Responding to the Activity Lifecycle

Xamarin Android Level 1, Chapter 5

Xamarin Inc.

Overview

Activities are an unusual programming concept specific to Android. In traditional application development there is usually a `static main` method, which is executed to launch the application. With Android, however, things are different; Android applications can be launched via any registered activity within an application. In practice, most applications will only have a specific activity that is specified as the application entry point. However, if an application crashes, or is terminated by the OS, the OS can try to restart the application at the last open activity or anywhere else within the previous activity stack. Additionally, the OS may pause activities when they're not active, and reclaim them if it is low on memory. Careful consideration must be made to allow the application to correctly restore its state in the event that an activity is restarted and in case that activity depends on data from previous activities.

The activity lifecycle is implemented as a collection of methods the OS calls throughout the lifecycle of an activity. These methods allow developers to implement the functionality that is necessary to satisfy the state and resource management requirements of their applications.

It is extremely important for the application developer to analyze the requirements of each activity to determine which methods exposed by the activity lifecycle need to be implemented. Failure to do this can result in application instability, crashes, resource bloat, and possibly even underlying OS instability.

This chapter examines the activity lifecycle in detail, including:

- ➔ Activity States
- ➔ Lifecycle Methods
- ➔ Retaining the State of an Application

This chapter also includes two walkthroughs that provide practical examples on how to efficiently save state during the Activity lifecycle. By the end of this chapter you should have an understanding of the Activity lifecycle and how to support it in an Android application.

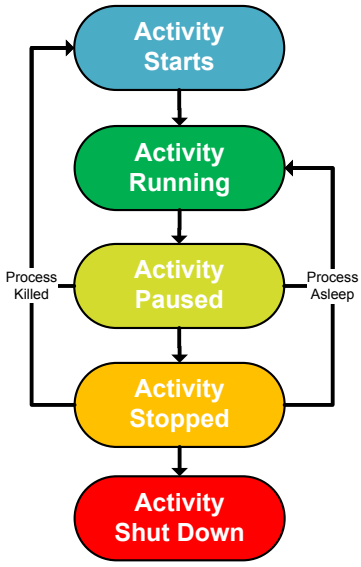
Activity Lifecycle

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 uses a priority queue to assist in managing activities running on the device. Based on the state a particular Android activity is in, it will be assigned a certain priority within the OS. This priority system 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:



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 - 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 was paused to stopped was dropped from memory by Android. If the user navigates back to the activity it must be restarted, restored to it's previously saved state, and then displayed to the user.

