Android Programming
Unit I

Introducing the android computing platform, History of android, Android software stack, Developing end user application using Android SDK, Android java packages, Setting up the development environment, Installing android development tools (ADT), Fundamental components, Android virtual devices, Running on real device, Structure of android application, Application life cycle.

WHAT IS ANDROID?

- Android is a software package and an open source linux- based operating system for mobile devices such as smartphones and tablet computers
- Android offers unified approach to application development for mobile devices
- Developers need only develop for android , and their applications should be able to run on different devices powered by android.
- **Irina Blok**: designed the official android logo
- The platform that's changing what mobile can do.

HISTORY

- Android Inc. was founded in Palo Alto of California by Andy Rubin, Rich Miner ,Nick sears and Chris White in 2003
- Android Inc. was acquired by Google in 2005 to start the development of Android platform.
- November 5 is celebrated as android's birthday

- Android SDK was first issued as an early look release in November 5, 2007 with the founding of Open Handset Alliance(OHA) , a consortium for 84 hardware ,software ,and telecommunication companies aimed to advancing open standards for mobile devices.
- In September 2008 HTC released T-mobile G1(HTC Dream), the first smartphone based on android platform.
- In September 2008 HTC released T-mobile G1(HTC Dream), the first smartphone based on android platform.

VERSION HISTORY

- The version history of the Android mobile operating system began with the public release of the Android beta on November 5, 2007
- The first commercial version, Android 1.0, was released on September 23, 2008.
- Android is continually developed by Google and the Open Handset Alliance, and it has seen several updates to its base operating system since the initial release.
- Android 1.0 and 1.1 were not released under specific code names, although Android 1.1 was unofficially known as Petit Four.
- Android code names were confectionery-themed and have been in alphabetical order since 2009's Android 1.5 Cupcake.
- Google ended the confectionery theming scheme in 2019 beginning with Android 10
- The most recent version of Android is Android 11, which was released on September 8 ,2020.

Code name	Version numbers	Initial release date	API level
No codename	1.0	September 23, 2008	1
Petit Four (only internally used)	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	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
Nougat	7.0	August 22, 2016	24
	7.1	October 4, 2016	25
	7.1.1 – 7.1.2	December 5, 2016	25
Oreo	8.0	August 21, 2017	26
	8.1	December 5, 2017	27
Pie	9.0	August 6, 2018	28
Android 10	**10.0**	September 3, 2019	29

CUPCAKE 1.5

- Released in April 2009
- First version of Android to have an on-screen keyboard
- Ability to upload videos to YouTube, support for third-party keyboards, and feature like automatically rotating phone’s screen to the right positions.
- The first Samsung Galaxy phone had the Android 1.5 cupcake.

DONUT 1.6

- Main feature included in Donut was that it supported carriers that used CDMA based networks
- Ability for the OS to operate on a variety of different screen sizes and resolutions
- It also included features like quick switching between the Cameras, Camcorder, and Gallery
- introduced the Quick Search Box.

ANDROID 2.0-2.1 ÉCLAIR

- First Android version with text-to-speech support.
- Introduced multiple account support, live wallpapers, navigation with Google Maps
- First smartphone with the Android 2.0 version was the Motorola Droid,

ANDROID 2.2 FROYO

- Froyo, short form for Frozen Yogurt was launched in May 2010.
- Wi-Fi mobile hotspot functions was introduced.
- It also included many other features such as flash support, push notifications via Android Cloud to Device Messaging (C2DM) service, and more.

ANDROID 2.3 GINGERBREAD

- Launched in September 2010.
- Updated UI design that provided increased efficiency and ease-of-use.
- It had support for extra-large screen sizes and resolution.
- More features such as native support for SIP VoIP internet telephones, improved text inputs using the virtual keyboard, better text suggestions and voice input capability were added
- One of the key features was its support for using NFC (near field communication) functions for smartphones.

ANDROID 3.0 HONEYCOMB

- Honeycomb was launched to be installed only for tablets and mobile devices with larger screens
- Google aimed for features that could not be handled by smartphones with smaller screens.
- But Honeycomb ended up as a version that not really required.
- Most of the features of Honeycomb were integrated with the next major version of Android.

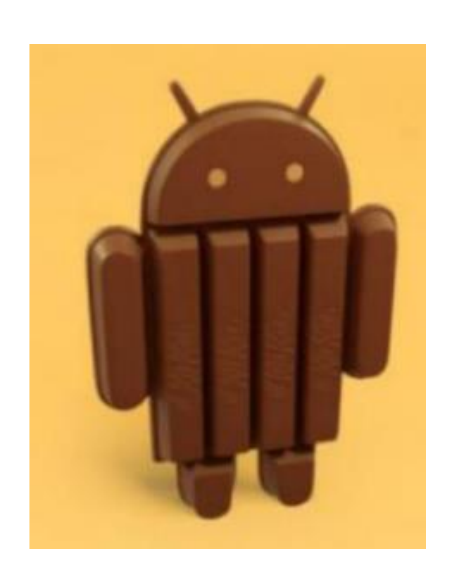
ANDROID 4.0 ICE CREAM SANDWICH

- Features of the previous version, Honeycomb, were integrated with the Ice Cream Sandwich version.
- This version was the first to introduce the feature to unlock the phone using its camera.
- Other notable changes with Ice Cream Sandwich included support for all the on-screen buttons, the ability to monitor the mobile and Wi-Fi data usage, and swipe gestures to dismiss notifications and browser tabs.

ANDROID 4.1-4.3 JELLY BEAN

- The notification part was improved a lot in this version.
- Full support for Google Chrome (Android version) was included in Android 4.2.
- Android's touch responsiveness was also improved.
- Jelly Bean was collectively the first Android version to support emoji and screensavers that are natively done.

ANDROID 4.4 KITKAT



- KitKat did not have many features.
- But one main feature was that KitKat could run on smartphones with even a 512 MB RAM.
- It was because KitKat used the Android Runtime (ART), though experimental, instead of the DVM (Dalvik Virtual Machine) originally used by Android.

ANDROID 5.0 LOLLIPOP

- Google's new Material Design language was first introduced in Lollipop
- It included changes in UIs like a revamped navigation bar and better-style notifications for the lock-screen etc.
- It brought the Flat Design concept into play.
- Google created more enhancements to Android devices' battery life with a Doze mode where background apps are killed once the show is turned off.

6.0 MARSHMALLOW

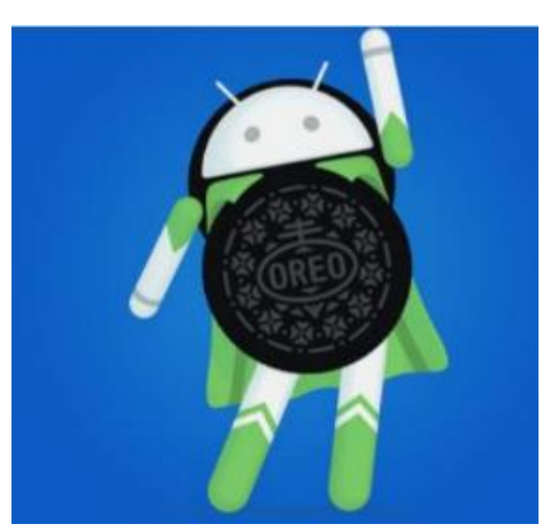
- First, the Android 6.0 version was to be called Macadamia Nut Cookie, but it was released as Marshmallow in May 2015.
- Marshmallow brought the addition of the memory manager,
- This was the first version that had native support for unlocking of the smartphone with biometric; fingerprint authentication.
- USB Type C support was included and Android pay was also introduced in Marshmallow.

ANDROID 7.0 NOUGAT

- Android 7.0 Nougat was released in August 2016.
- It came out with multitasking features, especially for smartphones with bigger screens.
- It included split-screen and fast switching between apps.
- Many changes behind the scenes were also made by Google such as switching to a new JIT compiler that could speed up apps.
- Google's own smartphone, the Pixel, and Pixel XL, and LG V20 came out with Android 7.0 Nougat.

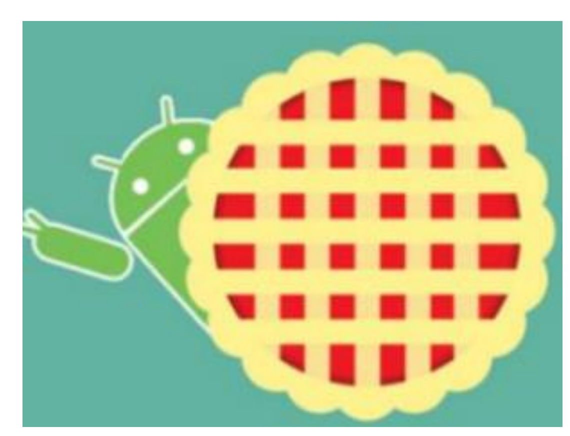
ANDROID 8.0 OREO

- It included many visual changes such as native support for picture-in-picture mode, new autofill APIs that could help in better managing the passwords and fill data, notification channels



ANDROID 9.0 PIE

- Released in August 2018.
- It came with a lot of new features and improvements.
- The new home-button was added in this version. When swiped up, it brings the apps used recently, a search bar and suggestions of five apps at the bottom.
- There was a new option added of swiping left to see the currently running apps.
- Improvements in battery life were also made in this version.
- Shush mode, a new feature was also added. It automatically puts the smartphone in Do not disturb mode. All you need to do is place your phone on any surface face down and DND will be automatically enabled.
- Added “Digital Wellbeing,” a feature that essentially tells you how often you use your phone, the apps that you use the most



ANDROID 10

- Along with the rollout of the latest version of Android, Google also announced a rebranding of the operating system, doing away with the naming scheme and instead sticking with version numbers only.
- Google also announced a new logo for Android, and a new color scheme.
- Android 10 marked the end of the Android navigation buttons.
- While Android 9 kept the back button, Android 10 now uses gestures instead
- Android 10 also brought a systemwide dark mode
- Another features include Sound Amplifier, faster security updates , Digital well being, set screen time limits, view app activity, manage apps and content restrictions

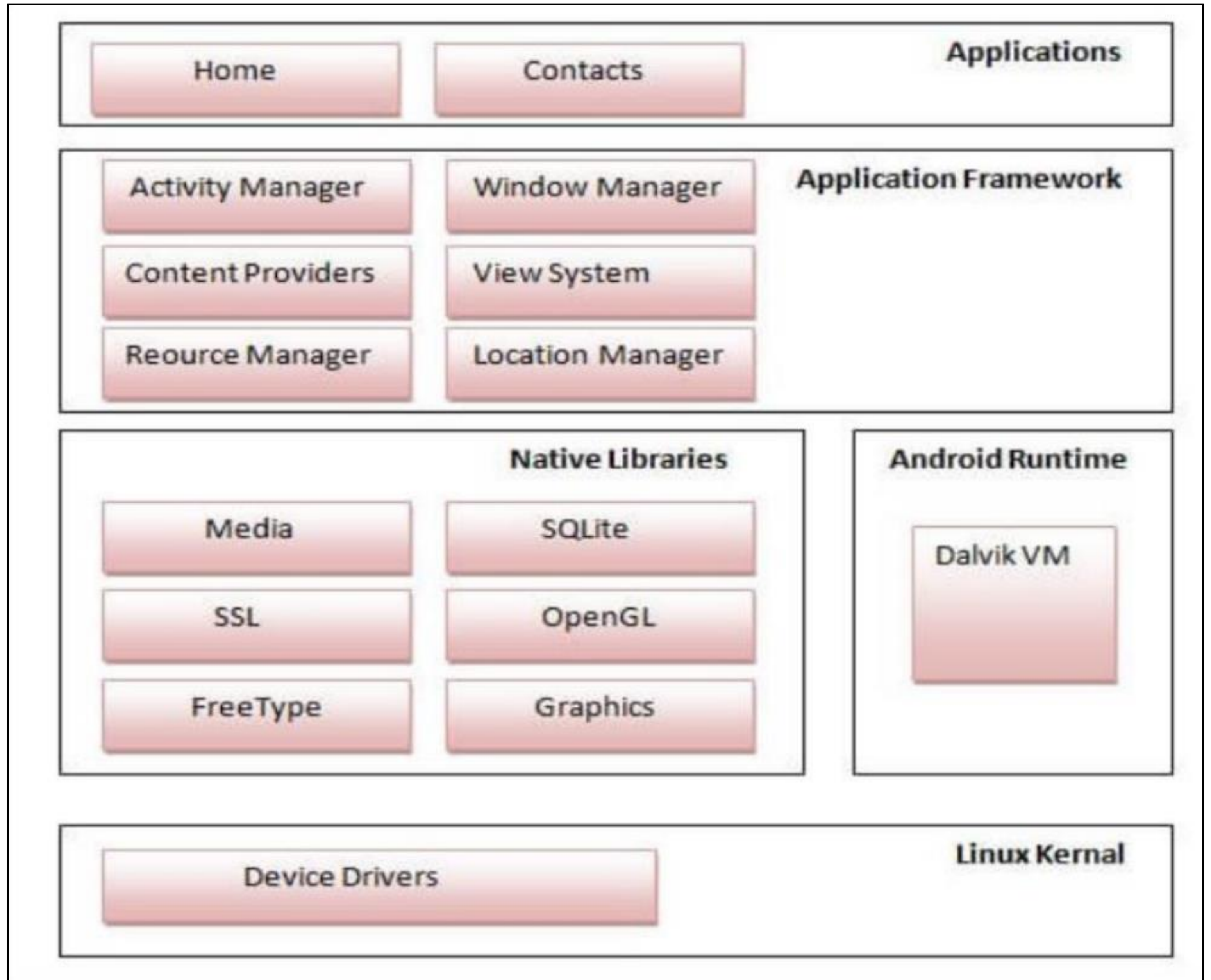
ANDROID 11

- Google released Android 11 on September 8, 2020
- **Features**
  - Conversations
  - Built-in screen recording
  - Predictive tools
  - Voice access
  - Device Controls
  - More security and privacy fixes sent to your phone from Google
  - Play.

ANDROID SOFTWARE STACK

- A solution stack or software stack is a set of software systems or components needed to create complete platform such that no additional software is needed to support applications
- Android OS is a stack of software components which is divided into 5 sections and 4 main layers
  - Linux kernel
  - Native libraries (middleware),
  - Android Runtime
  - Application Framework
  - Applications

ANDROID ARCHITECTURE



1. Linux kernel

- The term ‘kernel’ means ‘core’ as the name indicates ,It is the heart of android architecture that exists at the root of android architecture.
- Linux kernel is the base of the software layers upon which all other layers of the android are built.
- Users are not allowed to directly interact with this layer.
- All the essential hardware drivers are located in Linux Kernel which communicates with the hardware.
- Linux Kernel can also act as an abstraction layer between hardware and software layers

1. Linux kernel

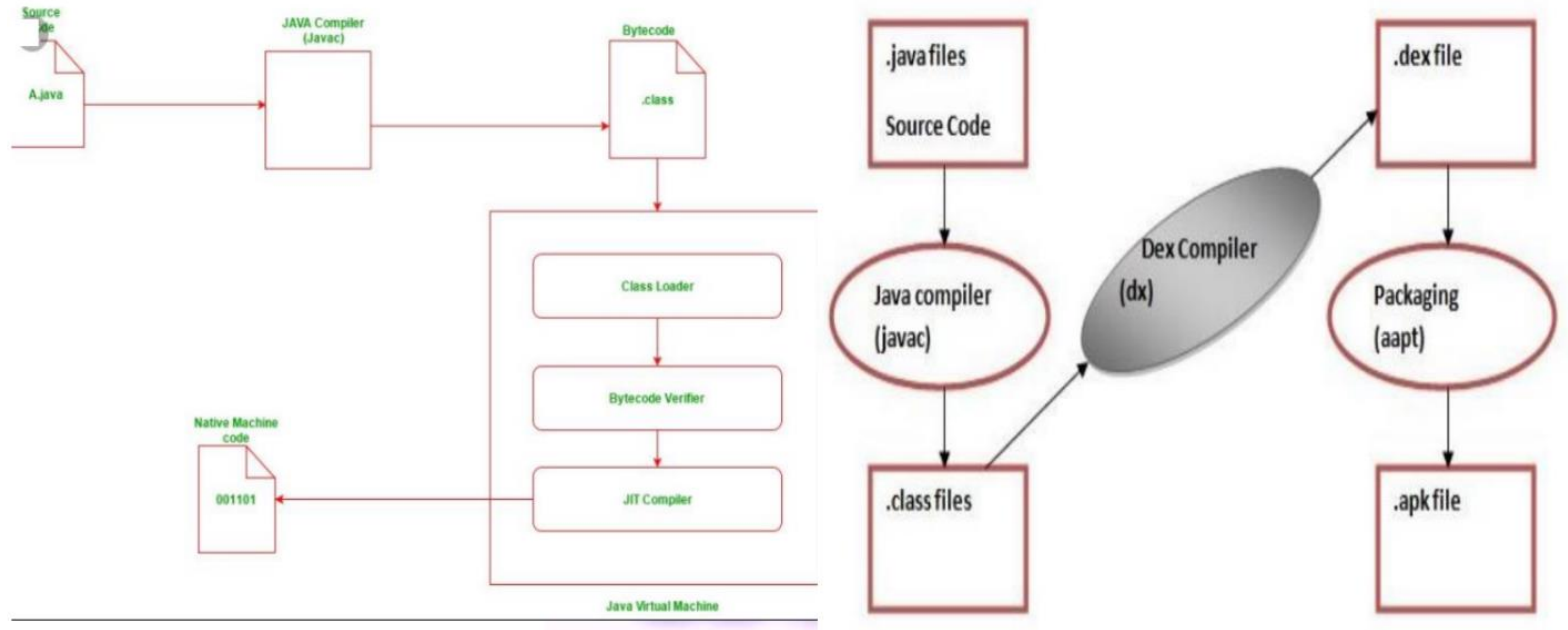
- The following are the important functions of kernel in the android system:
  - Hardware abstraction
  - Memory management programs
  - Security settings
  - Power management software
  - Support for shared libraries
  - Network stack
  - Device driver management

2. Native Libraries

- Next to Linux Kernel layer is the Android's Native Libraries.
- They provide capabilities for android's core features. Libraries help the device to handle different types of data.
- The following are some of the important native libraries:
  - **Surface Manager**: manages access to the display subsystem and 2D and 3D graphic layers from multiple applications .
  - **Media Framework**: for playing and recording audio and video formats.
  - **SQLite** : is an android database engine which stores data.
  - **WebKit** : is a browser engine that displays HTML content.
  - **OpenGL** : renders 2D or 3D graphic contents on the screen.
  - **FreeType** :for font support
  - **C runtime library (libc)**

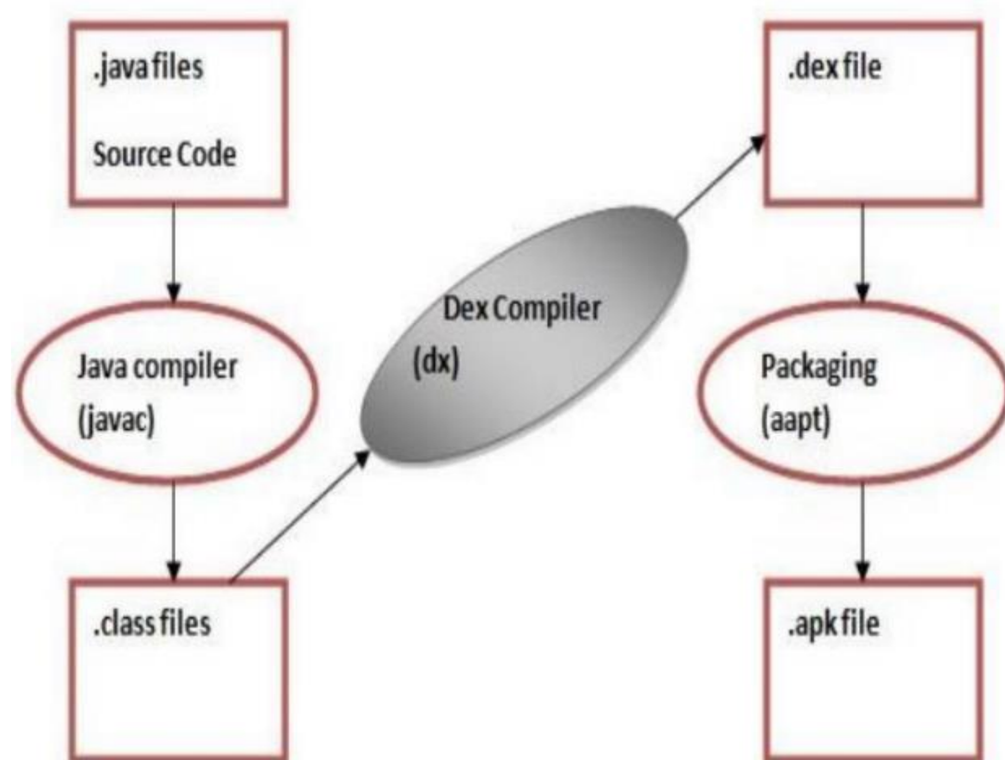
3. Android Runtime

- This is the 3rd section of the architecture and available on the second layer from the bottom
- In android runtime, there are core libraries and DVM (Dalvik Virtual Machine) which is responsible to run android application.
- DVM is like JVM but it is optimized for mobile devices. It consumes less memory and provides fast performance.
- DVM takes the generated java class files and combines them into one or more .dex files
- Goal of DVM is to find every possible ways to optimize the JVM for space, performance and battery life



Source
A.java
JAVA Compiler
(Javac)
Bytecode
.class
Class Loader
Bytecode Verifier
JIT Compiler
Native Machine
code
001101
Java Virtual Machine
.java files
Source Code
Java compiler
(javac)
.class files
Dex Compiler
(dx)
.dex file
Packaging
(aapt)
.apk file

- The javac tool compiles the java source file into the class file.
- The dx tool takes all the class files of your
- application and generates a single .dex file.
- It is a platform-specific tool.
- The Android Assets Packaging Tool (aapt) handles the packaging process.



4. Android Framework

- On the top of Native libraries and android runtime, there is android framework.
- Android framework layer which is written in Java includes Android API's such as UI (User Interface), telephony, resources, locations, Content Providers (data) and package managers.
- It provides a lot of classes and interfaces for android application development.
- The following are some of the basic tools with which applications are built:
  - **Activity Manager** : Activity life cycles of applications are managed.
  - **Content Providers** : data sharing among applications are managed.
  - **Telephony Manager** : manages the location via GPS or base station receiver.
  - **Resource Manager** : various types of resources used in applications are managed.

5. Applications

- On the top of android framework, there are applications.
- All applications such as home, contact, settings, games, browsers are using android framework that uses android runtime and libraries.
- Android runtime and native libraries are using linux kernel.

ANDROID JAVA PACKAGES

- **android.app**: implements application model for android
- **android.app.admin**: Provides device administration features at the system level
- **android.accounts**: provides classes to manage accounts such as google, facebook
- **android.animation**: to apply animation for different objects, animation classes and methods are included
- **Android.app.backup**: Contains the backup and restore functionality available to applications. If a user wipes the data on their device or upgrades to a new Android-powered device, all applications that have enabled backup can restore the user's previous data when the application is reinstalled.
- **Android.content**:Manages access to a central repository of data
- **Android.database**:Contains classes to explore data returned through a content provider.

ANDROID JAVA PACKAGES

- **android.database.sqlite**: Contains the SQLite database management classes that an application would use to manage its own private database.
- **android.graphics**: Provides low level graphics tools such as canvases, color filters, points, and rectangles that let you handle drawing to the screen directly.
- **android.media**: supports video streaming and play audio and video files
- **android.os**: provides operating system services through java such as IPC, file server management etc.
- **android.text**: provides text processing classes
- **android.view**: contains classes menu, view, viewgroups and a series of listeners
- **Android.webkit**: classes representimg web browser
- **Android.widget**: contains all of the UI controls(list, grid, image)

SETTING UP THE DEVELOPMENT ENVIRONMENT

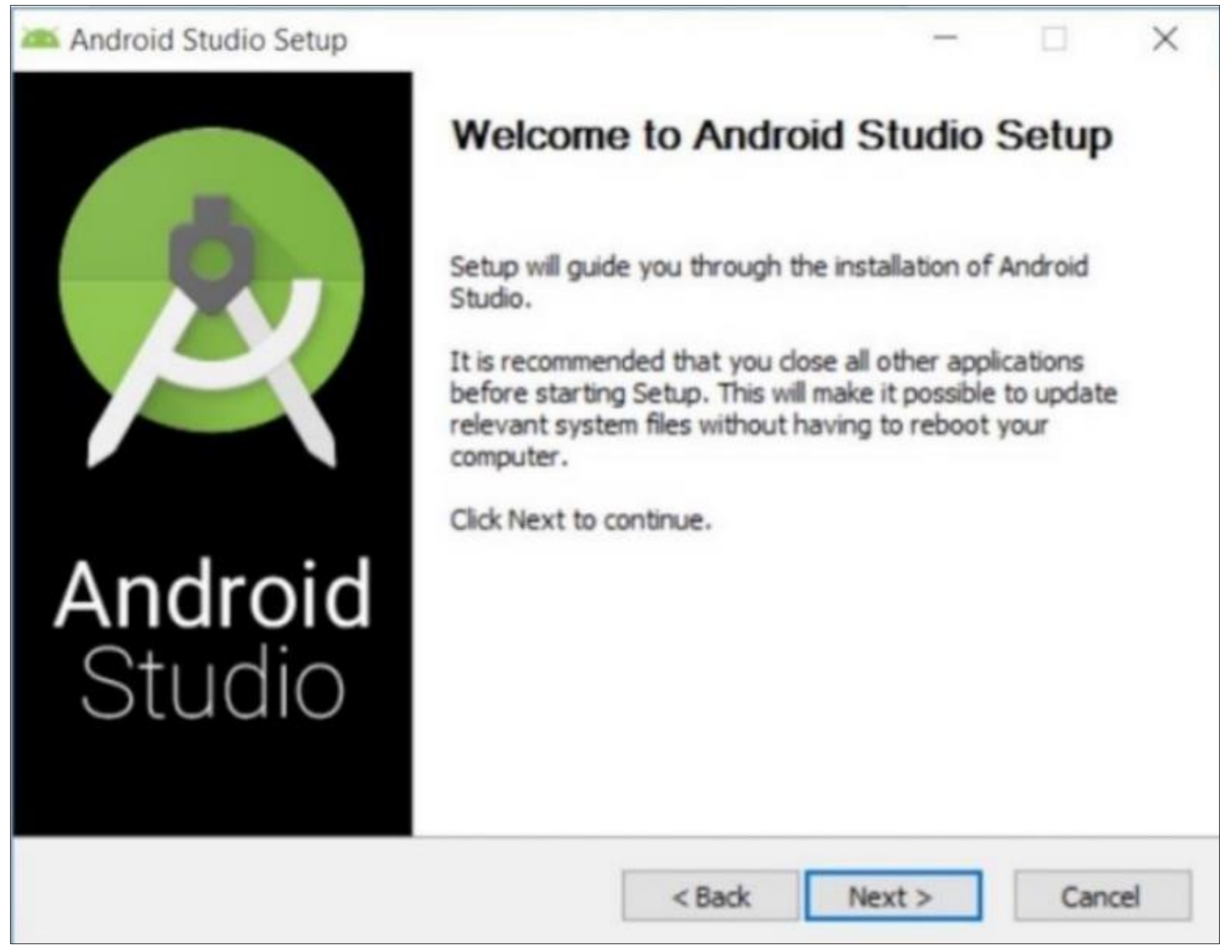
- Downloading JDK(The Android SDK requires JDK 5 or higher)
  - After installing set PATH variables
- Download the Eclipse IDE
- Downloading the Android SDK
  - The SDK contains 2 parts. base tools & package
  - The tools part includes an emulator and a setup utility to install the packages.
  - The packages are the files specific to a particular version of Android (called a platform) or a particular add-on to a platform(ex: google maps, vendor specific apps)
- Updating path environmental variable
- Installing Android Development Tools (ADT)
  - Eclipse plug-in that helps to build Android applications.
- From Eclipse, you can launch the SDK Manager. To do so, choose Window ➤ Android SDK Manager.

