event handlers are special cases here.

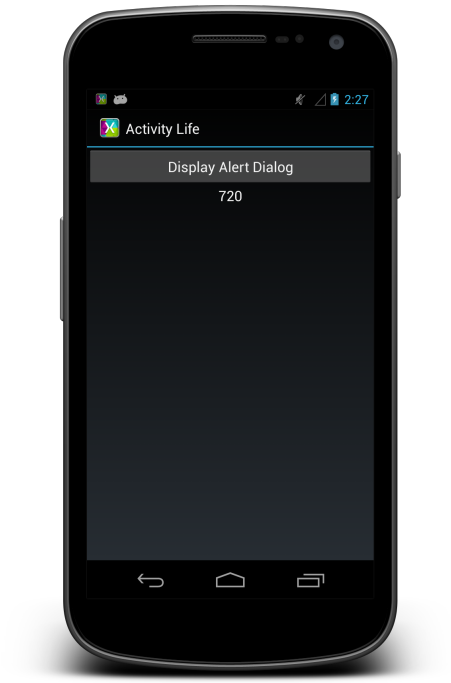
Dialogs must be dismissed (even if they'll be loaded again in OnResume) because they are actually tied to a window context under the Android hood, and will cause a memory leak.

External event handlers need to be removed for a similar reason. An event handler is tied to the object that it's listening to. Therefore, it won't get garbage collected until the external object does. As such, the Xamarin runtime will keep the Activity reference around because the handler is kept around. However, Android may destroy its instance of the Activity. The net effect is not only a memory leak, but if the Activity gets re-added, the updates might actually occur twice.

Example Functionality

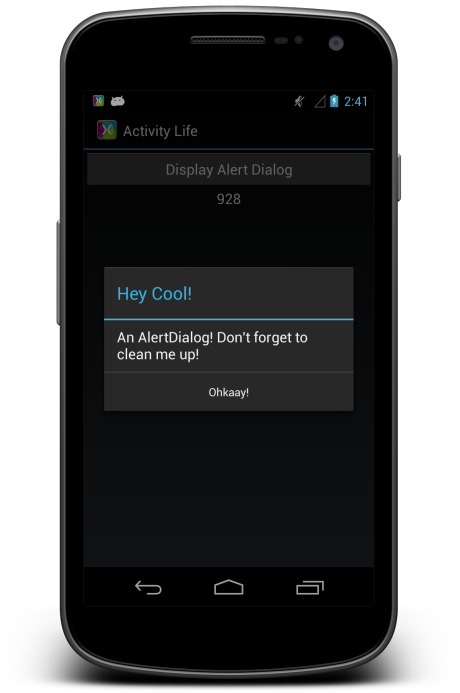
The ActivityLifecycle project in the App_Lifecycle_Demo illustrates these concepts via the following features:

Handling for External Updates



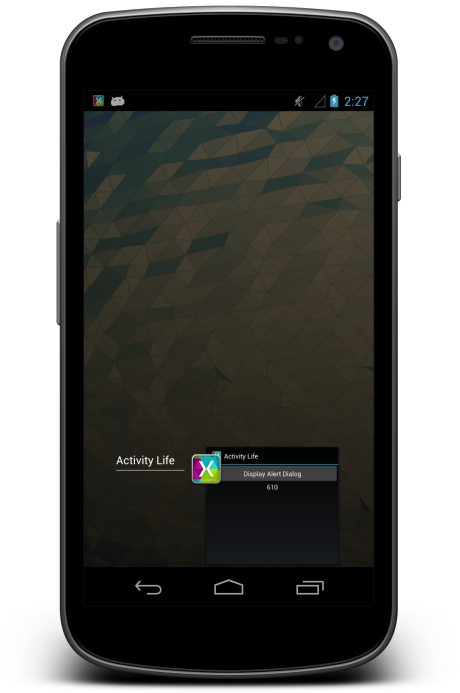
Updatesare raised via an event on a background thread that generates random numbers on an external object. The Activity subscribes to the event and displays them on the Activity.

