CSCI 322/522 6. Android Architecture, Application Anatomy and View Binding

- So far in our studies, steps have been taken to set up an environment suitable for the development of Android applications using Android Studio.

- An initial step has also been taken into the process of application development through the creation of an Android Studio application project.

- Before delving further into the practical matters of Android application development, however, it is important to gain an understanding of some of the more abstract concepts of both the Android SDK and Android development in general.

- Gaining a clear understanding of these concepts now will provide a sound foundation on which to build further knowledge.

- Starting with an overview of the Android architecture in this chapter, and continuing in the next few parts of this chapter, the goal is to provide a detailed overview of the fundamentals of Android development.

The Android Software Stack

- Android is structured in the form of a software stack comprising applications, an operating system, run-time environment, middleware, services and libraries.

- This architecture can, perhaps, best be represented visually as outlined in Figure 6-1.

- Each layer of the stack, and the corresponding elements within each layer, are tightly integrated and carefully tuned to provide the optimal application development and execution environment for mobile devices.

- The remainder of this chapter will work through the different layers of the Android stack, starting at the bottom with the Linux Kernel.

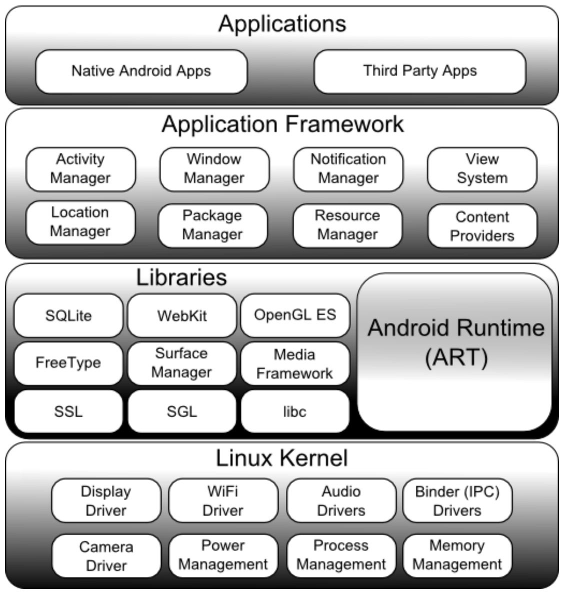

Figure 6-1

The Linux Kernel

- Positioned at the bottom of the Android software stack, the Linux Kernel provides a level of abstraction between the device hardware and the upper layers of the Android software stack.
- Based on Linux version 2.6, the kernel provides preemptive multitasking, low-level core system services such as memory, process and power management in addition to providing a network stack and device drivers for hardware such as the device display, Wi-Fi and audio.
- The original Linux kernel was developed in 1991 by Linus Torvalds and was combined with a set of tools, utilities and compilers developed by Richard Stallman at the Free Software Foundation to create a full operating system referred to as GNU/Linux.
- Various Linux distributions have been derived from these basic underpinnings such as Ubuntu and Red Hat Enterprise Linux.
- It is important to note, however, that Android uses ***only*** the Linux kernel.
- That said, it is worth noting that the Linux kernel was originally developed for use in traditional computers in the form of desktops and servers.
- In fact, Linux is now most widely deployed in mission critical enterprise server environments.
- It is a testament to both the power of today's mobile devices and the efficiency and performance of the Linux kernel that we find this software at the heart of the Android software stack.

Android Runtime – ART

- When an Android app is built within Android Studio it is compiled into an intermediate bytecode format (referred to as DEX format).
- When the application is subsequently loaded onto the device, the Android Runtime (ART) uses a process referred to as Ahead-of-Time (AOT) compilation to translate the bytecode down to the native instructions required by the device processor.
- This format is known as Executable and Linkable Format (ELF).
- Each time the application is subsequently launched, the ELF executable version is run, resulting in faster application performance and improved battery life.
- This contrasts with the Just-in-Time (JIT) compilation approach used in older Android implementations whereby the bytecode was translated within a virtual machine (VM) each time the application was launched.

Android Libraries

- In addition to a set of standard Java development libraries (providing support for such general purpose tasks as string handling, networking and file manipulation), the Android development environment also includes the Android Libraries.

- These are a set of Java-based libraries that are specific to Android development.

- Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access.

- A summary of some key core Android libraries available to the Android developer is as follows:

  - `android.app` – Provides access to the application model and is the cornerstone of all Android applications.

  - `android.content` – Facilitates content access, publishing and messaging between applications and application components.

