MODULE I

INTRODUCING THE ANDROID COMPUTING PLATFORM

The Android platform embraces the idea of general-purpose computing for handheld devices. It is a comprehensive platform that features a Linux-based operating system stack for managing devices, memory, and processes. Android's Java libraries cover telephony, video, speech, graphics, connectivity, UI programming, and a number of other aspects of the device.

Although built for mobile- and tablet-based devices, the Android platform exhibits the characteristics of a full-featured desktop framework. Google makes this framework available to Java programmers through a Software Development Kit (SDK) called the Android SDK. When we are working with the Android SDK, we rarely feel that we are writing to a mobile device because we have access to most of the class libraries that we use on a desktop or a server—including a relational database.

The Android SDK supports most of the Java Platform, Standard Edition (Java SE), except for the Abstract Window Toolkit (AWT) and Swing. In place of AWT and Swing, Android SDK has its own *extensive modern UI framework*. Because we're programming our applications in Java, we should expect that we need a Java Virtual Machine (JVM) that is responsible for interpreting the runtime Java byte code. A JVM typically provides the necessary optimization to help Java reach performance levels comparable to compiled languages such as C and C++. Android offers its own optimized JVM to run the compiled Java class files in order to counter the handheld device limitations such as memory, processor speed, and power. This virtual machine is called the Dalvik VM, which we'll explore in a later section, "Delving into the Dalvik VM."

HISTORY OF ANDROID

Mobile phones use a variety of operating systems, such as Symbian OS, Microsoft's Windows Phone OS, Mobile Linux, iPhone OS (based on Mac OS X), Moblin (from Intel), and many other proprietary OSs. So far, no single OS has become the de facto standard. The available APIs and environments for developing mobile applications are too restrictive and seem to fall behind when compared to desktop frameworks. In contrast, the Android platform promised openness, affordability, open source code, and, more important, a high-end, all-in-one-place, consistent development framework.

Google acquired the startup company Android Inc. in 2005 to start the development of the Android platform. The key players at Android Inc. included Andy Rubin, Rich Miner, Nick Sears, and Chris White.

The version history of the Android mobile operating system began with the release of the Android alpha in November 2007 called “Early Look”. The first commercial version, Android 1.0, was released in September 2008. Android is continually developed by Google and the Open Handset Alliance (OHA), and has seen a number of updates to its base operating system since the initial release.
Versions 1.0 and 1.1 were not released under specific code names, but since April 2009's Android 1.5 "Cupcake", Android versions have had confectionery-themedcode names. Each is in alphabetical order, with the most recent being Android 6.0"Marshmallow", released in October 2015.

Code name	**Version number**	**Initial release date**	**API level**
	1.0	September 23, 2008	1
	1.1	February 9, 2009	2
Cupcake	1.5	April 27, 2009	3
Donut	1.6	September 15, 2009	4
Eclair	2.0–2.1	October 26, 2009	5–7
Froyo	2.2–2.2.3	May 20, 2010	8
Gingerbread	2.3–2.3.7	December 6, 2010	9–10
Honeycomb	3.0–3.2.6	February 22, 2011	11–13
Ice Cream Sandwich	4.0–4.0.4	October 18, 2011	14–15
Jelly Bean	4.1–4.3.1	July 9, 2012	16–18
KitKat	4.4–4.4.4, 4.4W–4.4W.2	October 31, 2013	19–20
Lollipop	5.0–5.1.1	November 12, 2014	21–22
Marshmallow	6.0–6.0.1	October 5, 2015	23
N	Developer Preview 3		

Note:-
The main architectural goal of Android is to allow applications to interact with one another and reuse of components.

Android 1.0, 1.1

- Did not support soft keyboards.

Android 1.5 (Cupcake)

- Soft keyboard
- Advanced media recording
- Widget, live folders etc

Android 1.6 (Donut) 2.0 (Eclair)

- Advanced search capability
- Text to speech

Android 2.2 (Froyo)

- USB Tethering
- WiFi Hotspot
- Adobe Flash

Android 2.3 (Gingerbread)

- Camera and video in low light condition
- Installation of apps from SD card
- Sensors, Video Chats

Android 3.0 (Honeycomb)

- Focussed on Tablet
- Dual core Processor
- Use large screen
- Action bar, drag and drop

Android 4.0 (Ice Cream Sandwich)