Showing a Dialog



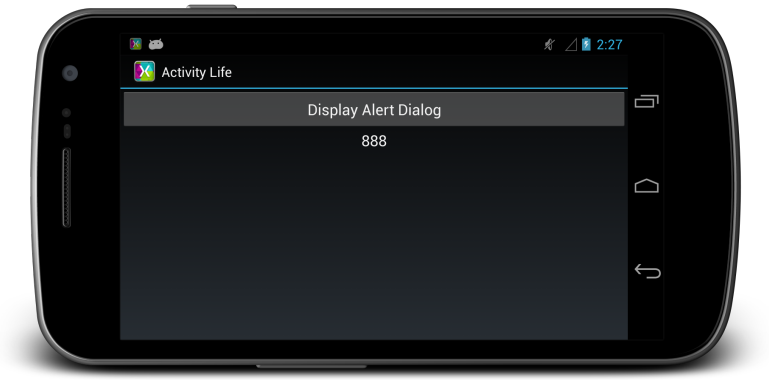
The Activity will show a dialog in response to button press and then preserve that dialog state through configuration changes (such as rotation), as well as pausing or destroying the Activity.

Handling Pause



The Activity can be paused, which will cause it to stop updating its UI (and listening for external updates).

Handles Rotation



In addition to preserving its dialog state, the Activity will also appropriately subscribe and unsubscribe to external updates during configuration changes.

Activity Lifecycle Walkthrough

Let's examine the application to see just how we accomplish the aforementioned functionality.

App.cs

App.cs contains a simple class that raises an `Updated` event on a background thread every second and populates the event data with a random number. It's in the project simply as an example to help illustrate the responsibility an Activity has when subscribing to events on a class whose lifecycle is not tied to the Activity; as opposed to subscribing to something like the `Click` event on a Button that is part of the Activity.

It utilizes the Singleton pattern to create a single instance of the class that is available application wide, via a static:

```
public class App
{
        public event EventHandler<UpdatingEventArgs> Updated = delegate {};

        public static App Current
        {
                get { return current; }
        } private static App current;

        static App ()
        {
                current = new App();
        }
        protected App ()
        {
                new Task (() => {
                        Random random =
                                new Random( DateTime.Now.Millisecond );
                        while(true){
                                int num = random.Next(0, 1000);
                                this.Updated ( this,
                                        new UpdatingEventArgs () {
                                                Message = num.ToString() } );
                                System.Threading.Thread.Sleep(500);
                        }
                }).Start ();
        }
}
```

To access this class, we can call `App.Current`, which will return the one and only App class instance.

MainActivity OnCreate Method

The *OnCreate* method of the `MainActivity` class sets it's view from a layout file, gets references to the controls, wires up a local event handler, and then rehydrates a Boolean value, `showingAlert`, from the bundle that is set later in the lifecycle:

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

        updateStatusText = FindViewById<TextView> (
                Resource.Id.UpdateStatusText);
        displayAlertDialogButton = FindViewById<Button> (
                Resource.Id.DisplayAlertDialogButton);

        // it's safe to wire up this event handler in OnCreate because
        // it's only referenced by this activity
        displayAlertDialogButton.Click += (sender, e) => {
                this.ShowAlertDialog();
        };

        // if we've stored the showingProgress bool, set it
        if(bundle != null)
                this.showingAlert = bundle.GetBoolean("showingAlert");
}
```

When we retrieve the `showingAlert` value, we have to first check to see if the bundle isn't `null`, because this value will only be persisted after the first load of the activity.

MainActivity OnResume Method

OnResume does two things in this example, first, it calls `AddHandlers`, which subscribes to our App Updated event, and second, if the `showingAlert Boolean` has been set to `true`, we call a method to show the alert. We do this in `OnResume` in the case that the activity has been launched, and then destroyed due to rotation or paused and we need to redisplay the alert:

```
protected override void OnResume ()
{
        base.OnResume ();
        this.AddHandlers();
        if(this.showingAlert) {
                this.ShowAlertDialog();
        }
}
```

The `AddHandlers` method instantiates an `Updated` event handler delegate, stores it in a class level variable (so we can reference it later to remove the event handler) and then adds that handler to the `App.Current.Updated` event:

```
protected void AddHandlers()
{
        Log.Debug (logTag, "MainActivity.AddHandlers");
        updateHandler = (object s, UpdatingEventArgs args) => {
                this.RunOnUiThread (() => {
                        this.updateStatusText.Text = args.Message;
                });
        };
        App.Current.Updated += updateHandler;
}
```

The important thing to note here is that instead of using an anonymous delegate, we actually create the delegate and store a reference to it. If we didn't store the reference, there would be no easy way to remove it from the event in `OnPause`.

OnPause

Our *OnPause* method does two things. First, it removes our event handlers so that the UI stops getting updated and we prevent memory leaks. And second, it dismisses our Alert. Even if the Alert needs to continue to be displayed when the Activity becomes active again, we still need to dismiss it, otherwise it'll cause a memory leak:

```
protected override void OnPause ()
{
        base.OnPause ();
        this.RemoveHandlers ();
        if (alert != null)
                alert.Dismiss();
}
```

Our `RemoveHandlers` method is very simple, it simply removes our `updateHandler` delegate from the Updated event:

```
protected void RemoveHandlers()
{
        App.Current.Updated -= updateHandler;
}
```

OnSaveInstanceState

In the example app, we also override *OnSaveInstanceState* so that we have an opportunity to persist a Boolean value of whether or not we're currently showing the alert. That way, in the case of a configuration change, such as rotation, in which the Activity gets completely destroyed and recreated, we know that the Alert should still be shown. `OnSaveInstanceState` is called before the application is stopped, and provides us with an opportunity to persist any data that allows us to rehydrate the UI to the state that it was in before it was destroyed:

```
protected override void OnSaveInstanceState (Bundle outState)
{
        base.OnSaveInstanceState (outState);
        outState.PutBoolean("showingAlert", this.showingAlert);
}
```

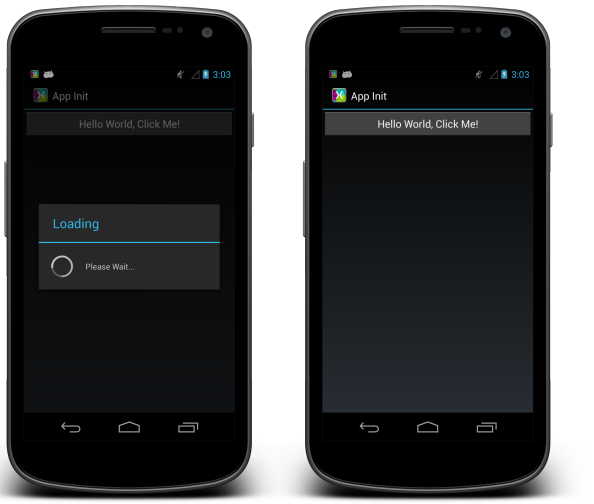
The Complete App

Running the app, we can see that a number of lifecycle scenarios and functionality are covered and we have a performant, leak-free application that behaves well under a number of conditions:

Waiting for App Initialization

The previous example illustrates the Activity lifecycle being exercised and provides guidance for a number of scenarios while the app is running. However, many applications need to perform initialization when the app is launched, such as starting services, performing authentication, accessing encrypted storage, etc., prior to allowing users to begin interacting with the app.

In this scenario, it's important that the app still launch quickly, so that it feels performant, but still prevent usage until it's fully initialized. A common approach here is to initialize things on a background thread, and show a loading screen until it's ready to go:



The *AppInitialization* project in the *App_Lifecycle_Demos* solution illustrates how to do exactly that, and the following code samples are from that application.

Singleton App Pattern

The rest of the samples in this chapter expand on the App singleton class introduced earlier, but the *AppInitialization* project adds a small bit of important code:

```
public class App
{
        public event EventHandler<EventArgs> Initialized = delegate {};
        public bool IsInitialized { get; set; }

        public static App Current
        {
                get { return current; }
        } private static App current;

        static App ()
        {
                current = new App();
        }
        protected App ()
        {
                new Task (() => {
                        // any real world inits should go here, such as
                        // starting services, database init, web calls, etc.