  - `android.database` – Used to access data published by content providers and includes SQLite database management classes.

  - `android.graphics` – A low-level 2D graphics drawing API including colors, points, filters, rectangles and canvases.

  - `android.hardware` – Presents an API providing access to hardware such as the accelerometer and light sensor.

  - `android.opengl` – A Java interface to the OpenGL ES 3D graphics rendering API.

  - `android.os` – Provides applications with access to standard operating system services including messages, system services and inter-process communication.

  - `android.media` – Provides classes to enable playback of audio and video.

  - `android.net` – A set of APIs providing access to the network stack.

    Includes `android.net.wifi`, which provides access to the device's wireless stack.

  - `android.print` – Includes a set of classes that enable content to be sent to configured printers from within Android applications.

  - `android.provider` – A set of convenience classes that provide access to standard Android content provider databases such as those maintained by the calendar and contact applications.

  - `android.text` – Used to render and manipulate text on a device display.

  - `android.util` – A set of utility classes for performing tasks such as string and number conversion, XML handling and date and time manipulation.

  - `android.view` – The fundamental building blocks of application user interfaces.

  - `android.widget` – A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.

  - `android.webkit` – A set of classes intended to allow web-browsing capabilities to be built into applications.

- Having covered the Java-based libraries in the Android runtime, it is now time to turn our attention to the C/ C++ based libraries contained in this layer of the Android software stack.

C/C++ Libraries

- The Android runtime core libraries outlined in the preceding section are Java-based and provide the primary APIs for developers writing Android applications.

- It is important to note, however, that the core libraries do not perform much of the actual work and are, in fact, essentially Java "wrappers" around a set of C/C++ based libraries.

- When making calls, for example, to the android.opengl library to draw 3D graphics on the device display, the library actually ultimately makes calls to the OpenGL ES C++ library which, in turn, works with the underlying Linux kernel to perform the drawing tasks.

- C/C++ libraries are included to fulfill a wide and diverse range of functions including 2D and 3D graphics drawing, Secure Sockets Layer (SSL) communication, SQLite database management, audio and video playback, bitmap and vector font rendering, display subsystem and graphic layer management and an implementation of the standard C system library (libc).

- In practice, the typical Android application developer will access these libraries solely through the Java based Android core library APIs.

- In the event that direct access to these libraries is needed, this can be achieved using the Android Native Development Kit (NDK),

- The purpose of NDK is to call the native methods of non-Java or Kotlin programming languages (such as C and C++) from within Java code using the Java Native Interface (JNI).

Application Framework

- The Application Framework is a set of services that collectively form the environment in which Android applications run and are managed.

- This framework implements the concept that Android applications are constructed from reusable, interchangeable and replaceable components.

- This concept is taken a step further in that an application is also able to publish its capabilities along with any corresponding data so that they can be found and reused by other applications.

- The Android framework includes the following key services:

  - Activity Manager – Controls all aspects of the application lifecycle and activity stack.

  - Content Providers – Allows applications to publish and share data with other applications.

  - Resource Manager – Provides access to non-code embedded resources such as strings, color settings and user interface layouts.

- Notifications Manager – Allows applications to display alerts and notifications to the user.
- View System – An extensible set of views used to create application user interfaces.
- Package Manager – The system by which applications are able to find out information about other applications currently installed on the device.
- Telephony Manager – Provides information to the application about the telephony services available on the device such as status and subscriber information.
- Location Manager – Provides access to the location services allowing an application to receive updates about location changes.

Applications

- Located at the top of the Android software stack are the applications.
- These comprise both the native applications provided with the particular Android implementation (for example web browser and email applications) and the third party applications installed by the user after purchasing the device.

The Anatomy of an Android Application

- Regardless of your prior programming experiences, be it Windows, macOS, Linux or even iOS based, the chances are good that Android development is quite unlike anything you have encountered before.
- The objective of this chapter, therefore, is to provide an understanding of the high-level concepts behind the architecture of Android applications.
- In doing so, we will explore in detail both the various components that can be used to construct an application and the mechanisms that allow these to work together to create a cohesive application.

Android Activities

- Those familiar with object-oriented programming languages such as Java, Kotlin, C++ or C# will be familiar with the concept of encapsulating elements of application functionality into classes that are then instantiated as objects and manipulated to create an application.
- Since Android applications are written in Java and Kotlin, this is still very much the case.
- Android, however, also takes the concept of re-usable components to a higher level.
- Android applications are created by bringing together one or more components known as Activities.
- An activity is a single, standalone module of application functionality that usually correlates directly to a single user interface screen and its corresponding functionality.
- An appointments application might, for example, have an activity screen that displays appointments set up for the current day.