ANDROID STUDIO

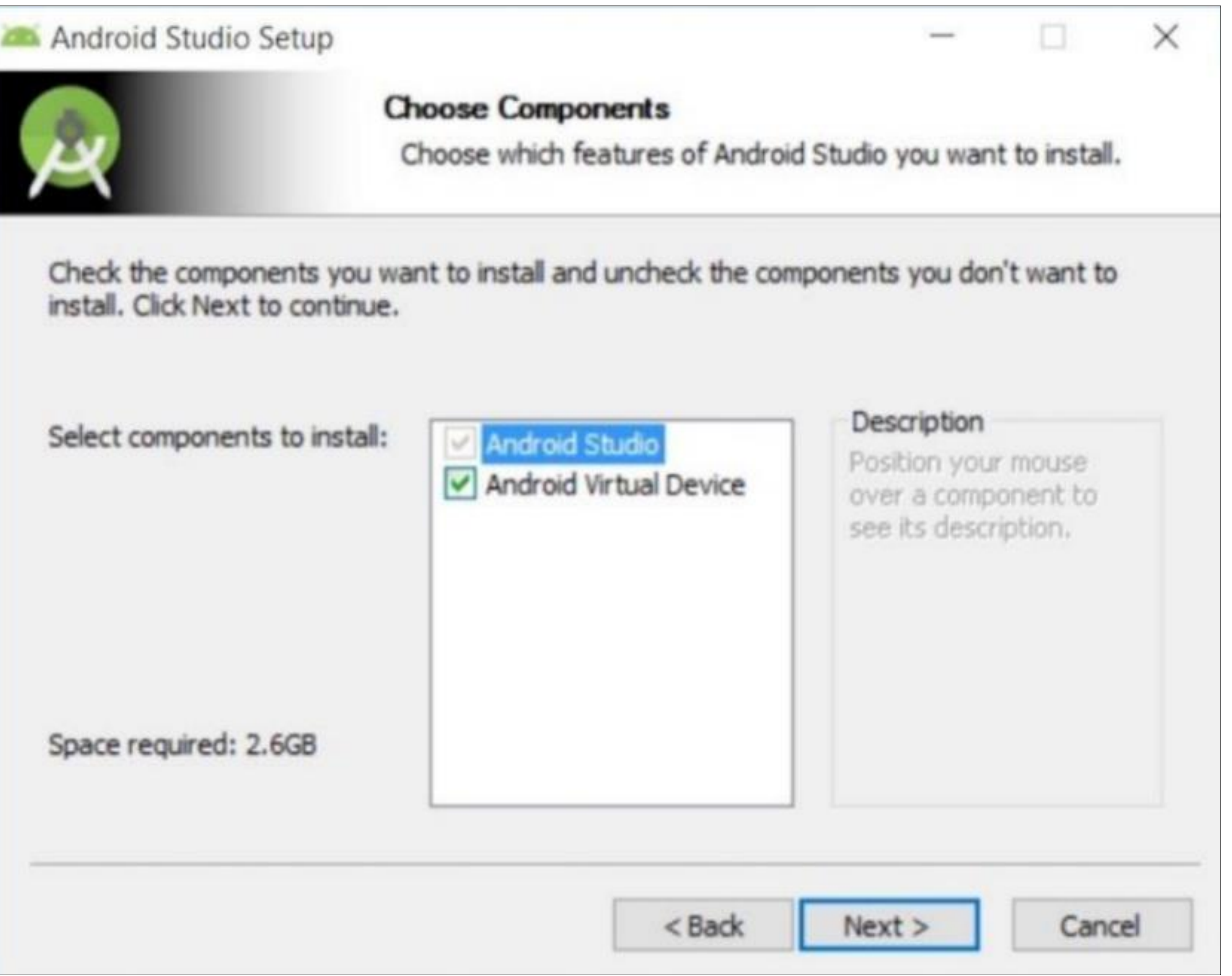
- Android Studio is the official Integrated Development Environment (IDE) for Android app development
- It is based on the IntelliJ IDEA, a Java integrated development environment for software, and incorporates its code editing and developer tools.
- The first stable build was released in December 2014, starting from version 1.0.
- It replaced Eclipse Android Development Tools (ADT) as the primary IDE for Android application development.
- Since May 7, 2019, Kotlin is Google's preferred language for Android app development

INSTALLING ANDROID STUDIO

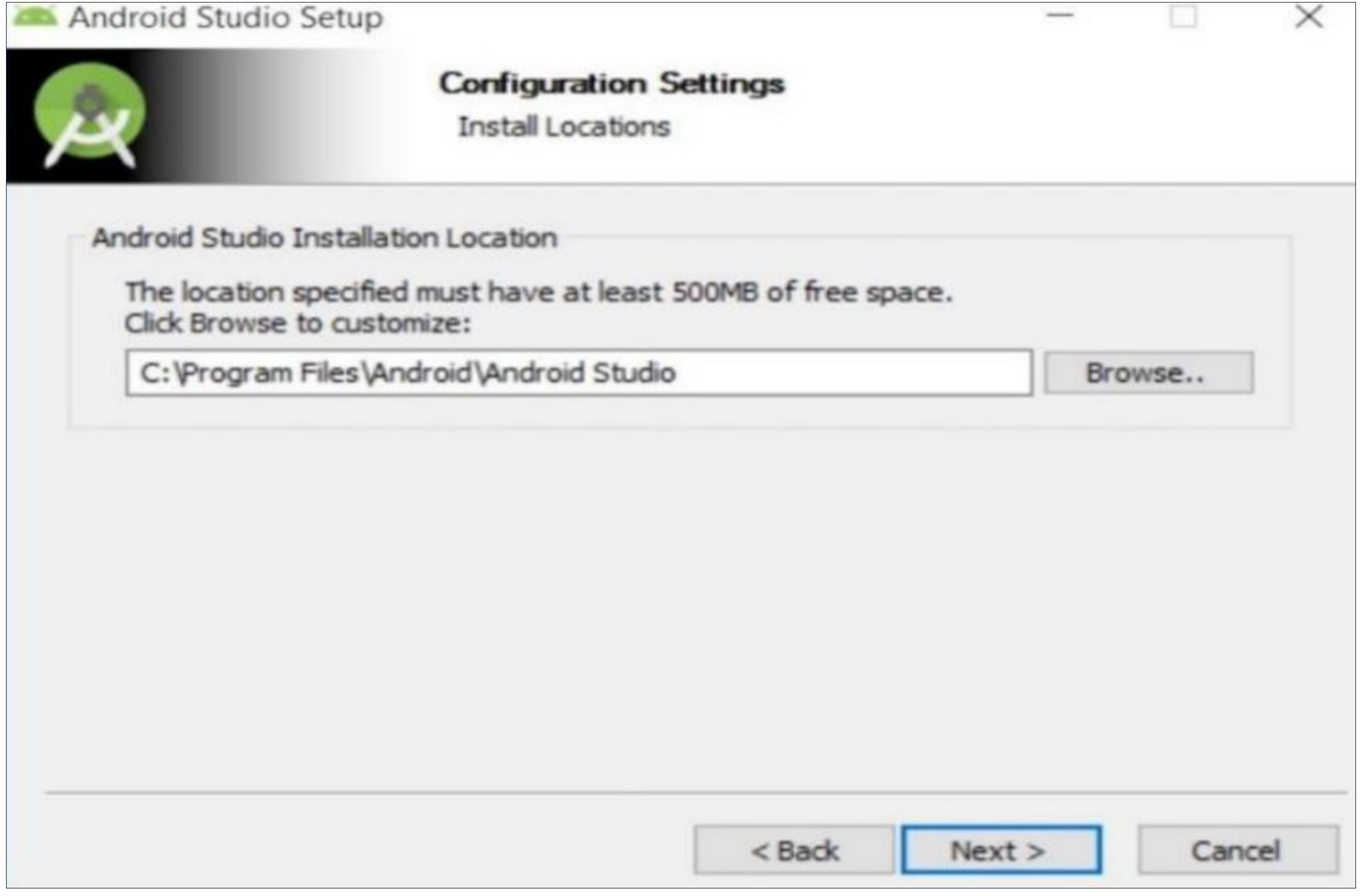
- Launch android studio.exe file
  - Before launching machine should require installed Java JDK
  - Studio setup will detect the jdk path automatically. if not specify the jdk path



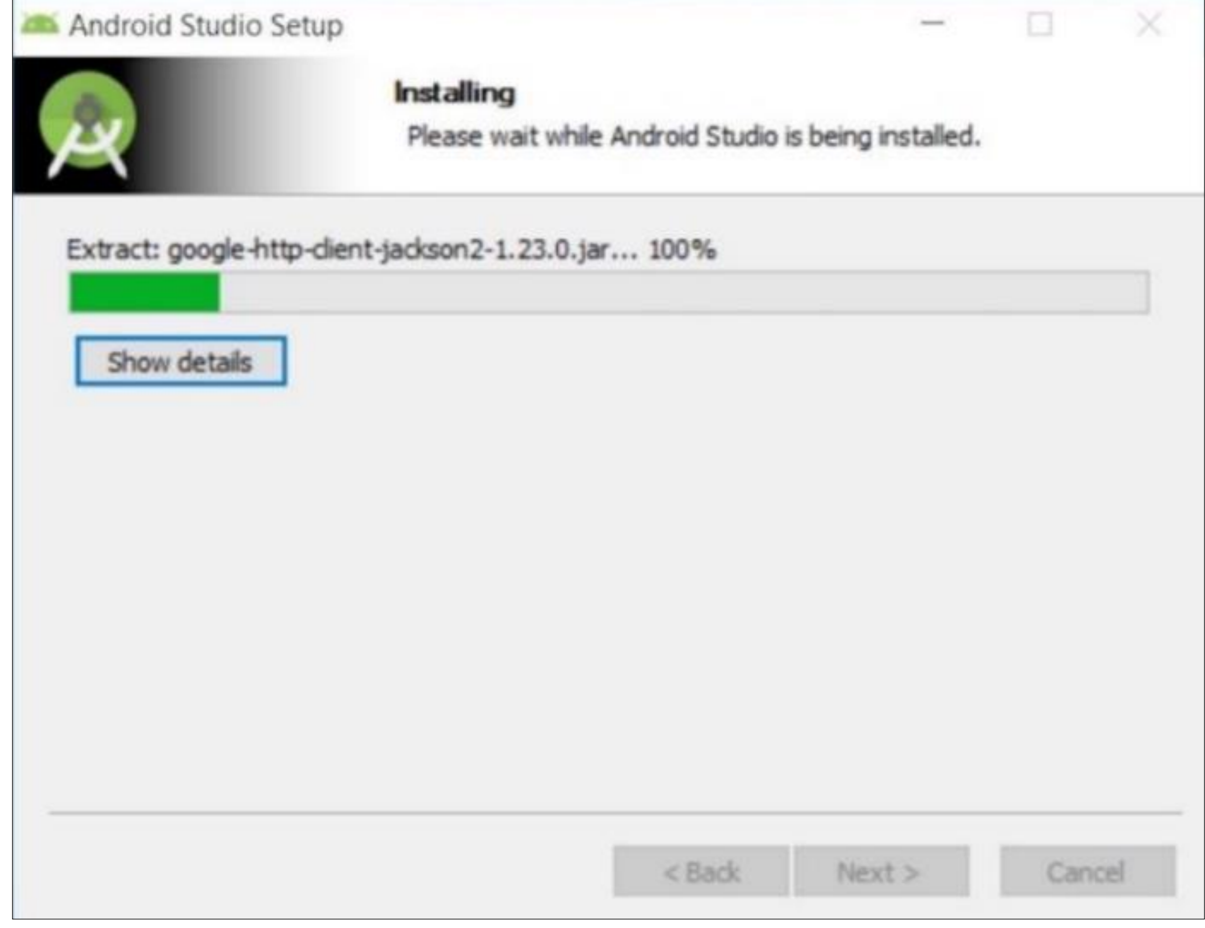
- Check the components required to create applications(like android studio, android SDK, android virtual device and performance)

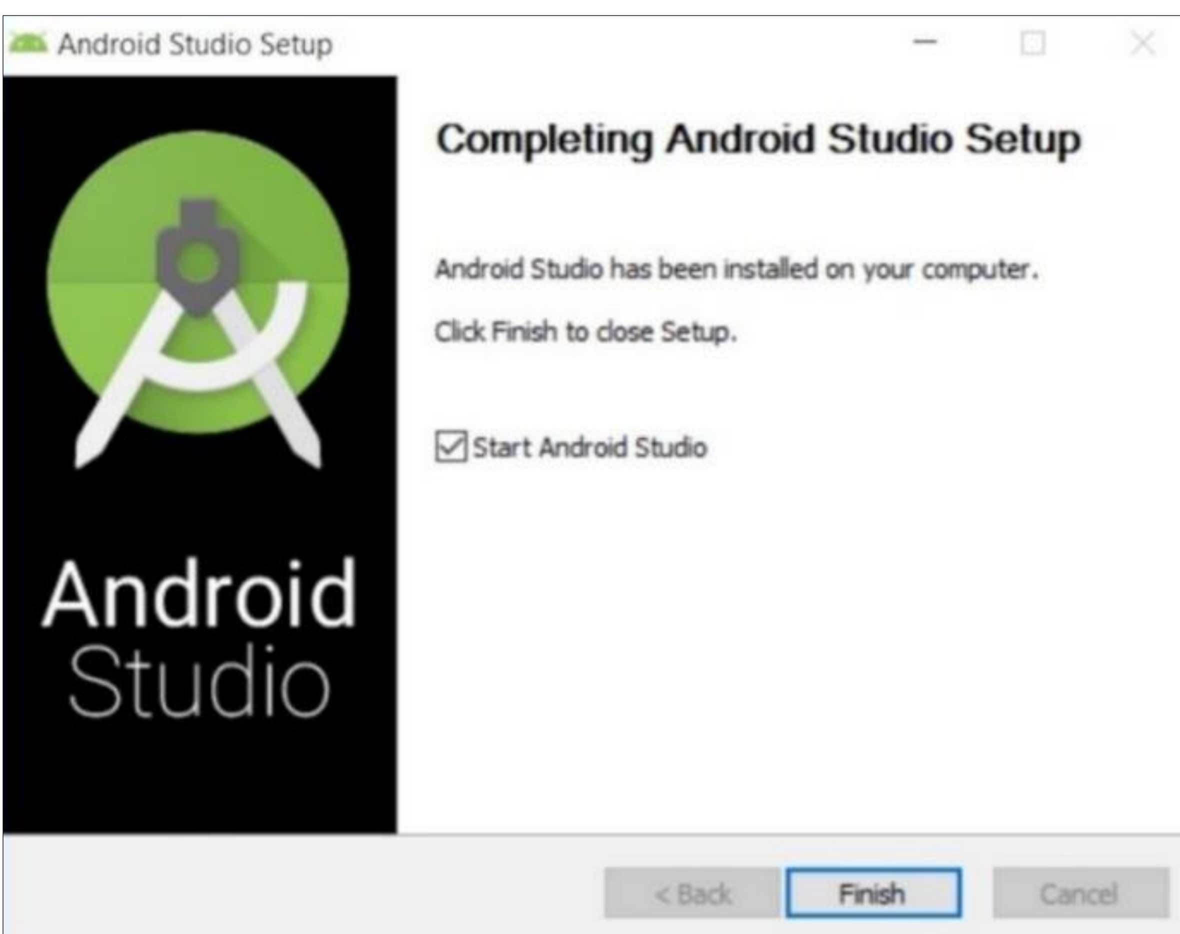


- Specify the location of local machine path for android studio and android sdk
    - Minimum 500 mb of free space for studio and 1.5 GB for SDK



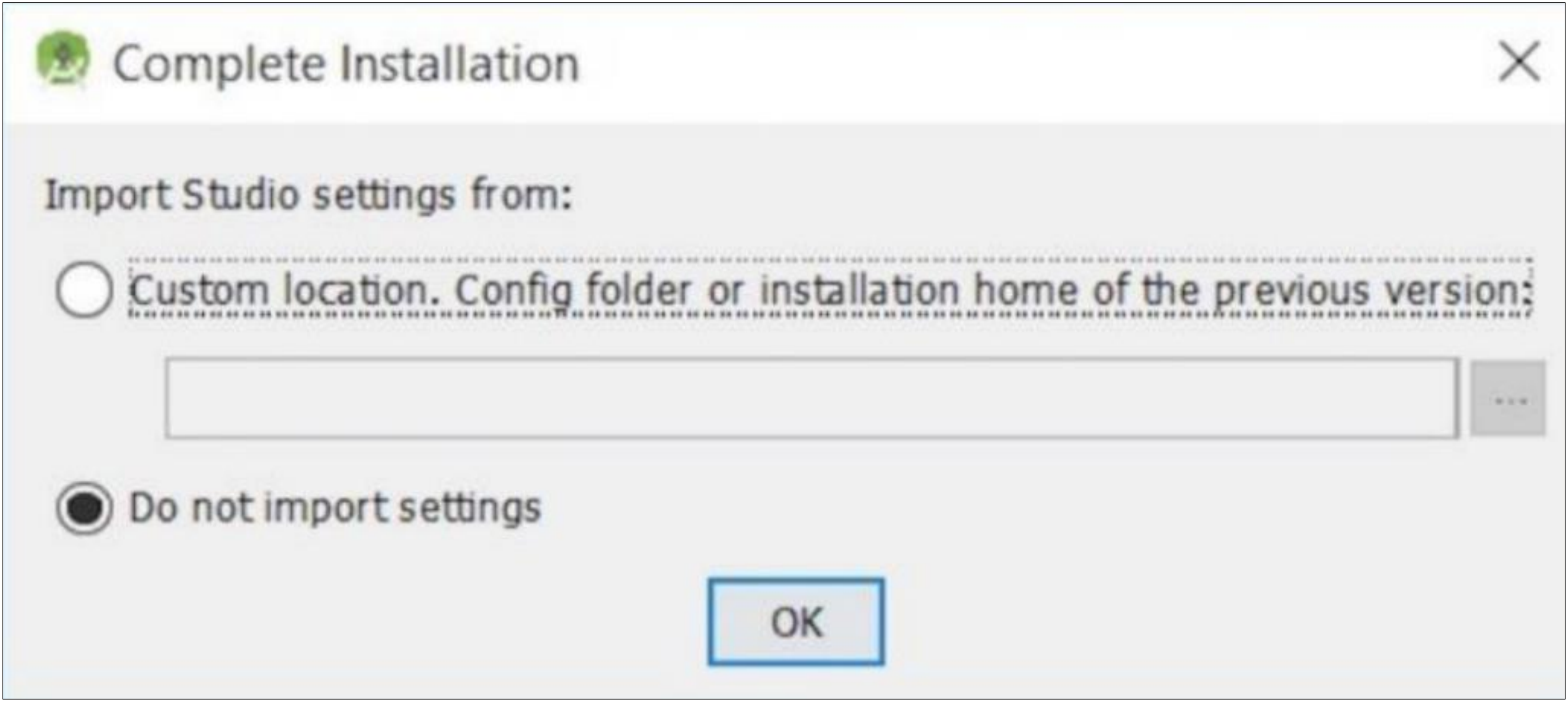
- Specify ram space for android emulator
  - By default it takes 512 MB
- At final stage it extracts SDK packages into local machine
- After completion get the finish button



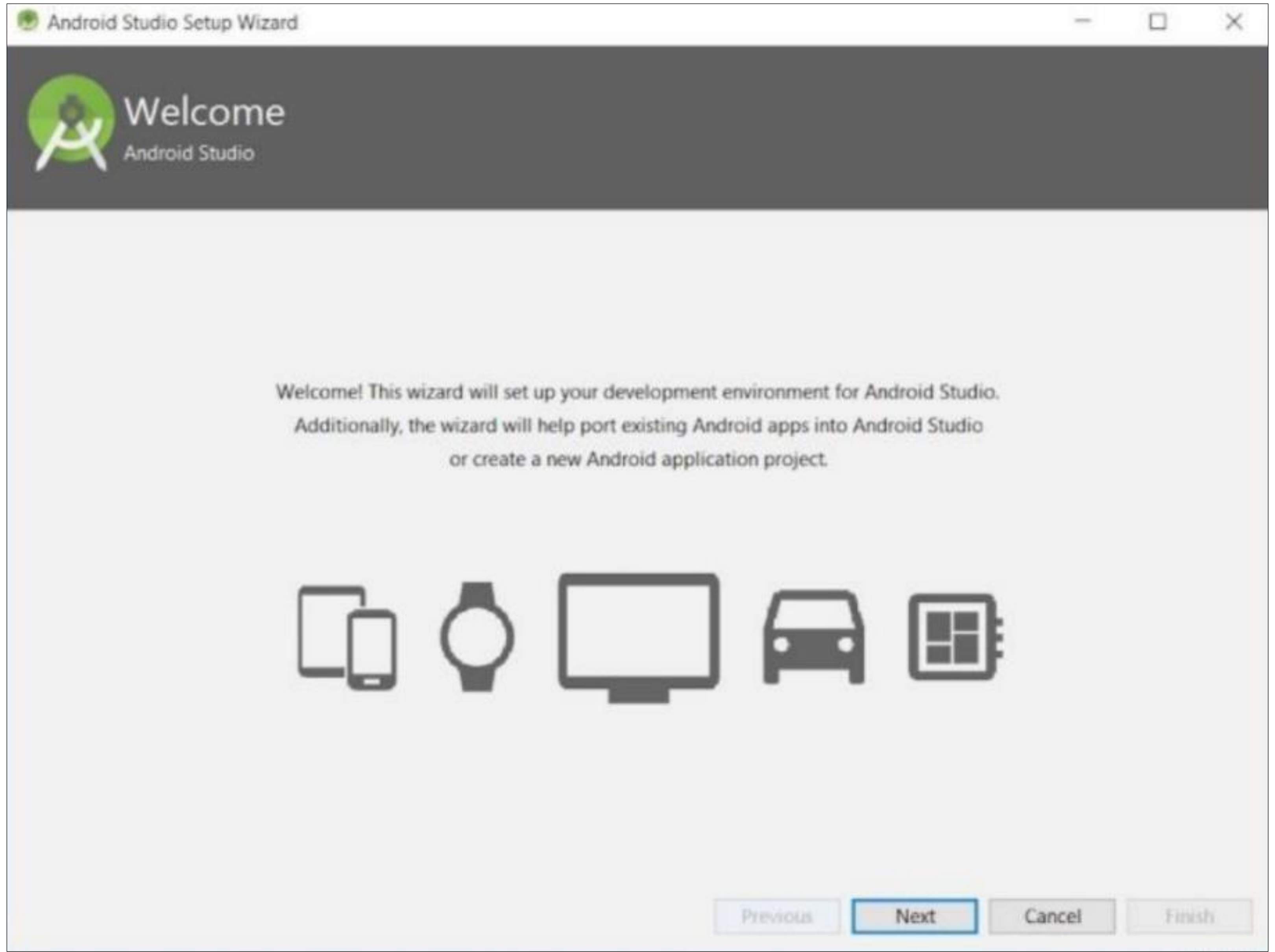


RUNNING ANDROID STUDIO

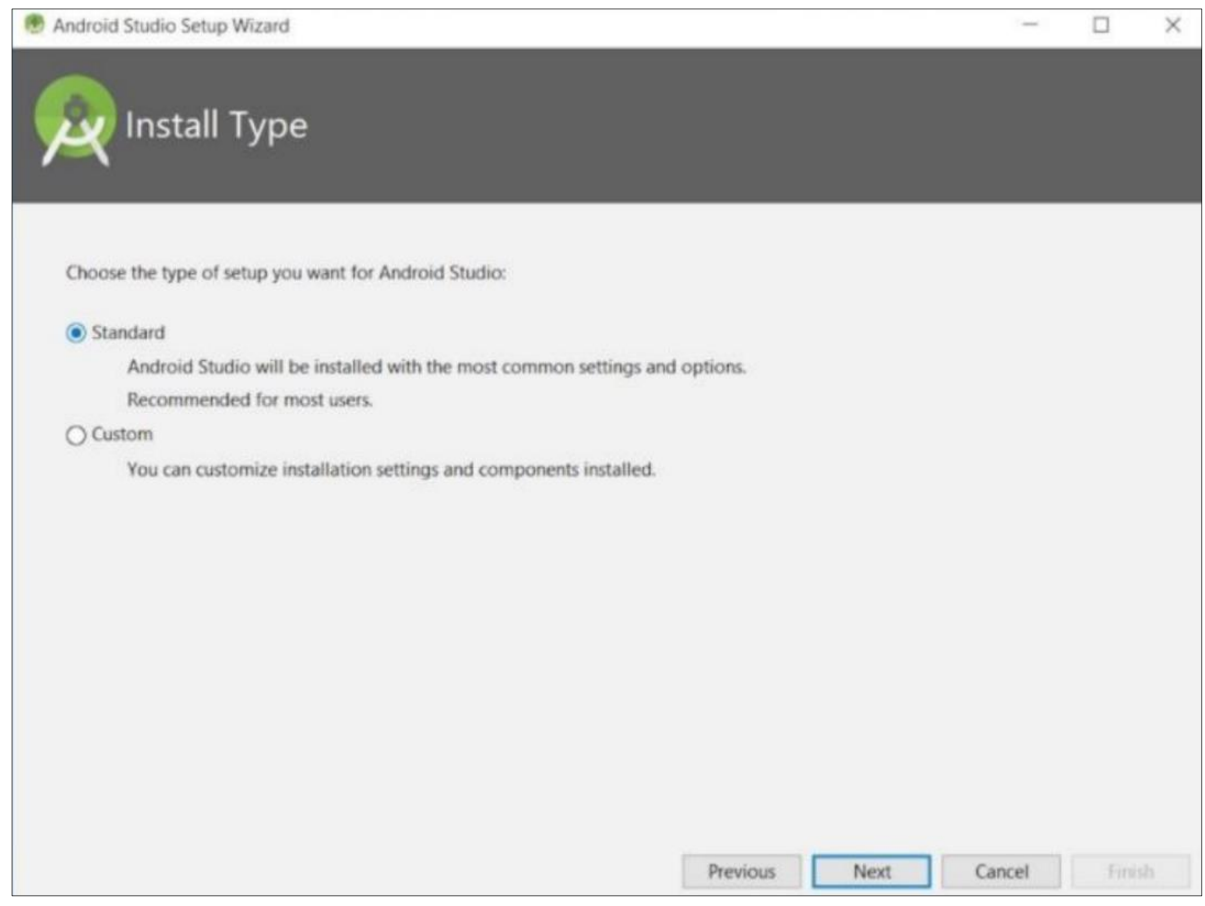
- The first time Android Studio runs, it presents a **Complete Installation** dialog box that offers the option of importing settings from a previous installation.