- HD screen
- Unlock screen in different ways
- Camera, face detection, auto focus, etc
- Video, photo, live effects
- Roboto font family installed
- Integrated screenshot capture

Android 4.1 (Jelly Bean)

- Smooth user interface
- Bidirectional text and other language support
- Multichannel audio
- Multiple user accounts (tablet only)
- Dial pad auto complete

Android 4.4 (Kitkat)

- Restrict for apps when accessing external storage
- Wireless printing
- Google chrome for android
- More sensors
- Verified booting
- GPS support

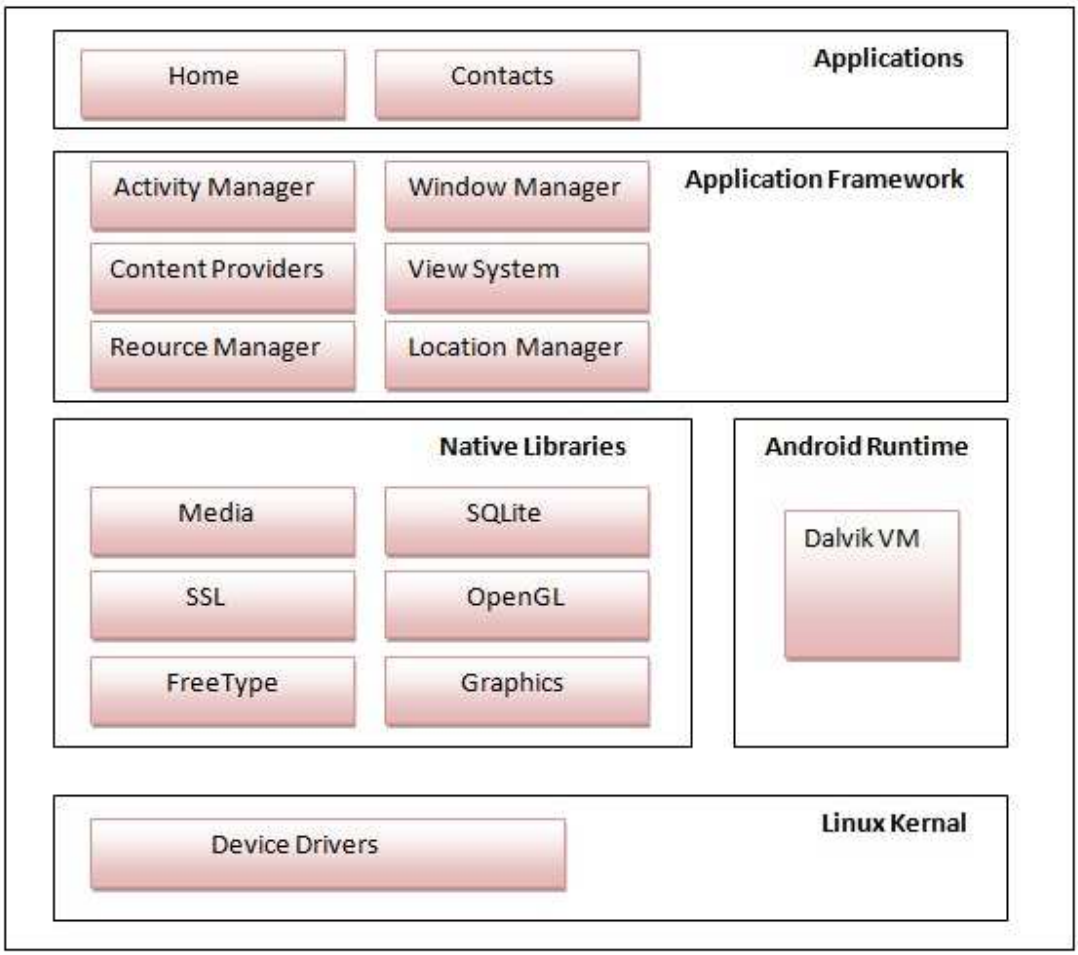
Android 5.0 (Lollipop)

- Improved garbage collection
- Dalvik replaced to Just-in-Time (JIT)
- Supports 64-bit, print preview
- Recently used apps are remembered even after restarts the device
- New 15 languages are added
- Smart lock added

Android 6.0 (Marshmallow)

- Introduce Doze mode to reduce CPU speed when screen is off mode
- Finger print reader support
- 4K display mode for apps
- MIDI supports
- USB Type- C supports

ANDROID SOFTWARE STACK



At the core of the Android platform is a Linux kernel responsible for device drivers, resource access, power management, and other OS duties. The supplied device

drivers include Display, Camera, Keypad, Wi-Fi, Flash Memory, Audio, and inter-process communication (IPC). Sitting at the next level, on top of the kernel, are a number of C/C++ libraries such as OpenGL, WebKit, FreeType, Secure Sockets Layer (SSL), the C runtime library (libc), SQLite, and Media. The media libraries are based on PacketVideo's OpenCORE. These libraries are responsible for recording and playback of audio and video formats. A library called Surface Manager Controls access to the display system and supports 2D and 3D.

