CSCI 322/522

7. Android Activity Lifecycles and State Changes

- In earlier chapters we have learned that Android applications run within processes and that they are comprised of multiple components in the form of activities, services and broadcast receivers.

- The goal of this chapter is to expand on this knowledge by looking at the lifecycle of applications and activities within the Android runtime system.

- Regardless of the fanfare about how much memory and computing power resides in the mobile devices of today compared to the desktop systems of yesterday, it is important to keep in mind that these devices are still considered to be "resource constrained" by the standards of modern desktop and laptop based systems, particularly in terms of memory.

- As such, a key responsibility of the Android system is to ensure that these limited resources are managed effectively and that both the operating system and the applications running on it remain responsive to the user at all times.

- In order to achieve this, Android is given full control over the lifecycle and state of both the processes in which the applications run, and the individual components that comprise those applications.

- An important factor in developing Android applications, therefore, is to gain an understanding of both the application and activity lifecycle management models of Android, and the ways in which an application can react to the state changes that are likely to be imposed upon it during its execution lifetime.

Android Applications and Resource Management

- Each running Android application is viewed by the operating system as a separate process.

- If the system identifies that resources on the device are reaching capacity it will take steps to terminate processes to free up memory.

- When making a determination as to which process to terminate in order to free up memory, the system takes into consideration both the priority and state of all currently running processes, combining these factors to create what is referred to by Google as an importance hierarchy.

- Processes are then terminated starting with the lowest priority and working up the hierarchy until sufficient resources have been liberated for the system to function.

Android Process States

- Processes host applications and applications are made up of components.

- Within an Android system, the current state of a process is defined by the highest-ranking active component within the application that it hosts.

- As outlined in Figure 7-1, a process can be in one of the following five states at any given time:

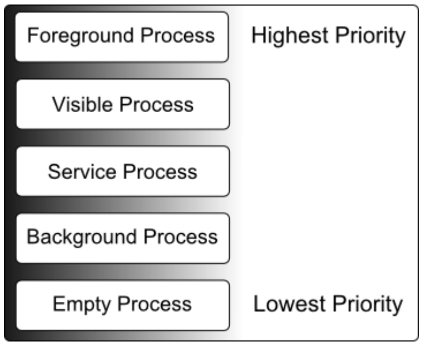


Figure 7-1

Foreground Process

- These processes are assigned the highest level of priority.
- At any one time, there are unlikely to be more than one or two foreground processes active and these are usually the last to be terminated by the system.
- A process must meet one or more of the following criteria to qualify for foreground status:
- Hosts an activity with which the user is currently interacting.
- Hosts a Service connected to the activity with which the user is interacting.
- Hosts a Service that has indicated, via a call to `startForeground()`, that termination would be disruptive to the user experience.
- Hosts a Service executing either its `onCreate()`, `onResume()` or `onStart()` callbacks.
- Hosts a Broadcast Receiver that is currently executing its `onReceive()` method.

Visible Process

- A process containing an activity that is visible to the user but is not the activity with which the user is interacting is classified as a “visible process”.
- This is typically the case when an activity in the process is visible to the user, but another activity, such as a partial screen or dialog, is in the foreground.
- A process is also eligible for visible status if it hosts a Service that is, itself, bound to a visible or foreground activity.

Service Process

Processes that contain a Service that has already been started and is currently executing.

Background Process

- A process that contains one or more activities that are not currently visible to the user, and does not host a Service that qualifies for Service Process status.

- Processes that fall into this category are at high risk of termination in the event that additional memory needs to be freed for higher priority processes.

- Android maintains a dynamic list of background processes, terminating processes in chronological order such that processes that were the least recently in the foreground are killed first.

Empty Process

- Empty processes no longer contain any active applications and are held in memory ready to serve as hosts for newly launched applications.

- This is somewhat analogous to keeping the doors open and the engine running on a bus in anticipation of passengers arriving.

- Such processes are, obviously, considered the lowest priority and are the first to be killed to free up resources.

Inter-Process Dependencies

- The situation with regard to determining the highest priority process is slightly more complex than outlined in the preceding section for the simple reason that processes can often be inter-dependent.

- As such, when making a determination as to the priority of a process, the Android system will also take into consideration whether the process is in some way serving another process of higher priority (for example, a service process acting as the content provider for a foreground process).

- As a basic rule, the Android documentation states that a process can never be ranked lower than another process that it is currently serving.

The Activity Lifecycle

- As we have previously determined, the state of an Android process is determined largely by the status of the activities and components that make up the application that it hosts.

- It is important to understand, therefore, that these activities also transition through different states during the execution lifetime of an application.

- The current state of an activity is determined, in part, by its position in something called the Activity Stack.

The Activity Stack

- For each application that is running on an Android device, the runtime system maintains an Activity Stack.

- When an application is launched, the first of the application's activities to be started is placed onto the stack.

- When a second activity is started, it is placed on the top of the stack and the previous activity is pushed down.

- The activity at the top of the stack is referred to as the active (or running) activity.

- When the active activity exits, it is popped off the stack by the runtime and the activity located immediately beneath it in the stack becomes the current active activity.

- The activity at the top of the stack might, for example, simply exit because the task for which it is responsible has been completed.

- Alternatively, the user may have selected a "Back" button on the screen to return to the previous activity, causing the current activity to be popped off the stack by the runtime system and therefore destroyed.

- A visual representation of the Android Activity Stack is illustrated in Figure 7-2.

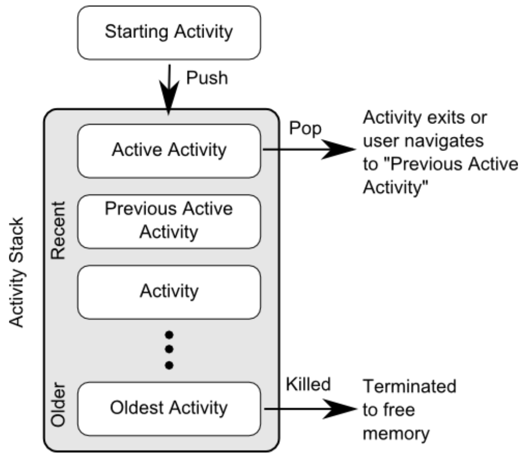


Figure 7-2

- As shown in the diagram, new activities are pushed on to the top of the stack when they are started.