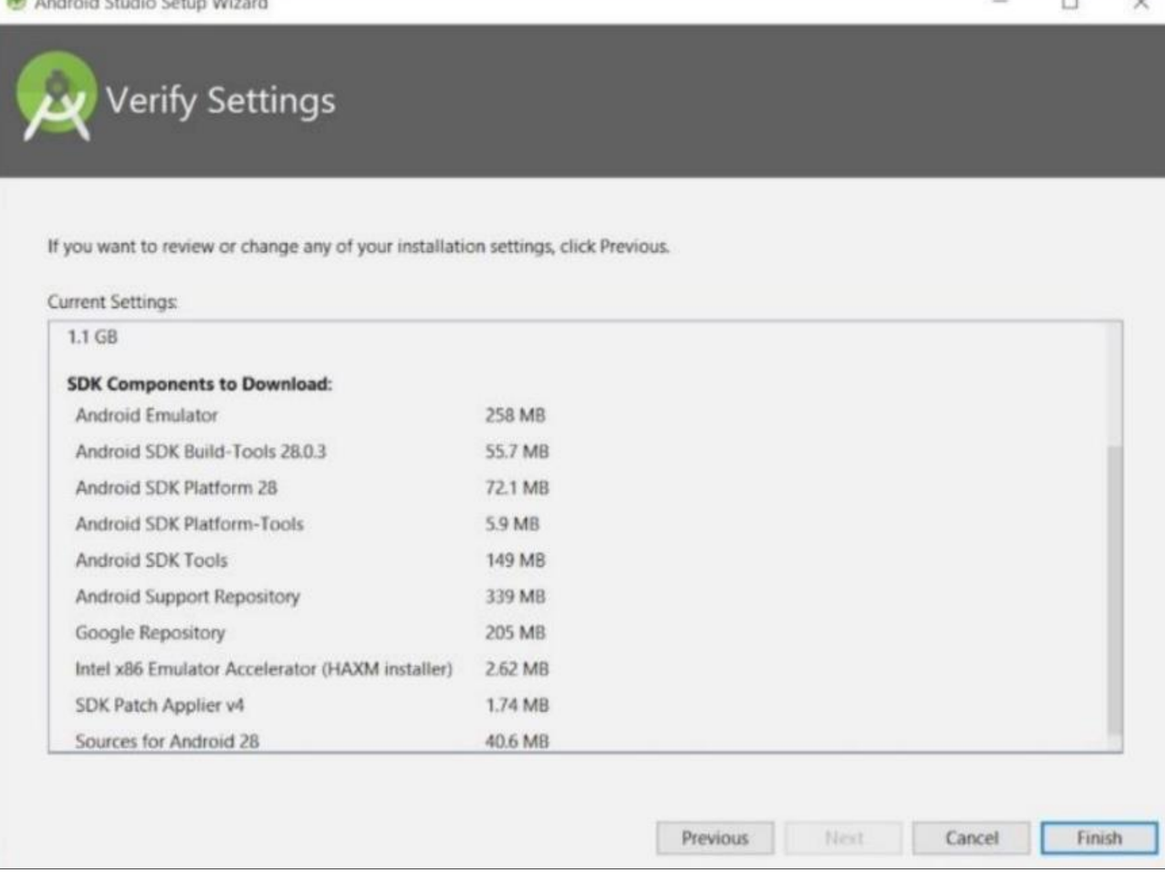
ANDROID STUDIO SETUP WIZARD



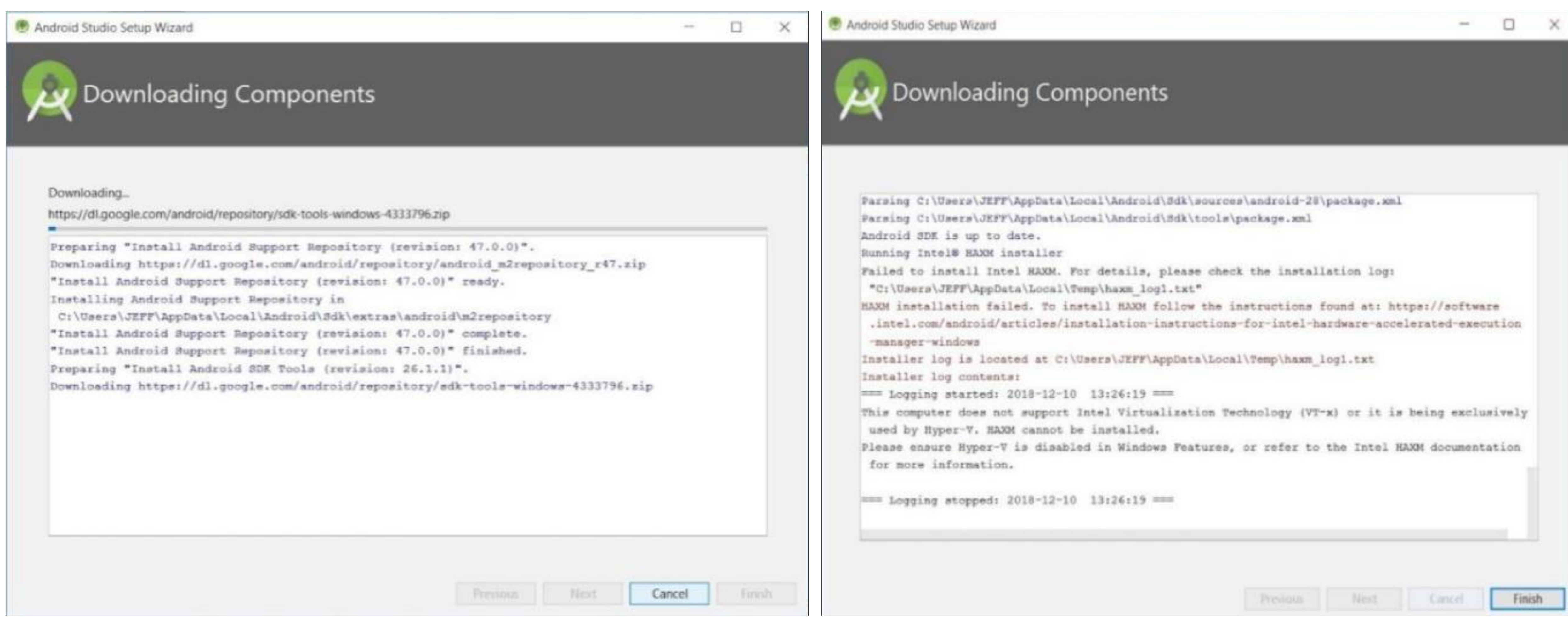
SELECT AN INSTALLATION TYPE



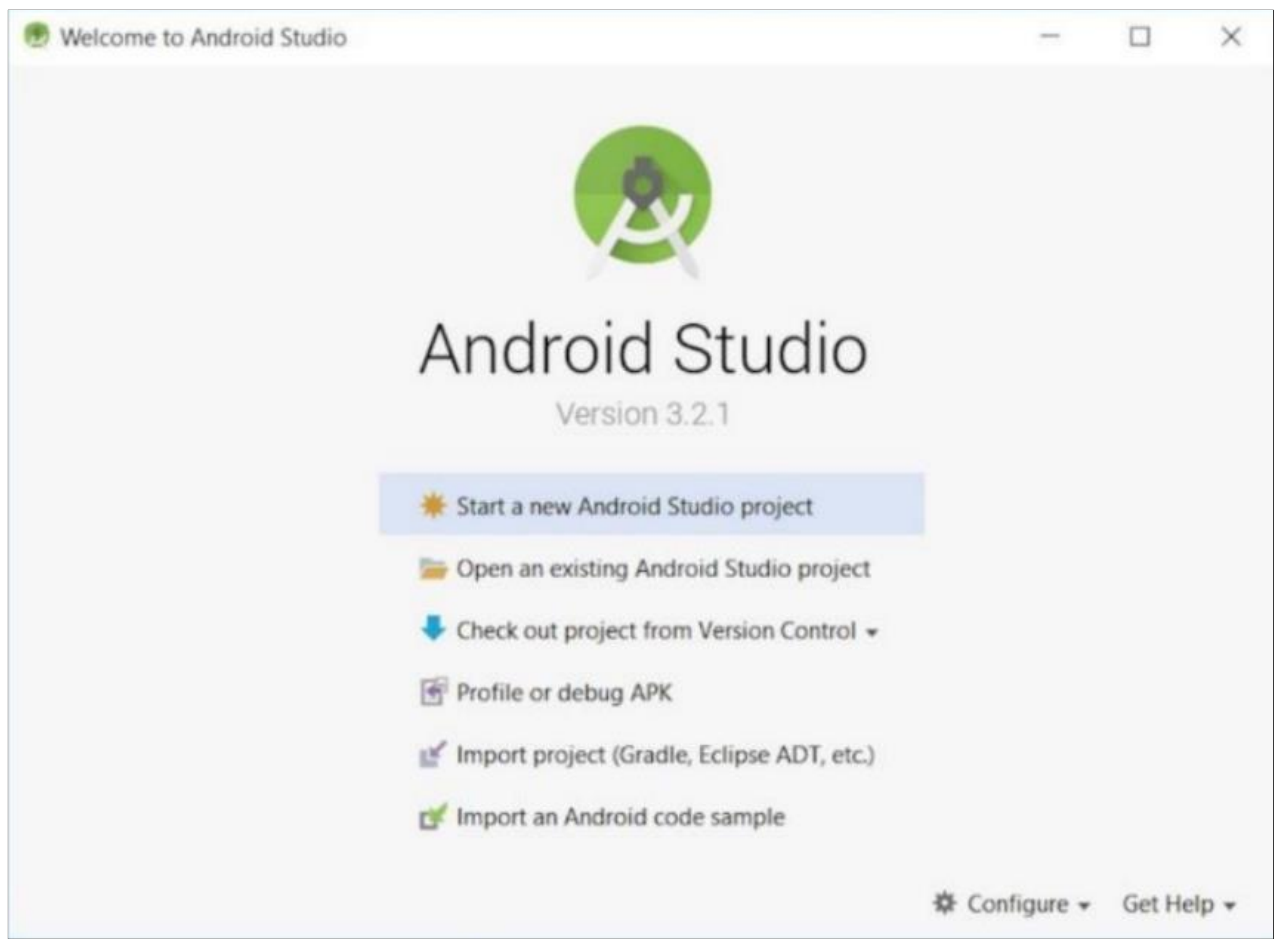
VERIFY SETTINGS



DOWNLOADING SDK COMPONENTS



WELCOME TO ANDROID STUDIO



- Android Studio 4.1 (August 2020) is the current major version

android studio

Powered by the IntelliJ® Platform

DEVELOPING END USER APPLICATION USING ANDROID SDK

1. Android Emulator

- The Android SDK includes a virtual mobile device emulator that runs on the computer.
- The emulator lets you prototype, develop and test Android applications without using a physical device.
- Emulator can used to run, debug, and test applications. You will not even need the real device for 90 percent of your application development
- The full-featured Android emulator mimics most of the device features, some limitations regarding USB connections, camera and video capture, head- phones, battery simulation, and Bluetooth.
- The Android emulator accomplishes its work through an open source “processor emulator” technology called QEMU developed by Fabrice Bellard.

DEVELOPING END USER APPLICATION USING ANDROID SDK

2. The Android UI

- Android uses a UI framework that resembles other desktop-based, full-featured UI frame-works, but it's more modern and more asynchronous in nature.
- Android is almost a fourth-generation UI framework
    - traditional C-based Microsoft Windows API -the first generation
    - C++-based MFC (Microsoft Foundation Classes) – second generation.
    - The Java-based Swing UI framework -third generation,
    - The Android UI, JavaFX, Microsoft Silverlight, and Mozilla XML User Interface Language (XUL) - fourth-generation UI framework
- UI is declarative and independently themed.

DEVELOPING END USER APPLICATION USING ANDROID SDK

2. The Android UI (Continues..)

- Programming in the Android UI involves declaring the interface in XML files.
- load these XML view definitions as windows in your UI application.
- Even menus in application are loaded from XML files
- Screens or windows in Android are referred to as activities,
- 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 it can be combined to form composite views called view groups.

DEVELOPING END USER APPLICATION USING ANDROID SDK

3. The Android Foundational Components

- The core building blocks or fundamental components of android are activities, views, intents, services, content providers, fragments and AndroidManifest.xml.
- **Activity:** An activity is a class that represents a single screen. It is like a Frame in AWT.
- **View:** A view is the UI element such as button, label, text field etc. Anything that you see is a view.
- **Intent:** Intent is used to invoke components. It is mainly used to:
    - Start the service
    - Launch an activity
    - Display a web page
    - Display a list of contacts
    - Broadcast a message
    - Dial a phone call etc.

DEVELOPING END USER APPLICATION USING ANDROID SDK

3. The Android Foundational Components (Continues..)

- **Service**
  - Service is a background process that can run for a long time.
  - There are two types of services local and remote. Local service is accessed from within the application whereas remote service is accessed remotely from other applications running on the same device.
- **Content Provider**
  - Content Providers are used to share data between the applications.
- **Fragment**
  - Fragments are like parts of activity.
  - An activity can display one or more fragments on the screen at the same time.
- **AndroidManifest.xml**
  - It contains information about activities, content providers, permissions etc.

FUNDAMENTAL COMPONENTS

- An android component is simply a piece of code that has a well defined life cycle
- Fundamental components or application components are the essential building blocks of an android application
  - **views**
  - **Activities**
  - **fragments**
  - **intents**
  - **content providers**
  - **Services**
  - **AndroidManifest.xml file**
  - **Broadcast receiver**

FUNDAMENTAL COMPONENTS

Views

- 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 means there's usually a hierarchy of views in the UI.
- In the end, everything you see is a view.

Activity

- An activity is a UI concept that usually represents a single screen in your application.
- It generally contains one or more views.
- Activity performs actions on the screen
- viewing data, creating data, or editing data

FUNDAMENTAL COMPONENTS

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 contain just one fragment

Intent

- Intent is used to invoke components
- “intention” to do some work
- Intents can be used to perform the following tasks:
    - Broadcast a message.
    - Start a service.

FUNDAMENTAL COMPONENTS

Intent (Continues…)

- 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 application—they're also used by the system to notify application of specific events (such as the arrival of a text message).

▪ Intents can be explicit or implicit.

▪ If you simply say that you want to display a URL, the system decides what component will fulfill the intention.

▪ You can also provide specific information about what should handle the intention.

FUNDAMENTAL COMPONENTS

Content Provider

- Content Providers are used to share data between the applications.
- Android defines a standard mechanism for applications to share data without exposing the underlying storage, structure, and implementation through content providers

Service

- Service is a background process that can run for a long time.
- Android service is a component that is used to perform operations on the background such as playing music, handle network transactions, interacting content providers etc.
- It doesn't has any UI
- two types of services:
- **Local services** are components that are only accessible by the application that is hosting the service.
- **Remote services** are services that are meant to be accessed remotely by other applications running on the device

FUNDAMENTAL COMPONENTS

Broadcast receiver

- Broadcast receiver is an Android component which allows you to send or receive Android system or application events. .
- All registered receivers for an event are notified by the Android runtime once this event happens.
- For example, applications can register for various system events like boot complete or battery low, and Android system sends broadcast when specific event occur.
- Applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

FUNDAMENTAL COMPONENTS

AndroidManifest.xml

- Defines the contents and behavior of your application
- It contains information about activities, content providers, permissions etc.
- For example, it lists your application's activities and services, along with the permissions and features the application needs to run.

ANDROID VIRTUAL DEVICES (AVD)

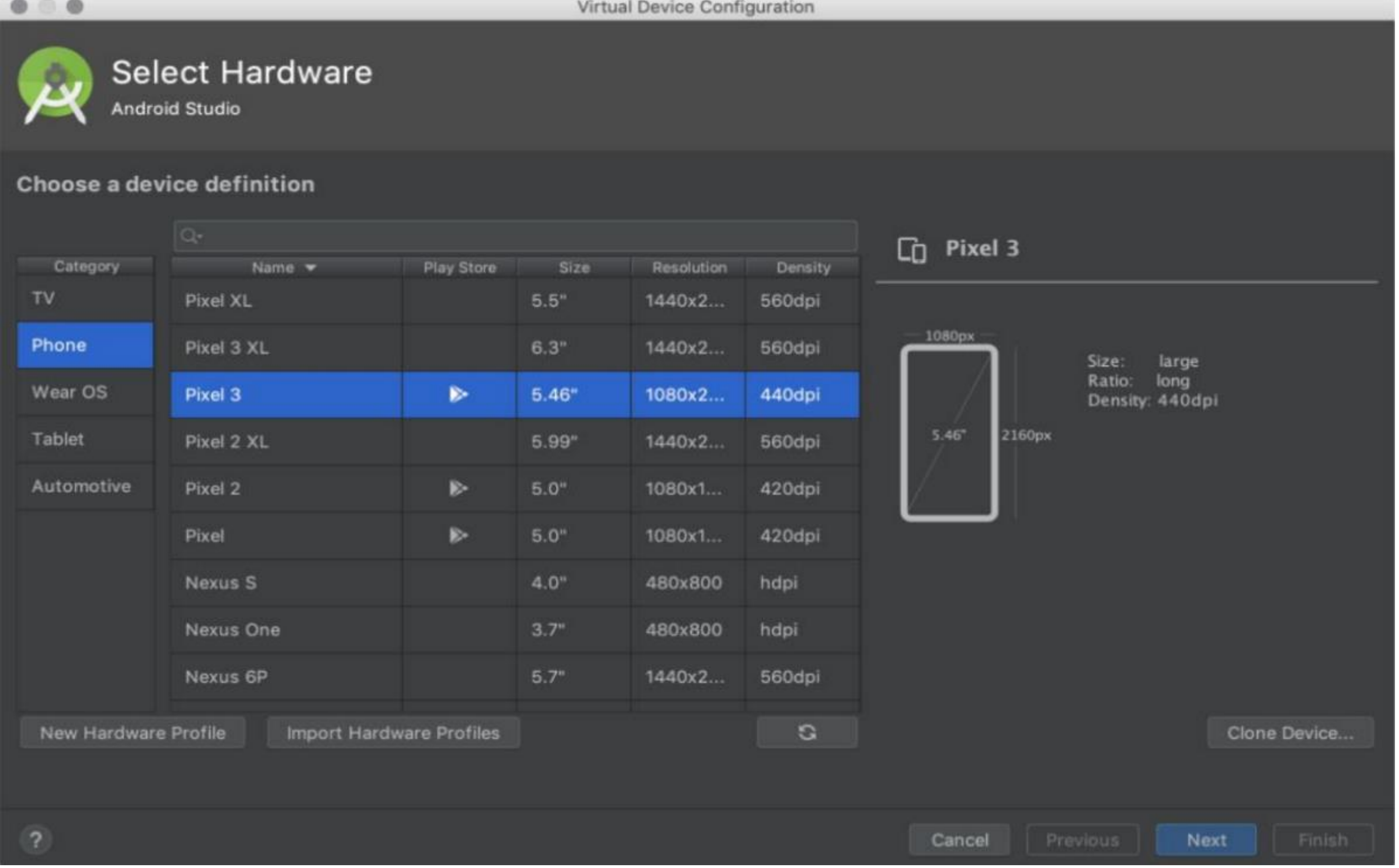
- AVD is a device configuration that is run with the android emulator
- It is an emulator configuration that allows developers to test the application by simulating the real device capabilities.
- It works with the emulator to provide a virtual device-specific environment in which to install and run Android apps.
- In computing, an emulator is hardware or software that enables one computer system(host) to behave like another computing system(guest)
- Emulator enables the host system to run software or use peripheral devices designed for the guest system

CREATING AVD

- The AVD Manager is a tool you can use to create update, delete, repair, and manage Android virtual devices (AVDs), which define device configurations for the Android Emulator.
- To open the AVD Manager, do one of the following:
  - Select Tools →AVD Manager.
  - Click AVD Manager icon in the toolbar.
  - In the your virtual devices screen click create Virtual device

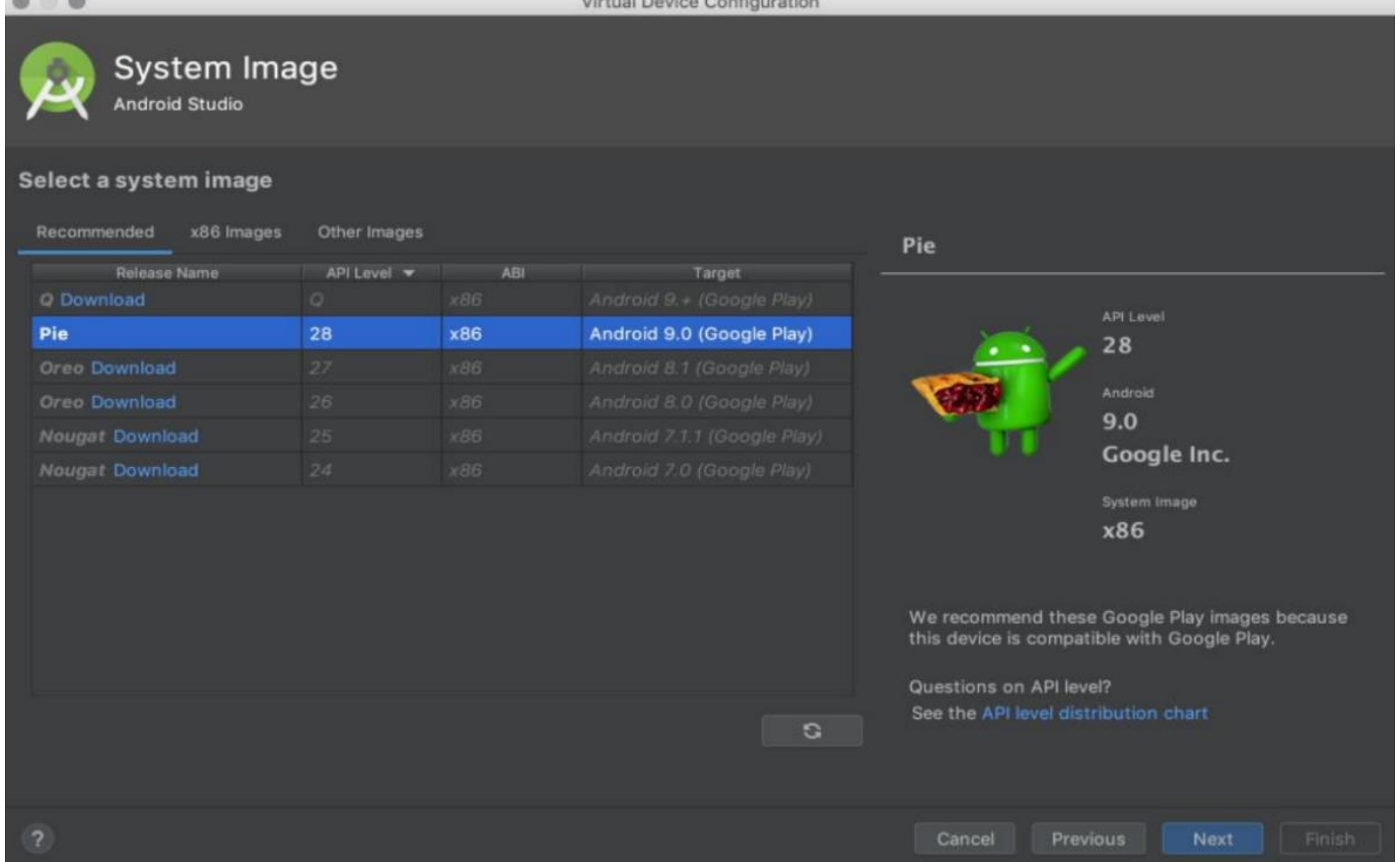
CREATING AVD

- In the select hardware screen select a hardware profile, and then click Next.
- If you don't see the hardware profile you want, you can create or import a hardware profile.



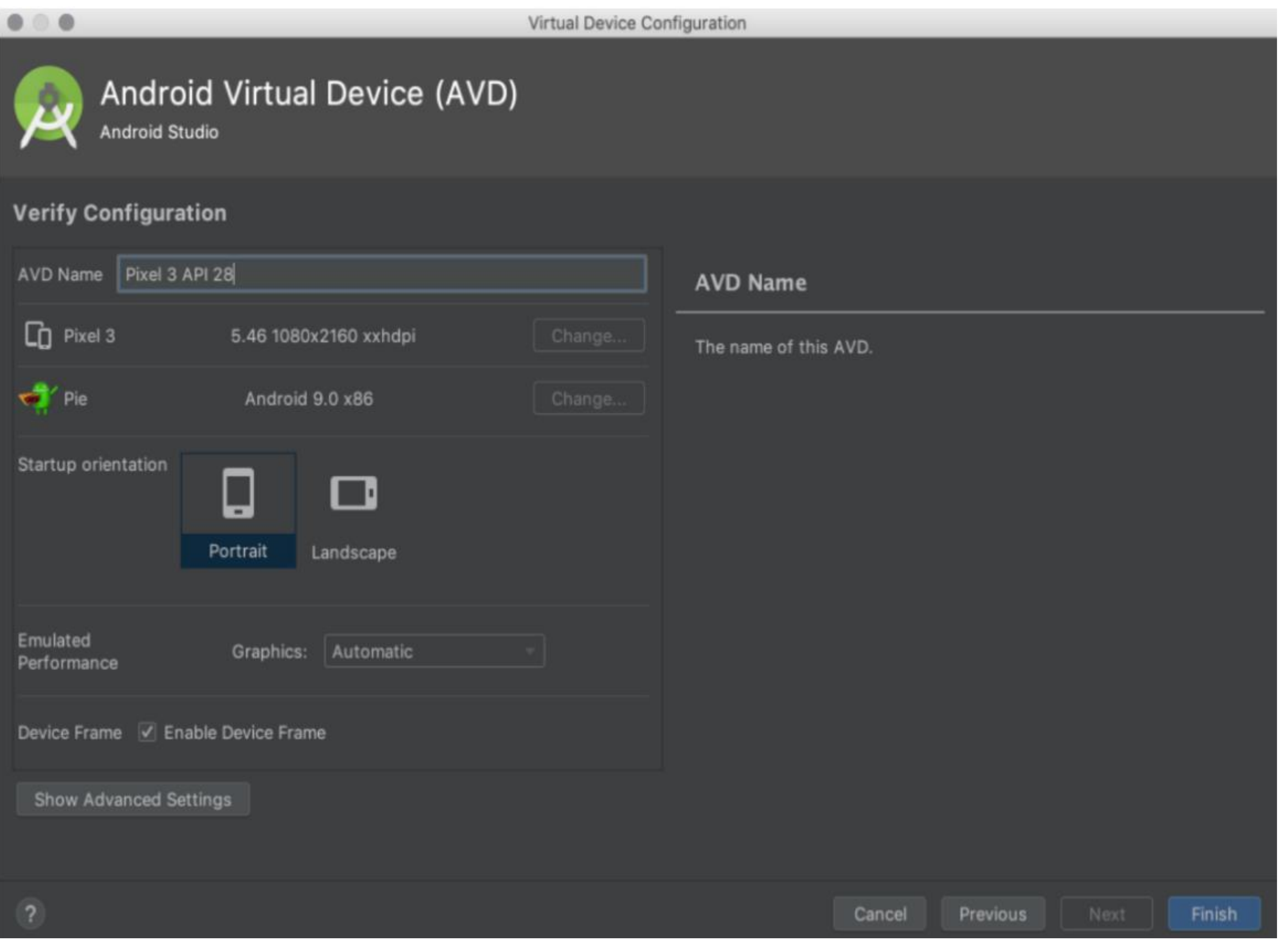
CREATING AVD

- In the System Image screen select the system image for a particular API level, and then click Next .
- x86 images run the fastest in the emulator.