The WebKit library is responsible for browser support; it is the same library that supports Google Chrome and Apple's Safari. The FreeType library is responsible for font support. SQLite is a relational database that is available on the device itself. SQLite is also an independent open source effort for relational databases and not directly tied to Android. Most of the application framework accesses these core libraries through the Dalvik VM, the gateway to the Android platform. As we indicated in the previous sections, Dalvik is optimized to run multiple instances of VMs. As Java applications access these core libraries, each application gets its own VM instance.

The Android Java API's main libraries include telephony, resources, locations, UI, content providers (data), and package managers (installation, security, and so on). From a media perspective, the Android platform supports the most common formats for audio, video, and images. From a wireless perspective, Android has APIs to support Bluetooth, EDGE, 3G, Wi-Fi, and Global System for Mobile Communication (GSM) telephony, depending on the hardware.

DELVING INTO DALVIK VM

Google optimizing designs for low-powered handheld devices The key figure in Google's implementation of this JVM is Dan Bornstein, who wrote the Dalvik VM—Dalvik is the name of a town in Iceland. Dalvik VM takes the generated Java class files and combines them into one or more Dalvik Executable (.dex) files. The goal of the Dalvik VM is to find every possible way to optimize the JVM for space, performance, and battery life. The final executable code in Android, as a result of the Dalvik VM, is based not on Java byte code but on .dex files instead. This means we cannot directly execute Java byte code; we have to start with Java class files and then convert them to linkable .dex files.

DEVELOPING END USER APPLICATION USING ANDROID SDK

To develop end-user applications on Android we need some technologies the Android emulator, Android foundational components, UI programming, services, media, telephony, animation, and more.

1. The Android Emulator

The Android SDK ships with an Eclipse plug-in called Android Development Tools (ADT). We will use this Integrated Development Environment (IDE) tool for developing, debugging, and testing our Java applications. We can also use the Android SDK without using ADT; we do not use command-line tools instead. Both approaches support an emulator that we can use to run, debug, and test our applications. We will not even need the real device for 90% of our application development. The full-featured Android emulator mimics most of the device features. The emulator limitations include USB connections, camera and video capture, headphones, battery simulation, Bluetooth, Wi-Fi, NFC, and OpenGL ES 2.0.

With the Android emulator, the processor is based on Advanced RISC Machine (ARM). ARM is a 32-bit microprocessor architecture based on Reduced Instruction Set Computing (RISC), in which design simplicity and speed is achieved through a reduced number of instructions in an instruction set. The emulator runs the Android version of Linux on this simulated processor. ARM is widely used in handhelds and other embedded electronics where lower power consumption is important. Much of the mobile market uses processors based on this architecture.

2. The android UI

It is a fourth-generation UI framework. if we consider the traditional C-based Microsoft Windows API the first generation and the C++-based Microsoft Foundation Classes (MFC) the second generation. The Java-based Swing UI framework would be the third generation, introducing design flexibility far beyond that offered by MFC. The Android UI, JavaFX, Microsoft Silverlight, and Mozilla XML User Interface Language (XUL) fall under this new type of fourth-generation UI framework. Programming in the Android UI involves declaring the interface in XML files. This is very much like HTML-based web pages. Even menus in our Android application are loaded from XML files. Screens or windows in Android are often referred to as *activities*, which comprise multiple views that a user needs in order to accomplish a logical unit of action. *Views* are Android's basic UI building blocks, and we can further combine them to form composite views called *view groups*.

Android 3.0 introduced a new UI concept called *fragments* to allow developers to chunk views and functionality for display on tablets. Tablets provide enough screen space for multi-pane activities, and fragments provide the abstraction for the panes.