- The current active activity is located at the top of the stack until it is either pushed down the stack by a new activity, or popped off the stack when it exits or the user navigates to the previous activity.

- In the event that resources become constrained, the runtime will kill activities, starting with those at the bottom of the stack.

- The Activity Stack is what is referred to in programming terminology as a Last-In-First-Out (LIFO) stack in that the last item to be pushed onto the stack is the first to be popped off.

Activity States

An activity can be in one of a number of different states during the course of its execution within an application:

- Active / Running – The activity is at the top of the Activity Stack, is the foreground task visible on the device screen, has focus and is currently interacting with the user.

  This is the least likely activity to be terminated in the event of a resource shortage.

- Paused – The activity is visible to the user but does not currently have focus (typically because this activity is partially obscured by the current active activity).

  Paused activities are held in memory, remain attached to the window manager, retain all state information and can quickly be restored to active status when moved to the top of the Activity Stack.

- Stopped – The activity is currently not visible to the user (in other words it is totally obscured on the device display by other activities).

  As with paused activities, it retains all state and member information, but is at higher risk of termination in low memory situations.

- Killed – The activity has been terminated by the runtime system in order to free up memory and is no longer present on the Activity Stack.

  Such activities must be restarted if required by the application.

Configuration Changes

- So far in this chapter, we have looked at two of the causes for the change in state of an Android activity, namely the movement of an activity between the foreground and background, and termination of an activity by the runtime system in order to free up memory.

- In fact, there is a third scenario in which the state of an activity can dramatically change and this involves a change to the device configuration.

- By default, any configuration change that impacts the appearance of an activity (such as rotating the orientation of the device between portrait and landscape, or changing a system font setting) will cause the activity to be destroyed and recreated.

- The reasoning behind this is that such changes affect resources such as the layout of the user interface and simply destroying and recreating impacted activities is the quickest way for an activity to respond to the configuration change.

- It is, however, possible to configure an activity so that it is not restarted by the system in response to specific configuration changes.

Handling State Change

- If nothing else, it should be clear from this chapter that an application and, by definition, the components contained therein will transition through many states during the course of its lifespan.

- Of particular importance is the fact that these state changes (up to and including complete termination) are imposed upon the application by the Android runtime subject to the actions of the user and the availability of resources on the device.

- In practice, however, these state changes are not imposed entirely without notice and an application will, in most circumstances, be notified by the runtime system of the changes and given the opportunity to react accordingly.

- This will typically involve saving or restoring both internal data structures and user interface state, thereby allowing the user to switch seamlessly between applications and providing at least the appearance of multiple, concurrently running applications.

- Android provides two ways to handle the changes to the lifecycle states of the objects within in app.

- One approach involves responding to state change method calls from the operating system and is covered in detail in an upcoming section titled Handling Android Activity State Changes.

- A new approach, and one that is recommended by Google, involves the lifecycle classes included with the Jetpack Android Architecture components introduced later.

- The activities and fragments that make up an application pass through a variety of different states during the course of the application's lifespan.

- The change from one state to the other is imposed by the Android runtime system and is, therefore, largely beyond the control of the activity itself.

- That does not, however, mean that the app cannot react to those changes and take appropriate actions.

- The primary objective of this chapter is to provide a high-level overview of the ways in which an activity may be notified of a state change and to outline the areas where it is advisable to save or restore state information.

- Having covered this information, the chapter will then touch briefly on the subject of activity lifetimes.

New vs. Old Lifecycle Techniques

- Up until recently, there was a standard way to build lifecycle awareness into an app.

- This is the approach covered in this chapter and involves implementing a set of methods (one for each lifecycle state) within an activity or fragment instance that get called by the operating system when the lifecycle status of that object changes.

- This approach has remained unchanged since the early years of the Android operating system and, while still a viable option today, it does have some limitations which will be explained later.

- With the introduction of the lifecycle classes with the Jetpack Android Architecture Components, a better approach to lifecycle handling is now available.

- This modern approach to lifecycle management (together with the Jetpack components and architecture guidelines) will be covered in detail later.

- It is still important, however, to understand the traditional lifecycle methods for a couple of reasons.

- First, as an Android developer you will not be completely insulated from the traditional lifecycle methods and will still make use of some of them.

- More importantly, understanding the older way of handling lifecycles will provide a good knowledge foundation on which to begin learning the new approach later in the book.

The Activity and Fragment Classes

- With few exceptions, activities and fragments in an application are created as subclasses of the Android `AppCompatActivity` class and `Fragment` classes respectively.

- Consider, for example, the project we created earlier to convert dollars to Euros.

- Locate the `MainActivity.java` file from that project and double-click on it to load it into the editor:

```
package edu.niu.your_Z-ID.androidsample;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
{
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }

  public void convertCurrency(View view)
  {
    EditText dollarText = findViewById(R.id.dollarText);
    TextView textView = findViewById(R.id.textView);
    if (!dollarText.getText().toString().equals(""))
    {
      Float dollarValue = Float.valueOf(dollarText.getText().toString());
      Float euroValue = dollarValue * 0.85F;
      textView.setText(euroValue.toString());
    }
    else
    {
      textView.setText(R.string.no_value_string);
    }
  }
}
```

- When the project was created, we instructed Android Studio also to create an initial activity named `MainActivity`.

- As is evident from the above code, the `MainActivity` class is a subclass of the `AppCompatActivity` class.

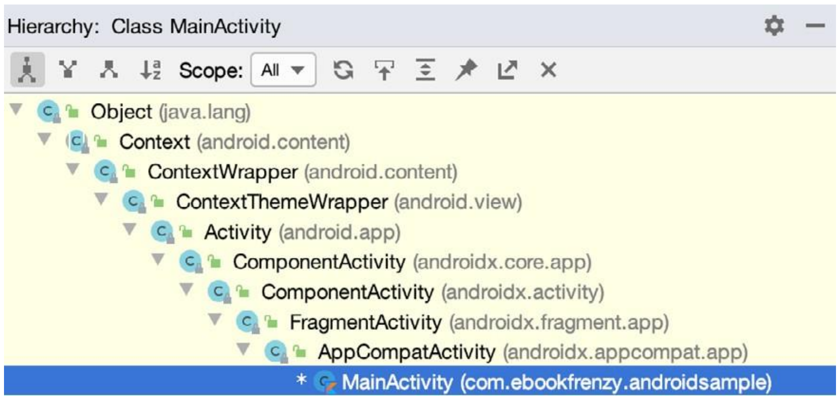
- A review of the reference documentation for the `AppCompatActivity` class would reveal that it is itself a subclass of the `Activity` class.