Activity Re-Creation in Response to Configuration Changes

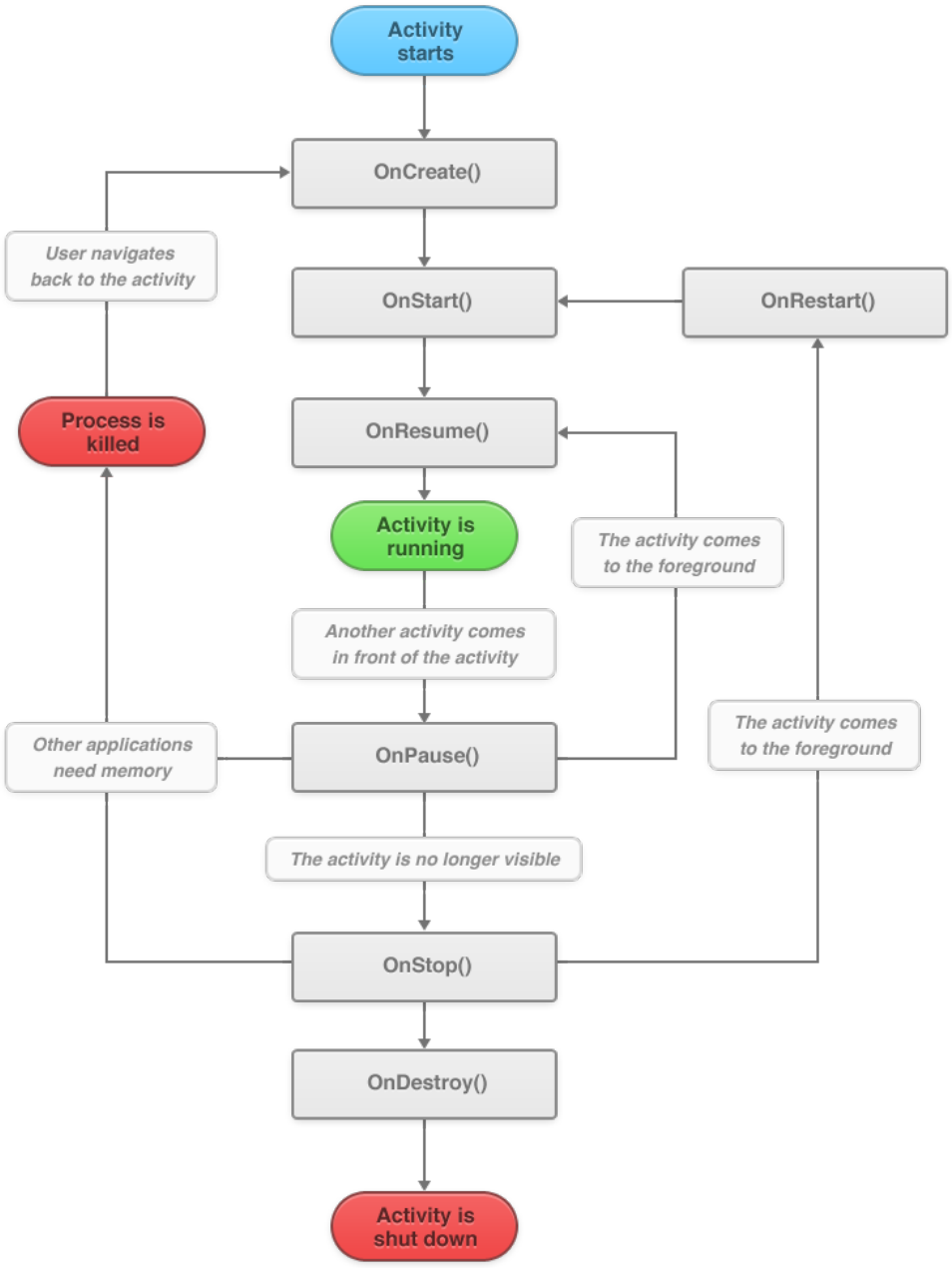
To make matters more complicated, in addition to the lifecycle states that occur when an activity is placed or retrieved from the background, etc., Android throws one more wrench in the mix called *Configuration Changes*. Configuration changes are rapid activity destruction/re-creation cycles that occur when the configuration of an activity changes, such as when the device is rotated (and the activity needs to get re-built in landscape or portrait mode), when the keyboard is displayed (and the activity is presented with an opportunity to resize itself), or when the device is placed in a dock, among others.

Configuration Changes still cause the same Activity State changes that would occur during stopping and restarting an activity. However, in order to make sure that an application feels responsive and performant during Configuration Changes, it's important that they be handled as quickly as possible. Because of this, Android has a specific API that can be used to persist state during them. We'll cover this later in the *Managing State Throughout the Lifecycle* section.

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 those methods and their names:



As a developer, you can handle state changes by overriding these methods within an activity. Below is a description of each lifecycle method:

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

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:

- ➔ Starting animations
- ➔ Listening for GPS updates

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
    }
}
```

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 cleanup other objects consuming resources
- ➔ Ramp down frame rates

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, while `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.

An Activity should override this method to handle any expensive or time-consuming cleanup that would slow down `OnPause`. This will allow the next Activity to promptly start while the current Activity is cleaning up.

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 cleanup and shut down has been done in the `OnPause` and `OnStop` methods. This method is typically overridden to cleanup 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 general guidelines for what kind of logic should be implemented `OnRestart`. This is because `OnStart` is always invoked regardless of the Activity is being created or being restarted, so any resources required by the Activity should be initialized in `OnStart`.

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

Managing the State Throughout the Lifecycle

When an Activity is stopped or destroyed the system provides an opportunity to save the state of the Activity for later rehydration. This saved state is referred to as *instance state*. Android provides three options for storing instance state during the Activity lifecycle. Which one you use depends on two factors; what kind of data you want to store, and whether or not the lifecycle state is changing due to a configuration change:

1. **Using A Bundle –** The first and foremost option for saving instance state is to use a key/value dictionary object know as a *Bundle*. Recall that when an Activity is created that the `OnCreate` method is passed a Bundle as a parameter, this Bundle can be used to restore the instance state. An Activity also has special APIs for saving and retrieving state from a Bundle. This option is best used when the state needed to be saved can easily/quickly serialized into string key/value pairs, but is not recommended for more complex data that isn't so easily (or quickly serialized).
2. **Using a Stateful Object during a Configuration Change –** A Bundle may not be the most efficient way to persist the instance state during a Configuration Change. For example, bitmaps may be too large to store a bundle, or may take too long to serialize and deserialize. In these circumstances it is possible for the Activity to instantiate a custom object and store the state in that object.
3. **Manually Handling the Configuration Change –** The last option available is for an Activity to assume all control over the configuration change. Android will not destroy and recreate the Activity – instead the Activity is responsible for handling all aspects of the Configuration Change. This option

is not recommended unless absolutely necessary and will not be covered in this chapter. For more information, see the Android documentation here.

Let's examine each of these options.

Using A Bundle

The primary mechanism that an Activity should use to persist its state is the `Bundle`. An Activity provides methods to help with saving and retrieving the instance state in the `Bundle`:

- **OnSaveInstanceState** – This is invoked by Android when the activity is being destroyed. Activities can implement this method if they need to persist any key/value state items.
- **OnRestoreInstanceState** – This is called after the `OnCreate` method is finished, and provides another opportunity for an Activity to restore it's state after initialization is complete.

The following diagram illustrates how these methods are used:

ONSAVEINSTANCESTATE

This method will be called as the Activity is being stopped. It will receive a Bundle parameter that the Activity can store its state in. For example, the following code snippet illustrates saving some simple state data:

```
protected override void OnSaveInstanceState(Bundle outState)
{
   outState.PutString("myString", "Hello Xamarin.Android
OnSaveInstanceState");
   outState.PutBoolean("myBool", true);

   base.OnSaveInstanceState(outState);

}
```

It is important to always call the base implementation of `OnSaveInstanceState` so that the state of the view hierarchy can also be saved.

ONRESTOREINSTANCESTATE

This method will be called after `OnStart`. It provides an activity the opportunity to restore any state that was previously saved to a `Bundle` during the previous `OnSaveInstanceState`. This is the same Bundle that is provided to `OnCreate`, however. As such, in many cases, it's not neccessary to override `OnRestoreInstanceState`, since most activities can restore state using the Bundle provided to `OnCreate`.

This method exists to provided some flexibility around when state should be restored – sometimes it is more appropriate to wait until all initializations are done before restoring instance state or a subclass of an existing Activity may only want to restore certain values from the instance state.

The following code demonstrates how state can be restored in `OnSaveInstanceState`:

```
protected override void OnSaveInstanceState(Bundle savedState)
{
    base.OnRestoreSaveInstanceState(savedState);
    var myString = savedState.GetString("myString");
    var myBool = GetBoolean("myBool");
}
```

Using a Stateful Object During a Configuration Change

The Activity restart that occurs during a configuration change may have a noticeable impact on an application. For example, an application may require a large amount data that it downloaded from the internet. A configuration change would force the Activity to download that data again, resulting in unwanted delays for the user. As another example it may be time consuming to serialize the state, store it in the `Bundle`, and then deserialize the bundle.

In these situations it is possible to use a stateful object between the two instances of the Activity. To do this, an Activity must implement the method `OnRetainNonConfigurationInstance` and return an object containing the instance state. Android will store this object in the `LastNonConfigurationInstance` property of the following Activity instance. The general pattern for this is as follows:

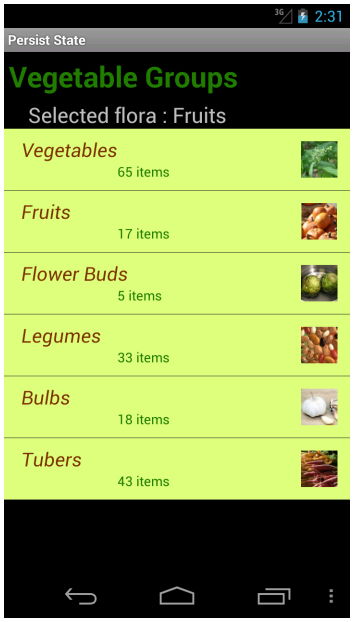
1. Create a new class that will be used to store the state.
2. Implement the method `OnRetainNonConfigurationInstance` so that it will instantiate an instance of this new class, populate it with data, and then return that instance.
3. Update `OnCreate` to check the value of the property `LastNonConfigurationInstance`. If this property is holding an instance of the new class, restore state from that.

Activity State Walkthrough

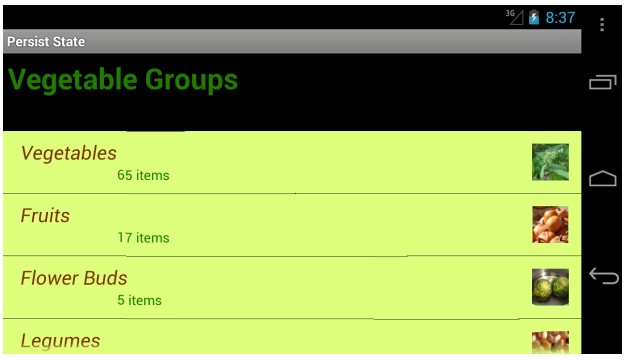
Learning about the activity lifecycle and actually being able to implement it are two different things. As such, let's take what we've learned here and apply it to one of

the applications we created in the last chapter so that it retains state during configuration changes and lifecycle changes due to backgrounding/foregrounding.

Recall from the last chapter the application that displays plants in a customized list:



The application creates a list of `Flora` objects each time the Activity is created. When a user selects an item from the list, the Activity displays the selection in a `TextView`. However, when the application is rotated, the currently select plant is lost – this is because the Activity does not properly persist its state during lifecycle changes:



There is also a second issue with this application; a noticeable pause on the configuration change as the Activity recreates the Flora list. Let's update the Activity to address these three issues by making the following changes:

- ➔ Save the selected Flora to instance state when the application goes into the background.
- ➔ Display the selected Flora at the appropriate point in the Activity lifecycle
- ➔ Preserve the list of Flora objects during a configuration change.

Let's get started with the first two of these three changes.

1. First, download the file `05 - Responding to the Activity Lifecycle - Persist State - start.zip` and extract the contents of that file. This file contains a starter project that we will modify throughout this walkthrough.

2. Add an instance variable named `selectedPosition` to `HomeScreen` to hold the position of the item that the user selected:

```
[Activity(Label = "Persist State", MainLauncher = true, Icon =
"@drawable/icon")]
public class HomeScreen : Activity
{
    private List<Flora> listOfFlora = new List<Flora>();
    private ListView listView;
    private int selectedPosition = -1;
```

3. Now we need to add a method to display the selected item in the user interface. Add the following method to `Activity1`:

```
private void DisplayLastSelectedFlora()
{
    if (selectedPosition > -1)
    {
        var flora = listOfFlora [selectedPosition];
        lastSelectedFloraText.Text = "Selected flora : " + flora.Name;
    }
}
```

This method will be used by changes we make in the following steps.

4. Next, edit the method `OnListItemClick` to update this new variable each time the user makes a selection from the list. Change the method so that it contains the code displayed below:

```
protected void OnListItemClick(object sender,
AdapterView.ItemClickEventArgs e)
{
    selectedPosition = e.Position;
    DisplayLastSelectedFlora();
}
```

5. Now that we're saving the position, we need to make sure it gets persisted in instance state. To do this, override the method `OnSaveInstanceState` on the Activity `HomeScreen`:

```
protected override void OnSaveInstanceState(Bundle outState)
{
    base.OnSaveInstanceState(outState);
    outState.PutInt("listview_selecteditemposition", selectedPosition);
}
```

6. At this point the Activity will now save the last selected item to instance state, but it will not restore it. This next change will restore the instance state when the Activity is created. Update the `OnCreate` method so that it will retrieve the instance state from the Bundle:

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

```
        SetContentView(Resource.Layout.HomeScreen);
        listView = FindViewById<ListView>(Resource.Id.List);

        listView.Adapter = new HomeScreenAdapter(this, listOfFlora);
        listView.ItemClick += OnListItemClick;
        if (bundle != null)
        {
            selectedPosition = bundle.GetInt("listview_selecteditemposition",
-1);
        }
    }
```

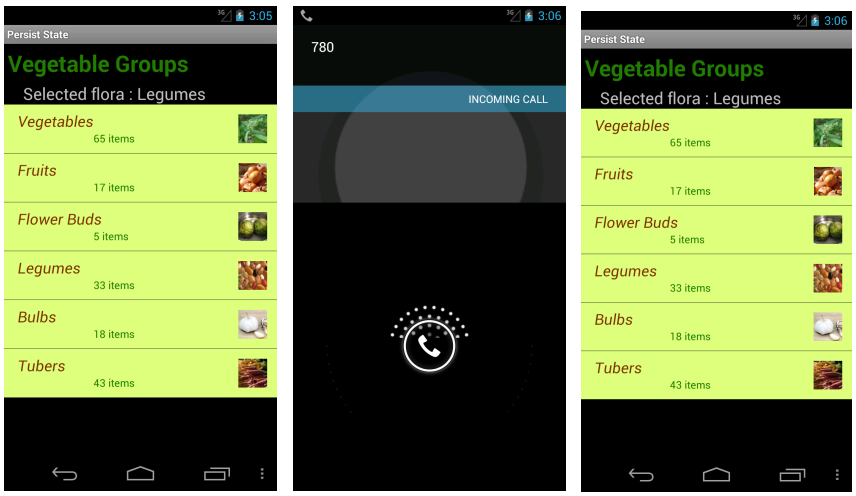
If the bundle is not null, we will restore the value of `selectedPosition` from the previous instance of this activity. If by some chance the key value `listview_selecteditemposition` is not in the Bundle, then `selectedPosition` will be initialized to a default value of `-1`.

7. Next we need ensure that the UI is updated to show the selected Flora. Override `OnResume` so that it looks like the following:

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

        DisplayLastSelectedFlora();

        // This will cause the list to scroll so that the selected item is in
the view.
        listView.SetSelection(selectedPosition);
    }
```

We update the UI in `OnResume` because `OnResume` will always be called before the Activity is displayed to the user. `OnCreate` is only called when an instance of the Activity is being created, and not when the Activity moves from background to foreground.

At this point we've implement the first two of our three changes – preserving the selected index in instance state and ensuring that the UI displays the last selected. Now when this Activity returns to the foreground it will display the name of last selected `Flora` object. The following screenshots illustrates what happens when we run the application now and get a phone call, which puts the application in a background state. Notice that the selected flora is still populated:



Finally, we will tackle the last change - we don't want the user to have to experience delay caused by the Activity in creating the Flora list each time the device is rotated. Lets use a custom object to persist the list of `Flora` on a configuration change.

8. We will need a class for the retained state. Add a new class to the project called `HomeScreenRetainedState` and edit it to contain the following code:

```
public class HomeScreenState: Java.Lang.Object
{
        public HomeScreenState (List<Flora> list)
        {
                ListOfFlora = list;
        }

        public List<Flora> ListOfFlora { get; private set; }
}
```

This class must subclasses `Java.Lang.Object`. This is because `OnRetainNonConfigurationInstance` returns `Java.Lang.Object`, a limitation imposed by the Android SDK that Xamarin.Android must respect.