3. The Android Functional Components

Android relies on a new concept called an *intent*. An intent is an intra- and inter-process mechanism to invoke components in Android. A component in Android is a piece of code that has a well defined life cycle. An activity representing a window in an Android application is a component. A service that runs in an Android process and serves other clients is a component. A receiver that wakes up in response to an event is another example of a component in Android.

```
public static void invokeWebBrowser(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.google.com"));
activity.startActivity(intent);
}
```

Another new concept in Android is the *content provider*. A content provider is an abstraction of a data source that makes it look like an emitter and consumer of RESTful services. The underlying SQLite database makes this facility of content providers a powerful tool for application developers.

4. The Advanced UI Concept

Android supports dialogs, and all dialogs in Android are asynchronous. These asynchronous dialogs present a special challenge to developers accustomed to the synchronous modal dialogs in some windowing frameworks. Android offers extensive support for animation. There are three fundamental ways to accomplish animation. We can do frame-by-frame animation. Or we can provide tweening animation by changing view transformation matrices (position, scale, rotation, and alpha). Or we can also do tweening animation by changing properties of objects.

Android also supports 3D graphics through its implementation of the OpenGL ES 1.0 and 2.0 standards. OpenGL ES, like OpenGL, is a C-based flat API. The Android SDK, because it's a Java-based programming API, needs to use Java binding to access the OpenGL ES. Java ME has already defined this binding through Java Specification Request (JSR) 239 for OpenGL ES, and Android uses the same Java binding for OpenGL ES in its implementation.

Android has a number of new concepts that revolve around *information at our fingertips* using the home screen. The first of these is *live folders*. Using live folders, we can publish a collection of items as a folder on the homepage. The contents of this collection change as the underlying data changes. This changing data could be either on the device or from the Internet. Due to space limitations, we are not able to cover live folders in the fourth edition of the book.

Integrated Android Search is the third homepage-based idea. Using integrated search, we can search for content both on the device and also across the Internet. Android also supports touch screen and gestures based on finger movements on the device. Android allows we to record any random motion on the screen as a named gesture. This gesture can then be used by applications to indicate specific actions.

Another necessary innovation required for a mobile device is the dynamic nature of its configurations. For instance, it is very easy to change the viewing mode of a handheld between portrait and landscape. Drag-and-drop is introduced for tablets in 3.0.

5. The Android Service Components

Security is a fundamental part of the Android platform. In Android, security spans all phases of the application life cycle—from design-time policy considerations to runtime boundary checks. Location-based service is another of the more exciting components of the Android SDK. This portion of the SDK provides application developers with APIs to display and manipulate maps, as well as obtain real-time device-location information.

6. The Android Media And Telephony Components

Android has APIs that cover audio, video, and telephony components. Starting with Android 2.0, Android includes the Pico Text-to-Speech engine. Due to space limitations, we are not able to include Text-to-Speech. The third edition does cover the Text-to-Speech API. Android ties all these concepts into an application by creating a single XML file that defines what an application package is. This file is called the application's *manifest file* (AndroidManifest.xml).

ANDROID JAVA PACKAGES

One way to get a quick snapshot of the Android platform is to look at the structure of Java packages. Because Android deviates from the standard JDK distribution, it is important to know what is supported and what is not. Here's a brief description of the important packages that are included in the Android SDK:

android.app*:* Implements the Application model for Android. Primary classes include Application, representing the start and stop semantics, as well as a number of activity-related classes, fragments, controls, dialogs, alerts, and notifications. We work with most of these classes through out this book.
android.app.admin*:* Provides the ability to control the device by folks such as enterprise administrators.
android.accounts*:* Provides classes to manage accounts such as Google, Facebook, and so on. The primary classes are AccountManager and Account.