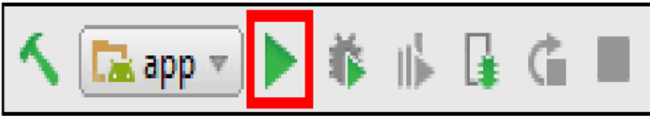
CREATING AVD

- Verify the configuration settings and then finish

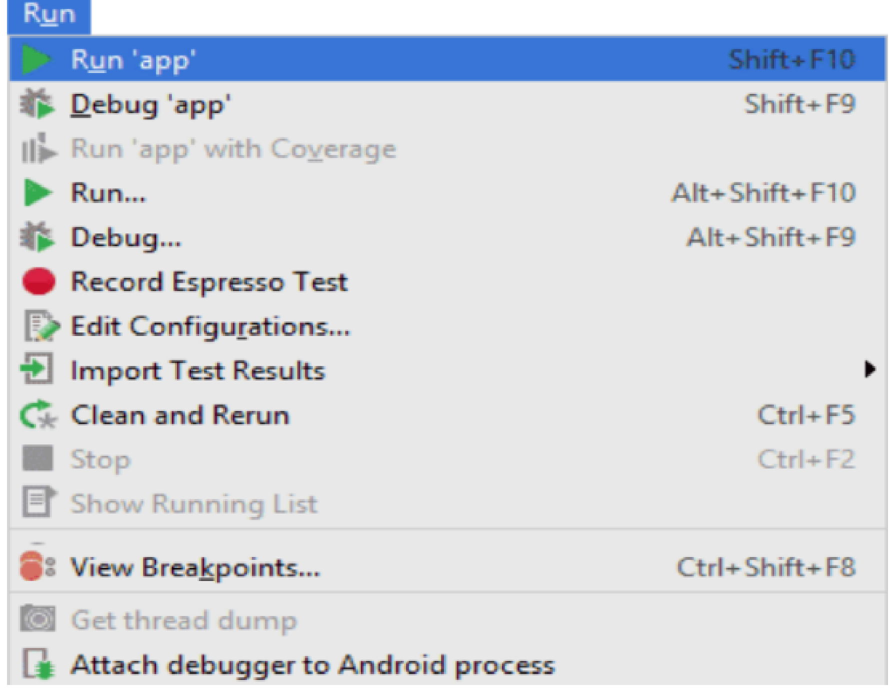


RUN THE APP FROM ANDROID STUDIO

- To run an app in Android Studio, you can click the green arrow in the menu bars at the top:



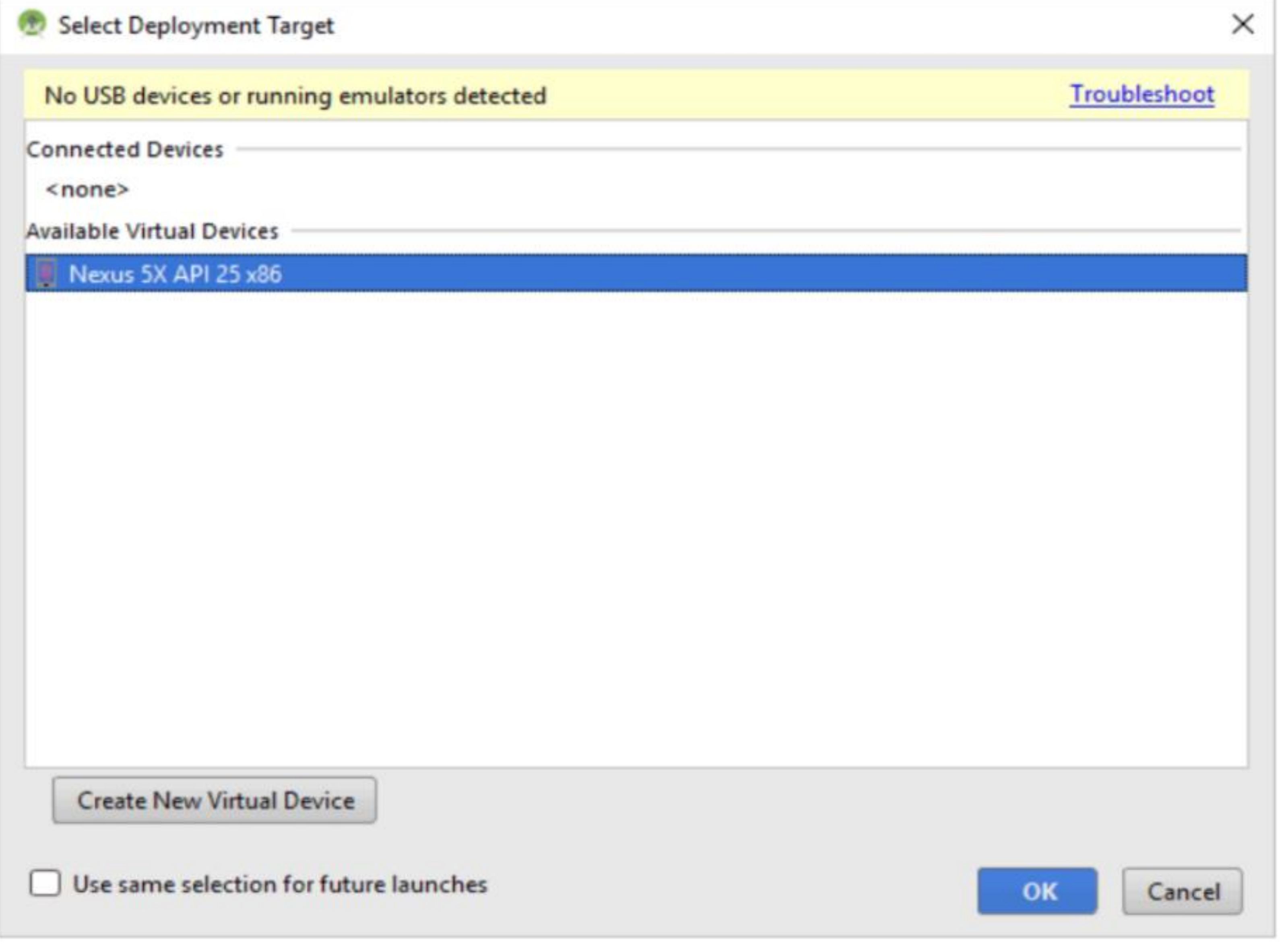
- Or you can click the Run menu, then select Run



- In the select deployment target window, select emulator and click OK

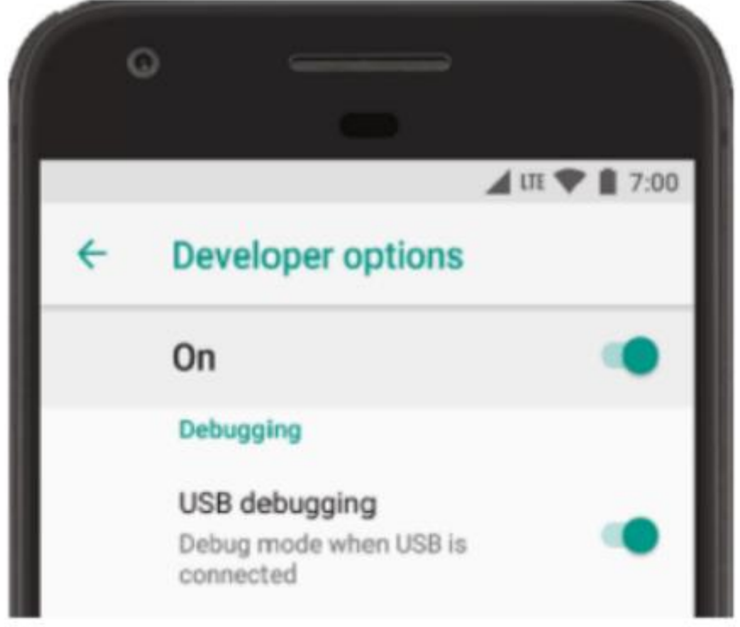
RUN THE APP FROM ANDROID STUDIO

- In the select deployment target window, select emulator and click OK



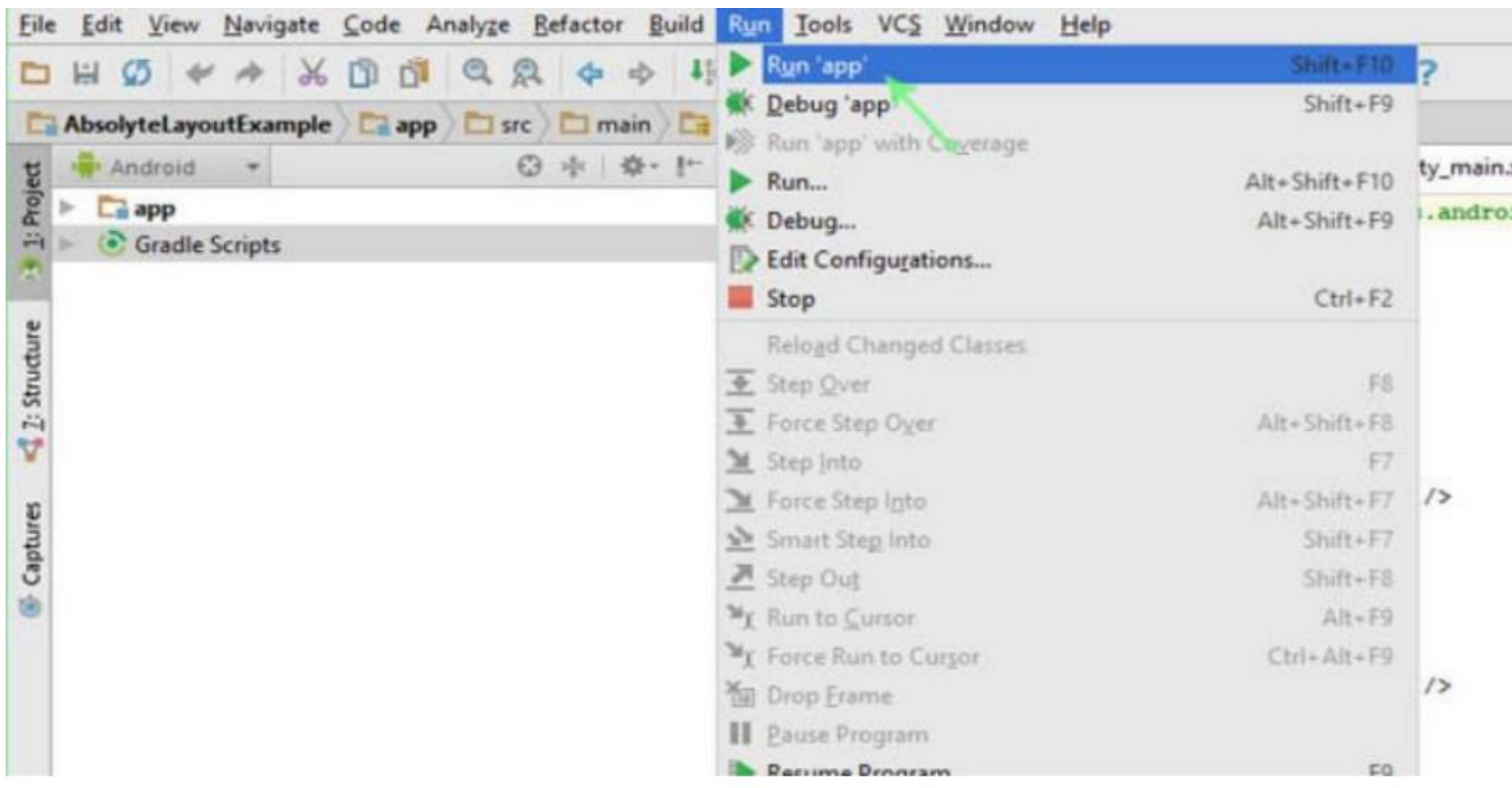
RUNNING ON REAL DEVICES

- To run android on real device
  - Enable USB debugging from settings → developer options
  - Connect device with development machine with USB and after that allow USB debugging message shown on your device and press OK.



RUNNING ON REAL DEVICES

- After that Go to the menu bar and Run app



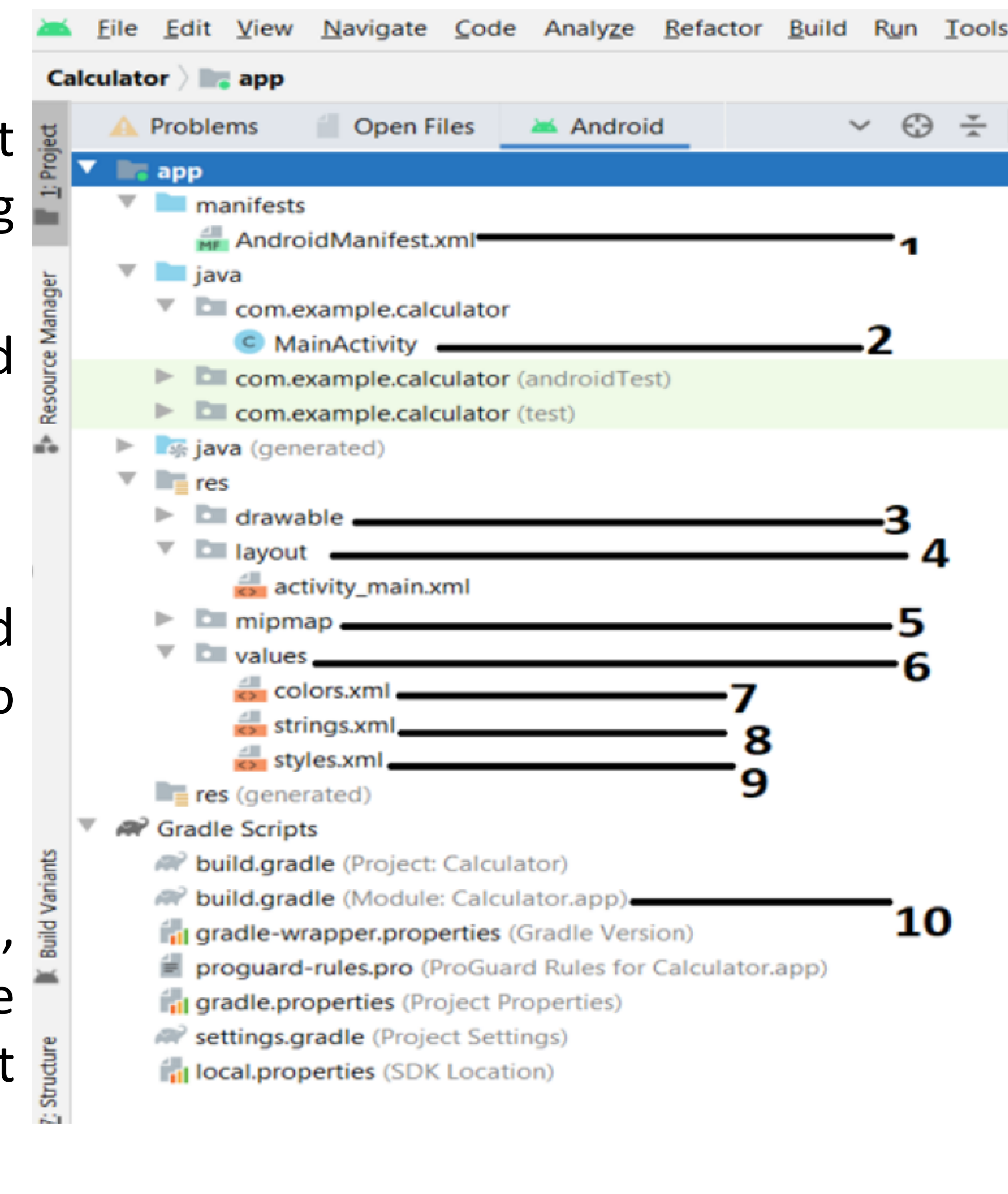
RUNNING ON REAL DEVICES

- If real device is connected to your system then it will show Online.

- Now click on your Mobile phone device and click OK
- Studio installs the app on device and starts it

ANDROID APPLICATION FILE STRUCTURE

- Android application contains different components such as java source code, string resources, images, manifest file, apk file etc.
- Let's understand the project structure of android application.



❑ Modules

- Module is a collection of source files and build settings that allow to divide the project into discrete units of functionality

1. Android app module

- Provides a container for app's source code, resource files, and app level settings like module level build file and Android Manifest file
- Major sub divisions are: **Manifest, java, Res**.

1. AndroidManifest.xml

- Every project in Android includes a manifest file, which is AndroidManifest.xml, stored in the root directory of its project hierarchy
- It defines the structure and metadata of our application, its components, and its requirements.
- This file includes nodes for each of the Activities, Services, Content Providers and Broadcast Receiver that make the application and using Intent Filters and Permissions, determines how they co-ordinate with each other and other applications.

2. Java

- The Java folder contains the Java source code files.
- These files are used as a controller for controlled UI (Layout file).
- It gets the data from the Layout file and after processing that data output will be shown in the UI layout.
- It works on the backend of an Android application.

3. Res

- Resource folder is the most important folder because it contains all the non-code sources like images, XML layouts, UI strings for our android application
- Major subdivisions are:
    - **Drawable**: A Drawable folder contains resource type file (something that can be drawn). Drawable may take a variety of file like Bitmap (PNG, JPEG), Nine Patch, Vector (XML), Shape, Layers, States, Levels, and Scale.
    - **Layout**: Defines the visual structure for a user interface, such as the UI for an Android application. This folder stores Layout files that are written in XML language.
    - **Mipmaps**: contains launcher.xml files to define icons which are used to show on the home screen.
    - **Values**: Values folder contains a number of XML files like strings, dimens, colors and styles definitions.
        - **colors.xml**: contains color resources of the Android application. Different color values are identified by a unique name that can be used in the Android application program.

- **strings.xml**: contains string resources of the Android application. The different string value is identified by a unique name that can be used in the Android application program. This file also stores string array by using XML language.
- **styles.xml**: The styles.xml file contains resources of the theme style in the Android application. This file is written in XML language.

❑ **Gradle script**

- Gradle is a build system (open source) which is used to automate building, testing, deployment etc. "Build.gradle" are scripts where one can automate the tasks
- Every Android project needs a gradle for generating an apk from the .java and .xml files in the project.
- A gradle takes all the source files (java and XML) and apply appropriate tools, e.g., converts the java files into dex files and compresses all of them into a single file known as apk that is actually used.
- Two types of build.gradle scripts: **Top-level build.gradle** and **Module-level build.gradle**

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 contain just one fragment

INTENT

- Intent is used to invoke components
- “intention” to do some work
- Intents can be used 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 application—they're also used by the system to notify application of specific events (such as the arrival of a text message).
- Intents can be explicit or implicit.
- If you simply say that you want to display a URL, the system decides what component will fulfill the intention.
- You can also provide specific information about what should handle the intention.

CONTENT PROVIDER

- Content Providers are used to share data between the applications.
- Android defines a standard mechanism for applications to share data without exposing the underlying storage, structure, and implementation through content providers

SERVICE

- Service is a background process that can run for a long time.
- Android service is a component that is used to perform operations on the background such as playing music, handle network transactions, interacting content providers etc.
- It doesn't has any UI
- two types of services:
- **Local services** are components that are only accessible by the application that is hosting the service.
- **Remote services** are services that are meant to be accessed remotely by other applications running on the device

BROADCAST RECEIVER

- Broadcast receiver is an Android component which allows you to send or receive Android system or application events. .

- All registered receivers for an event are notified by the Android runtime once this event happens.
- For example, applications can register for various system events like boot complete or battery low, and Android system sends broadcast when specific event occur.
- Applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

AndroidManifest.xml

- Defines the contents and behavior of your application
- It contains information about activities, content providers, permissions etc.
- For example, it lists your application’s activities and services, along with the permissions and features the application needs to run.

ANDROID APPLICATION LIFE CYCLE

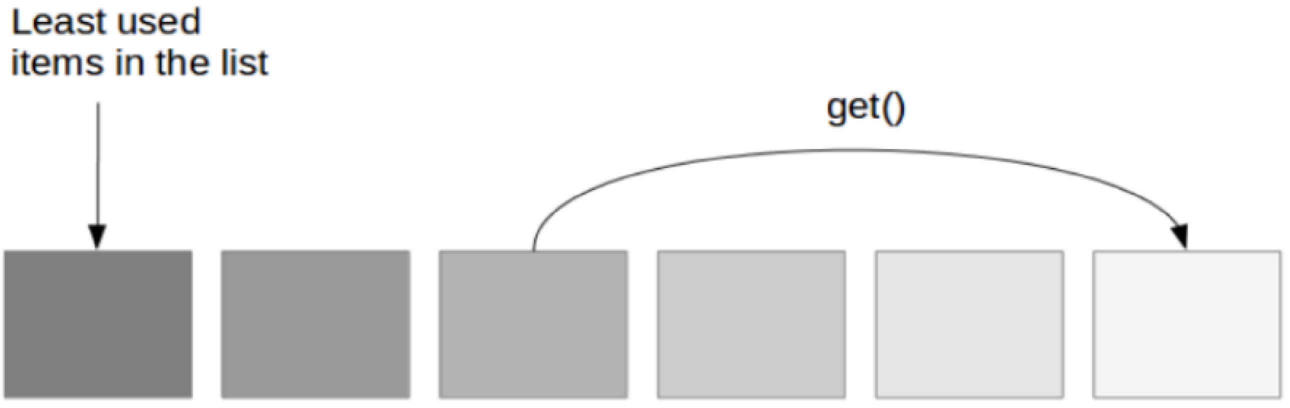
- The phases that an application goes through from start to end is called application life cycle
- An android application is a collection of activities and an activity corelates to a screen
- Each Android application runs inside its own instance of a Virtual Machine (VM)
- At any point in time several parallel VM instances could be active
- Unlike a common Windows or Unix process, an Android application does not completely control the completion of its lifecycle.
- Every android application should be prepared for untimely termination
- Occasionally hardware resources may become critically low and the OS could order early termination of any process .
- The decision considers factors such as:
  1. Number and age of the application's components currently running
  2. relative importance of those components to the user

3. how much free memory is available in the system.

- To manage limited system resources the Android system can terminate running applications.
- If the Android system needs to terminate processes it follows the following priority system.

Process status	Description	Priority
Foreground	An application in which the user is interacting with an activity, or which has an service which is bound to such an activity. Also if a service is executing one of its lifecycle methods or a broadcast receiver which runs its `onReceive()` method.	1
Visible	User is not interacting with the activity, but the activity is still (partially) visible or the application has a service which is used by a inactive but visible activity.	2
Service	Application with a running service which does not qualify for 1 or 2.	3
Background	Application with only stopped activities and without a service or executing receiver. Android keeps them in a least recent used (LRU) list and if requires terminates the one which was least used.	4
Empty	Application without any active components.	5

- All processes in the empty list are added to a least recently used list (LRU list).
- The processes which are at the beginning of this lists will be the ones killed by the out-of-memory killer.
- If an application is restarted by the user, its gets moved to the end of this queue.



STATES OF AN ACTIVITY

1. Active State

- When an Activity is in active state, it means it is active and running.
- It is visible to the user and the user is able to interact with it.
- Process state is foreground
- Android Runtime treats the Activity in this state with the highest priority and never tries to kill it.

2. Paused State

- An activity being in this state means that the user can still see the Activity in the background such as behind a transparent window or a dialog box ie. it is partially visible.
- The user cannot interact with the Activity until he/she is done with the current view.
- Process state is visible
- Android Runtime usually does not kill an Activity in this state but may do so in an extreme case of resource crunch.

3. Stopped State

- When a new Activity is started on top of the current one or when a user hits the Home key, the activity is brought to Stopped state.
- The activity in this state is invisible, but it is not destroyed.
- Process state is background
- Android Runtime may kill such an Activity in case of resource crunch.

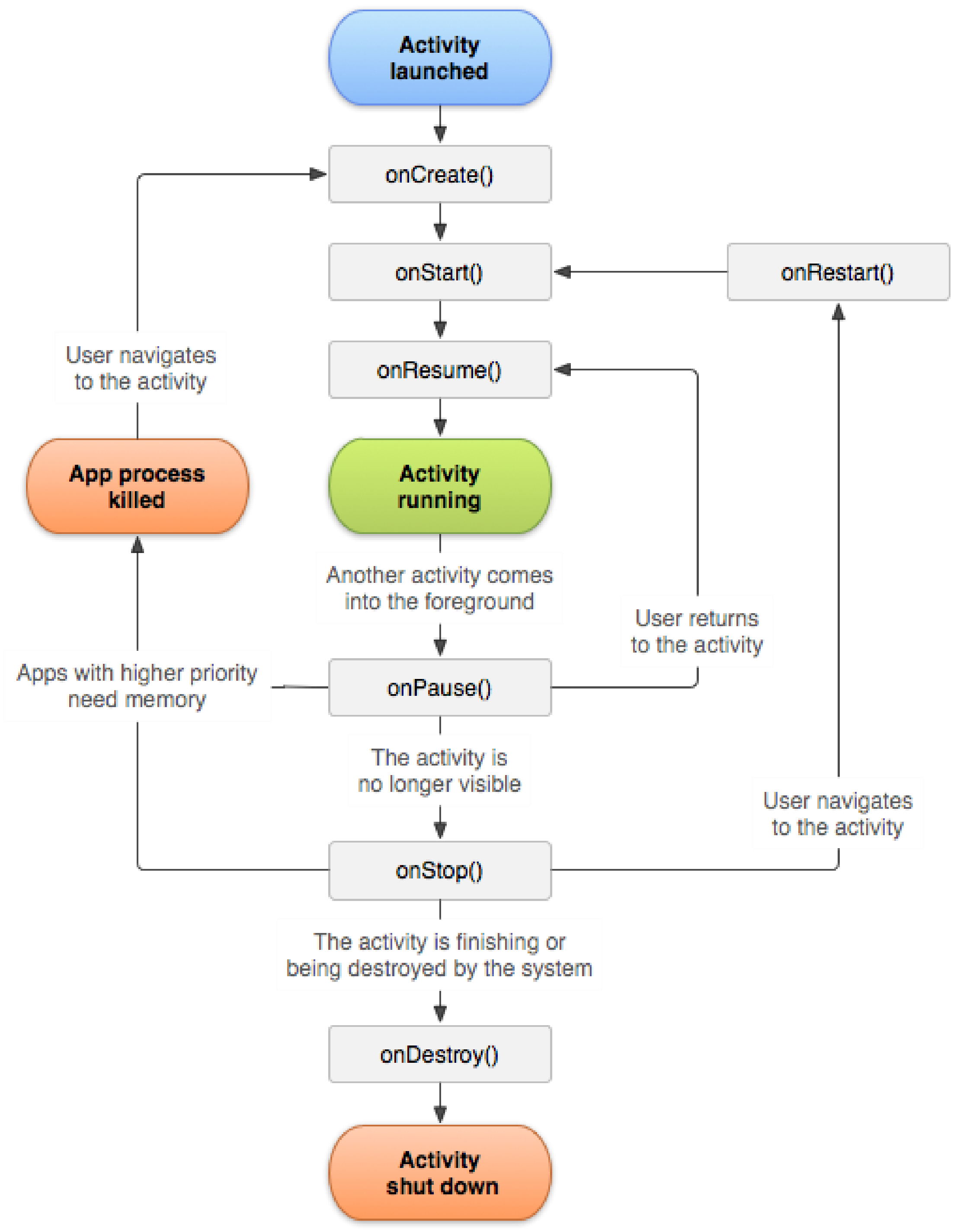
4. Destroyed State

When a user hits a Back key or Android Runtime decides to reclaim the memory allocated to an Activity i.e in the paused or stopped state, It goes into the Destroyed state.