- The application might also utilize a second activity consisting of a screen where new appointments may be entered by the user.
- Activities are intended as fully reusable and interchangeable building blocks that can be shared amongst different applications.
- An existing email application, for example, might contain an activity specifically for composing and sending an email message.
- A developer might be writing an application that also has a requirement to send an email message.
- Rather than develop an email composition activity specifically for the new application, the developer can simply use the activity from the existing email application.
- Activities are created as subclasses of the Android Activity class and must be implemented so as to be entirely independent of other activities in the application.
- In other words, a shared activity cannot rely on being called at a known point in a program flow (since other applications may make use of the activity in unanticipated ways).
- One activity cannot directly call methods or access instance data of another activity.
- This, instead, is achieved using Intents and Content Providers.
- By default, an activity cannot return results to the activity from which it was invoked.
- If this functionality is required, the activity must be specifically started as a sub-activity of the originating activity.

Android Fragments

- An activity, as described above, typically represents a single user interface screen within an app.
- One option is to construct the activity using a single user interface layout and one corresponding activity class file.
- A better alternative, however, is to break the activity into different sections.
- Each of these sections is referred to as a fragment, each of which consists of part of the user interface layout and a matching class file (declared as a subclass of the Android Fragment class).
- In this scenario, an activity simply becomes a container into which one or more fragments are embedded.
- In fact, fragments provide an efficient alternative to having each user interface screen represented by a separate activity.
- Instead, an app can consist of a single activity that switches between different fragments, each representing a different app screen.

Android Intents

- Intents are the mechanism by which one activity is able to launch another and implement the flow through the activities that make up an application.
- Intents consist of a description of the operation to be performed and, optionally, the data on which it is to be performed.
- Intents can be explicit, in that they request the launch of a specific activity by referencing the activity by class name, or implicit by stating either the type of action to be performed or providing data of a specific type on which the action is to be performed.
- In the case of implicit intents, the Android runtime will select the activity to launch that most closely matches the criteria specified by the Intent using a process referred to as Intent Resolution.

Broadcast Intents

- Another type of Intent, the Broadcast Intent, is a system wide intent that is sent out to all applications that have registered an "interested" Broadcast Receiver.
- The Android system, for example, will typically send out Broadcast Intents to indicate changes in device status such as the completion of system start up, connection of an external power source to the device or the screen being turned on or off.
- A Broadcast Intent can be normal (asynchronous) in that it is sent to all interested Broadcast Receivers at more or less the same time, or ordered in that it is sent to one receiver at a time where it can be processed and then either aborted or allowed to be passed to the next Broadcast Receiver.

Broadcast Receivers

- Broadcast Receivers are the mechanism by which applications are able to respond to Broadcast Intents.
- A Broadcast Receiver must be registered by an application and configured with an Intent Filter to indicate the types of broadcast in which it is interested.
- When a matching intent is broadcast, the receiver will be invoked by the Android runtime regardless of whether the application that registered the receiver is currently running.
- The receiver then has 5 seconds in which to complete any tasks required of it (such as launching a Service, making data updates or issuing a notification to the user) before returning.
- Broadcast Receivers operate in the background and do not have a user interface.

Android Services

- Android Services are processes that run in the background and do not have a user interface.
- They can be started and subsequently managed from activities, Broadcast Receivers or other Services.
- Android Services are ideal for situations where an application needs to continue performing tasks but

does not necessarily need a user interface to be visible to the user.

- Although Services lack a user interface, they can still notify the user of events using notifications and toasts (small notification messages that appear on the screen without interrupting the currently visible activity) and are also able to issue Intents.

- Services are given a higher priority by the Android runtime than many other processes and will only be terminated as a last resort by the system in order to free up resources.

- In the event that the runtime does need to kill a Service, however, it will be automatically restarted as soon as adequate resources once again become available.

- A Service can reduce the risk of termination by declaring itself as needing to run in the foreground.

- This is achieved by making a call to `startForeground()`.

- This is only recommended for situations where termination would be detrimental to the user experience (for example, if the user is listening to audio being streamed by the Service).

- Example situations where a Service might be a practical solution include, as previously mentioned, the streaming of audio that should continue when the application is no longer active, or a stock market tracking application that needs to notify the user when a share hits a specified price.