- This can be verified within the Android Studio editor using the Hierarchy tool window.

- With the `MainActivity.java` file loaded into the editor, click on `AppCompatActivity` in the class declaration line and press the Ctrl-H keyboard shortcut.

- The hierarchy tool window will subsequently appear displaying the class hierarchy for the selected class.

- As illustrated in Figure 7-3, `AppCompatActivity` is clearly subclassed from the `FragmentActivity` class which is itself ultimately a subclass of the `Activity` class:



```
Object (java.lang)
  Context (android.content)
    ContextWrapper (android.content)
      ContextThemeWrapper (android.view)
        Activity (android.app)
          ComponentActivity (androidx.core.app)
            ComponentActivity (androidx.activity)
              FragmentActivity (androidx.fragment.app)
                AppCompatActivity (androidx.appcompat.app)
                  * MainActivity (com.ebookfrenzy.androidsample)
```

Figure 7-3

- The `Activity` and `Fragment` classes contain a range of methods that are intended to be called by the Android runtime to notify the object when its state is changing.

- For the purposes of this chapter, we will refer to these as the lifecycle methods.

- An activity or fragment class simply needs to override these methods and implement the Handling Android Activity State Changes necessary functionality within them in order to react accordingly to state changes.

- One such method is named `onCreate()` and, turning once again to the above code fragment, we can see that this method has already been overridden and implemented for us in the `MainActivity` class.

- In a later section we will explore in detail both `onCreate()` and the other relevant lifecycle methods of the `Activity` and `Fragment` classes.

Dynamic State vs. Persistent State

- A key objective of lifecycle management is ensuring that the state of the activity is saved and restored at appropriate times.
- When talking about state in this context we mean the data that is currently being held within the activity and the appearance of the user interface.
- The activity might, for example, maintain a data model in memory that needs to be saved to a database, content provider or file.
- Such state information, because it persists from one invocation of the application to another, is referred to as the persistent state.
- The appearance of the user interface (such as text entered into a text field but not yet committed to the application's internal data model) is referred to as the dynamic state, since it is typically only retained during a single invocation of the application (and also referred to as user interface state or instance state).
- Understanding the differences between these two states is important because both the ways they are saved, and the reasons for doing so, differ.
- The purpose of saving the persistent state is to avoid the loss of data that may result from an activity being killed by the runtime system while in the background.
- The dynamic state, on the other hand, is saved and restored for reasons that are slightly more complex.
- Consider, for example, that an application contains an activity (which we will refer to as Activity A) containing a text field and some radio buttons.
- During the course of using the application, the user enters some text into the text field and makes a selection from the radio buttons.
- Before performing an action to save these changes, however, the user then switches to another activity causing Activity A to be pushed down the Activity Stack and placed into the background.
- After some time, the runtime system ascertains that memory is low and consequently kills Activity A to free up resources.
- As far as the user is concerned, however, Activity A was simply placed into the background and is ready to be moved to the foreground at any time.
- On returning Activity A to the foreground the user would, quite reasonably, expect the entered text and radio button selections to have been retained.
- In this scenario, however, a new instance of Activity A will have been created and, if the dynamic state was not saved and restored, the previous user input lost.
- The main purpose of saving dynamic state, therefore, is to give the perception of seamless switching between foreground and background activities, regardless of the fact that activities may actually have been killed and restarted without the user's knowledge.

- The mechanisms for saving persistent and dynamic state will become clearer in the following sections of this chapter.

The Android Lifecycle Methods

- As previously explained, the Activity and Fragment classes contain a number of lifecycle methods which act as event handlers when the state of an instance changes.

- The primary methods supported by the Android `Activity` and `Fragment` class are as follows:

  - `onCreate(Bundle savedInstanceState)` – The method that is called when the activity is first created and the ideal location for most initialization tasks to be performed.

    The method is passed an argument in the form of a `Bundle` object that may contain dynamic state information (typically relating to the state of the user interface) from a prior invocation of the activity.

  - `onRestart()` – Called when the activity is about to restart after having previously been stopped by the runtime Handling Android Activity State Changes system.

  - `onStart()` – Always called immediately after the call to the `onCreate()` or `onRestart()` methods, this method indicates to the activity that it is about to become visible to the user.

    This call will be followed by a call to `onResume()` if the activity moves to the top of the activity stack, or `onStop()` in the event that it is pushed down the stack by another activity.

  - `onResume()` – Indicates that the activity is now at the top of the activity stack and is the activity with which the user is currently interacting.

  - `onPause()` – Indicates that a previous activity is about to become the foreground activity.

    This call will be followed by a call to either the `onResume()` or `onStop()` method depending on whether the activity moves back to the foreground or becomes invisible to the user.

    Steps may be taken within this method to store persistent state information not yet saved by the app.

    To avoid delays in switching between activities, time consuming operations such as storing data to a database or performing network operations should be avoided within this method.

    This method should also ensure that any CPU intensive tasks such as animation are stopped.

  - `onStop()` – The activity is now no longer visible to the user.

    The two possible scenarios that may follow this call are a call to `onRestart()` in the event that the activity moves to the foreground again, or `onDestroy()` if the activity is being terminated.

  - `onDestroy()` – The activity is about to be destroyed, either voluntarily because the activity has completed its tasks and has called the `finish()` method or because the runtime is terminating it either to release memory or due to a configuration change (such as the orientation of the device changing).

It is important to note that a call will not always be made to `onDestroy()` when an activity is terminated.

- `onConfigurationChanged()` – Called when a configuration change occurs for which the activity has indicated it is not to be restarted.

  The method is passed a `Configuration` object outlining the new device configuration and it is then the responsibility of the activity to react to the change.

- The following lifecycle methods only apply to the `Fragment` class:
  - `onAttach()` - Called when the fragment is assigned to an activity.
  - `onCreateView()` - Called to create and return the fragment's user interface layout view hierarchy.
  - `onActivityCreated()` - The `onCreate()` method of the activity with which the fragment is associated has completed execution.
  - `onViewStatusRestored()` - The fragment's saved view hierarchy has been restored.