***android.animation**:* Hosts all the new property animation classes.
***android.app.backup**:* Provides hooks for applications to back up and restore their data when folks switch their devices.
***android.appwidget**:* Provides functionality for home screen widgets.
***android.bluetooth**:* Provides a number of classes to work with Bluetooth functionality. The main classes include BluetoothAdapter, BluetoothDevice, BluetoothSocket, BluetoothServerSocket, and BluetoothClass. We can use BluetoothAdapter to control the locally installed Bluetooth adapter.
***android.content**:* Implements the concepts of content providers. Content providers abstract out data access from data stores.
***android.content.res**:* Provides access to resource files, both structured and unstructured. The primary classes are AssetManager (for unstructured resources) and Resources.
***android.database**:* Implements the idea of an abstract database. The primary interface is the Cursor interface.
***android.database.sqlite**:* Implements the concepts from the android.database package using SQLite as the physical database. Primary classes are SQLiteCursor, SQLiteDatabase, SQLiteQuery, SQLiteQueryBuilder, and SQLiteStatement.
***android.drm**:* Classes related to Digital Rights Management.
***android.graphics**:* Contains the classes Bitmap, Canvas, Camera, Color, Matrix, Movie, Paint, Path, Rasterizer, Shader, SweepGradient, and TypeFace.
***android.graphics.drawable**:* Implements drawing protocols and background images, and allows animation of drawable objects.
***android.graphics.drawable.shapes**:* Implements shapes including ArcShape, OvalShape, PathShape, RectShape, and RoundRectShape.
***android.hardware**:* Implements the physical Camera-related classes. The Camera represents the hardware camera, whereas
android.graphics.Camera: Rrepresents a graphical concept that's not related to a physical camera at all.
***android.hardware.usb**:* Let us talk to USB devices from Android.
***android.location**:* Contains the classes Address, GeoCoder, Location, LocationManager, and LocationProvider.
***android.media**:* Contains the classes MediaPlayer, MediaRecorder, Ringtone, AudioManager, and FaceDetector. MediaPlayer, which supports streaming, is used to play audio and video. MediaRecorder is used to record audio and video. The Ringtone class is used to play short sound snippets that could serve as ringtones and notifications.
***android.media.audiofx**:* Provides audio effects.
***android.media.effect**:* Provides video effects.
***android.mtp**:* Provides the ability to interact with cameras and music devices.
***android.net**:* Implements the basic socket-level network APIs. Primary classes include Uri, ConnectivityManager, LocalSocket, and LocalServerSocket. It is also

worth noting here that Android supports HTTPS at the browser level and also at the network level. Android also supports JavaScript in its browser.
android.net.rtp: Supports streaming protocols.
android.net.sip: Provides support for VOIP.
android.net.wifi: Manages Wi-Fi connectivity. Primary classes include WifiManager and WifiConfiguration. WifiManager is responsible for listing the configured networks and the currently active Wi-Fi network.
android.net.wifi.p2p*:* Supports P2P networks with Wi-Fi Direct.
android.telephony*:* Contains the classes CellLocation, PhoneNumberUtils, and TelephonyManager. TelephonyManager lets we determine cell location, phone number, network operator name, network type, phone type, and Subscriber Identity Module (SIM) serial number.
android.telephony.gsm*:* Allows we to gather cell location based on cell towers and also hosts classes responsible for SMS messaging. This package is called GSM because Global System for Mobile Communication is the technology that originally defined the SMS data messaging standard.
android.telephony.cdma*:* Provides support for CDMA telephony. *android.test*, *android.test.mock*, *android.test.suitebuilder:* Packages to support writing unit tests for Android applications.
android.text*:* Contains text-processing classes.
android.text.style*:* Provides a number of styling mechanisms for a span of text.
android.utils*:* Contains the classes Log, DebugUtils, TimeUtils, and Xml.
android.view*:* Contains the classes Menu, View, and ViewGroup, and a series of listeners and callbacks.
android.view.animation*:* Provides support for tweening animation. The main classes include Animation, a series of interpolators for animation, and a set of specific animator classes that include AlphaAnimation, ScaleAnimation, TranslationAnimation, and RotationAnimation.Some of the classes.
android.webkit*:* Contains classes representing the web browser. The primary classes include WebView, CacheManager, and CookieManager. *android.widget:* Contains all of the UI controls usually derived from the View class. Primary widgets include Button, Checkbox, Chronometer, AnalogClock, DatePicker, DigitalClock, EditText, ListView, FrameLayout, GridView, ImageButton, MediaController, ProgressBar, RadioButton, RadioGroup, RatingButton, Scroller, ScrollView, Spinner, TabWidget, TextView, TimePicker, VideoView, and ZoomButton.
com.google.android.maps*:* Contains the classes MapView, MapController, and MapActivity, essentially classes required to work with Google maps.

In addition, Android provides a number of packages in the java.* namespace. These include awt.font, beans, io, lang, lang.annotation, lang.ref, lang.reflect, math, net, nio, nio.channels, nio.channels.spi, nio.charset, security, security.acl, security.cert, security.interfaces, security.spec, sql, text, util,

util.concurrent, util.concurrent.atomic, util.concurrent.locks, util.jar, util.logging, util.prefs, util.regex, and util.zip.