The Activity is out of the memory and it is invisible to the user.

LIFE CYCLE METHODS

Method	What does it do?
`onCreate()`	Whenever an Activity starts running, the first method to get executed is `onCreate()`. This method is executed only once during the lifetime. If we have any instance variables in the Activity, the initialization of those variables can be done in this method. After `onCreate()` method, the `onStart()` method is executed.
`onStart()`	During the execution of `onStart()` method, the Activity is not yet rendered on screen but is about to become visible to the user. In this method, we can perform any operation related to UI components.
`onResume()`	When the Activity finally gets rendered on the screen, `onResume()` method is invoked. At this point, the Activity is in the active state and is interacting with the user.
`onPause()`	If the activity loses its focus and is only partially visible to the user, it enters the paused state. During this transition, the `onPause()` method is invoked. In the `onPause()` method, we may commit database transactions or perform light-weight processing before the Activity goes to the background.
`onStop()`	From the active state, if we hit the Home key, the Activity goes to the background and the Home Screen of the device is made visible. During this event, the Activity enters the stopped state. Both `onPause()` and `onStop()` methods are executed.
`onDestroy()`	When an activity is destroyed by a user or Android system, `onDestroy()` function is called.


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

- When the Activity comes back to focus from the paused state, onResume() is invoked.
- When we reopen any app(after pressing Home key), Activity now transits from stopped state to the active state.
- Thus, onStart() and onResume() methods are invoked.
  - Note: onCreate() method is not called, as it is executed only once during the Activity life-cycle.
- To destroy the Activity on the screen, we can hit the Back key.
- This moves the Activity into the destroyed state.
- During this event, onPause(), onStop() and onDestroy() methods are invoked.
- Whenever we change the orientation of the screen i.e from portrait to landscape or vice-versa, lifecycle methods start from the start i.e from onCreate() method.
- This is because the complete spacing and other visual appearance gets changed and adjusted.

Android Programming

Unit II

Understanding android resources - String resources, Layout resources, Resource reference syntax, Defining own resource IDs - Enumerating key android resources, string arrays, plurals, Colour resources, dimension resources, image resources, Understanding content providers - Android built in providers, exploring databases on emulator, architecture of content providers, structure of android content URIs, reading data using URIs, using android cursor, working with where clause, inserting updates and deletes, implementing content, Understanding intents basics of intents, available intents, exploring intent composition, Rules for Resolving Intents to Their Components, ACTION PICK, GET CONTENT, pending intents

ANDROID RESOURCE

- A resource in Android is a file or a value that is bound to an executable application in such a way that you can change them or provide alternatives without recompiling the application.
- Examples of resources include strings, colors, bitmaps, and layouts

STRING RESOURCES

- String resource provides text strings for applications with optional text styling and formatting
- 3 types of string resources
  - **String**–XML resource that provides a single string
  - **String Array** –XML resource that provides an array of strings
  - **Quantity strings(plurals)-**XML resource that carries different strings for pluralization

STRING

- A single string that can be referenced from the application or from other resource file
- Located in res/values/strings.xml
- getString() method is used to retrieve strings
- Syntax of string resource:

```
<?xml version=“1.0” encoding=“utf-8”?>
<resources>
<string name=“string_name”>text_string</string>
</resources>
```

- Example:

```
<?xml version=“1.0” encoding=“utf-8”?>
<resources>
<string name=“hello”>hello world</string>
</resources>
```

- Retrieves a string:

```
String name=getString(R.string.hello);
```

STRING ARRAY

- String arrays are used to describe the array of strings that can be referenced from the application
- getStringArray() method is used to retrieve strings
- Syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string-array name="string_array_name">
<item>text_string</tem>
</string-array>
</resources>
```

- Example:

```
<resources >
    <string-array name="test_array">
    <item>one</item>
    <item>two</item>
    <item>three</item>
    </string-array>
</resources>
```

- retrieve this array in the Java code as shown

```
//Get access to Resources object
Resources res = getResources();
String strings[] = res.getStringArray(R.array.test_array);
//Print strings
for (String s: strings)
{     Log.d("example", s); }
```

PLURALS

- Also known as quantity strings
- The resource plurals is a set of strings.
- These strings are various ways of expressing a numerical quantity, such as how many eggs are in a nest
    - There is 1 egg.
    - There are 2 eggs.
    - There are 0 eggs.
    - There are 100 eggs.

- Full set supported by android is zero, one, two, few, manyand other
- getQuantityString() method is used to retrieve strings
- Syntax:

```
<resources...>
<plurals name="plural_name">
<item
quantity=["zero"|"one"|"two"|"few"|"many"|"other"]>text_string</item>
</plurals> </resources>
```

- Example:

```
<resources...>
<plurals name="EggsInANestText">
<item quantity="one">There is 1 egg</item>
<item quantity="other">There are %d eggs</item> </plurals> </resources>
```

- Application code for retrieving plural strings array:

```
int count=getEggsInANestText();
Resources res=getResources();
String eggs=res.getQuantityString(R.plurals.EggsInANestText,count,count);
```

FORMATTING AND STYLING

- **Escaping apostrophes and quotes**

```
<string name="ex1">"this'll work"</string>
<string name="ex2">this\'ll work</string>
<string name="ex3">this doesn't  work</string>
```

- **Styling with HTML markup**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="welcome">
Welcome to <b>Android</b>
</string>
</resources>
```

LAYOUT RESOURCES

- A layout resource defines the architecture for the UI in an activity or a component of a UI
- Located in res/layout/filename.xml
- In Android, the view for a screen is often loaded from an XML file as a resource.
- These XML files are called layout resources

```
<?xml version="1.0" encoding="utf-8"?>
<ViewGroupxmlns:android="http://schemas.android.com/apk/res/android"
android:id="@[+][package:]id/resource_name"
android:layout_height=["dimension"|"match_parent"|"wrap_content">
android:layout_width=["dimension"|" match _parent"|"wrap_content">
[ViewGroup-specific attributes]>
<View    android:id="@[+][package:]id/resource_name"
android:layout_height=["dimension"|" match _parent"|"wrap_content">
android:layout_width=["dimension"|" match _parent"|"wrap_content">
[View-specific attributes]>
</view>
</ViewGroup>
```

- ViewGroup–root element.it must contain xmlns:androidattribute with android namespace
- Different ViewGroupobjects include LinearLayout, RelativeLayout, FrameLayout
- Individual UI component generally referred to as widget can be defined using <view>

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical" android:layout_width="fill_parent"
android:layout_height="fill_parent">

   <TextViewandroid:id="@+id/text" android:layout_width="wrap_content"
   android:layout_height="wrap_content" android:text="@string/hello"/>

   <Button  android:id="@+id/button" android:layout_width="fill_parent"
   android:layout_height="wrap_content" android:text="hello I am a button"/>

</LinearLayout>
```

COLOR RESOURCE

- A color value defined in xml
- Color is specified with an RGB value and alpha channel
- Value always begins with # and followed by Alpha-Red-Green-Blue information
  - #RGB
  - #ARGB
  - #RRGGBB
  - #AARRGGBB
- File is saved in res/values/colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<colorname="color_name">
Hex_color
</color>
</resources>
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<colorname="opaque_red">#f00</color>
<colorname="translucent_red">#80ff0000</color>
</resources>
```

- Code that retrieves colorresource

```
Resources res=getResources();
int color=res.getColor(R.color.opaque_red);
```

- Layout XML that applies colorto an attribute

```
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColor="@color/translucent_red"
android:text="Hello"/>
```

DIMENSION RESOURCES

- Dimension resources can be used to style and localize Android UIs without changing the source code
- Dimension is specified with a number followed by unit of measure :10px,2in
- Dimensions can be specified in any of the following units:
  - px: Pixels
  - in: Inches
  - mm: Millimeters
  - pt: Points
  - dp: Density-independent pixels based on a 160dpi (pixel density per inch) screen (dimensions adjust to screen density)
  - sp: Scale-independent pixels (dimensions that allow for user sizing; helpful for use in fonts)

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<dimenname="dimension_name"> dimen</dimen>
</resources>

<?xml version="1.0" encoding="utf-8"?>
<resources>
<dimenname="textview_height">25dp</dimen>
<dimenname="textview_ width">150dp </dimen>
<dimenname="font_size">16sp </dimen>
</resources>
```

- Code that retrieves a dimension

```
Resources res=getResources();
float fontSize=res.getDimension(R.dimen.font_size);
```

- Layout XML that applies dimension to an attribute

```
<TextView
android:layout_height="@dimen/textview_height"
android:layout_width="@dimen/textview_width"
android:textSize="@dimen/font_size"/>
```

IMAGE RESOURCES

- A drawable resource is a general concept for a graphic that can be drawn to the screen
- It can be retrieved using getDrawable()
- image files are placed in the /res/drawable subdirectory.
- The supported image types include .gif, .jpg, and .png.

```
<ImageView
android:layout_height="wrap_content"
android:layout_width="wrap_content"
android:src="@drawable/myimage"/>
```

- Code that retrieves an image

```
Resources res=getResources();
Drawable drawable=res.getDrawable(R.drawable.myimage);
```

RESOURCE REFERENCE SYNTAX

- Regardless of the type of resource all Android resources are identified (or referenced) by their IDs in Java source code.
- The syntax that is used to allocate an ID to a resource in the XML file is called **resource reference syntax.**
- This syntax is not limited to allocating just ids: it is a way to identify any resource such as a string, a layout file, or an image
- This resource reference has the following formal structure:

```
@[package:]type/name
```

- The type corresponds to one of the resource-type namespaces available in R.java, some of which follow:
  - R.drawable
  - R.id
  - R.layout

- R.string
- R.attr
- R.plural
- R.array

- The name part in the resource reference @[package:]type/name is the name given to the resource
- If you don’t specify any package in the syntax @[package:]type/name, the pair type/name is resolved based on local resources and the application’s local R.java package.
- If you specify android:type/name, the reference is resolved using the package android and specifically through the android.R.java file.
- You can use any Java package name in place of the package

```
<TextViewandroid:id=”@+id/text1” …./>
```

  - “The + indicates that if the id of text1 is not defined as a resource, go ahead and define it with a unique number.

DEFINING OWN RESOURCE IDs

- The general pattern for allocating an ID is either to create a new one or to use the one created by the Android package.
- However, it is possible to create IDs beforehand and use them later in your own packages
- The line <TextViewandroid:id="@+id/text"> indicates that an ID named text is used if it already exists.
- If the ID doesn’t exist, a new one is created.
- **Predefining an ID**

```
<resources> <item type="id" name="text"/> </resources>
```

- **Reusing a Predefined ID**

```
<TextViewandroid:id="@id/text"> .. </TextView>
```

ENUMERATING KEY ANDROID RESOURCES

Resource Type	Location	Description
Color	/res/values/any-file	Represents color identifiers pointing to color codes. These resource IDs are exposed in R.java as R.color.*.
String	/res/values/any-file	Represents string resources. String resources allow Java-formatted strings and raw HTML in addition to simple strings. These resource IDs are exposed in R.java as R.string.*.
String-array	/res/values/any-file	Represents a resource that is an array of strings. These resource IDs are exposed in R.java as R.array.*.
plurals	/res/values/any-file	Represents a suitable collection of strings based on the value of a quantity. The quantity is a number. The resource IDs are exposed in R.java as R.plural.*
Dimension	/res/values/any-file	Represents dimensions or sizes of various elements or views in Android. Supports pixels, inches, millimeters, density independent pixels, and scale independent pixels. These resource IDs are exposed in R.java as R.dimen.*
Image	/res/values/multiple-files	Represents image resources. Supported images include .jpg, .gif, .png, and so on. Each image is in a separate file and gets its own ID based on the file name. These resource ids are exposed in R.java as R.drawable.*

Resource type	Location	description
Color drawable	/res/values/any-file Also /res/values/multiple-files	Represents rectangles of colors to be used as view backgrounds or general drawables like bitmaps. In Java, this is equivalent to creating a colored rectangle and setting it as a background for a view.
Arbitrary XML files	/res/xml/*.xml	Android allows arbitrary XML files as resources. These files are compiled by the AAPT compiler. These resource IDs are exposed in R.java as R.xml.*.
Arbitrary raw resources	/res/raw/*.*	Android allows arbitrary noncompiled binary or text files under this directory. Each file gets a unique resource ID. These resource IDs are exposed in R.java as R.raw.*
Arbitrary raw assets	/assets/*.*/*.*	Android allows arbitrary files in arbitrary subdirectories starting at the /assets subdirectory. These are not really resources, just raw files.

CONTENT PROVIDERS

- Content providers are Android's central mechanism that enables to access data of other applications -mostly information stored in databases or flat files.
- Content providers support the four basic operations, normally called CRUD-operations.

  Create, Read, Update and Delete
- Provide data abstraction and encapsulation and provide mechanism for defining data security
- A content provider component supplies data from one application to others on request.
- Such requests are handled by the methods of the ContentResolverclass.
- A content provider can use different ways to store its data and the data can be stored in a database, in files, or even over a network.

NEED OF CONTENT PROVIDER

- Database in android is private to the application that creates it
- No common storage area in android that every applications can access
- For different applications to use a database android needs an interface that allows inter-application and inter-process data exchange

ANDROID BUILT IN PROVIDERS

Number of content providers are part of Android's API.

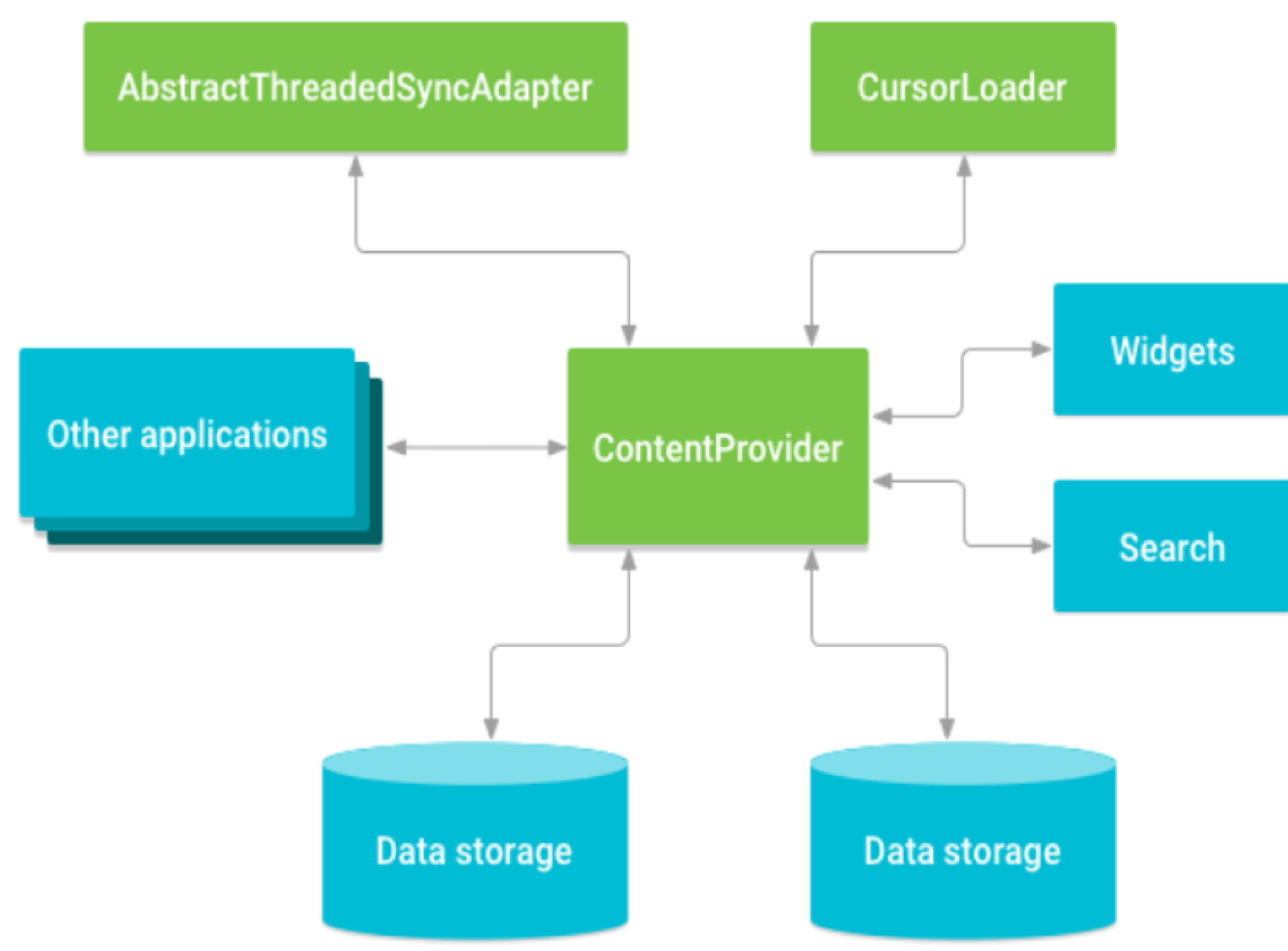
All these standard providers are defined in the package **android.provider**

Provider	Since	Usage
Browser	SDK 1	Manages your web-searches, bookmarks and browsing-history.
CalendarContract	SDK 14	Manages the calendars on the user's device.
CallLog	SDK 1	Keeps track of your call history.
Contacts	SDK 1	The old and deprecated content provider for managing contacts. You should only use this provider if you need to support an SDK prior to SDK 5!
ContactsContract	SDK 5	Deals with all aspects of contact management. Supersedes the Contacts-content provider.
MediaStore	SDK 1	The content provider responsible for all your media files like music, video and pictures.
Settings	SDK 1	Manages all global settings of your device.
UserDictionary	SDK 3	Keeps track of words you add to the default dictionary.

ARCHITECTURE OF CONTENT PROVIDER

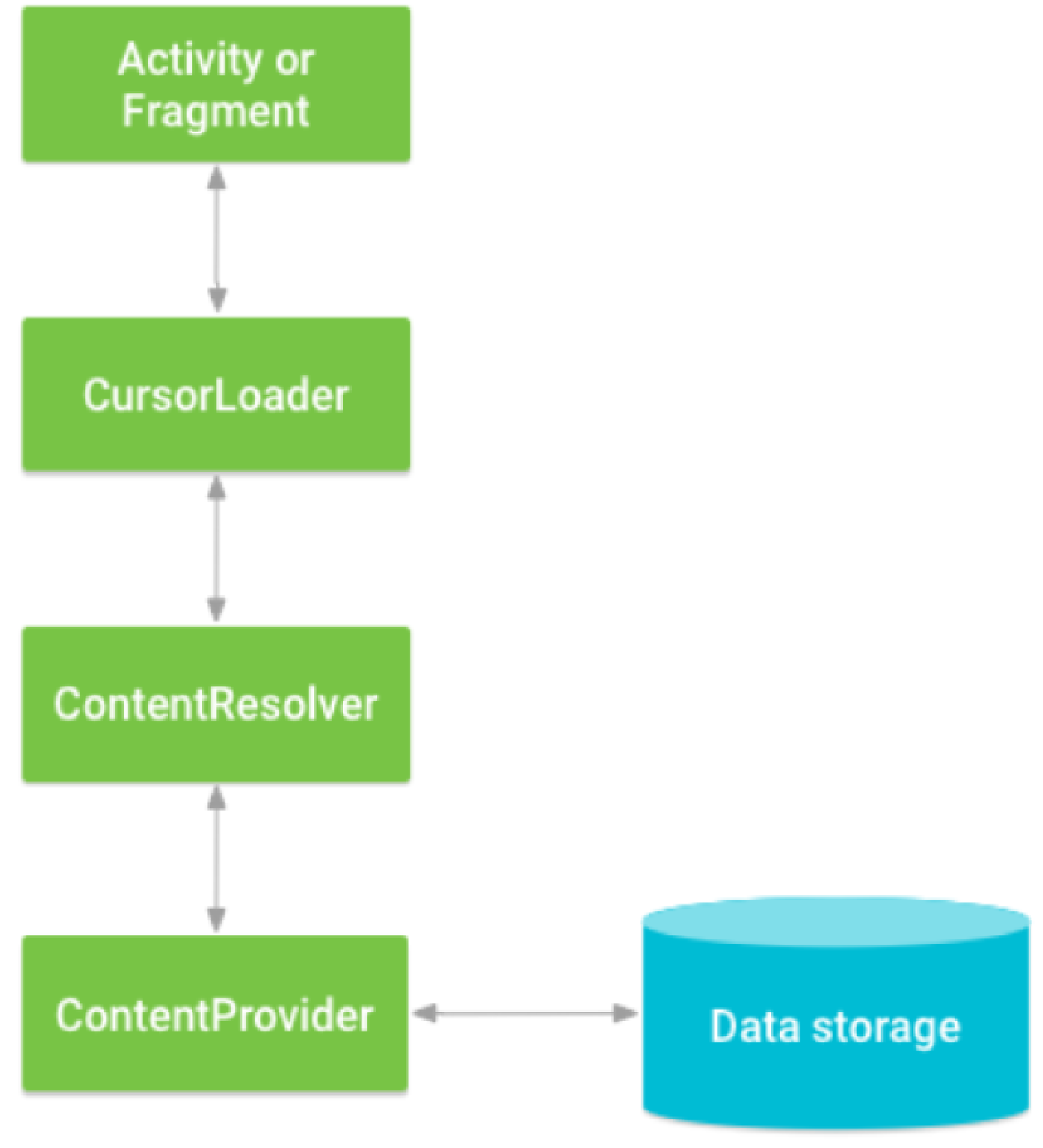
- A content provider presents data to external applications as one or more tables similar to tables in relational database
- A content provider coordinates access to the data storage layer in your application for a number of different APIs and components
  - Sharing access to your application data with other applications
  - Sending data to a widget
  - Returning custom search suggestions for your application through the search framework using **SearchRecentSuggestionsProvider**
  - Synchronizing application data with your server using an implementation of **AbstractThreadedSyncAdapter**
  - Loading data in your UI using a **CursorLoader**

Relationship between content provider and other components



- To access data in a content provider, use the ContentResolverobject to communicate with the provider as a client.
- The ContentResolverobject communicates with the provider object, an instance of a class that implements ContentProvider.
- The provider object receives data requests from clients, performs the requested action, and returns the results.
- The ContentResolvermethods provide the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage.
- A common pattern for accessing a ContentProviderfrom UI uses a **CursorLoader** to run an asynchronous query in the background.
- The Activity or Fragment in UI call a CursorLoader to the query, which in turn gets the ContentProvider using the ContentResolver.
- This allows the UI to continue to be available to the user while the query is running.
- This pattern involves the interaction of a number of different objects, as well as the underlying storage mechanism,

Interaction between ContentProvider, other classes, and storage



STRUCTURE OF ANDROID CONTENT URI

- Whenever you want to access data from a content provider you have to specify a URI.
- To query a content provider, you specify the query string in the form of a URI
- has following format −
- <scheme>://<authority>/<path>/<id>
- 4 parts
  - **Scheme :**It has a constant value: content
  - **Authority:** symbolic name of the provider for ex contacts , browser. unique for each one
  - **Path:** helps distinguish the required data from complete database
  - **Id:** specifies specific record requested . should be numeric
- Looking for contact number 5 in contacts then URI

  Content://contacts/people/5

READING DATA USING URI

- To retrieve data from a content provider, you need to use URIs supplied by that content provider.
- Because the URIs defined by a content provider are unique to that provider, it is important that these URIs are documented and available to programmers to see and then call.
- The providers that come with Android do this by defining constants representing these URI strings.
- Consider these three URIs defined by helper classes in the Android SDK:
- MediaStore.Images.Media.INTERNAL_CONTENT_URI
  MediaStore.Images.Media.EXTERNAL_CONTENT_URI
  ContactsContract.Contacts.CONTENT_URI
- The equivalent textual URI strings would be as follows:

  content://media/internal/images
  content://media/external/images
  content://com.android.contacts/contacts/

- Code to retrieve a single row of people from the Contacts provider

```
Uri peopleBaseUri= ContactsContract.Contacts.CONTENT_URI;
Uri myPersonUri= Uri.withAppendedPath(peopleBaseUri, "23");
```

- To retrieve data from a provider,
  - Request the read access permission for the provider.specifythat you need this permission in the manifest using<uses-permission>element

USING ANDROID CURSOR

- A cursor represents the result of a query and basically points to one row of the query result
- It is an interface which represents a 2D table of any database
- When retrieving data using SELECT statement, database will first create CURSOR object and return its reference
- The pointer of this returned reference is pointing to the 0thlocation which is otherwise called as before first location of the cursor
- To retrieve data from cursor, we have to move to the first record using moveToFirst()

CURSOR

- moveToFirst() method takes the cursor pointer to the first location and data in the first record can be accessed
- moveToNext()-to get the next data from cursor
- moveToLast()-last data in the result
- moveToPrevious()
- isAfterLast()-checks whether the end of the query result has been reached
- isBeforeFirst()-used to determine whether the cursor is at the default position of the Resultset
- getCount()-To find the number of rows in a cursor

WORKING WITH WHERE CLAUSE

- Content providers offer two ways of passing a where clause to retrieve data:
  - Through the URI
  - Through the combination of a string clause and a set of replaceable string-array arguments (explicit where clause)

Through the URI

- To retrieve a note whose ID is 23 from the Google notes database

```
Activity someActivity;
//initialize someActivity
String noteUri= "content://com.google.provider.NotePad/notes/23";
Cursor managedCursor= someActivity.managedQuery( noteUri,
projection, //Which columns to return.
null,       // WHERE clause
null); // Order-by clause.
```

  - Here where clause argument of the managedQuerymethod is null
  - id is embedded in the URI itself
  - URI is used as a vehicle to pass the where clause

Using Explicit where Clauses

- Method by which Android send a list of explicit columns and their corresponding values as a where clause