- In addition to the lifecycle methods outlined above, there are two methods intended specifically for saving and restoring the dynamic state of an activity:
  - `onRestoreInstanceState(Bundle savedInstanceState)` – This method is called immediately after a call to the `onStart()` method in the event that the activity is restarting from a previous invocation in which state was saved.

    As with `onCreate()`, this method is passed a `Bundle` object containing the previous state data.

    This method is typically used in situations where it makes more sense to restore a previous state after the initialization of the activity has been performed in `onCreate()` and `onStart()`.

  - `onSaveInstanceState(Bundle outState)` – Called before an activity is destroyed so that the current dynamic state (usually relating to the user interface) can be saved.

    The method is passed the `Bundle` object into which the state should be saved and which is subsequently passed through to the `onCreate()` and `onRestoreInstanceState()` methods when the activity is restarted.

    Note that this method is only called in situations where the runtime ascertains that dynamic state needs to be saved.

Handling Android Activity State Changes

- When overriding the above methods, it is important to remember that, with the exception of `onRestoreInstanceState()` and `onSaveInstanceState()`, the method implementation must include a call to the corresponding method in the super class.

- For example, the following method overrides the onRestart() method but also includes a call to the super class instance of the method:

```
protected void onRestart()
{
 super.onRestart(); Log.i(TAG, "onRestart");
}
```

- Failure to make this super class call in method overrides will result in the runtime throwing an exception during execution.

- While calls to the super class in the `onRestoreInstanceState()` and `onSaveInstanceState()` methods are optional (they can, for example, be omitted when implementing custom save and restoration behavior) there are considerable benefits to using them, a subject that will be covered later in Saving and Restoring the State of an Android Activity.

Lifetimes

- The final topic to be covered in this section involves an outline of the entire, visible and foreground lifetimes through which an activity or fragment will transition during execution:

  - Entire Lifetime – The term "entire lifetime" is used to describe everything that takes place between the initial call to the `onCreate()` method and the call to `onDestroy()` prior to the object terminating.

  - Visible Lifetime – Covers the periods of execution between the call to `onStart()` and `onStop()`.

    During this period the activity or fragment is visible to the user though may not be the object with which the user is currently interacting.

  - Foreground Lifetime – Refers to the periods of execution between calls to the `onResume()` and `onPause()` methods.

- It is important to note that an activity or fragment may pass through the foreground and visible lifetimes multiple times during the course of the entire lifetime.

- The concepts of lifetimes and lifecycle methods are illustrated in Figure 7-4:

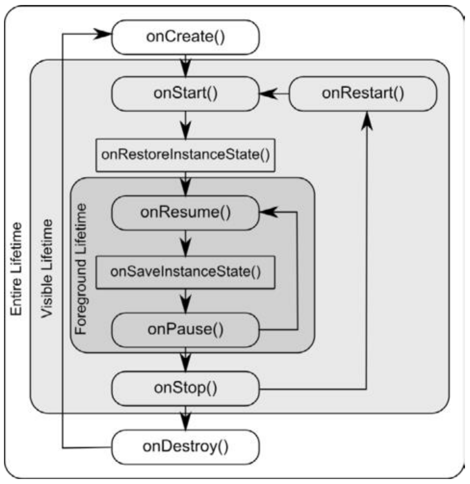


Figure 7-4

Foldable Devices and Multi-Resume

- As discussed previously, an activity is considered to be in the resumed state when it has moved to the foreground and is the activity with which the user is currently interacting.

- On standard devices an app can have one activity in the resumed state at any one time and all other activities are likely to be in the paused or stopped state.

- For some time now, Android has included multi-window support, allowing multiple activities to appear simultaneously in either split-screen or freeform configurations.

- Although originally used primarily on large screen tablet devices, this feature is likely to become more popular with the introduction of foldable devices.

- On devices running Android 10 and on which multi-window support is enabled (as will be the case for most foldables), it will be possible for multiple app activities to be in the resumed state at the same time (a concept referred to as multi-resume) allowing those visible activities to continue functioning (for example streaming content or updating visual data) even when another activity currently has focus.

- Although multiple activities can be in the resumed state, only one of these activities will be considered to the topmost resumed activity (in other words, the activity with which the user most recently interacted).

- An activity can receive notification that it has gained or lost the topmost resumed status by implementing the `onTopResumedActivityChanged()` callback method.

Disabling Configuration Change Restarts

- As previously outlined, an activity may indicate that it is not to be restarted in the event of certain configuration changes.

- This is achieved by adding an android:configChanges directive to the activity element within the project manifest file.

- The following manifest file excerpt, for example, indicates that the activity should not be restarted in the event of configuration changes relating to orientation or device-wide font size:

```
<activity
  android:name=".MainActivity"
  android:configChanges="orientation|fontScale"
  android:label="@string/app_name"
>
```

Lifecycle Method Limitations

- As discussed at the start of this section, lifecycle methods have been in use for many years and, until recently, were the only mechanism available for handling lifecycle state changes for activities and fragments.

- There are, however, shortcomings to this approach.

- One issue with the lifecycle methods is that they do not provide an easy way for an activity or fragment to find out its current lifecycle state at any given point during app execution.
- Instead the object would need to track the state internally, or wait for the next lifecycle method call.
- Also, the methods do not provide a simple way for one object to observe the lifecycle state changes of other objects within an app.
- This is a serious consideration since many other objects within an app can potentially be impacted by a lifecycle state change in a given activity or fragment.
- The lifecycle methods are also only available on subclasses of the Fragment and Activity classes.
- It is not possible, therefore, to build custom classes that are truly lifecycle aware.
- Finally, the lifecycle methods result in most of the lifecycle handling code being written within the activity or fragment which can lead to complex and error prone code.
- Ideally, much of this code should reside in the other classes that are impacted by the state change.
- An app that streams video, for example, might include a class designed specifically to manage the incoming stream.
- If the app needs to pause the stream when the main activity is stopped, the code to do so should reside in the streaming class, not the main activity.
- All of these problems and more are resolved by using lifecycle-aware components, a topic which will be covered Handling Android Activity State Changes starting with a later chapter .