Android comes with these packages from the javax namespace: crypto, crypto.spec, microedition.khronos.egl, microedition.khronos.opengles, net, net.ssl, security.auth, security.auth.callback, security.auth.login, security.auth.x500, security.cert, sql, xml, and xmlparsers.

In addition to these, it contains a lot of packages from org.apache.http.* as well as org.json, org.w3c.dom, org.xml.sax, org.xml.sax.ext, org.xml.sax.helpers, org.xmlpull.v1, and org.xmlpull.v1.sax2. Together, these numerous packages provide a rich computing platform to write applications for handheld devices.

SETTING UP THE DEVELOPMENT ENVIRONMENT

To build applications for Android, we need the Java SE Development Kit (JDK), the Android SDK, and a development environment. Strictly speaking, we can develop our applications using a primitive text editor, but for the purposes of this book, we use the commonly available Eclipse IDE. The Android SDK requires JDK 5 or JDK 6 (the examples use JDK 6) and Eclipse 3.5 or higher (this note uses Eclipse 3.5, also known as Galileo, and 3.6, also known as Helios).

The Android SDK is compatible with Windows (Windows XP, Windows Vista, and Windows 7), Mac OS X (Intel only), and Linux (Intel only). In terms of hardware, we need an Intel machine, the more powerful the better. To make our life easier, we want to use Android Development Tools (ADT). ADT is an Eclipse plug-in that supports building Android applications with the Eclipse IDE. The Android SDK is made up of two main parts: the tools and the packages. When we first install the SDK, all we get are the base tools. These are executables and supporting files to help we develop applications. The packages are the files specific to a particular version of Android (called a *platform*) or a particular add-on to a platform. The platforms include Android 1.5 through 4.0.

To build Android applications, we need to establish a development environment. In this section, we walk through downloading JDK 6, the Eclipse IDE, the Android SDK (tools and packages), and ADT.

Steps:

1. Installing JDK 6.0 and set the path

The first thing we need is the Java SE Development Kit. The Android SDK requires JDK 5 or higher; we developed the examples using JDK 6. For Windows, download JDK 6 and install it. We only need the JDK, not the bundles.

To set the JAVA_HOME environment variable to point to the JDK install folder. To do this on a Windows XP machine, choose Start ➤ My Computer, right-click, selects Properties, choose the Advanced tab, and click Environment Variables. Click New to add the variable or Edit to modify it if it already exists. The value of JAVA_HOME is something like C:\Program Files\Java\jdk1.6.0_27.

2. Installing Eclipse 3.6

The Eclipse distribution is a .zip file that can be extracted just about anywhere. The simplest place to extract to on Windows is C:\, which results in a C:\eclipse folder where we find eclipse.exe. When we first start up Eclipse, it asks we for a location for the workspace. To make things easy, we can choose a simple location such as C:\android or a directory under our home directory.

3. Installing Android SDK

To build applications for Android, we need the Android SDK. The tools part of the SDK includes an emulator so we don't need a mobile device with the Android OS to develop Android applications. It also has a setup utility to allow we to install the packages that we want to download. We can download the Android SDK from http://developer.android.com/sdk. It ships as a .zip file, similar to the way Eclipse is distributed, so we need to unzip it to an appropriate location. For Windows, unzip the file to a convenient location (we used the C: drive), after which we should have a folder called something like C:\android-sdkwindows that contains the files. To set environment variable path, For Windows, get back to the Environment Variables window. Edit the PATH variable and add a semicolon (;) on the end, followed by the path to the Android SDK tools folder, followed by another semicolon, followed by the path to the Android SDK platform-tools folder, following by another semicolon, and then %JAVA_HOME%\bin. Click OK when we're done.

4. The Tool Window

The easiest way to create a tools window in Windows is to choose Start ➤ Run, type in cmd, and click OK. to know the IP address of our workstation To find this in Windows, launch a tools window and enter the command ipconfig. The results contain an entry for IPv4 (or something like that) with our IP address listed next to it. An IP address looks something like this: 192.168.1.25. We may see a network connection called localhost or lo; the IP address for this network connection is 127.0.0.1. This is a special network connection used by the operating system and is not the same as our workstation's IP address. Look for a different number for our workstation's IP address.