Content Providers

- Content Providers implement a mechanism for the sharing of data between applications.

- Any application can provide other applications with access to its underlying data through the implementation of a Content Provider including the ability to add, remove and query the data (subject to permissions).

- Access to the data is provided via a Universal Resource Identifier (URI) defined by the Content Provider. Data can be shared in the form of a file or an entire `SQLite` database.

- The native Android applications include a number of standard Content Providers allowing applications to access data such as contacts and media files.

- The Content Providers currently available on an Android system may be located using a Content Resolver.

The Application Manifest

- The glue that pulls together the various elements that comprise an application is the Application Manifest file.

- It is within this XML based file that the application outlines the activities, services, broadcast receivers, data providers and permissions that make up the complete application.

Application Resources

- In addition to the manifest file and the Dex files that contain the byte code, an Android application package will also typically contain a collection of resource files.

- These files contain resources such as the strings, images, fonts and colors that appear in the user interface together with the XML representation of the user interface layouts.

- By default, these files are stored in the `/res` sub-directory of the application project's hierarchy.

Application Context

- When an application is compiled, a class named `R` is created that contains references to the application resources.

- The application manifest file and these resources combine to create what is known as the Application Context.

- This context, represented by the Android Context class, may be used in the application code to gain access to the application resources at runtime.

- In addition, a wide range of methods may be called on an application's context to gather information and make changes to the application's environment at runtime.

An Overview of View Binding

- An important part of developing Android apps involves the interaction between the code and the views that make up the user interface layouts.

- This part of the chapter will look at the options available for gaining access to layout views in code with a particular emphasis on an option introduced with Android Studio 3.6 known as view binding.

- Once the basics of view bindings have been covered, the chapter will outline the changes necessary to convert the AndroidSample project to use this approach.

Find View by ID

- As outlined above, all of the resources that make up an application are compiled into a class named `R`.

- Amongst those resources are those that define layouts.

- Within the `R` class is a subclass named layout, which contains the layout resources, including the views that make up the user interface.

- Most apps will need to implement interaction between the code and these views, for example when reading the value entered into the `EditText` view or changing the content displayed on a `TextView`.

- Prior to the introduction of Android Studio 3.6, the only option for gaining access to a view from within the app code involved writing code to manually find a view based on its id via a method named `findViewById()`.

- For example:

  ```
  TextView exampleView = findViewById(R.id.exampleView);
  ```

  With the reference obtained, the properties of the view can then be accessed.

- For example:

  ```
  exampleView.setText("Hello");
  ```

- While finding views by id is still a viable option, it has some limitations.

- The biggest disadvantage of `findViewById()` being that it is possible to obtain a reference to a view that has not yet been created within the layout, leading to a null pointer exception when an attempt is made to access the view's properties.

- Since Android Studio 3.6, an alternative way of accessing views from the app code has been available in the form of view bindings.

View Bindings

- When view bindings are enabled in an app module, Android Studio automatically generates a binding class for each layout file within the module.

- Using this binding class, the layout views can be accessed from within the code without the need to use `findViewById()`.

- The name of the binding class generated by Android Studio is based on the layout file name converted to so- called "camel case" with the word "Binding" appended to the end. In the case of the `activity_main.xml` file, for example, the binding class will be named `ActivityMainBinding`.

- The process for using view bindings within a project module can be summarized as follows:

  - Enable view binding for any project modules where support is required.
  - Edit code to import the auto-generated view binding class.
  - Inflate the binding class to obtain a reference to the binding.
  - Access the root view within the binding and use it to specify the content view of the user interface.
  - Access views by name as properties of the binding object.

Converting the AndroidSample Project

- The remainder of this chapter will demonstrate the use of view bindings by converting the AndroidSample project to use view bindings instead of using `findViewById()`.

- Begin by launching Android Studio and opening the AndroidSample project created in the chapter entitled ***3. Creating a First Android App***.

Enabling View Binding

- As of Android Studio 4.0, view binding is not enabled by default.

- To use view binding, therefore, some changes must be made to the build.gradle file for each module in which view binding is needed.

- In the case of the AndroidSample project, this will require changes to the `Gradle Scripts > build.gradle (Module: app)` file.

- To begin with, the `viewBinding` property must be enabled within the android section of the file:

```
.
.
android
{
  buildFeatures
  {
    viewBinding = true

}
.
.
```

- Once these changes have been made, use the Build menu to clean and then rebuild the project to make sure the binding class is generated.