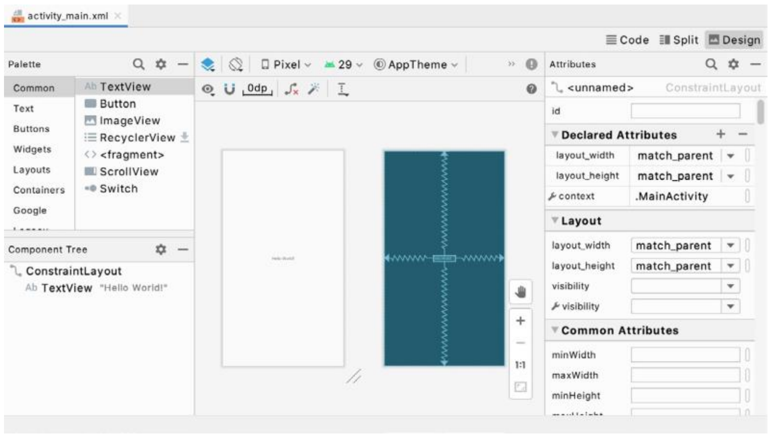
Android Activity State Changes by Example

- The previous sections have discussed in some detail the different states and lifecycles of the activities that comprise an Android application.
- In this section, we will put the theory of handling activity state changes into practice through the creation of an example application.
- The purpose of this example application is to provide a real world demonstration of an activity as it passes through a variety of different states within the Android runtime.
- In the next section, the example project constructed in this chapter will be extended to demonstrate the saving and restoration of dynamic activity state.

Creating the State Change Example Project

- The first step in this exercise is to create the new project.
- Begin by launching Android Studio and, if necessary, closing any currently open projects using the File > Close Project menu option so that the Welcome screen appears.

- Select the Start a new Android Studio project quick start option from the welcome screen.
- Within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.
- Enter `StateChange` into the Name field and specify `edu.niu.your_Z-ID.statechange` as the package name.
- Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.
- Upon completion of the project creation process, the `StateChange` project should be listed in the Project tool window located along the left-hand edge of the Android Studio main window.
- The next action to take involves the design of the user interface for the activity.
- This is stored in a file named `activity_main.xml` which should already be loaded into the Layout Editor tool.
- If it is not, navigate to it in the project tool window where it can be found in the app > res > layout folder.
- Once located, double-clicking on the file will load it into the Android Studio Layout Editor tool.



id	
layout_width	match_parent
layout_height	match_parent
context	.MainActivity
layout_width	match_parent
layout_height	match_parent
visibility	
visibility	
minWidth	
maxWidth	
minHeight	

Figure 7-5

Designing the User Interface

- With the user interface layout loaded into the Layout Editor tool, it is now time to design the user interface for the example application.
- Instead of the “Hello world!” `TextView` currently present in the user interface design, the activity actually requires an `EditText` view.
- Select the `TextView` object in the Layout Editor canvas and press the Delete key on the keyboard to remove it from the design.

- From the Palette located on the left side of the Layout Editor, select the `Text` category and, from the list of text components, click and drag a `Plain Text` component over to the visual representation of the device screen.

- Move the component to the center of the display so that the center guidelines appear and drop it into place so that the layout resembles that of Figure 7-6 below.

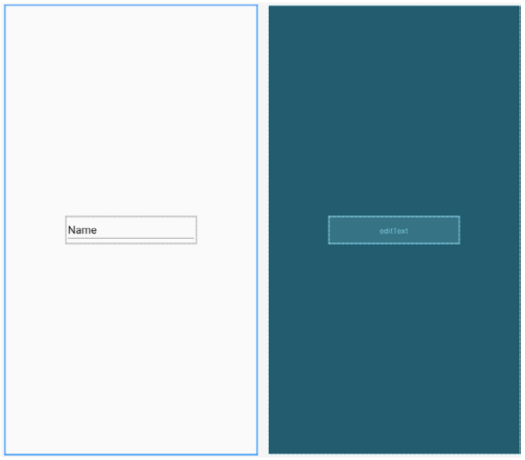


Figure 7-6

- When using the `EditText` widget it is necessary to specify an input type for the view.

- This simply defines the type of text or data that will be entered by the user.

- For example, if the input type is set to Phone, the user will be restricted to entering numerical digits into the view.

- Alternatively, if the input type is set to `TextCapCharacters`, the input will default to upper case characters.

- Input type settings may also be combined.

- For the purposes of this example, we will set the input type to support general text input.

- To do so, select the `EditText` widget in the layout and locate the `inputType` entry within the `Attributes` tool window.

- Click on the flag icon to the left of the current setting to open the list of options and, within the list, switch off `textPersonName` and enable text before clicking on the Apply button.

- Remaining in the Attributes tool window, change the id of the view to `editText`.

- By default the `EditText` is displaying text which reads “Name”.

- Remaining within the Attributes panel, delete this from the text property field so that the view is blank within the layout.

Overriding the Activity Lifecycle Methods

- At this point, the project contains a single activity named `MainActivity`, which is derived from the Android `AppCompatActivity` class.
- The source code for this activity is contained within the `MainActivity.java` file which should already be open in an editor session and represented by a tab in the editor tab bar.
- In the event that the file is no longer open, navigate to it in the Project tool window panel (app > java > edu.niu.your_Z-ID.statechange > MainActivity) and double-click on it to load the file into the editor.
- Once loaded the code should read as follows:

```
package edu.niu.your_Z-ID.statechange;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity
{
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
  }
}
```

- So far the only lifecycle method overridden by the activity is the `onCreate()` method which has been implemented to call the super class instance of the method before setting up the user interface for the activity.
- We will now modify this method so that it outputs a diagnostic message in the Android Studio Logcat panel each time it executes.
- For this, we will use the `Log` class, which requires that we `import android.util.Log` and declare a tag that will enable us to filter these messages in the log output:

```
package edu.niu.your_Z-ID.statechange;
.
.
import android.util.Log;

public class MainActivity extends AppCompatActivity
{
  private static final String TAG = "StateChange";

  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.i(TAG, "onCreate");
  }
}
```

- The next task is to override some more methods, with each one containing a corresponding log call.
- Note that the Log calls will still need to be added manually if the methods are being auto-generated:

```
@Override
protected void onStart()
{
  super.onStart();
  Log.i(TAG, "onStart");
}

@Override
protected void onResume()
{
  super.onResume();
  Log.i(TAG, "onResume");
}

@Override
protected void onPause()
{
  super.onPause();
  Log.i(TAG, "onPause");
}

@Override
protected void onStop()
{
  super.onStop();
  Log.i(TAG, "onStop");
}

@Override
protected void onRestart()
{
  super.onRestart();
  Log.i(TAG, "onRestart");
}

@Override
protected void onDestroy()
{
  super.onDestroy();
  Log.i(TAG, "onDestroy");
}
```

```
@Override
protected void onSaveInstanceState(Bundle outState)
{
  super.onSaveInstanceState(outState);
  Log.i(TAG, "onSaveInstanceState");
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState)
{
  super.onRestoreInstanceState(savedInstanceState);
  Log.i(TAG, "onRestoreInstanceState");
}
```

Filtering the Logcat Panel

- The purpose of the code added to the overridden methods in `MainActivity.java` is to output logging information to the Logcat tool window.