                        this.IsInitialized = true;
                        this.Initialized (this, new EventArgs ());
                }).Start ();
        }
}
```

Now, in addition to simply having an App singleton class, we've added two important things:

- *Initialized* Event – The class constructor now does initialization work on a separate thread and raises an Initialized event when it's finished.
- *IsInitialized* Property – In addition to the event, a Boolean property is available to determine if it has been initialized.

As a side benefit, this pattern provides a static way to access application wide members. To access it, we can simply use the `App.Current` property. For example, the following code snippet checks to see if the app has been initialized:

```
if ( App.Current.IsInitialized ) { ... }
```

If we had any other public members, they would be just as easily accessed. For example, if we had some sort of Authentication Manager class, and we wanted to see if the app has been authorized, we might call:

```
If ( App.Current.AuthManager.IsAuthed ) { ... }
```

And because we're raising an event when the app is initialized, we can wire up a handler to listen for the event:

```
App.Current.Initialized += (s, e) => { /* do work here */ };
```

In fact, we're going to examine doing just that in the next section.

> **Note:** Any work in the static constructor is likely to be blocking (static constructors run on whatever thread that first access its instance members, which in our case is an activity doing an initialization check), so we do it on a background thread here.

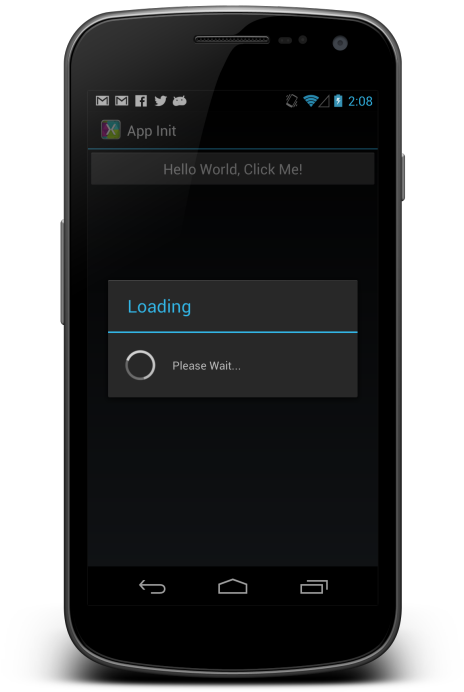
Showing a Loading Screen While the App is Initializing

It's all well and good to allow our app to initialize certain things in the background while it's loading up, however, as previously discussed, we want to wait for those things to initialize before the user begins interacting with the app. This can be a welcome user experience in making the app feel responsive because the app starts quickly, and then notifies the user it's doing some work.

One of the easiest ways to do this is to show a *ProgressDialog* in the `OnResume` method of our `MainActivity` class (or any activities that are registered as *Launchable*), as illustrated in the following code snippet:

```
if (!App.Current.IsInitialized){
        progress = ProgressDialog.Show(this, "Loading",
                "Please Wait...", true);
        App.Current.Initialized += (s, e) => {
                RunOnUiThread( () => {
                        this.FinishedInitializing();
                        // hide the progress bar
                        if (progress != null)
                                progress.Dismiss();
                });
        };
}
```

This code shows our `ProgressDialog` and then hides it when the app has finished initializing. The `ProgressDialog` is a modal overlay that prevents user interaction with the activity below it:



While the code above works great, it presents a bit of a limitation, in that each time we wanted to use this pattern, we'd have to put this code into every activity that we want this functionality on. It also makes our `OnResume` code kind of messy.

However, through the magic of inheritance, we can create a base Activity class so we have a single, reusable file, and wherever we want activities to use it, we simply inherit from this new class. Additionally, we can provide a method for the derived classes to put code into that will execute after the app is initialized, as shown in the following `ActivityBase` code:

```
public abstract class ActivityBase : Activity
{
        ProgressDialog progress;

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

                if (!App.Current.IsInitialized){
                        progress = ProgressDialog.Show(this, "Loading",
                                "Please Wait...", true);

                        App.Current.Initialized += (s, e) => {
                                RunOnUiThread( () => {
                                        this.FinishedInitializing();
                                        if (progress != null)
                                                progress.Dismiss();
                                });
```

```
                };
            }
        }

        /// <summary>
        /// Override this method to perform tasks after the app class
        /// has fully initialized
        /// </summary>
        protected abstract void FinishedInitializing ();
    }
```

Now, we can get the loading functionality in our activities by simply inheriting from `ActivityBase`:

```
public class MainActivity : ActivityBase
```

And then override `FinishedInitializing` to provide a place for any post-initialization code to run:

```
protected override void FinishedInitializing ()
{
        // do post initialization work here
}
```

Now we have a fully re-usable, drop in way to handle app initialization on any activity!

Modifying the Pattern to be Activity Lifecycle Compliant

There is still one issue with this pattern. Recall from our first application example, *ApplicationLifecycle*, if the application has a configuration change, such as the device being rotated, `OnResume` will be called at least twice. Once when the Activity is started, and then once for every time the device is rotated because Android tears down and recreates the Activity each time. And because that handler is still around (per garbage collection rules, since there's a reference to it on the App class now), it'll still get called.

To prevent it from getting called multiple times, we need to be a good App citizen and remove the handler during `OnPause`, which is called by the OS before the activity gets torn down. In order to do that, however, we have to keep a reference to the handler around, rather than using an anonymous delegate. So, to fix, first we declare the handler at a class level:

```
public abstract class ActivityBase : Activity
{
        protected EventHandler<EventArgs> initializedEventHandler;
        ...
}
```

Then, when we wire the handler up, we actually wire it up to a reference to the handler, like so:

```
initializedEventHandler = (s, e) => {
        RunOnUiThread( () => {
                this.FinishedInitializing();
                if (progress != null)
                        progress.Dismiss();
```

```
        });
};
App.Current.Initialized += initializedEventHandler;
```

Finally, we need to override the `OnPause` event method, and remove the handler there:

```
protected override void OnPause ()
{
        base.OnPause ();
        if (this.initializedEventHandler != null)
                App.Current.Initialized -= initializedEventHandler;
        if (progress != null)
                progress.Dismiss ();
}
```

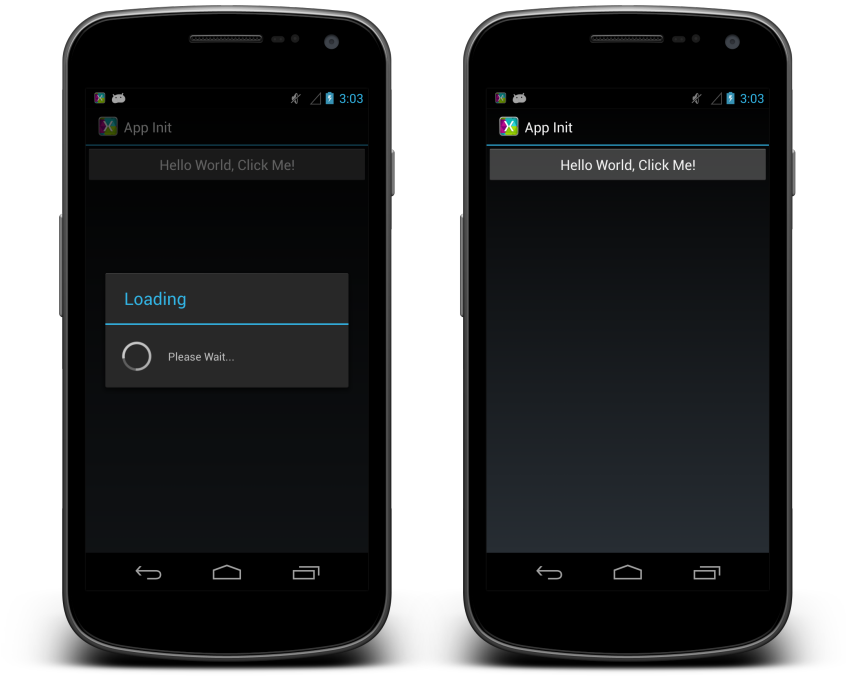
Notice that we're also dismissing our `ProgressDialog`. If we didn't, android will keep a reference to the Activity around and it will leak.

Our final `ActivityBase` now looks like this:

```
public abstract class ActivityBase : Activity
{
        ProgressDialog progress;
        protected EventHandler<EventArgs> initializedEventHandler;