```
public final Cursor managedQuery(Uri uri,
    String[] projection,
    String selection,
    String[] selectionArgs,
    String sortOrder);
```

- Argument named selection represents a filter (a where clause) declaring which rows to return, Passing null will return all rows for the given URI.
- In the selection string we can include ?s, which will be replaced by the values from selectionArgsin the order that they appear in the selection.
- Query for a note whose ID is 23 using either of these two methods:

```
//URI method
managedQuery("content://com.google.provider.NotePad/notes/23" ,null ,null
,null ,null);
```

OR

```
//explicit where clause
managedQuery("content://com.google.provider.NotePad/notes" ,null ,"_id=?" ,new
String[] {23} ,null);
```

- The convention is to use where clauses through URIs where applicable and use the explicit option as a special case

INSERTING RECORDS

- Android uses a class called android.content.ContentValues to hold the values for a single record that is to be inserted. ContentValues is a dictionary of key/value pairs, like column names and their values.
- Records are inserted by first populating a record into ContentValues and then asking android.content.ContentResolver to insert that record using a URI.
- Populating a single row of notes in ContentValues in preparation for an insert:

```
ContentValuesvalues = new ContentValues();
values.put("title", "New note");
values.put("note","Thisis a new note");
//values object is now ready to be inserted
```

```
//get a reference to ContentResolverby asking the Activity class:
ContentResolvercontentResolver= activity.getContentResolver();
```

- URI for notepad is Notepad.Notes.CONTENT_URI
- take this URI and the ContentValuesand make a call to insert the row:

```
Uri uri= contentResolver.insert(Notepad.Notes.CONTENT_URI, values);
```

- This call returns a URI pointing to the newly inserted record.
- This returned URI would match the following structure:

```
Notepad.Notes.CONTENT_URI/new_id
```

UPDATES AND DELETES

- Performing an update is similar to performing an insert, in which changed column values are passed through a ContentValuesobject.
- int numberOfRowsUpdated= activity.getContentResolver().update(Uri uri,

  ContentValuesvalues, String whereClause, String[] selectionArgs)

- The whereClauseargument constrains the update to the applicable rows.
- The signature for the delete method is
- int numberOfRowsDeleted = activity.getContentResolver().delete(Uri uri, String whereClause, String[] selectionArgs)

IMPLEMENTING CONTENT

- To write a content provider, you have to extend android.content.ContentProvider and implement the following key methods:
  - onCreate()-method is called when the provider is started
  - Query()-receives a request from a client.Result is returned as cursor object
  - Insert()-inserts a new record into the content provider
  - Delete()-deletes an existing record from the content provider
  - Update()-this method updates an existing record from the content provider
  - getType()-returns the MIME type of the data at the given URI

Content-provider implementation steps :

1. Plan your database, URIs, column names, and so on, and create a metadata class that defines constants for all of these metadata elements.
2. Extend the abstract class ContentProvider.
3. Implement these methods: query, insert, update, delete, and getType
4. Register the provider in the manifest file.

INTENTS

- At the simplest level, an intent is an action that you can tell Android to perform (or invoke). The action Android invokes depends on what is registered for that action.
- Intents facilitate communication b/w components in several ways.
- 3 fundamental use cases
- **Starting an activity:-**an activity represents single screen in an app.to start a new instance of an activity ,pass an intent to startActivity(). To receive a result from an activity when finishes, call startActivityForResult()
- **Starting a service:-**a service is a component that performs operations in the background without a user interface.to start a service to perform a one-time operation, pass an intent to startService()
- **Delivering a broadcast:-**broadcast is a message that any app can receive.to deliver a broadcast to other apps, pass an intent to sendBroadcast() or sendOrderedBroadcast()

AVAILABLE INTENTS IN ANDROID

- The set of available applications could include
- A browser application to open a browser window
- An application to call a telephone number
- An application to present a phone dialer so the user can enter the numbers and make a call through the UI
- A mapping application to show the map of the world at a given latitude and longitude coordinate

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

INTENT STRUCTURE

- Primary pieces of information in an intent
  - **ACTION:-**The general action to be performed.ACTION_VIEW,ACTION_EDIT,ACTION_MAIN
  - **DATA :-**The data to operate on such as person record in the contact database ,expressed as uri

- **Examples of action/data pairs:**
  - ACTION_VIEW content://contacts/people/1 :- display information about person whose ID=1
  - ACTION_DIAL content://contacts/people/1 :- display phone dialer with person filled in
  - ACTION_VIEW tel:123,ACTION_DIAL tel:123 :- display phone dialer with given number filled in
  - ACTION_EDIT content://contacts/people/1 :- EDIT information about person whose ID=1

SECONDARY ATTRIBUTES OF INTENTS

- **CATEGORY**:-gives additional information about kind of component that should handle the intent. addCategory() places a category in an intent object, removeCategory() deletes a category previously added, getCategories() gets the set of all categories currently in the object.
- For example, CATEGORY_LAUNCHER means it should appear in the Launcher as a top-level application, while CATEGORY_ALTERNATIVE means it should be included in a list of alternative actions the user can perform on a piece of data.
- **EXTRAS:** This is a Bundle of anyadditional information. Used to provide extended information to the component.canbe set and read using putExtras() and getExtras()
- For example, if we have a action to send an e-mail message, we could also include extra pieces of data here to supply a subject, body, etc.
- **FLAGS**: optional part of intent object that instruct android system how to launch an activity and how to treat it after it is launched
- FLAG_ACTIVITY_CLEAR_TASK, FLAG_ACTIVITY_NEW_TASK

TYPES OF INTENTS

- **Implicit intents**: Do not specify the android components which should be called .
- Only specifies action to be performed. A Uri can be used with implicit intent to specify the datatype
- Intent intent=new Intent(ACTION_VIEW,Uri.parse(http://www.google.com”));
- **Explicit intent:** Use explicit intents when you know exactly which activity can handle the request.
- Explicit intents are used in applications wherein one activity can switch to other activity.
- Typically used for application-internal-messages Such as an activity starting a subroutine service or launching a sister activity

```
Intent i=newIntent(FirstActivity.this,SecondActivity.class)
startActivity(i);
```

RULES FOR RESOLVING INTENTS TO THEIR COMPONENTS

- Android uses multiple strategies to match intents to their target activities based on intent filters.
- At the top of the hierarchy is the component name attached to an intent. If this is set, the intent is known as an explicit intent.
- For an explicit intent, only the component name matters; every other aspect or attribute of the intent is ignored.
- When a component name is not present on an intent, the intent is said to be an implicit intent.
- When the system receives an implicit intent to start an activity, it searches for the best activity for the intent by comparing it to intent filters based on 3 aspects
  - Action
  - Data(URI and datatype)
  - Category
- Intent filter will specify type of intents it will accept. They are declared in manifest file

- **Action test:**
  - If an intent has an action on it, the intent filter must have that action as part of its action list
  - To specify accepted intent actions, an intent filter can declare zero or more <action> elements

```
<intent-filter>
<action android:name="android.intent.action.EDIT"/>
<action android:name="android.intent.action.VIEW"/>
….
</intent-filter>
```

  - To pass this filter, the action specified in the intent must match one of the actions listed in the filter
  - If the filter doesn't list any action,thenall intents fail the test
  - If an intent doesn't specify an action, it passes the test as long as filter contains at least one action

- **Category Test:**
  - To specify accepted intent categories, an intent filter can declare zero or more <category> elements

```
<intent-filter>
<category android:name="android.intent.category.DEFAULT"/>
<category android:name="android.intent.category.BROWSABLE"/>
….
</intent-filter>
```

  - For an intent to pass the category test, every category in the intent must match a category in the filter.
  - Reverse is not necessary
  - An intent with no categories always passes the test regardless of what categories are declared in the filter

- **Data test:**
  - To specify accepted intent data, filter can declare zero or more <data> elements

```
<intent-filter>
<data android:mimeType="video/mpeg" android:scheme="http"…./>
<data android:mimeType="audio/mpeg" android:scheme="http"…./>
….
</intent-filter>
```

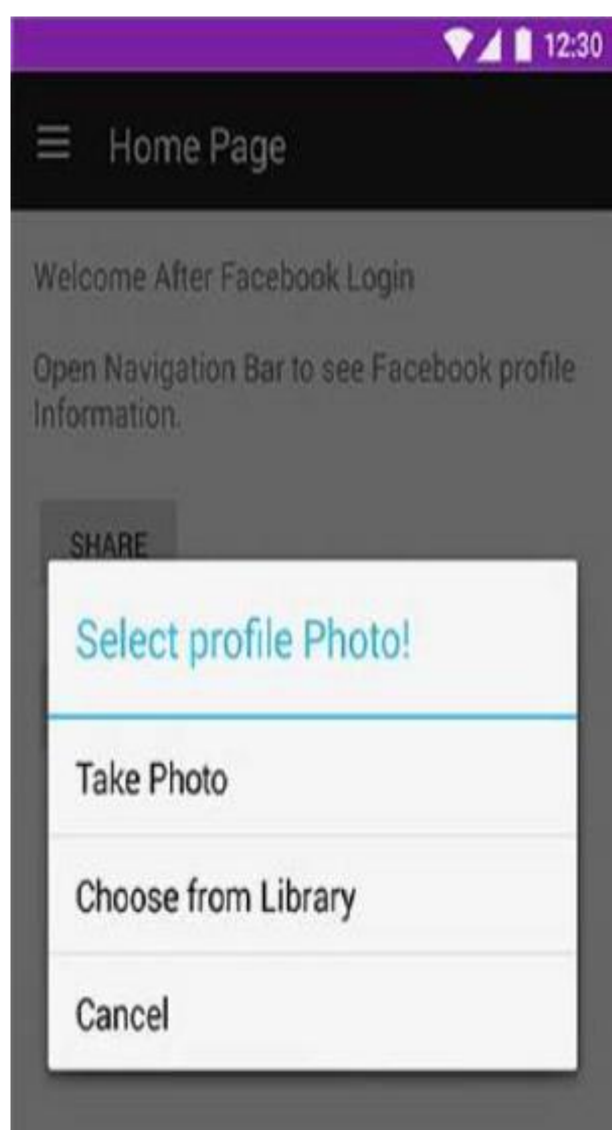
  - Each <Data> element contains a URI structure and a data type(mime type)
  - Each part of URI is a separate attribute
  - <scheme>://<host>:<port>/<path>
  - Ex: content://com.example.project:200/folder/subfolder/etc
  - Data test compares both URI and MIME type in the intent to that of the filter
  - An intent that contains neither URI nor MIME passes the test only if filter doesn't specify any URIs or MIME types

- An intent that contains a URI but no MIME passes the test only if its URI matches the filter's URI and filter doesn't specify any MIME types
- An intent that contains MIME but not URI passes the test only if filter list same MIME types and doesn't specify any URI
- An intent that contains both URI & MIME passes the MIME type part of the test if it matches a type listed in the filter. It passes URI part of test if its URI matches a URI in the filter or if it has a content: or file: URI and filter doesn't specify any URI

Action_pick

- The idea of ACTION_PICK is to start an activity that displays a list of items.
- The activity then should allow a user to pick one item from that list.
- Once the user picks the item, the activity should return the URI of the picked item to the caller.
- This allows reuse of the UI's functionality to select items of a certain type.
- It helps to pick an image item from a data source like camera or gallery

```
Intent photoPickerIntent= new
Intent(Intent.ACTION_PICK);

photoPickerIntent.setType("image/*");

startActivityForResult(photoPickerIntent,
SELECT_PHOTO);
```



Action_get_content

- Allow the user to select a particular kind of data and return it.
- This is different than ACTION_PICK in that here we just say what kind of data is desired, not a URI of existing data from which the user can pick

```
Intent pickIntent= new Intent(Intent.ACTION_GET_CONTENT);
int requestCode= 2;
pickIntent.setType("vnd.android.cursor.item/vnd.google.note");
activity.startActivityForResult(pickIntent, requestCode);
```

PENDING INTENT

- Android allows a component to store an intent for future use in a location from which it can be invoked again.
- For example, in an alarm manager, you want to start a service when the alarm goes off.
- Android does this by creating a wrapper pending intent around a normal corresponding intent and storing it away so that even if the calling process dies off, the intent can be dispatched to its target.

Creating a pending intent:

- Syntax:

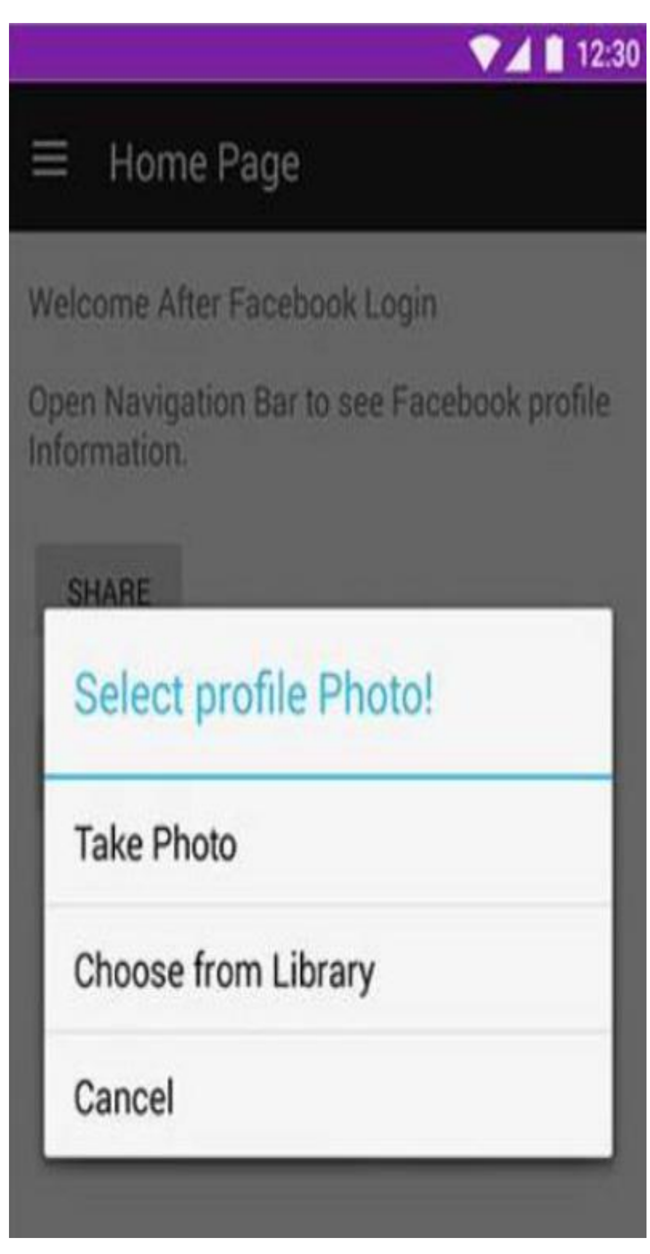
```
PendingIntent.getActivity(Context context, //originating context    int
requestCode, //1,2, 3, etc      Intent intent, //original intent    int
flags //flags )
```

- Usually, you can pass a zero for requestCodeand flags to get the default behavior.

```
Intent regularIntent;
PendingIntentpi = PendingIntent.getActivity(context, 0, regularIntent,0);
```

ACTION_PICK

- The idea of ACTION_PICK is to start an activity that displays a list of items.
- The activity then should allow a user to pick one item from that list.
- Once the user picks the item, the activity should return the URI of the picked item to the caller.
- This allows reuse of the UI's functionality to select items of a certain type.
- It helps to pick an image item from a data source like camera or gallery
- Launches a sub-Activity that lets you pick an item from the Content Provider specified by the Intent's data URI.
- When closed, it should return a URI to the item that was picked.
- The Activity launched depends on the data being picked



- for example, passing *content://contacts/people* will invoke the native contacts list
- Almost all core android applications (eg: Messaging, Gallery, Contacts etc) provide this facility.
- All you need is the URI of the data you need and required permissions to access that data.

startActivityForResult:

- public void startActivityForResult(Intent intent, int requestCode)
- The requestCodehelps you to identify from which Intent you came back.
- For example, imagine your Activity A (Main Activity) could call Activity B (Camera Request), Activity C (Audio Recording), Activity D (Select a Contact).
- Whenever the subsequently called activities B, C or D finish and need to pass data back to A, now you need to identify in your *onActivityResult*from which Activity you are returning from and put your handling logic accordingly.

```
private static final int REQUEST_PICK_IMAGE = 1;
Intent pickImageIntent= new Intent(Intent.ACTION_PICK,
android.provider.MediaStore.Images.
Media. EXTERNAL_CONTENT_URI);
startActivityForResult(pickImageIntent, REQUEST_PICK_IMAGE);
```

```
private static final int PICK_CONTACT_SUBACTIVITY = 2;
Uri uri= Uri.parse("content://contacts/people");
Intent intent= new Intent(Intent.ACTION_PICK, uri);
startActivityForResult(intent, PICK_CONTACT_SUBACTIVITY);
```

ACTION_GET_CONTENT

- Allow the user to select a particular kind of data and return it.
- This is different than ACTION_PICK in that here we just say what kind of data is desired, not a URI of existing data from which the user can pick
- An ACTION_GET_CONTENT could allow the user to create the data as it runs (for example taking a picture or recording a sound), let them browse over the web and download the desired data, etc.

- There are two main ways to use this action:
- For a specific kind of data, such as a person contact, set the MIME type to the kind of data you want and launch it with Context #startActivity(Intent).
- The system will then launch the best application to select that kind of data

```
Intent intent= new Intent(Intent.ACTION_GET_CONTENT);
intent.setType(ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE);
startActivityForResult(intent, 1);
```

- It is possible to wrap the GET_CONTENT intent with a chooser (through createChooser(Intent, CharSequence)), which will give the proper interface for the user to pick how to send your data and allow you to specify a prompt indicating what they are doing.

```
Intent intent= new Intent();

intent.setType("image/*");

intent.setAction(Intent.ACTION_GET_CONTENT);

startActivityForResult(Intent.createChooser(intent,"SelectPicture"),
SELECT_PICTURE);
```

PENDING INTENT

- Android allows a component to store an intent for future use in a location from which it can be invoked again.
- For example, in an alarm manager, you want to start a service when the alarm goes off.
- Android does this by creating a wrapper pending intent around a normal corresponding intent and storing it away so that even if the calling process dies off, the intent can be dispatched to its target.
- Android PendingIntent is an object that wraps up an intent object and it specifies an action to be taken place in future.
- In other words, PendingIntent pass a future Intent to another application and allow that application to execute that Intent as if it had the same permissions as our application, whether or not our application is still around when the Intent is eventually invoked.
- A PendingIntent is generally used in cases were an AlarmManager needs to be executed or for Notification (that we'll implement later in this tutorial).
- A PendingIntent provides a means for applications to work, even after their process exits.

- Each explicit intent is supposed to be handled by a specific app component like Activity, BroadcastReceiveror a Service.
- Hence PendingIntentuses the following methods to handle the different types of intents:
- PendingIntent.getActivity() : Retrieve a PendingIntentto start an Activity
- PendingIntent.getBroadcast() : Retrieve a PendingIntentto perform a Broadcast
- PendingIntent.getService() : Retrieve a PendingIntentto start a Service

```
Intent intent= new Intent(this, SomeActivity.class);
// Creating a pending intent and wrapping our intent
PendingIntentpendingIntent= PendingIntent.getActivity(this, 1, intent,
PendingIntent.FLAG_UPDATE_CURRENT);
pendingIntent.send();
```

- The operation associated with the pendingIntent is executed using the send() method.
- The parameters inside the getActivity() method :
  - **this (context)** : This is the context in which the PendingIntentstarts the activity
  - **requestCode**: “1” is the private request code for the sender

- **intent :** Explicit intent object of the activity to be launched
- **flag :** FLAG_UPDATE_CURRENT. This one states that if a previous PendingIntent already exists, then the current one will update it with the latest intent. There are many other flags like FLAG_CANCEL_CURRENT etc.

THANK YOU

MODULE III

USER INTERFACES DEVELOPMENT IN ANDROID

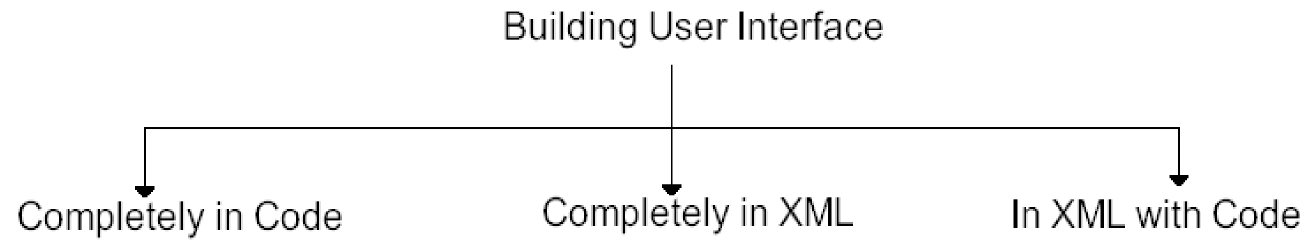
UI development in Android is fun. It's fun because it's relatively easy. With Android, we have a simple-to-understand framework with a limited set of out-of-the-box controls. The available screen area is generally limited. Android also takes care of a lot of the heavy lifting normally associated to designing and building quality UIs. This, combined with the fact that the user usually wants to do one specific action, allows us to easily build a good UI to deliver a good user experience. The Android SDK provides text fields, buttons, lists, grids, and so on. In addition, Android provides a collection of controls that are appropriate for mobile devices.

At the heart of the common controls are two classes: android.view.View and android.view.ViewGroup. As the name of the first class suggests, the View class represents a general-purpose View object. The common controls in Android ultimately extend the View class. ViewGroup is also a view, but it contains other views too. ViewGroup is the base class for a list of layout classes. Android, like Swing, uses the concept of *layouts* to manage how controls are laid out within a container view. Using layouts, as we'll see, makes it easy for us to control the position and orientation of the controls in our UIs.

We can construct UIs entirely in code. We can also define UIs in XML. We can even combine the two define the UI in XML and then refer to it, and modify it, in code. We will find the terms *view*, *control*, *widget*, *container*, and *layout* in discussions regarding UI development. If we are new to Android programming or UI development in general, we might not be familiar with these terms. We'll briefly describe them before we get started (see Table 6–1).

Table 6–1. *UI Nomenclature*

Term	**Description**
View, widget, control	Each of these represents a UI element. Examples include a button, a grid, a list, a window, a dialog box, and so on. The terms *view*, *widget*, and *control* are used interchangeably in this chapter.
Container	This is a view used to contain other views. For example, a grid can be considered a container because it contains cells, each of which is a view.
Layout	This is a visual arrangement of containers and views and can include other layouts.



an double OS

BUILDING UI COMPLETELY IN CODE

The first example, Listing 6–1, demonstrates how to build the UI entirely in code. To try this, create a new Android project with an activity named MainActivity and then copy the code from Listing 6–1 into our MainActivity class.

Listing 6–1. *Creating a Simple User Interface Entirely in Code*

```
package com.androidbook.controls;
import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup.LayoutParams;
import android.widget.LinearLayout;
import android.widget.TextView;
public class MainActivity extends Activity
{
        private LinearLayout nameContainer;
        private LinearLayout addressContainer;
        private LinearLayout parentContainer;
        /** Called when the activity is first created. */
        @Override
        public void onCreate(Bundle savedInstanceState)
        {
                super.onCreate(savedInstanceState);
                createNameContainer();
                createAddressContainer();
                createParentContainer();
                setContentView(parentContainer);
        }
        private void createNameContainer()
        {
                nameContainer = new LinearLayout(this);
                nameContainer.setLayoutParams(new
                            LayoutParams(LayoutParams.FILL_PARENT,
                            LayoutParams.WRAP_CONTENT));
                nameContainer.setOrientation(LinearLayout.HORIZONTAL);
                TextView nameLbl = new TextView(this);
                nameLbl.setText("Name: ");
                TextView nameValue = new TextView(this);
                nameValue.setText("John Doe");
                nameContainer.addView(nameLbl);
                nameContainer.addView(nameValue);
        }
        private void createAddressContainer()
        {
                addressContainer = new LinearLayout(this);
                addressContainer.setLayoutParams(new
                          LayoutParams(LayoutParams.FILL_PARENT,
```

an double os

```
                LayoutParams.WRAP_CONTENT));
        addressContainer.setOrientation(LinearLayout.VERTICAL);
        TextView addrLbl = new TextView(this);
        addrLbl.setText("Address:");
        TextView addrValue = new TextView(this);
        addrValue.setText("911 Hollywood Blvd");
        addressContainer.addView(addrLbl);
        addressContainer.addView(addrValue);
    }
    private void createParentContainer()
    {
        parentContainer = new LinearLayout(this);
        parentContainer.setLayoutParams(new
        LayoutParams(LayoutParams.FILL_PARENT,
        LayoutParams.FILL_PARENT));
        parentContainer.setOrientation(LinearLayout.VERTICAL);
        parentContainer.addView(nameContainer);
        parentContainer.addView(addressContainer);
    }
}
```

As shown in Listing 6–1, the activity contains three LinearLayout objects. As we mentioned earlier, layout objects contain logic to position objects within a portion of the screen. A LinearLayout, for example, knows how to lay out controls either vertically or horizontally. Layouts objects can contain any type of view even other layouts. The nameContainer object contains two TextView controls: one for the label Name: and the other to hold the actual name (such as John Doe). The addressContainer also contains two TextView controls. The difference between the two containers is that the nameContainer is laid out horizontally and the addressContainer is laid out vertically. Both of these containers live within the parentContainer, which is the root view of the activity.

After the containers have been built, the activity sets the content of the view to the root view by calling setContentView(parentContainer). When it comes time to render the UI of the activity, the root view is called to render itself. The root view then calls its children to render themselves, and the child controls call their children, and so on, until the entire UI is rendered. As shown in Listing 6–1, we have several LinearLayout controls. Two of them are laid out vertically, and one is laid out horizontally. The nameContainer is laid out horizontally. This means the two TextView controls appear side by side horizontally. The addressContainer is laid out vertically, which means the two TextView controls are stacked one on top of the other. The parentContainer is also laid out vertically, which is why the nameContainer appears above the addressContainer. Note a subtle difference between the two vertically laid-out containers, addressContainer and parentContainer. parentContainer is set to take up the entire width and height of the screen:

an double os

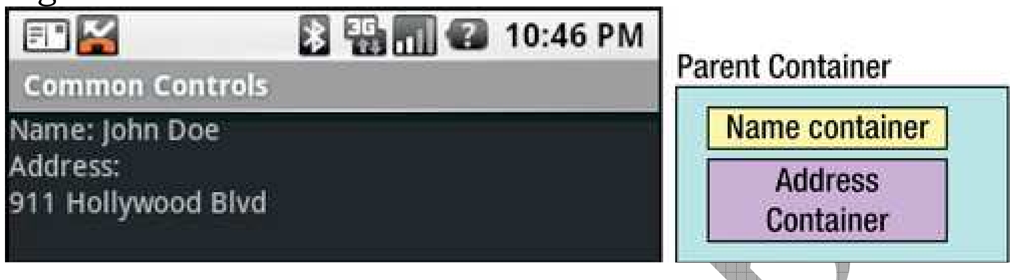
```
parentContainer.setLayoutParams(new
        LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.FILL_PARENT));
```

And addressContainer wraps its content vertically:

```
addressContainer.setLayoutParams(new
     LayoutParams(LayoutParams.FILL_PARENT,
LayoutParams.WRAP_CONTENT));
```

Said another way, WRAP_CONTENT means the view should take just the space it needs in that dimension and no more, up to what the containing view will allow. For the addressContainer, this means the container will take two lines vertically, because that's all it needs.

Figure 6–1



BUILDING UI USING XML

XML layout files are stored under the resources (/res/) directory in a folder called layout. By default, we will get an XML layout file named main.xml, located under the res/layout folder. Double-click main.xml to see the contents. Eclipse will display a visual editor for our layout file. We probably have a string at the top of the view that says "Hello World, MainActivity!" or something like that. Click the main.xml tab at the bottom of the view to see the XML of the main.xml file. This reveals a LinearLayout and a TextView control. Using either the Layout or main.xml tab, or both, re-create Listing 6–2 in the main.xml file. Save it.

Listing 6–2. *Creating a User Interface Entirely in XML*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <!-- NAME CONTAINER -->
        <LinearLayout
                xmlns:android="http://schemas.android.com/apk/res/android"
                android:orientation="horizontal" android:layout_width="fill_parent"
                android:layout_height="wrap_content">

                        <TextView android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content" android:text="Name:" />

        <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text="John Doe" />
</LinearLayout>

<!-- ADDRESS CONTAINER -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="wrap_content">

        <TextView android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:text="Address:" />

        <TextView android:layout_width="fill_parent"
android:layout_height="wrap_content" android:text="911 Hollywood Blvd."
/>
</LinearLayout>
</LinearLayout>
```

Under our new project's src directory, there is a default .java file containing an Activity class definition. Double-click that file to see its contents. Notice the statement setContentView(R.layout.main). The XML snippet shown in Listing 6–2, combined with a call to setContentView(R.layout.main), will render the same UI as before when we generated it completely in code. The XML file is self-explanatory, but note that we have three container views defined. The first LinearLayout is the equivalent of our parent container. This container sets its orientation to vertical by setting the corresponding property like this: android:orientation="vertical". The parent container contains two LinearLayout containers, which represent nameContainer and addressContainer. Running this application will produce the same UI as our previous example application. The labels and values will be displayed as shown in Figure 6–1.