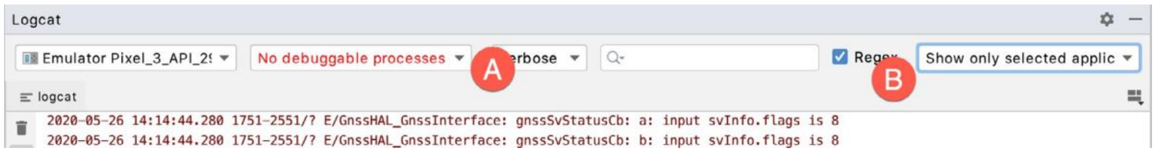
- This output can be configured to display all events relating to the device or emulator session, or restricted to those events that relate to the currently selected app.

- The output can also be further restricted to only those log events that match a specified filter.

- Display the Logcat tool window and click on the filter menu (marked as B in Figure 7-7) to review the available options.

- When this menu is set to Show only selected application, only those messages relating to the app selected in the menu marked as A will be displayed in the Logcat panel.

- Choosing No Filters, on the other hand, will display all the messages generated by the device or emulator.

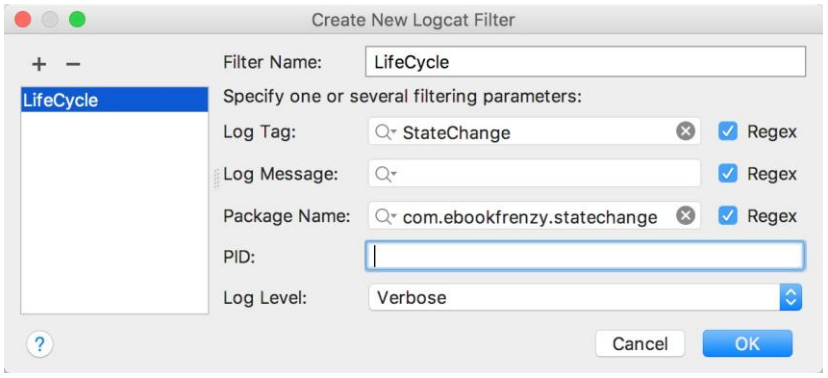


```
2020-05-26 14:14:44.280 1751-2551/? E/GnssHAL_GnssInterface: gnssSvStatusCb: a: input svInfo.flags is 8
2020-05-26 14:14:44.280 1751-2551/? E/GnssHAL_GnssInterface: gnssSvStatusCb: b: input svInfo.flags is 8
```

Figure 7-7

- Before running the application, it is worth demonstrating the creation of a filter which, when selected, will further restrict the log output to ensure that only those log messages containing the tag declared in our activity are displayed.rom the filter menu (B), select the Edit Filter Configuration menu option.

- In the Create New Logcat Filter dialog (Figure 7-8), name the filter Lifecycle and, in the Log Tag field, enter the Tag value declared in `MainActivity.java` (in the above code example this was `StateChange`).



Filter Name: LifeCycle
Specify one or several filtering parameters:
Log Tag: StateChange ✓ Regex
Log Message: ✓ Regex
Package Name: com.ebookfrenzy.statechange ✓ Regex
PID:
Log Level: Verbose

Figure 7-8

- Enter the package identifier in the Package Name field and, when the changes are complete, click on the OK button to create the filter and dismiss the dialog.
- Instead of listing No Filters, the newly created filter should now be selected in the Logcat tool window.

Running the Application

- For optimal results, the application should be run on a physical Android device or emulator.
- With the device configured and connected to the development computer, click on the run button represented by a green triangle located in the Android Studio toolbar as shown in Figure 7-9 below, select the Run > Run… menu option or use the Shift+F10 keyboard shortcut:

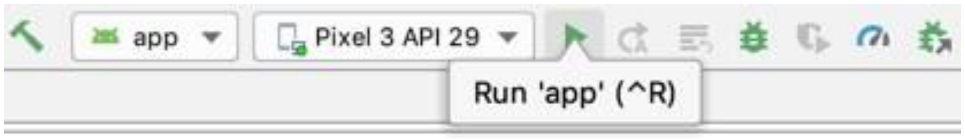
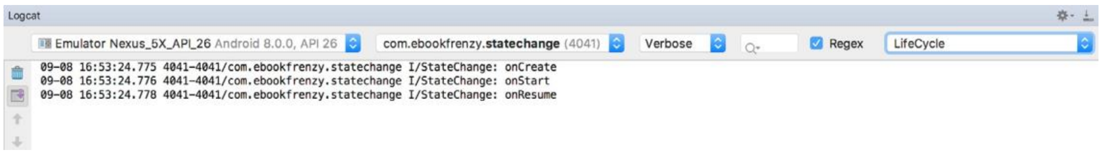


Figure 7-9

- Select the physical Android device from the Choose Device dialog if it appears (assuming that you have not already configured it to be the default target).
- After Android Studio has built the application and installed it on the device it should start up and be running in the foreground.
- A review of the Logcat panel should indicate which methods have so far been triggered (taking care to ensure that the Lifecycle filter created in the preceding section is selected to filter out log events that are not currently of interest to us):


Logcat
Emulator Nexus_5X_API_26 Android 8.0.0, API 26
com.ebookfrenzy.statechange (4041)
Verbose
✓ Regex
LifeCycle

```
09-08 16:53:24.775 4041-4041/com.ebookfrenzy.statechange I/StateChange: onCreate
09-08 16:53:24.776 4041-4041/com.ebookfrenzy.statechange I/StateChange: onStart
09-08 16:53:24.778 4041-4041/com.ebookfrenzy.statechange I/StateChange: onResume
```

Figure 7-10

Experimenting with the Activity

- With the diagnostics working, it is now time to exercise the application with a view to gaining an understanding of the activity lifecycle state changes.