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

                if (!App.Current.IsInitialized){
                        progress = ProgressDialog.Show(this, "Loading",
                                "Please Wait...", true);

                        initializedEventHandler = (s, e) => {
                                RunOnUiThread( () => {
                                        this.FinishedInitializing();
                                        if (progress != null)
                                                progress.Dismiss();
                                });
                        };
                        App.Current.Initialized += initializedEventHandler;
                }
        }

        protected override void OnPause ()
        {
                base.OnPause ();
                if (this.initializedEventHandler != null)
                        App.Current.Initialized -= initializedEventHandler;
                if (progress != null)
                        progress.Dismiss ();
        }

        protected void RestartApp ()
        {
```

```
                Log.Debug (logTag, "ActivityBase.RestartApp");
                StartActivity (typeof(MainActivity));
                base.Finish ();
            }

            protected abstract void FinishedInitializing ();
        }
```

If we run the *AppInitilization* project, we can see this pattern in action:



Next, we can extend this pattern to handle application errors.

Handling Crashes and Waiting for App Initialization

Unhandled exceptions in an Activity or Service will, at a minimum, cause Android to destroy them, but more often than not, Android will destroy the entire application process.

When this happens, Android will try to restart the application from the penultimate Activity. If that Activity fails to start, then Android will move backwards in the navigation stack and continue launching activities until one successfully work.

In activities that require app initialization, this could cause a real problem, as they can extend the chain reaction of failing activities and the user experience would be poor.

Fortunately, the loading pattern we just created works great and has a beneficial side effect in that if the app crashes, as long as the Activity that is re-launched

inherits from our ActivityBase, they'll wait for the app to load and then be on their merry way.

However, for some apps, even this can cause a bit of a data and/or UX dilemma, and it may make sense to relaunch the initial activity.

Additionally, when an unhandled exception occurs, we probably want to log it. Especially while in development.

In the *HandlingCrashes* project in the *App_Lifecycle_Demos* solution we're going to take a look at an app that does just that: it ensures in a crash that the `MainActivity` is relaunched, and also attempts to log unhandled exceptions.

Let's first look at how to log app crashes, and then we'll examine how to force the application to go back to the first activity.

Logging Unhandled Exceptions at the App Level

As with other .NET/Mono applications, we can subscribe to them in via the `UnhandledException` event on the current *AppDomain*. For example, the following code snippet is in the constructor of the `App` class and does just that:

```
// subscribe to app wide unhandled exceptions so that we can log them.
AppDomain.CurrentDomain.UnhandledException += HandleUnhandledException;
```

Our handler then looks like this:

```
protected void HandleUnhandledException (object sender,
         UnhandledExceptionEventArgs args)
{
        Exception e = (Exception) args.ExceptionObject;
        Console.WriteLine ("========= MyHandler caught : " + e.Message);
}
```

An important note to observe here however; in the event of an unhandled exception in which Android is destroying the process, we will likely not have access to Android objects, so if we want to log errors, we have to rely on objects in our application. For instance, while we can call `Console.WriteLine`, calling the `Android.Util.Log` class would just throw another exception.

Let's look now at how to ensure that a particular Activity is launched in the event of a crash.

Forcing the Launch of the First (or other) Activity

In the `App` class of the *HandlingCrashes* project, there is a public field that allows us to configure whether or not to restart the application back at the main activity in the case of a crash. Setting it to true will cause the app to restart back to square one:

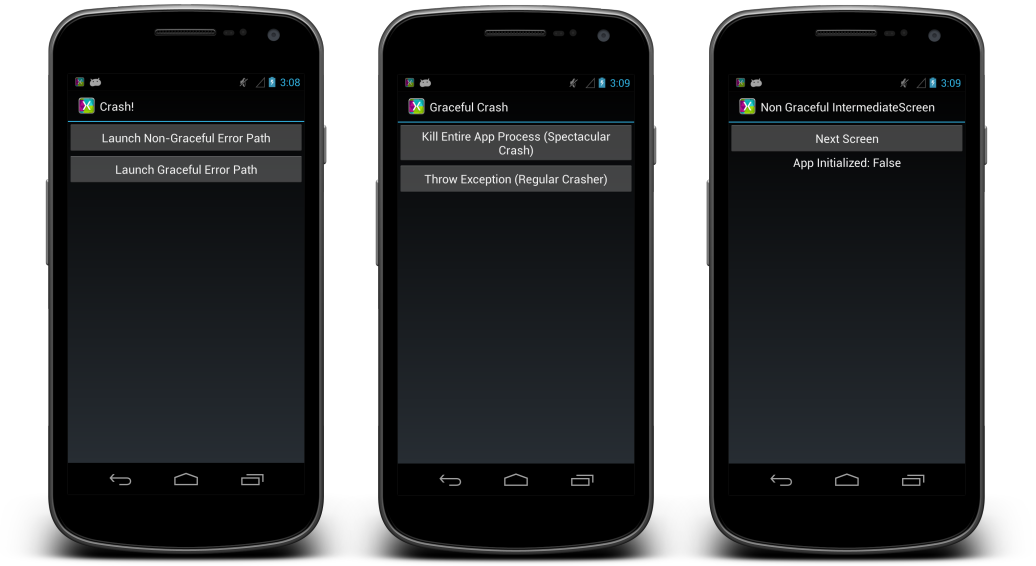
```
public readonly bool RestartMainActivityOnCrash = true;
```

To use this then, we only need to apply a small logic change to the `OnCreate` method in our `ActivityBase` class to check to see whether or not the current activity is the `MainActivity`, and if not, start the `MainActivity` and finish the current Activity:

```
if (!App.Current.IsInitialized){
        if (App.Current.RestartMainActivityOnCrash
                && (this.GetType() != typeof(MainActivity))){
                StartActivity (typeof(MainActivity));
                base.Finish ();
                return;
        } else {
                progress = ProgressDialog.Show(this, "Loading",
                        "Please Wait...", true);

                initializedEventHandler = (s, e) => {
                        RunOnUiThread( () => {
                                this.FinishedInitializing();
                                if (progress != null)
                                        progress.Dismiss();
                        });
                };
                App.Current.Initialized += initializedEventHandler;
        }
}
```

Of course, we could extend this logic in an infinite number of ways in order to restart on other activities as well, depending on the current state of the app.

We can see this in action by launching the HandlingCrashes application, and running it. There are two paths that can be taken through the application, a Graceful Error Path, and a Non-Graceful Error Path. The Graceful Error Path illustrates the app waiting for initialization, regardless of whether or not the `RestartMainActivityOnCrash` field has been set to true, and re-launching at the `MainActivity`, if it has. The Non-Graceful Path illustrates a normal crash scenario, with the application starting at the penultimate activity, and not waiting for the app to initialize:



Let's look at one final app scenario, in which we need to wait for multiple asynchronous items to initialized before user interaction.

Waiting for Multiple Async Items to Initialize at App Start

In the previous examples, we created a framework for waiting for items to initialize before we allowed the application to accept user input. This is great for synchronous things that can be called one after another. However, sometimes you need to initialize multiple, asynchronous processes. A good example of this are starting and connecting to Bound Services. Binding to services is asynchronous by nature because you create a connection and Android goes and finds the service for you, and then at some point notifies you that it has been connected, or *Bound*.

If we have multiple asynchronous items to initialize, we don't want to raise the `App.Initialized` event until they have all completed their initialization.

An easy pattern that solves this is to keep a count of the items that need to initialized and then increment that count every time one gets initialized. When the count has reached the number of items that need to be completed, we can then raise this event.

To implement this, we first need to declare how many initializations have to occur, and how many we're current at (as well as a locker object, which we'll see why in a moment), in our `App` class. For example, if we have two async items that need to be initialized, we would have the following:

```
public class App
{
	protected readonly int totalInitCount = 2;
	protected int currentInitCount = 0;
	protected object locker = new object();
	...
}
```

Then, we need to create a method that increments our `currentInitCount`, and when it's reached it's threshold (`totalInitCount`), we raise our Initialized event:

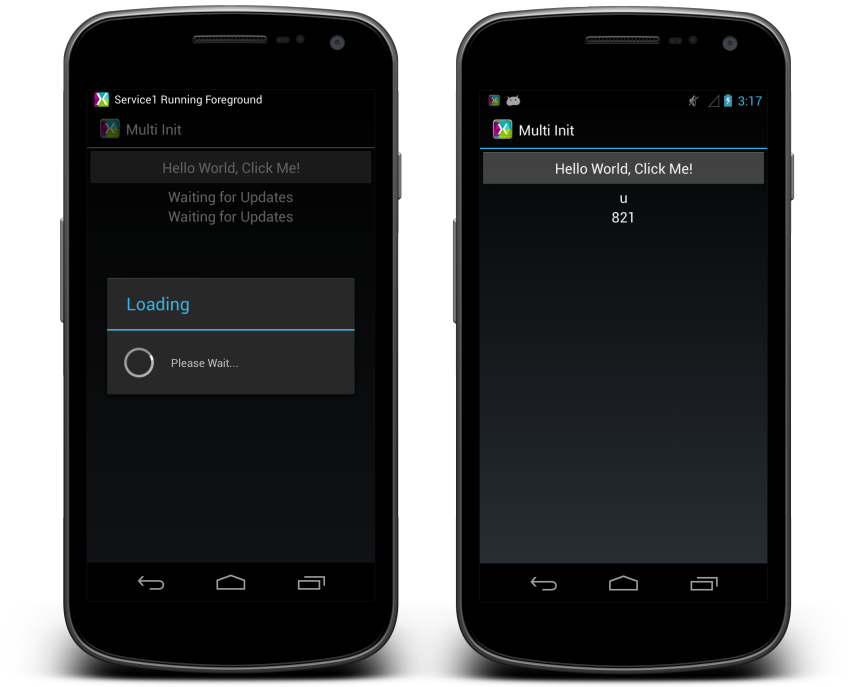
```
protected void IncrementInitCount()
{
	lock (this.locker) {
		Log.Debug (logTag, "App.IncrementInitCount");
		this.currentInitCount++;
		if (this.currentInitCount == this.totalInitCount) {
			this.IsInitialized = true;
			this.Initialized (this, new EventArgs ());
		}
	}
}
```

Notice that we use our locker object here to create a critical section so that multiple threads can access this and still only raise the event once.

To use this counting method, we just have to make sure that as items get initialized, we call `IncrementInitCount`. For example, the following code, from the *InitializingAppAndServices* demo project, illustrates starting and binding to two different services in the App constructor:

```
protected App ()
{
        new Task ( () => {
                // start the services
                Android.App.Application.Context.StartService (
                        new Intent ( Android.App.Application.Context,
                        typeof(Service1) ) );
                Android.App.Application.Context.StartService (
                        new Intent ( Android.App.Application.Context,
                        typeof(Service2) ) );

                // create their connections
                this.service1Connection =
                        new ServiceConnection<Service1> (null);
                this.service1Connection.ServiceConnected += (object sender,
                        ServiceConnectedEventArgs<Service1> e) => {
                        this.Service1Connected ( this, e );
                        this.IncrementInitCount();
                };

                this.service2Connection =
                        new ServiceConnection<Service2> (null);
                this.service2Connection.ServiceConnected += (object sender,
                        ServiceConnectedEventArgs<Service2> e) => {
                        this.Service2Connected ( this, e );
                        this.IncrementInitCount();
                };

                // bind them
                Intent service1Intent =
                        new Intent (Android.App.Application.Context,
                        typeof(Service1));
                Intent service2Intent =
                        new Intent (Android.App.Application.Context,
                        typeof(Service2));
                Android.App.Application.Context.BindService
                        ( service1Intent, service1Connection,
                        Bind.AutoCreate );
                Android.App.Application.Context.BindService
                        ( service2Intent, service2Connection,
                        Bind.AutoCreate );
        } ).Start ();
}
```

If we run this, the app shows a loading screen while the services initialize, and then when they're both up, the loading screen gets dismissed and the app is usable:



In this example, we used services, but any kind of asynchronous initializations could be used here, such as web requests or other async processes.

Summary

In this chapter, we covered a lot of interesting advanced application lifecycle scenarios, as well as the Activity lifecycle as it relates to them. In doing so, we also examined patterns for dealing with them and creating well behaved, performant applications.