BUILDING UI IN XML WITH CODE

Listing 6–2 is a contrived example. It doesn't make any sense to hard-code the values of the TextView controls in the XML layout. Ideally, we should design our UIs in XML and then reference the controls from code. This approach enables us to bind dynamic data to the controls defined at design time. In fact, this is the recommended approach. It is fairly easy to build layouts in XML and then use code to populate the dynamic data. Listing 6–3 shows the same UI with slightly different XML. This XML assigns IDs to the TextView controls so that we can refer to them in code.

Listing 6–3. *Creating a User Interface in XML with IDs*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

an double os

```
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <!-- NAME CONTAINER -->
    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal" android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text="@string/name_text" />

        <TextView android:id="@+id/nameValue"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
    </LinearLayout>

    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="wrap_content">

    <TextView android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:text="@string/addr_text" />

    <TextView android:id="@+id/addrValue"
    android:layout_width="fill_parent" android:layout_height="wrap_content" />
    </LinearLayout>
</LinearLayout>
```

In addition to adding the IDs to the TextView controls that we want to populate from code, we also have label TextView controls that we're populating with text from our strings resource file. These are the TextViews without IDs that have an android:text attribute. As we may recall from Chapter 3, the actual strings for these TextViews will come from our strings.xml file in the /res/values folder. The developers of Android have done a nice job of making just about every aspect of a control settable via XML or code. It's usually a good idea to set the control's attributes in the XML layout file rather than using code. However, there will be lots of times when we need to use code, such as setting a value to be displayed to the user.

FILL_PARENT vs. MATCH_PARENT

The constant FILL_PARENT was deprecated in Android 2.2 and replaced with MATCH_PARENT. This was strictly a name change, though. The value of this constant is still -1. Similarly, for XML layouts, fill_parent was replaced with match_parent. So what value do we use? Instead of FILL_PARENT or MATCH_PARENT, we could simply use the value -1, and you'd be fine. However, this isn't very easy to read, and we don't have an equivalent unnamed value to use with our XML layouts. There's a better way.

an double OS

ANDROID'S COMMON CONTROLS

We will now start our discussion of the common controls in the Android SDK. We'll start with text controls and then cover buttons, check boxes, radio buttons, lists, grids, date and time controls, and a map-view control. We will also talk about layout controls.

TEXT CONTROLS

Text controls are likely to be the first type of control that we'll work with in Android. Android has a complete but not overwhelming set of text controls. In this section, we are going to discuss the TextView, EditText, AutoCompleteTextView, and MultiCompleteTextView controls. Figure 6–2 shows the controls in action.

TextView:-

A TextView displays text to the user and optionally allows them to edit it. It is a complete text editor but the basic class is not allowing to editing. The TextView control knows how to display text but does not allow editing. This might lead us to conclude that the control is essentially a dummy label. We can set the autoLink property to emai or web, and the control will find and highlight any e-mail addresses and URLs. TextView is utilizing the android.text.util.Linkify class. The main attributes are id, hint, maxHeight, maxWidth, password, minHeight, minWidth, text, phoneNumber, textColor, textStyle etc.

EditText:-

The EditText control is a subclass of TextView. As suggested by the name, the EditText control allows for text editing. EditText is not as powerful as the text-editing controls that we find on the Internet, but users of Android-based devices probably won't type documents they'll type a couple paragraphs at most. Therefore, the class has limited but appropriate functionality and may even surprise us. For example, one of the most significant properties of an EditText is the inputType. We can set the inputType property to textAutoCorrect have the control correct common misspellings. We can set it to textCapWords to have the control capitalize words. Other options expect only phone numbers or passwords. The old default behavior of the EditText control is to display text on one line and expand as needed. The user to a single line by setting the singleLine property to true. The user will have to continue typing on the same line. With inputType, if we don't specify textMultiLine, the EditText will default to single-line only. So if we want the old default behavior of multiline typing, we need to specify inputType with textMultiLine.

AutoCompleteTextView:-

The AutoCompleteTextView control is a TextView with auto-complete functionality. In other words, as the user types in the TextView, the control can display suggestions for selection. For example, if the user types **en**, the control suggests English. If the user types **gr**, the control recommends Greek, and so on.

an double os

MultiAutoCompleteTextView:-

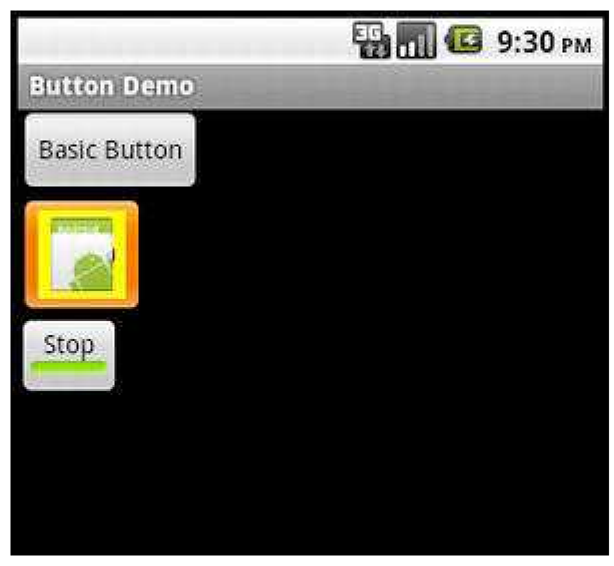
The control offers suggestions only for the *entire* text in the text view. In other words, if we type a sentence, we don't get suggestions for each word. That's where MultiAutoCompleteTextView comes in. We can use the MultiAutoCompleteTextView to provide suggestions as the user types. For example, the user typed the word **English** followed by a comma, and then **Ge**, at which point the control suggested **German**. If the user were to continue, the control would offer additional suggestions. Using the MultiAutoCompleteTextView is like using the AutoCompleteTextView. The difference is that we have to tell the control where to start suggesting again. For example, in Figure 6–2, we can see that the control can offer suggestions at the beginning of the sentence and after it sees a comma. The MultiAutoCompleteTextView control requires that we give it a tokenizer that can parse the sentence and tell it whether to start suggesting again. Listing 6–8 demonstrates using the MultiAutoCompleteTextView control with the XML and then the Java code.

Figure 6–2. *Text controls in Android*

BUTTON CONTROLS

Buttons are common in any widget toolkit, and Android is no exception. Android offers the typical set of buttons as well as a few extras. We will discuss three types of button controls: the basic button, the image button, and the toggle button. The figure shows a UI with these controls. The button at the top is the basic button, the middle button is an image button, and the last one is a toggle button.

The Basic Button Control:-



The basic button class in Android is android.widget.Button. There's not much to this type of button, beyond how we use it to handle click events. We register for the on-click event by calling the setOnClickListener() method with an OnClickListener. When the button is clicked, the onClick() method of the listener is called. The handler method will be called with target set to the View object representing the button that was clicked. Notice how the switch statement in the click handler method uses the resource IDs of the buttons to select the logic to run. Using this method means we won't have to explicitly create each Button object in our code, and we can reuse the same method across multiple buttons. This makes things easier to understand and maintain. This works with the other button types as well.

The ImageButton Control:-

Android provides an image button via android.widget.ImageButton. Using an image button is similar to using the basic button. The image file for the button must exist under /res/drawable. In our case, we're simply reusing the Android icon for the button. Note that we only need to do one or the other. We don't need to specify the button image in both the XML file and in code. One of the nice features of an image button is that we can specify a transparent background for the button. The result will be a clickable image that acts like a button but can look like whatever we want it to look like. Just set android:background="@null" for the image button.

The ToggleButton Control:-

The ToggleButton control, like a check box or a radio button, is a two-state button. This button can be in either the ON or OFF state. As shown in Figure 6–3, the ToggleButton's default behavior is to show a green bar when in the ON state and a grayed-out bar when in the OFF state. Moreover, the default behavior also sets the button's text to ON when it's in the ON state and OFF when it's in the OFF state. We can modify the text for the ToggleButton if ON/OFF is not appropriate for our application. For example, if we have a background process that we want to start and stop via a ToggleButton, we could set the button's text to Stop and Run by using android:textOn and android:textOff properties.

an double

CHECKBOX CONTROL:-

The CheckBox control is another two-state button that allows the user to toggle its state. The difference is that, for many situations, the users don't view it as a button that invokes immediate action. From Android's point of view, however, it is a button, and we can do anything with a check box that we can do with a button. We manage the state of a check box by calling setChecked() or toggle(). We can obtain the state by calling isChecked().If we need to implement specific logic when a check box is checked or unchecked, we can register for the on-checked event by calling setOnCheckedChangeListener() with an implementation of the OnCheckedChangeListener interface. We'll then have to implement the onCheckedChanged() method, which will be called when the check box is checked or unchecked. The nice part of setting up the OnCheckedChangeListener is that we are passed the new state of the CheckBox button. We could instead use the OnClickListener technique as we used with basic buttons. When the onClick() method is called, we would need to determine the new state of the button by casting it appropriately and then calling isChecked() on it.

RADIO BUTTON CONTROLS

RadioButton controls are an integral part of any UI toolkit. A radio button gives the users several choices and forces them to select a single item. To enforce this single-selection model, radio buttons generally belong to a group, and each group is forced to have only one item selected at a time. To create a group of radio buttons in Android, first create a RadioGroup, and then populate the group with radio buttons. Note that the radio buttons within the radio group are, by default, unchecked to begin with, although we can set one to checked in the XML definition. To set one of the radio buttons to the checked state programmatically, we can obtain a reference to the radio button and call setChecked().We can also use the toggle() method to toggle the state of the radio button. As withthe CheckBox control, we will be notified of on-checked or on-unchecked events if we call the setOnCheckedChangeListener() with an implementation of the OnCheckedChangeListener interface. There is a slight difference here, though. This is a different class than before. This time, it's technically the RadioGroup.OnCheckedChangeListener class, whereas before it was the CompoundButton.OnCheckedChangeListener class. The RadioGroup can also contain views other than the radio button.

IMAGE VIEW CONTROL

This is used to display an image, where the image can come from a file, a content provider, or a resource such as a drawable. We can even specify just a color, and the ImageView will display that color. Listing 6–21 shows some

an double os

XML examples of ImageViews, followed by some code that shows how to create an ImageView.

Listing 6–21. *ImageViews in XML and in Code*

```
<ImageView android:id="@+id/image1"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:src="@drawable/icon" />

<ImageView android:id="@+id/image2"
    android:layout_width="125dip" android:layout_height="25dip"
    android:src="#555555" />

<ImageView android:id="@+id/image3"
    android:layout_width="wrap_content"
android:layout_height="wrap_content"/>

<ImageView android:id="@+id/image4"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:src="@drawable/manatee02"
    android:scaleType="centerInside"
    android:maxWidth="35dip" android:maxHeight="50dip" />

ImageView imgView = (ImageView)findViewById(R.id.image3);
imgView.setImageResource( R.drawable.icon );
imgView.setImageBitmap(BitmapFactory.decodeResource(
    this.getResources(), R.drawable.manatee14) );
imgView.setImageDrawable(
Drawable.createFromPath("/mnt/sdcard/dave2.jpg") );
imgView.setImageURI(Uri.parse("file://mnt/sdcard/dave2.jpg"));
```

In this example, we have four images defined in XML. The first is simply the icon for our application. The second is a gray bar that is wider than it is tall. The third definition does not specify an image source in the XML, but we associate an ID with this one (image3) that we can use from our code to set the image. The fourth image is another of our drawable image files where we not only specify the source of the image file but also set the maximum dimensions of the image on the screen and define what to do if the image is larger than our maximum size. In this case, we tell the ImageView to center and scale the image so it fits inside the size we specified.

In the Java code of Listing 6–21 we show several ways to set the image of image3. We first of course must get a reference to the ImageView by finding it using its resource ID. The first setter method, setImageResource(), simply uses the image's resource ID to locate the image file to supply the image for our ImageView. The second setter uses the BitmapFactory to read in an image resource into a Bitmap object and then sets the ImageView to that Bitmap. Note that we could have done some modifications to the Bitmap before

an double OS

applying it to our ImageView, but in our case, we used it as is. In addition, the BitmapFactory has several methods of creating a Bitmap, including from a byte array and an InputStream. We could use the InputStream method to read an image from a web server, create the Bitmap image, and then set the ImageView from there.

The third setting uses a Drawable for our image source. In this case, we're showing the source of the image coming from the SD card. We 'll need to put some sort of image file out on the SD card with the proper name for this to work for us. Similar to BitmapFactory, the Drawable class has a few different ways to construct Drawables, including from an XML stream.

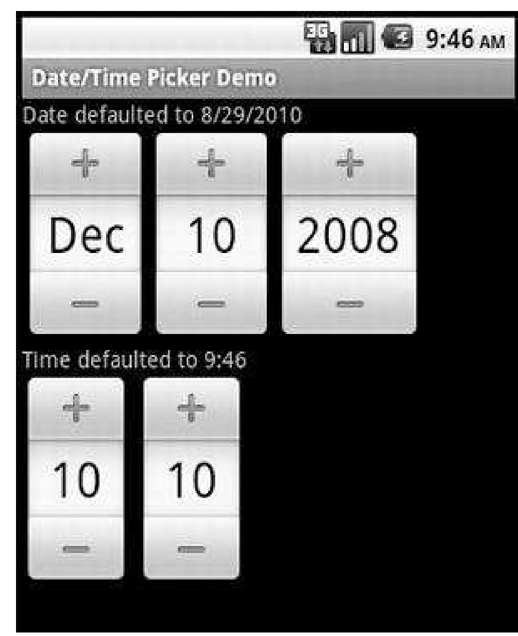
The final setter method takes the URI of an image file and uses that as the image source. For this last call, don't think that we can use any image URI as the source. This method is really only intended to be used for local images on the device, not for images that we might find through HTTP. To use Internet-based images as the source for our ImageView, we 'd most likely use BitmapFactory and an InputStream.

DATE AND TIME CONTROLS

Date and time controls are common in many widget tool kits. Android offers several date- and time-based controls. We are going to introduce the DatePicker, TimePicker, DigitalClock, and AnalogClock controls.

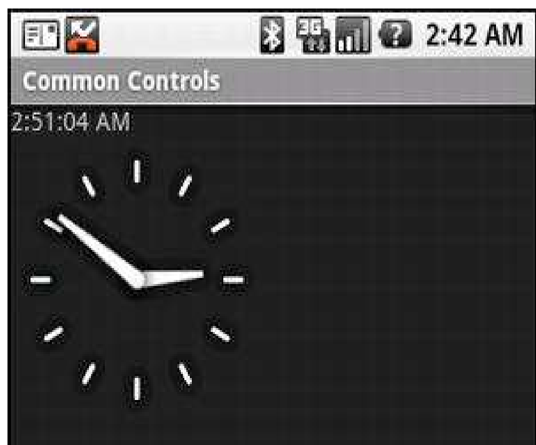
The DatePicker and TimePicker Controls:-

As the names suggest, we use the DatePicker control to select a date and the TimePicker control to pick a time. If we look at the XML layout, we can see that defining these controls is easy. As with any other control in the Android toolkit, we can access the controls programmatically to initialize them or to retrieve data from them. Note that for the month, the internal value is zero-based, which means that January is 0 and December is 11. For the TimePicker, the number of hours and minutes is set to 10. Note also that this control supports 24–hour view. If we do not set values for these controls, the default values will be the current date and time as known to the device. Finally, note that Android offers versions of these controls as modal windows, such as DatePickerDialog and TimePickerDialog. These controls are useful if we want to display the control to the user and force the user to make a selection.



an double os

The DigitalClock and AnalogClock Controls:-



Android also offers DigitalClock and AnalogClock controls. As shown, the digital clock supports seconds in addition to hours and minutes. The analog clock in Android is a two-handed clock, with one hand for the hour indicator and the other hand for the minute indicator. These two controls are really just for displaying the current time, as they don't let we modify the date or time. In other words, they are controls whose only capability is to display the current time. Thus, if we want to change the date or time, we'll need to stick to the DatePicker/TimePicker or DatePickerDialog/TimePickerDialog. The nice part about these two clocks, though, is that they will update themselves without having to do anything. That is, the seconds tick away in the DigitalClock, and the hands move on the AnalogClock without anything extra from us.

MAP VIEW CONTROL

The com.google.android.maps.MapView control can display a map. We can instantiate this control either via XML layout or code, but the activity that uses it must extend MapActivity. MapActivity takes care of multithreading requests to load a map, perform caching, and so on. Listing 6–25 shows an example instantiation of a MapView.