- To begin with, consider the initial sequence of log events in the Logcat panel:

```
onCreate
onStart
onResume
```

- Clearly, the initial state changes are exactly as outlined earlier in this chapter.

- Note, however, that a call was not made to `onRestoreInstanceState()` since the Android runtime detected that there was no state to restore in this situation.

- Tap on the Home icon in the bottom status bar on the device display and note the sequence of method calls reported in the log as follows:

```
onPause
onStop
onSaveInstanceState
```

- In this case, the runtime has noticed that the activity is no longer in the foreground, is not visible to the user and has stopped the activity, but not without providing an opportunity for the activity to save the dynamic state.

- Depending on whether the runtime ultimately destroyed the activity or simply restarted it, the activity will either be notified it has been restarted via a call to `onRestart()` or will go through the creation sequence again when the user returns to the activity.

- As outlined earlier in this chapter, the destruction and recreation of an activity can be triggered by making a configuration change to the device, such as rotating from portrait to landscape.

- To see this in action, simply rotate the device while the `StateChange` application is in the foreground.

- When using the emulator, device rotation may be simulated using the rotation button located in the emulator toolbar.

- The resulting sequence of method calls in the log should read as follows:

```
onPause onStop
onSaveInstanceState onDestroy
onCreate onStart
onRestoreInstanceState onResume
```

- Clearly, the runtime system has given the activity an opportunity to save state before being destroyed and restarted.

- In the next section, we will extend the `StateChange` example project to demonstrate how to save and restore an activity's dynamic state.

- If the previous few sections have achieved their objective, it should now be a little clearer as to the importance of saving and restoring the state of a user interface at particular points in the lifetime of an activity.

- In this section, we will extend the example application created earlier in the chapter to demonstrate the steps involved in saving and restoring state when an activity is destroyed and recreated by the runtime system.

- A key component of saving and restoring dynamic state involves the use of the Android SDK Bundle class, a topic that will also be covered in this section.

Saving Dynamic State

- An activity, as we have already learned, is given the opportunity to save dynamic state information via a call from the runtime system to the activity's implementation of the `onSaveInstanceState()` method.

- Passed through as an argument to the method is a reference to a `Bundle` object into which the method will need to store any dynamic data that needs to be saved.

- The `Bundle` object is then stored by the runtime system on behalf of the activity and subsequently passed through as an argument to the activity's `onCreate()` and `onRestoreInstanceState()` methods if and when they are called.

- The data can then be retrieved from the `Bundle` object within these methods and used to restore the state of the activity.

Default Saving of User Interface State

- Earlier in this chapter, the diagnostic output from the `StateChange` example application showed that an activity goes through a number of state changes when the device on which it is running is rotated sufficiently to trigger an orientation change.

- Launch the `StateChange` application once again, this time entering some text into the `EditText` field prior to performing the device rotation (on devices or emulators running Android 9 it may be necessary to tap the rotation button in the located in the status bar to complete the rotation).

- Having rotated the device, the following state change sequence should appear in the Logcat window:

```
onPause
onStop
onSaveInstanceState
onDestroy
onCreate
onStart
onRestoreInstanceState
onResume
```

- Clearly this has resulted in the activity being destroyed and re-created.

- A review of the user interface of the running application, however, should show that the text entered into the `EditText` field has been preserved.

- Given that the activity was destroyed and recreated, and that we did not add any specific code to make sure the text was saved and restored, this behavior requires some explanation.

- In fact most of the view widgets included with the Android SDK already implement the behavior necessary to automatically save and restore state when an activity is restarted.

- The only requirement to enable this behavior is for the `onSaveInstanceState()` and `onRestoreInstanceState()` override methods in the activity to include calls to the equivalent methods of the super class:

```
@Override
protected void onSaveInstanceState(Bundle outState)
{
  super.onSaveInstanceState(outState);
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState)
{
  super.onRestoreInstanceState(savedInstanceState);
}
```

- The automatic saving of state for a user interface view can be disabled in the XML layout file by setting the `android:saveEnabled` property to false.

- For the purposes of an example, we will disable the automatic state saving mechanism for the `EditText` view in the user interface layout and then add code to the application to manually save and restore the state of the view.

- To configure the `EditText` view such that state will not be saved and restored in the event that the activity is restarted, edit the `activity_main.xml` file so that the entry for the view reads as follows (note that we can edit XML directly. Click the Text tab on the bottom edge of the Layout Editor panel):

```
<EditText
  android:id="@+id/editText"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:ems="10"
  android:inputType="text"
  android:saveEnabled="false"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent"
/>
```

- After making the change, run the application, enter text and rotate the device to verify that the text is no longer saved and restored before proceeding.

The Bundle Class

- For situations where state needs to be saved beyond the default functionality provided by the user interface view components, the Bundle class provides a container for storing data using a key-value pair mechanism.

- The keys take the form of string values, while the values associated with those keys can be in the form of a primitive value or any object that implements the Android `Parcelable` interface.

- A wide range of classes already implements the `Parcelable` interface.

- Custom classes may be made "parcelable" by implementing the set of methods defined in the `Parcelable` interface, details of which can be found in the Android documentation at:

  `https://developer.android.com/reference/android/os/Parcelable.html`

- The `Bundle` class also contains a set of methods that can be used to get and set key-value pairs for a variety of data types including both primitive types (including `Boolean`, `char`, `double` and `float` values) and objects (such as `Strings` and `CharSequences`).

- For the purposes of this example, and having disabled the automatic saving of text for the `EditText` view, we need to make sure that the text entered into the `EditText` field by the user is saved into the `Bundle` object and subsequently restored.

- This will serve as a demonstration of how to manually save and restore state within an Android application and will be achieved using the `putCharSequence()` and `getCharSequence()` methods of the `Bundle` class respectively.

Saving the State

- The first step in extending the `StateChange` application is to make sure that the text entered by the user is extracted from the `EditText` component within the `onSaveInstanceState()` method of the `MainActivity` activity, and then saved as a key-value pair into the `Bundle` object.

- In order to extract the text from the `EditText` object we first need to identify that object in the user interface.

- Clearly, this involves bridging the gap between the Java code for the activity (contained in the `MainActivity. java` source code file) and the XML representation of the user interface (contained within the `activity_main.xml` resource file).