9. Now, lets modify the Activity `HomeScreen` and implement the method `OnRetainNonConfigurationInstance` to instantiate a `HomeScreenRetainedState` object and store the list of `Flora` objects:

```
public override Java.Lang.Object OnRetainNonConfigurationInstance ()
{
        var savedState = new HomeScreenState (listOfFlora);
        return savedState;
}
```

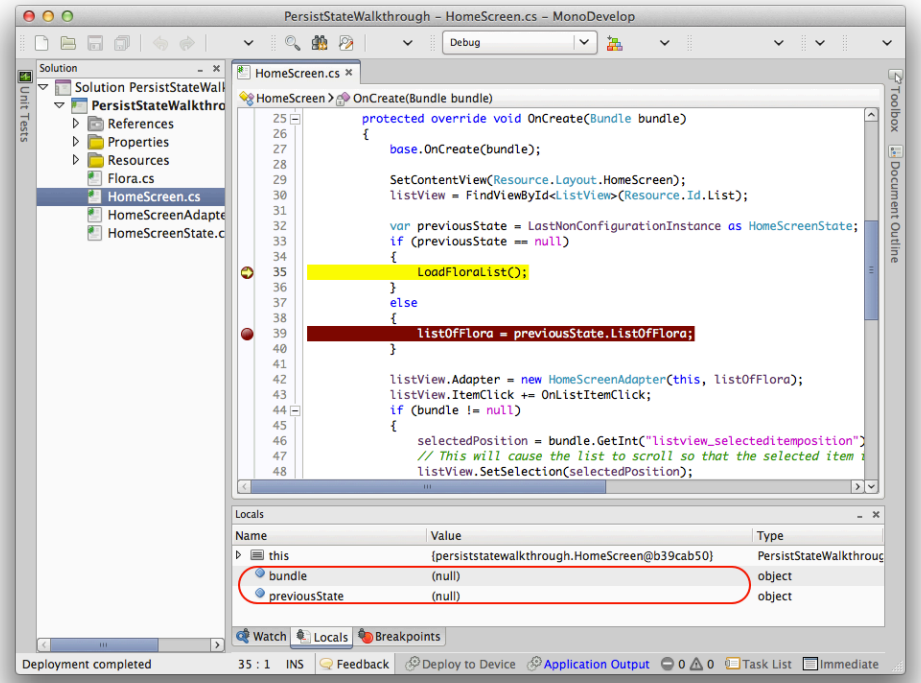
10. The final step is to use this instance of `HomeScreenState` when the Activity is recreated. Edit the method `OnCreate` so that it matches the following code:

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

    SetContentView(Resource.Layout.HomeScreen);
```

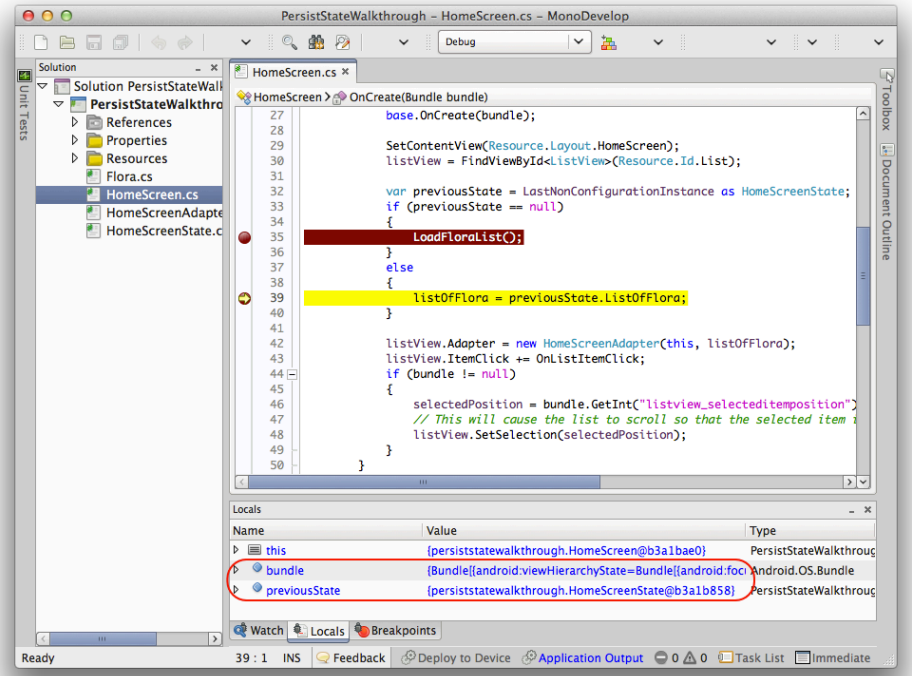
```
        listView = FindViewById<ListView>(Resource.Id.List);