5. Installing Android Development Tool (ADT)

It is an Eclipse plug-in that helps we build Android applications. Specifically, ADT integrates with Eclipse to provide facilities for we to create, test,

and debug Android applications. Eclipse downloads the Developer Tools and installs them. We need to restart Eclipse for the new plug-in to show up in the IDE. The final step to make ADT functional in Eclipse is to point it to the Android SDK. In Eclipse, select Window ➤ Preferences. In the Preferences dialog box, select the Android node and set the SDK Location field to the path of the Android SDK (see Figure 2–4), and then click the Apply button. We may want to make one more Preferences change on the Android ➤ Build page. The Skip Packaging option should be checked if we'd like to make our file saves faster. By default, the ADT readies our application for launch every time it builds it. By checking this option, packaging and indexing occur only when truly needed.

FUNDAMENTAL COMPONENTS

When we build applications for Android we need to understand JavaServer Pages (JSP) and servlets in order to write Java 2 Platform, Enterprise Edition (J2EE) applications. Similarly, we need to understand views, activities, fragments, intents, content providers, services, and the AndroidManifest.xml file.

1. View

Views are user interface (UI) elements that form the basic building blocks of a user interface. A view can be a button, a label, a text field, or many other UI elements. Views are also used as containers for views, which mean there's usually a hierarchy of views in the UI.

2. Activity

An *activity* is a UI concept that usually represents a single screen in our application. It generally contains one or more views, but it doesn't have to. An activity is pretty much like it sounds—something that helps the user do one thing, which could be viewing data, creating data, or editing data. Most Android applications have several activities within them.

3. Fragment

When a screen is large, it becomes difficult to manage all of its functionality in a single activity. *Fragments* are like sub-activities, and an activity can display one or more fragments on the screen at the same time. When a screen is small, an activity is more likely to contain just one fragment, and that fragment can be the same one used within larger screens.

4. Intent

An *intent* generically defines an "intention" to do some work. Intents encapsulate several concepts, so the best approach to understanding them is to see examples of their use. We can use intents to perform the following tasks:

Broadcast a message.
Start a service.
Launch an activity.
Display a web page or a list of contacts.
Dial a phone number or answer a phone call.

Intents are not always initiated by our application—they're also used by the system to notify our application of specific events (such as the arrival of a text message). Intents can be explicit or implicit. If we simply say that we want to display a URL, the system decides what component will fulfill the intention. We can also provide specific information about what should handle the intention.

5. Content Provider

Data sharing among mobile applications on a device is common. Therefore, Android defines a standard mechanism for applications to share data (such as a list of contacts) without exposing the underlying storage, structure, and implementation. Through content providers, we can expose our data and have our applications use data from other applications.

6. Service

Services in Android resemble services we see in Windows or other platforms—they're background processes that can potentially run for a long time. Android defines two types of services: local services and remote services. Local services are components that are only accessible by the application that is hosting the service. Conversely, remote services are services that are meant to be accessed remotely by other applications running on the device.

7. AndroidManifest.xml

AndroidManifest.xml, which is similar to the web.xml file in the J2EE world, defines the contents and behavior of our application. For example, it lists our application's activities and services, along with the permissions and features the application needs to run.

ANDROID VIRTUAL DEVICES

An Android Virtual Device (AVD) represents a device configuration. It allows developers to test their applications without hooking up an actual Android device (typically a phone or a tablet). AVDs can be created in various configurations to emulate different types of real devices.

RUNNING ON REAL DEVICE

The best way to test an Android app is to run it on a real device. Any commercial Android device should work when connected to our workstation, but we may need to do a little work to set it up.

We have to deal with USB drivers. Google supplies some with the Android packages, which are placed under the usb_driver subdirectory of the Android SDK directory. Other device vendors provide drivers for we, so look for them on their web sites. When we have the drivers set up, enable USB debugging on the device, and we're ready. Now that our device is connected to our workstation, when we try to launch an app, either it launches directly on the device a window opens in which we choose which device or emulator to launch into. If not, try editing our Run Configuration to manually select the target.

STRUCTURE OF ANDROID APPLICATION

An Android application is primarily made up of three pieces: the application descriptor, a collection of various resources, and the application's source code. Android applications have some artifacts that are required and some that are optional. The following table summarizes the elements of an Android application.