- In order to extract the text entered into the `EditText` component we need to gain access to that user interface object.

- Each component within a user interface has associated with it a unique identifier.

- By default, the Layout Editor tool constructs the id for a newly added component from the object type.
- If more than one view of the same type is contained in the layout the type name is followed by a sequential number (though this can, and should, be changed to something more meaningful by the developer).
- As can be seen by checking the Component Tree panel within the Android Studio main window when the `activity_main.xml` file is selected and the Layout Editor tool displayed, the `EditText` component has been assigned the id `editText`:

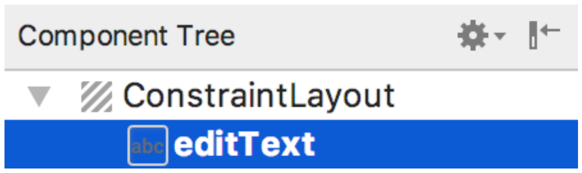


Figure 7-11

- With knowledge of the view id, we can obtain a reference to the view from within the Activity code:

```
final EditText editText = findViewById(R.id.editText);
```

- We can now obtain the text that the `editText` view contains via the object's `getText()` method, which, in turn, returns the current text:

```
CharSequence userText = editText.getText();
```

- Finally, we can save the text using the `Bundle` object's `putCharSequence()` method, passing through the key (this can be any string value but in this instance, we will declare it as "savedText") and the `userText` object as arguments:

```
outState.putCharSequence("savedText", userText);
```

- Bringing this all together gives us a modified `onSaveInstanceState()` method in the `MainActivity.java` file that reads as follows (noting also the additional import directive for `android.widget.EditText`):

```
package edu.niu.your_Z-ID.statechange;
.
.
import android.widget.EditText;

public class MainActivity extends AppCompatActivity
{
.
.
.
protected void onSaveInstanceState(Bundle outState)
{
  super.onSaveInstanceState(outState);
  Log.i(TAG, "onSaveInstanceState");
```

```
    final EditText editText = findViewById(R.id.editText);
    CharSequence userText = editText.getText();
    outState.putCharSequence("savedText", userText);
  }
  .
  .
```

- Now that steps have been taken to save the state, the next phase is to ensure that it is restored when needed.

Restoring the State

- The saved dynamic state can be restored in those lifecycle methods that are passed the `Bundle` object as an argument.
- This leaves the developer with the choice of using either `onCreate()` or `onRestoreInstanceState()`.
- The method to use will depend on the nature of the activity.
- In instances where state is best restored after the activity's initialization tasks have been performed, the `onRestoreInstanceState()` method is generally more suitable.
- For the purposes of this example we will add code to the `onRestoreInstanceState()` method to extract the saved state from the `Bundle` using the "savedText" key.
- We can then display the text on the `editText` component using the object's `setText()` method:

```
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState)
{
  super.onRestoreInstanceState(savedInstanceState);
  Log.i(TAG, "onRestoreInstanceState");

  final EditText editText = findViewById(R.id.editText);

  CharSequence userText = savedInstanceState.getCharSequence("savedText");
  editText.setText(userText);
}
```

Testing the Application

- All that remains is once again to build and run the `StateChange` application.
- Once running and in the foreground, touch the `EditText` component and enter some text before rotating the device to another orientation.
- Whereas the text changes were previously lost, the new text is retained within the `editText` component thanks to the code we have added to the activity in this chapter.

- Having verified that the code performs as expected, comment out the `super.onSaveInstanceState()` and `super. onRestoreInstanceState()` calls from the two methods, re-launch the app and note that the text is still preserved after a device rotation.

- The default save and restoration system has essentially been replaced by a custom implementation, thereby providing a way to dynamically and selectively save and restore state within an activity.

Summary

- Mobile devices are typically considered to be resource constrained, particularly in terms of on-board memory capacity.

- Consequently, a prime responsibility of the Android operating system is to ensure that applications, and the operating system in general, remain responsive to the user.

- Applications are hosted on Android within processes. Each application, in turn, is made up of components in the form of activities and services.

- The Android runtime system has the power to terminate both processes and individual activities in order to free up memory.

- Process state is taken into consideration by the runtime system when deciding whether a process is a suitable candidate for termination.

- The state of a process is largely dependent upon the status of the activities hosted by that process.

- The key message of this chapter is that an application moves through a variety of states during its execution lifespan and has very little control over its destiny within the Android runtime environment.

- Those processes and activities that are not directly interacting with the user run a higher risk of termination by the runtime system.

- An essential element of Android application development, therefore, involves the ability of an application to respond to state change notifications from the operating system.

- All activities are derived from the Android `Activity` class which, in turn, contains a number of lifecycle methods that are designed to be called by the runtime system when the state of an activity changes.

- Similarly, the Fragment class contains a number of comparable methods.

- By overriding these methods, activities and fragments can respond to state changes and, where necessary, take steps to save and restore the current state of both the activity and the application.

- Lifecycle state can be thought of as taking two forms.

- The persistent state refers to data that needs to be stored between application invocations (for example to a file or database).

- Dynamic state, on the other hand, relates instead to the current appearance of the user interface.

- Although lifecycle methods have a number of limitations that can be avoided by making use of lifecycle-aware components, an understanding of these methods is important in order to fully understand the new approaches to lifecycle management covered later in this book.

- In this chapter, we have highlighted the lifecycle methods available to activities and covered the concept of activity lifetimes.

- The old adage that a picture is worth a thousand words holds just as true for examples when learning a new programming paradigm.

- We have created an example Android application for the purpose of demonstrating the different lifecycle states through which an activity is likely to pass. In the course of developing the project in this chapter, we also looked at a mechanism for generating diagnostic logging information from within an activity.

- The saving and restoration of dynamic state in an Android application is simply a matter of implementing the appropriate code in the appropriate lifecycle methods.

- For most user interface views, this is handled automatically by the Activity super class. In other instances, this typically consists of extracting values and settings within the `onSaveInstanceState()` method and saving the data as key-value pairs within the Bundle object passed through to the activity by the runtime system.

- State can be restored in either the `onCreate()` or the `onRestoreInstanceState()` methods of the activity by extracting values from the Bundle object and updating the activity based on the stored values.

- In this chapter, we have used these techniques to update the `StateChange` project so that the `Activity` retains changes through the destruction and subsequent recreation of an activity.