        var previousState = LastNonConfigurationInstance as HomeScreenState;
        if (previousState == null)
        {
            LoadFloraList();
        }
        else
        {
            // Use the listOfFlora from the previous instance of this Activity.
            listOfFlora = previousState.ListOfFlora;
        }

        listView.Adapter = new HomeScreenAdapter(this, listOfFlora);
        listView.ItemClick += OnListItemClick;
        if (bundle != null)
        {
            selectedPosition = bundle.GetInt("listview_selecteditemposition");
            // This will cause the list to scroll so that the selected item is
    in the view.
            listView.SetSelection(selectedPosition );
        }
    }
```

In the changes above, `LastNonConfigurationInstance` is checked to see if it contains a `HomeScreenRetainedState` object. If it does not, then a new `List<Flora>` will be created for the `HomeScreenAdapter`. If it does, then the `List<Flora>` from the previous instance will be used.

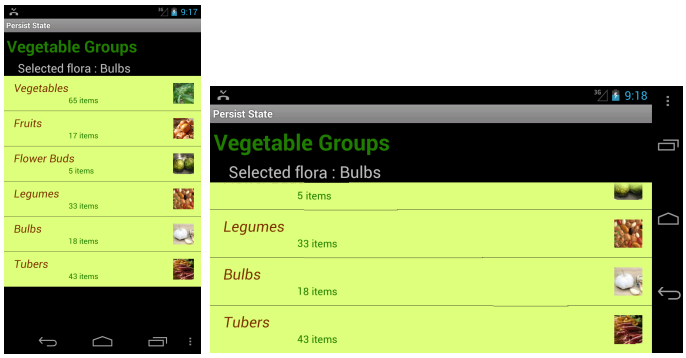
It will not be possible to directly observe what is going on, so set a breakpoint in the `OnCreate` method after the declaration and initialization of `previousState`. Run the application in the Android emulator. Notice that as the application is being created, the value `previousState` is `null`, and the Activity will create a new `List<Flora>`. The following screenshot shows the state of the Activity as it is being created for the first time:



Next simulate a configuration change by pressing Ctrl-F12 (or Fn-Ctrl-F12 on OS X). The emulator will switch to landscape mode and the value of `previousState` is no longer `null`, and the `List<Flora>` from the previous instance will be used. The following screen shot shows the state of the Activity as it is being created after a configuration change:



The following screenshots show the application running first in portrait mode, and then rotated to landscape mode:



Now that we're using configuration state, the activity allows the rotation (configuration change) to occur much more quickly, which makes the application feel much more responsive and performant.

Congratulations! The activity lifecycle is one of the trickiest concepts to deal with when building Android applications, and you've just finished modifying an application to deal with it properly. At this point you should have a solid understanding of the activity lifecycle and how to create applications that are performant and well-behaved during activity lifecycle change. For a completed example of the modifications we just did, see the `05 - Responding to the Activity Lifecycle - Walkthrough 2 - final.zip` file.

Summary

The Android activity lifecycle provides a powerful framework for state management of activities within an application but it can be tricky to understand and implement. This chapter introduced the different states that an activity may go through during its lifetime, as well as the lifecycle methods that are associated with those states. Next, guidance was provided as to what kind of logic should be performed in each of these methods. Finally, we finished up by walking through giving a practical implementation of the lifecycle methods to handle activity lifecycle state changes. We saved both simple data in the bundle, as well as complex data in instance state during a configuration change, and modified the activity to rehydrate itself accordingly when it's resumed.