Artifact	Description	Required?
AndroidManifest.xml	The Android application descriptor file. This file defines the activities, content providers, services, and intent receivers of the application. We can also use this file to declaratively define permissions required by the application, as well as grant specific permissions to other applications using the services of the application. Moreover, the file can contain instrumentation detail that we can use to test the application or another application.	Yes
src	A folder containing all of the source code of the application	Yes
assets	An arbitrary collection of folders and files	No
Res	A folder containing the resources of the application. This is the parent folder of drawable, anim, layout, menu, values, xml, and raw.	Yes
drawable	A folder containing the images or image-descriptor files used by the application.	No
Animator	A folder containing the XML-descriptor files that describe the animations used by the application. On older Android versions, this is called anim.	No
Layout	A folder containing views of the application.	No

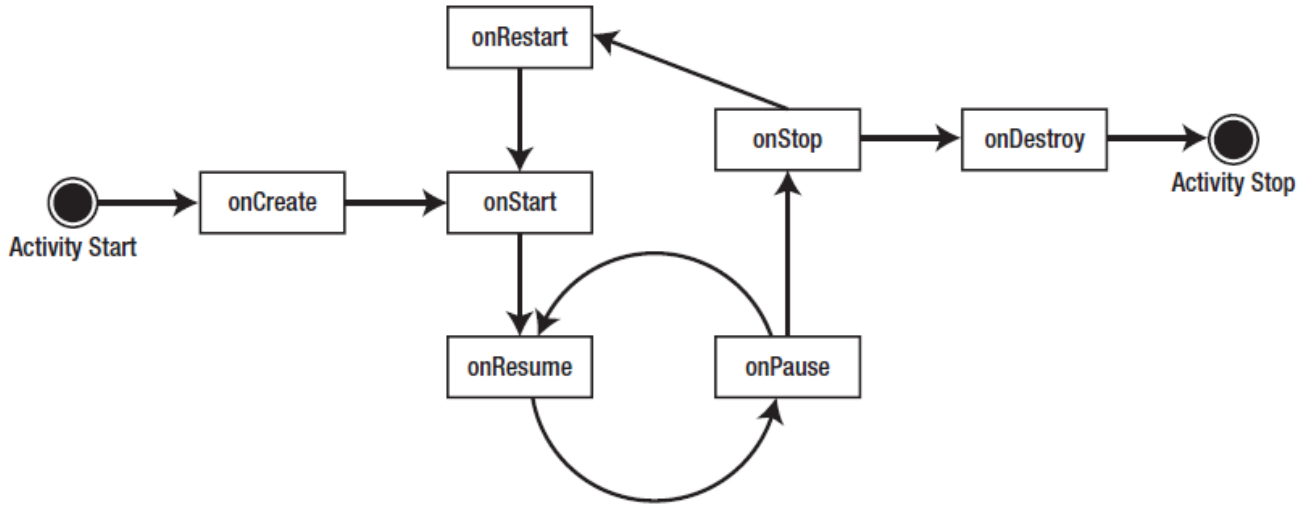
	We should create our application's views by using XML descriptors rather than coding them.	
Menu	A folder containing XML-descriptor files for menus in the application.	No
Values	A folder containing other resources used by the application. Examples of resources found in this folder include strings, arrays, styles, and colors.	No
Xml	A folder containing additional XML files used by the application.	No
raw	A folder containing additional data—possibly non-XML data— that is required by the application.	No

APPLICATION LIFE CYCLE.

The life cycle of an Android application is strictly managed by the system, based on the user's needs, available resources, and so on. Android is sensitive to the life cycle of an application and its components.

Life-Cycle Methods of an Activity

protected void onCreate(Bundle savedInstanceState);
protected void onStart();
protected void onRestart();
protected void onResume();
protected void onPause();
protected void onStop();
protected void onDestroy();



The system can start and stop our activities based on what else is happening. Android calls the onCreate() method when the activity is freshly created. onCreate() is always followed by a call to onStart(), but onStart() is not always preceded by a call to onCreate() because onStart() can be called if our application was stopped. When onStart() is called, our activity is not visible to the user, but it's about to be. onResume() is called after onStart(), just when the activity is in the foreground and accessible to the user. At this point, the user can interact with our activity. When the user decides to move to another activity, the system calls our activity's onPause() method. From onPause(), we can expect either onResume() or onStop() to be called. onResume() is called, for example, if the user brings our activity back to the foreground. onStop() is called if our activity becomes invisible to the user. If our activity is brought back to the foreground after a call to onStop(), then onRestart() is called. If our activity sits on the activity stack but is not visible to the user, and the system decides to kill our activity, onDestroy() is called.