- The next step is to use the binding class within the code.

Using View Bindings

- The first step in this process is to “inflate” the view binding class so that we can access the root view within the layout.

- This root view will then be used as the content view for the layout.

- The logical place to perform these tasks is within the `onCreate()` method of the activity associated with the layout.

- A typical `onCreate()` method will read as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
}
```

- To switch to using view binding, the view binding class will need to be imported and the class modified as follows.

- Note that since the layout file is named `activity_main.xml`, we can surmise that the binding class generated by Android Studio will be named `ActivityMainBinding`:

```
.
.
import android.widget.EditText;
import android.widget.TextView;
import com.example.androidsample.databinding.ActivityMainBinding;
.
.
public class MainActivity extends AppCompatActivity
{
  private ActivityMainBinding binding;
  .
  .
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    binding = ActivityMainBinding.inflate(getLayoutInflater());
    View view = binding.getRoot();
    setContentView(view);
  }
```

- Now that we have a reference to the binding we can access the views by name as follows:

```
public void convertCurrency(View view)
{
  EditText dollarText = findViewById(R.id.dollarText);
  TextView textView = findViewById(R.id.textView);

  if (!binding.dollarText.getText().toString().equals(""))
  {
    Float dollarValue =
      Float.valueOf(binding.dollarText.getText().toString());
    Float euroValue = dollarValue * 0.84F;
    binding.textView.setText(euroValue.toString());
  }
  else
  {
    binding.textView.setText(R.string.no_value_string);
  }
}
```

- Compile and run the app and verify that the currency conversion process still works as before.

Choosing an Option

- The introduction of view binding does not invalidate the previous options and both will continue to be widely supported for the foreseeable future.

- In terms of avoiding null pointer exceptions, however, view bindings are clearly the safer option when compared to using the `findViewById()` method.

- When developing your own projects, therefore, view binding should probably be used.

- With regards to the examples in our studies, however, it is important to keep in mind that view bindings are not enabled by default in Android Studio 4.0.

- Unfortunately, to use view bindings in the code examples it would be necessary to manually repeat each of the `build.gradle` and `onCreate()` method changes in this chapter over 50 times.

- For this reason alone, the examples in our studies continue to use `findViewById()`.

- That being said, there is no reason why you should not follow the steps in this chapter to adapt the examples in our studies to use view bindings if you wish to do so.

- In fact, we would encourage you to convert at least a few of the tutorials in our studies to use view binding as an exercise to gain familiarity with the concepts.

Summary

- A good Android development knowledge foundation requires an understanding of the overall architecture of Android.

- Android is implemented in the form of a software stack architecture consisting of a Linux kernel, a runtime environment and corresponding libraries, an application framework and a set of applications.

- Applications are predominantly written in Java or Kotlin and compiled down to bytecode format within the Android Studio build environment.

- When the application is subsequently installed on a device, this bytecode is compiled down by the Android Runtime (ART) to the native format used by the CPU.

- The key goals of the Android architecture are performance and efficiency, both in application execution and in the implementation of reuse in application design.

- A number of different elements can be brought together in order to create an Android application.

- In this chapter, we have provided a high-level overview of Activities, Fragments, Services, Intents and Broadcast Receivers together with an overview of the manifest file and application resources.

- Maximum reuse and interoperability are promoted through the creation of individual, standalone modules of functionality in the form of activities and intents, while data sharing between applications is achieved by the implementation of content providers.

- While activities are focused on areas where the user interacts with the application (an activity essentially equating to a single user interface screen and often made up of one or more fragments), background processing is typically handled by Services and Broadcast Receivers.

- The components that make up the application are outlined for the Android runtime system in a manifest file which, combined with the application's resources, represents the application's context.

- Much has been covered in this chapter that is most likely new to the average developer.
- Rest assured, however, that extensive exploration and practical use of these concepts will be made in subsequent chapters to ensure a solid knowledge foundation on which to build your own applications.
- Prior to the introduction of Android Studio 3.6, access to layout views from within the code of an app involved the use of the `findViewById()` method.
- An alternative is now available in the form of view bindings.
- View bindings consist of classes which are automatically generated by Android Studio for each XML layout file.
- These classes contain bindings to each of the views in the corresponding layout, providing a safer option to that offered by the `findViewById()` method.
- As of Android Studio 3.6, however, view bindings are not enabled by default and additional steps are required to manually enable and configure support within each project module.