Listing 6–25. *Creating a MapView Control via XML Layout*

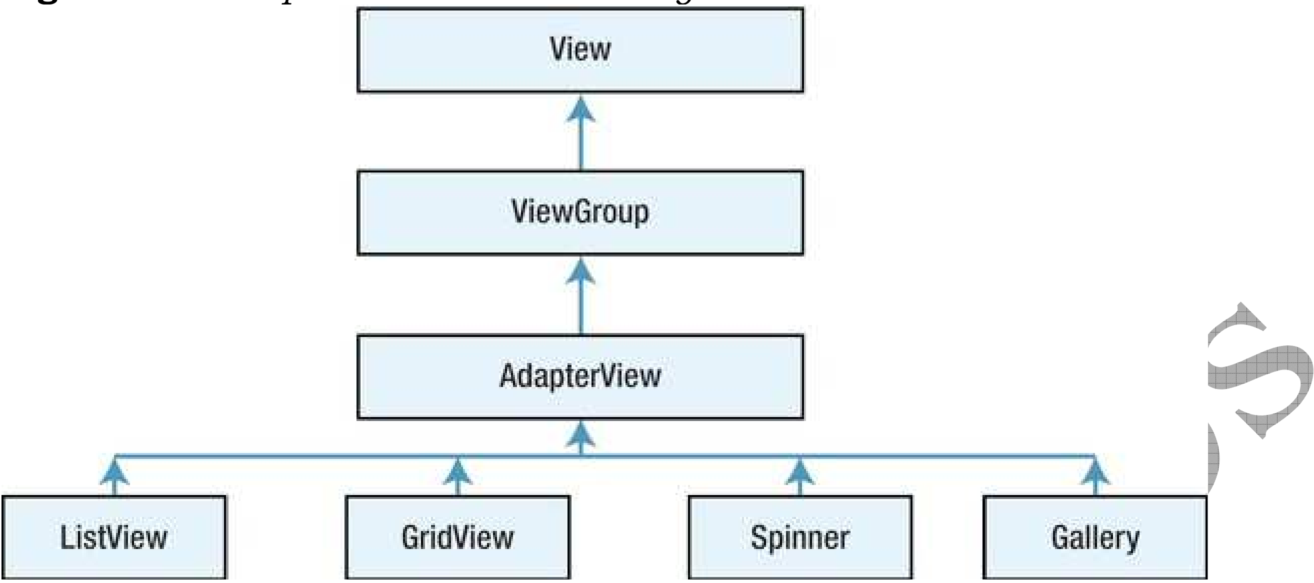
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <com.google.android.maps.MapView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey="myAPIKey"/>
</LinearLayout>
```

UNDERSTANDING ADAPTERS

List controls are used to display collections of data. But instead of using a single type of control to manage both the display and the data, Android separates these two responsibilities into list controls and adapters. List controls are classes that extend android.widget.AdapterView and include ListView, GridView, Spinner, and Gallery.

andoublec

Figure 6–8. *AdapterView class hierarchy*



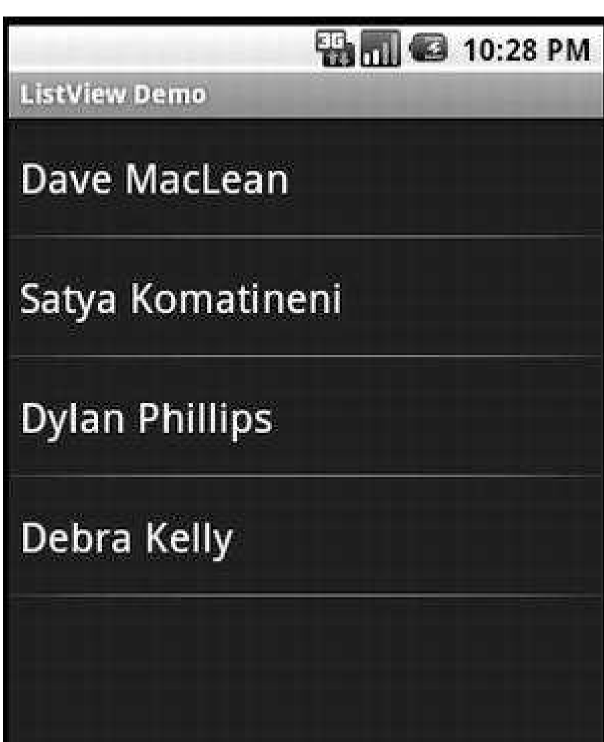
AdapterView itself extends android.widget.ViewGroup, which means that ListView, GridView, and so on are container controls. In other words, list controls contain collections of child views. The purpose of an adapter is to manage the data for an AdapterView and to provide the child views for it. Let's see how this works by examining the SimpleCursorAdapter.

ADAPTER VIEWS

Now that we've been introduced to adapters, it is time to put them to work for us, providing data for list controls. In this section, we're going to first cover the basic list control, the ListView. Then, we'll describe how to create our own custom adapter, and finally, we'll describe the other types of list controls: GridViews, spinners, and the gallery.

LIST VIEW

The ListView control displays a list of items vertically. That is, if we've got a list of items to view and the number of items extends beyond what we can currently see in the display, we can scroll to see the rest of the items. We generally use a ListView by writing a new activity that extends android.app.ListActivity. ListActivity contains a ListView, and we set the data for the ListView by calling the setListAdapter() method. As we described previously, adapters link list controls to the data and help prepare the child views for the list control. Items in a ListView can be clicked to take immediate action or selected to act on the set of selected



an doubles

items later. We're going to start really simple and then add functionality as we go.

Clickable Items in a ListView:-

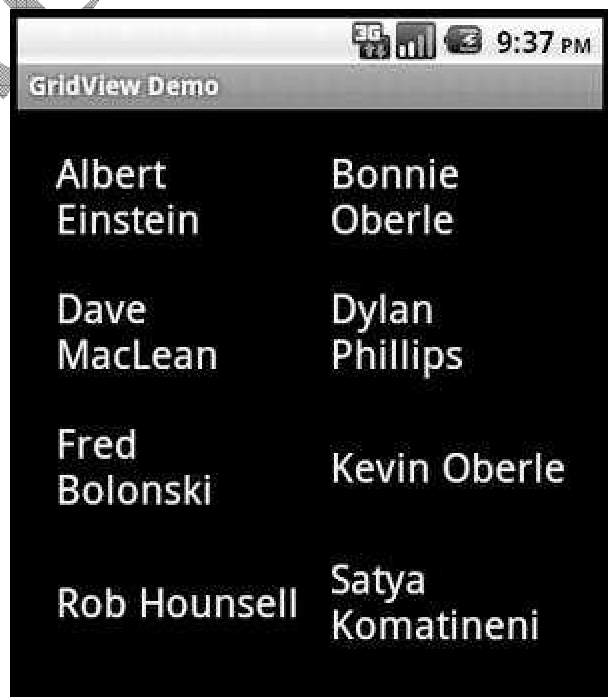
We're able to scroll up and down the list to see all our contact names, but that's about it. What if we want to do something a little more interesting with this example, like launch the Contact application when a user clicks one of the items in our ListView?

Adding Other Controls with a ListView:-

If we want additional controls in our main layout, we can provide our own layout XML file, put in a ListView, and add other desired controls. For example, we could add a button below the ListView in the UI to submit an action on the selected items, as shown in the figure.

GRID VIEW

Android has a GridView control that can display data in the form of a grid. We use the term *data* here; the contents of the grid can be text, images, and so on. The GridView control displays information in a grid. The usage pattern for the GridView is to define the grid in the XML layout and then bind the data to the grid using an android.widget.ListAdapter. Don't forget to add the uses-permission tag to the AndroidManifest.xml file to make this example work.



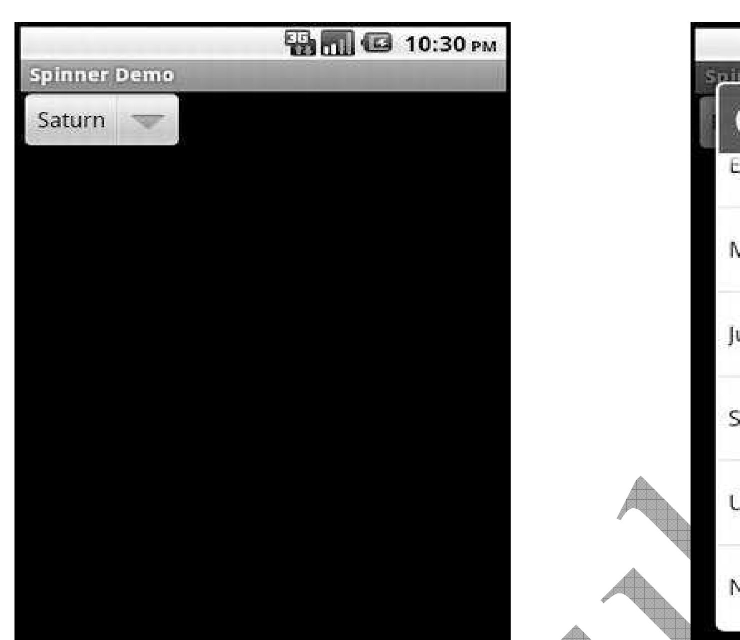
We've no doubt noticed that the adapter used by the grid is a ListAdapter. Lists are generally one-dimensional, whereas grids are two-dimensional. We can conclude, then, that the grid actually displays list-oriented data. And it turns out that the list is displayed by rows. That is, the list goes across the first row, then across the second row, and so on.

As before, we have a list control that works with an adapter to handle the data management, and the generation of the child views. The same techniques we used before should work just fine with GridViews. One exception

relates to making selections: there is no way to specify multiple choices in a GridView.

SPINNER CONTROL

The Spinner control is like a drop-down menu. It is typically used to select from a relatively short list of choices. If the choice list is too long for the display, a scrollbar is automatically added for us. Although a spinner is technically a list control, it will appear to us more like a simple TextView control.



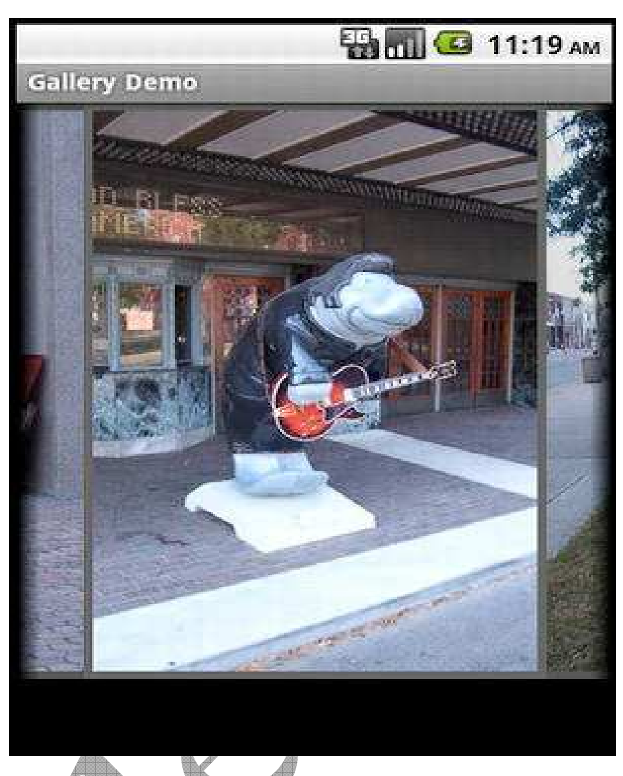
In other words, only one value will be displayed when the spinner is at rest. The purpose of the spinner is to allow the user to choose from a set of predetermined values: when the user clicks the small arrow, a list is displayed, and the user is expected to pick a new value. Populating this list is done in the same way as the other list controls: with an adapter. Because a spinner is often used like a drop-down menu, it is common to see the adapter get the list choices from a resource file. Notice the new attribute called android:prompt for setting a prompt at the top of the list to choose from. The actual text for our spinner prompt is in our /res/values/strings.xml file. As we should expect, the Spinner class has a method for setting the prompt in code as well.

STYLES AND THEMES

Android provides several ways to alter the style of views in our application. We'll first cover using markup tags in strings and then how to use spannables to change specific visual attributes of text. But what if we want to control how things look using a common specification for several views or across an entire activity or application?

GALLERY CONTROL

The Gallery control is a horizontally scrollable list control that always focuses at the center of the list. This control generally functions as a photo gallery in touch mode. The Gallery control is typically used to display images, so our adapter is likely going to be specialized for images. We'll show a custom image adapter in next section on custom adapters. See the figure.

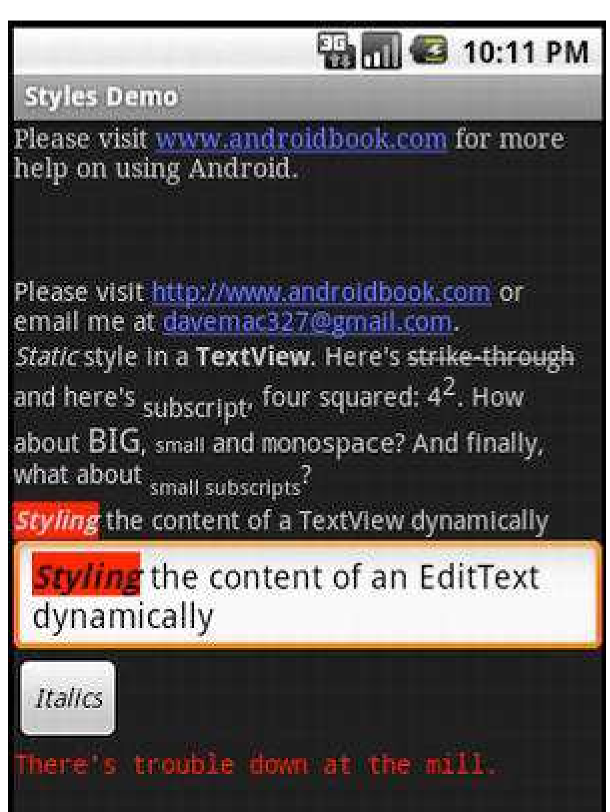


Using Styles:-

Sometimes, we want to highlight or style a portion of the View's content. We can do this statically or dynamically. Statically, we can apply markup directly to the strings in our string resources, as shown here:

<string name="styledText"><i>Static</i> style in a TextView.</string>

We can then reference it in our XML or from code. Note that we can use the following HTML tags with string resources: <i>, <b>, and <u> for italics, bold, and underlined, respectively, as well as <sup> (superscript), <sub> (subscript), <strike> (strikethrough), <big>, <small>, and <monospace>. We can even nest these to get, for example, small superscripts. This works not just in TextViews but also in other views, like buttons. The figure shows what styled and themed text looks like, using many of the examples in this section.



Styling a TextView control's content programmatically requires a little additional work but allows for much more flexibility (see Listing 6–35), because we can style it at runtime. This flexibility can only be applied to a spannable, though, which is how EditText normally manages the internal text, whereas TextView does not normally use Spannable. Spannable is basically a String that we can apply styles to. To get a TextView to store text as a spannable, we can call setText() this way:

tv.setText("This text is stored in a Spannable",

TextView.BufferType.SPANNABLE);

Then, when we call tv.getText(), we 'll get a spannable. We can get the content of the EditText (as a Spannable object) and then set styles for portions of the text. The code in the listing sets the text styling to bold and italics and sets the background to red. We can use all the styling options as we have with the HTML tags as described previously, and then some.

Using Themes:-

One problem with styles is that we need to add an attribute specification of style="@style/..." to every view definition that we want it to apply to. If we have some style elements we want applied across an entire activity, or across the whole application, we should use a theme instead. A *theme* is really just a style applied broadly; but in terms of defining a theme, it's exactly like a style. In fact, themes and styles are fairly interchangeable: we can extend a theme into a style or refer to a style as a theme. Typically, only the names give a hint as to whether a style is intended to be used as a style or a theme. To specify a theme for an activity or an application, we would add an attribute to the <activity> or <application> tag in the AndroidManifest.xml file for our project. The code might look like one of these:

<activity android:theme="@style/MyActivityTheme">
<application android:theme="@style/MyApplicationTheme">
<application android:theme="@android:style/Theme.NoTitleBar">

We can find the Android-provided themes in the same folder as the Android-provided styles, with the themes in a file called themes.xml. When we look inside the themes file, we will see a large set of styles defined, with names that start with Theme. We will also notice that within the Android-provided themes and styles, there is a lot of extending going on, which is why we end up with styles called Theme.Dialog.AppError", This concludes our discussion of the Android control set. As we mentioned in the beginning of the chapter, building UIs in Android requires we to master two things: the control set and the layout managers. In the next section, we are going to discuss the Android layout managers.

UNDERSTANDING LAYOUT MANAGERS

Android offers a collection of view classes that act as containers for views. These container classes are called *layouts* (or *layout managers*), and each implements a specific strategy to manage the size and position of its children. For example, the LinearLayout class lays out its children either horizontally or vertically, one after the other. All layout managers derive from the View class; therefore we can nest layout managers inside of one another. The layout managers that ship with the Android SDK are defined in Table 6–2.

Table 6–2. *Android Layout Managers*

Layout Manager	**Description**
LinearLayout	Organizes its children either horizontally or vertically
TableLayout	Organizes its children in tabular form
RelativeLayout	Organizes its children relative to one another or to the parent
FrameLayout	Allows us to dynamically change the control(s) in the layout
GridLayout	Organizes its children in a grid arrangement

LINEAR LAYOUT MANAGER

The LinearLayout layout manager is the most basic. This layout manager organizes its children either horizontally or vertically based on the value of the orientation property. We've used LinearLayout in several of our examples so far. We can create a vertically oriented LinearLayout by setting the value of orientation to vertical. Because layout managers can be nested, we could, for example, construct a vertical layout manager that contained horizontal layout managers to create a fill-in form, where each row had a label next to an EditText control. Each row would be its own horizontal layout, but the rows as a collection would be organized vertically.

Layout managers extend android.widget.ViewGroup, as do many control-based container classes such as ListView. Although the layout managers and control-based containers extend the same class, the layout manager classes strictly deal with the sizing and position of controls and not user interaction with child controls. For example, compare LinearLayout to the ListView control. On the screen, they look similar in that both can organize children vertically. But the ListView control provides APIs for the user to make selections, whereas LinearLayout does not. In other words, the control-based container (ListView) supports user interaction with the items in the container, whereas the layout manager (LinearLayout) addresses sizing and positioning only.

Using the LinearLayout layout manager

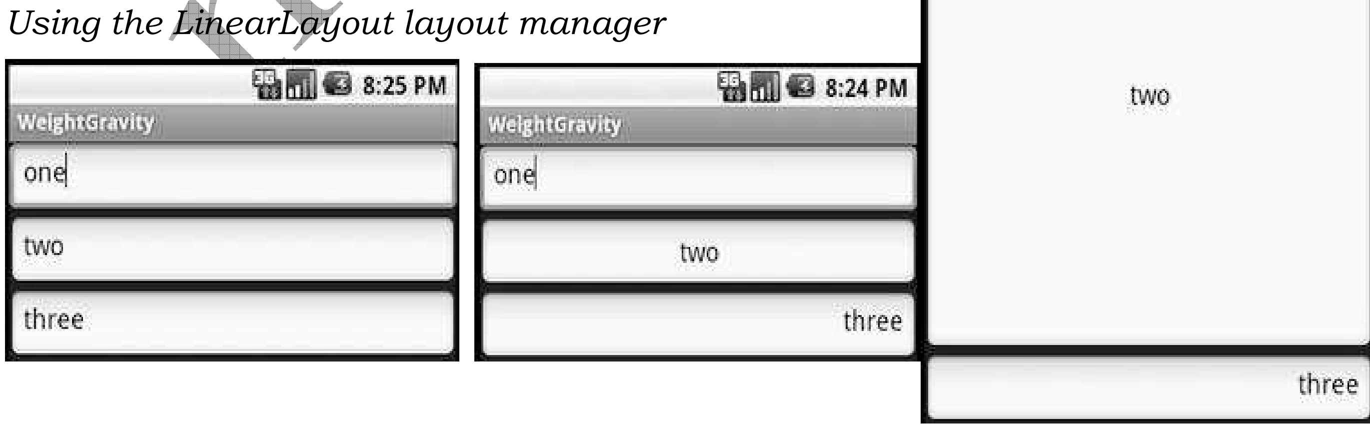
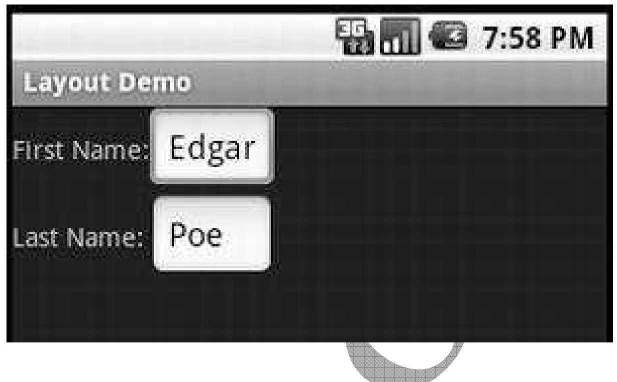


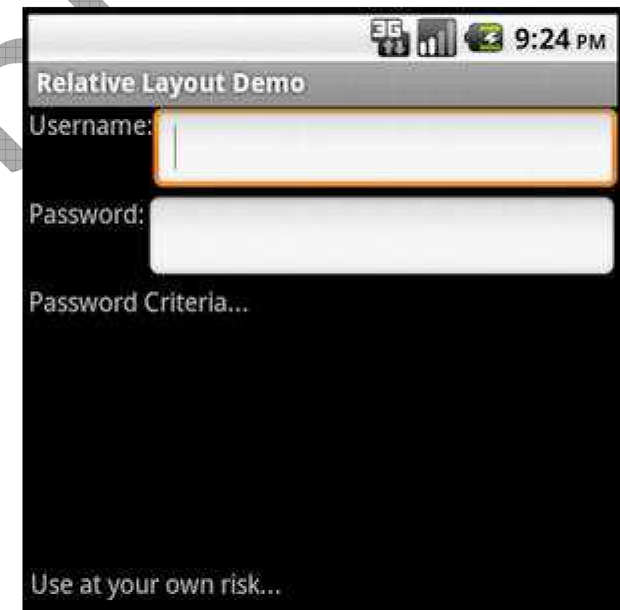
TABLE LAYOUT MANAGER

The TableLayout layout manager is an extension of LinearLayout. This layout manager structures its child controls into rows and columns. To use this layout manager, we create an instance of TableLayout and place TableRow elements within it. These TableRow elements contain the controls of the table. Because the contents of a TableLayout are defined by rows as opposed to columns, Android determines the number of columns in the table by finding the row with the most cells. Android creates a table with two rows and three columns. The last column of the first row is an empty cell.



RELATIVE LAYOUT MANAGER

Another interesting layout manager is RelativeLayout. As the name suggests, this layout manager implements a policy where the controls in the container are laid out relative to either the container or another control in the container. As shown, the UI looks like a simple login form. The username label is pinned to the top of the container, because we set android:layout_alignParentTop to true. Similarly, the Username input field is positioned below the Username label because we set android:layout_below. The Password label appears below the Username label, and the Password input field appears below the Password label. The disclaimer label is pinned to the bottom of the container because we set android:layout_alignParentBottom to true.



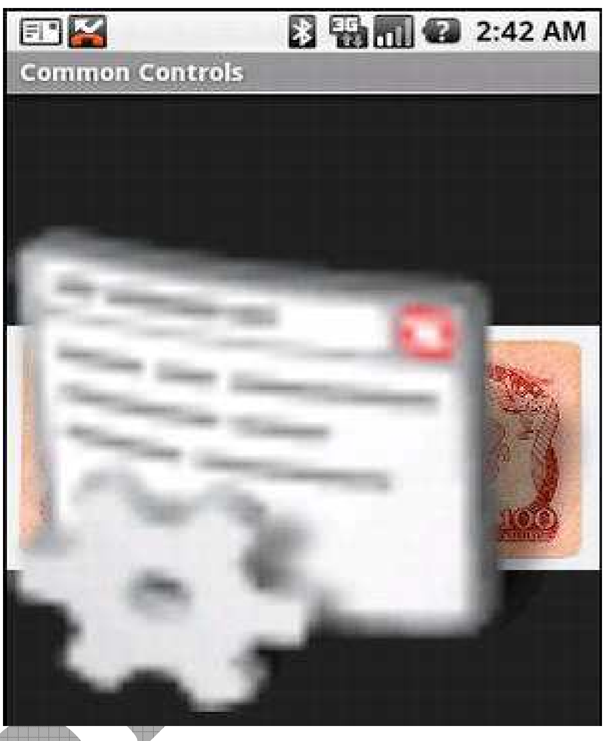
Besides these three layout attributes, we can also specify layout_above, layout_toRightOf, layout_toLeftOf, layout_centerInParent, and several more. Working with RelativeLayout is fun due to its simplicity. In fact, once we start using it, it'll become our favorite layout manager we'll find our self going back to it over and over again.

FRAME LAYOUT MANAGER

The layout managers that we've discussed so far implement various layout strategies. In other words, each one has a specific way that it positions and orients its children on the screen. With these layout managers, we can have many controls on the screen at one time, each taking up a portion of the screen. Android also offers a layout manager that is mainly used to display a single item: FrameLayout. We mainly use this utility layout class to dynamically

display a single view, but we can populate it with many items, setting one to visible while the others are invisible.

As we said earlier, we generally use FrameLayout when we need to dynamically set the content of a view to a single control. Although this is the general practice, the control will accept many children, as we demonstrated. Adds two controls to the layout but has one of the controls visible at a time. FrameLayout, however, does not force us to have only one control visible at a time. If we add many controls to the layout, FrameLayout will simply stack the controls, one on top of the other, with the last one on top. This can create an interesting UI. For example, Figure 6–25 shows a FrameLayout control with two ImageView objects that are visible. We can see that the controls are stacked, and that the top one is partially covering the image behind it.



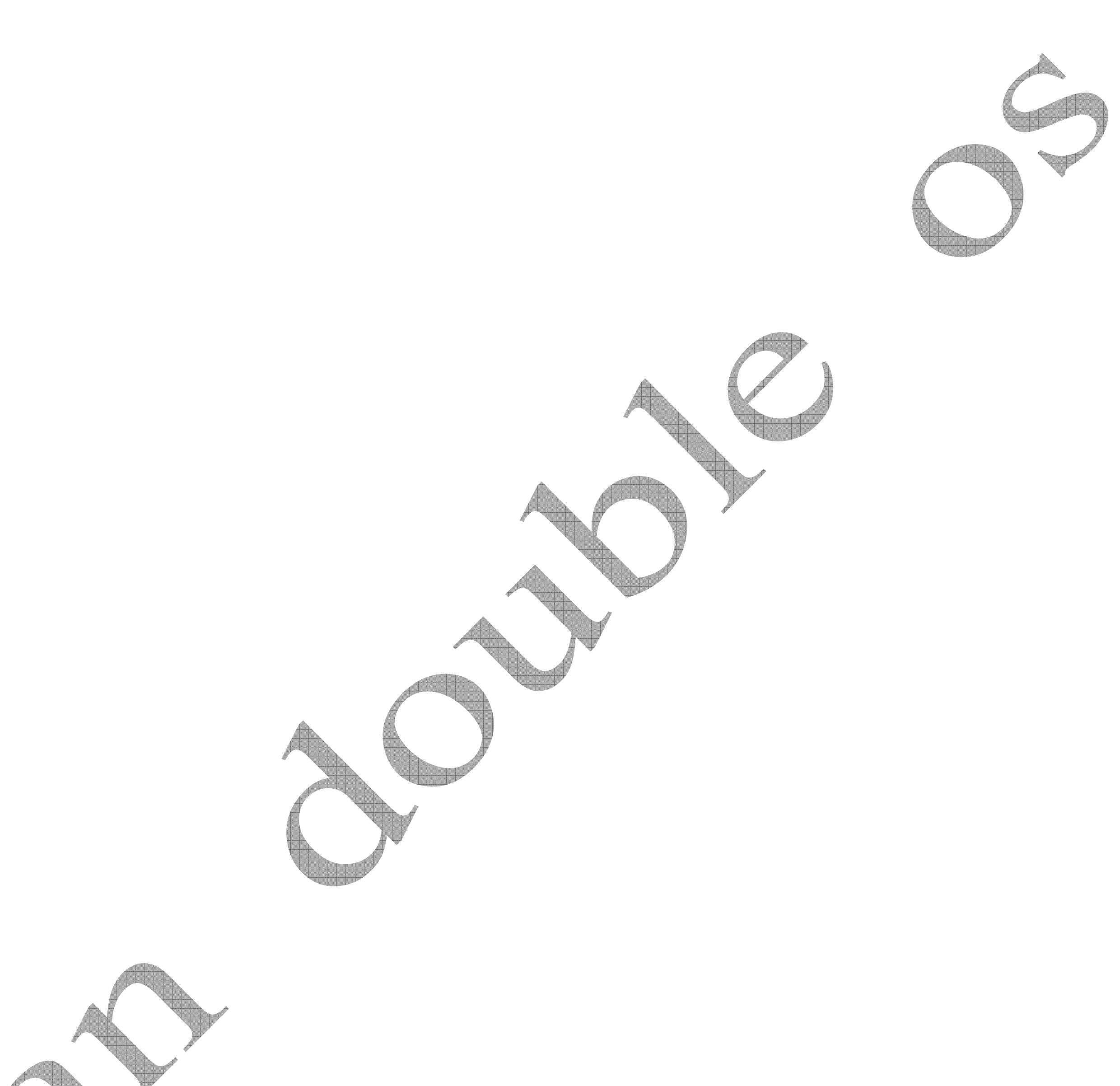
Another interesting aspect of the FrameLayout is that if we add more than one control to the layout, the size of the layout is computed as the size of the largest item in the container, the top image is actually much smaller than the image behind it, but because the size of the layout is computed based on the largest control, the image on top is stretched. Also note that if we put many controls inside a FrameLayout with one or more of them invisible to start, we might want to consider using setMeasureAllChildren(true) on our FrameLayout. Because the largest child dictates the layout size, you'll have a problem if the largest child is invisible to begin with: when it becomes visible, it is only partially visible. To ensure that all items are rendered properly, call setMeasureAllChildren() and pass it a value of true. The equivalent XML attribute for FrameLayout is android:measureAllChildren="true".

GRID LAYOUT MANAGER

Android 4.0 brought with it a new layout manager called GridLayout. As we might expect, it lays out views in a grid pattern of rows and columns, somewhat like TableLayout. However, it's easier to use than TableLayout. With a GridLayout, we can specify a row and column value for a view, and that's where it goes in the grid. This means we don't need to specify a view for every cell, just those that we want to hold a view. Views can span multiple grid cells. We can even put more than one view into the same grid cell.

When laying out views, we must not use the weight attribute, because it does not work in child views of a GridLayout. We can use the layout_gravity attribute instead. Other interesting attributes we can use with

GridLayout child views include layout_column and layout_columnSpan to specify the left-most column and the number of columns the view takes up, respectively. Similarly, there are layout_row and layout_rowSpan attributes. Interestingly, we do not need to specify layout_height and layout_width for GridLayout child views; they default to WRAP_CONTENT.



MODULE IV

ANDROID MENUS

The key class in Android menu support is *android.view.Menu.* Every activity in Android is associated with one menu object of this type. The menu object then contains a number of menu items and submenus. Menu items are represented by *android.view.MenuItem.* Submenus are represented by *android.view.SubMenu.*

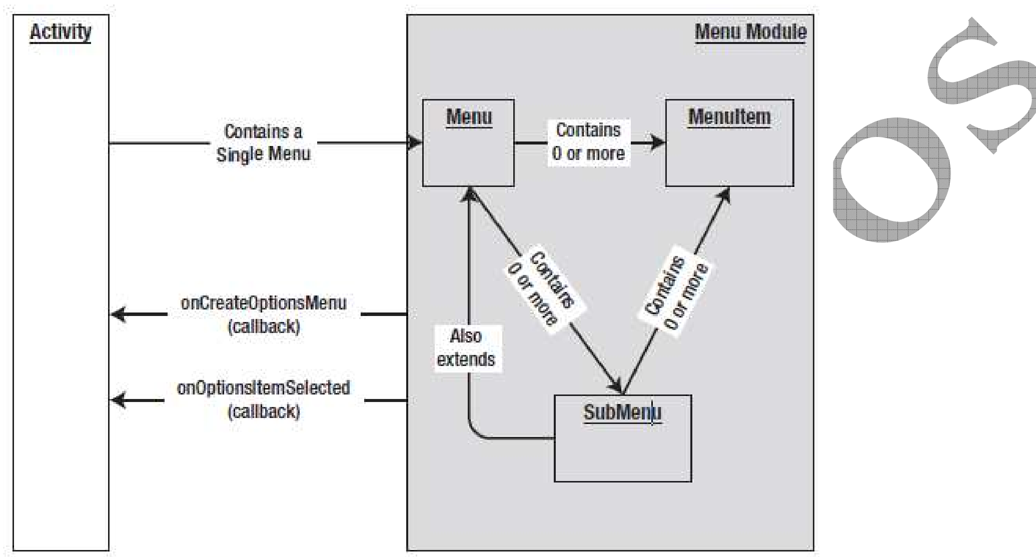


Figure 7–1. *Structure of Android menu-related classes*

A Menu object contains a set of menu items. A menu item carries the following attributes:

- *Name*: A string title
- *Menu item ID*: An integer
- *Group ID*: An integer representing which group this item should be part of
- *Sort order*: An integer identifying the order of this menu item when it is displayed in the menu.

The name and menu item ID attributes are self explanatory. We can group menu items together by assigning each one a group ID. Multiple menu items that carry the same group ID are considered part of the same group. The sort-order attribute demands a bit of coverage. If one menu item carries an order number of 4 and another menu item carries a order number of 6, the first menu item will appear above the second menu item in the menu. Some of these menu item sort-ordernumber ranges are reserved for certain kinds of menus. These are called menu categories. The available menu categories are as follows:

- *Secondary*: Secondary menu items, which are considered less important
- *System*: This sort-order range is reserved for menu items added by the Android system.
- *Alternative*: They're usually contributed by external applications that provide alternative ways to deal with the data that is under consideration.
- *Container*: In Android, the parents of views, such as layouts, are considered containers. The documentation is not clear about whether this

category pertains to layouts, but it is as good a guess as any. Most likely, container-related menu items can be placed in this range.

CREATING MENUS

In the Android SDK, we don't need to create a menu object from scratch. Because an activity is associated with a single menu, Android creates this single menu for that activity and passes it to the *onCreateOptionsMenu()* callback method of the activity class. (As the name of the method indicates, menus in Android are also known as options menus). Starting with 3.0, this method is called as part of activity creation. This change is due to the fact that the action bar is always present in an activity. A menu item that we create in this method for the options menu may sit in an action bar. Because an action bar is always visible (unlike the options menu), the action bar must know its menu items from the beginning. So Android cannot wait until the user opens an options menu to call the *onCreateOptionsMenu()* method. This callback menu setup method allows us to populate the single passed-in menu with a set of menu items (see Listing 7–1).

Listing 7–1. Signature for the onCreateOptionsMenu Method

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
        // populate menu items
        .....
        ...return true;
}
```

Once the menu items are populated, the code should return true to make the menu visible. If this method returns false, the menu is invisible. The code in Listing 7–2 shows how to add three menu items using a single group ID along with incremental menu item IDs and order IDs.

Listing 7–2. Adding Menu Items

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    //call the base class to include system menus
    super.onCreateOptionsMenu(menu);
    menu.add(0 // Group
        ,1 // item id
        ,0 //order
        ,"append"); // title
    menu.add(0,2,1,"item2");
    menu.add(0,3,2,"clear");
    //It is important to return true to see the menu
    return true;
}
```

an double os

WORKING WITH MENU GROUPS

Following code shows how to work with menu groups. Using Group IDs to Create Menu Groups

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
//Group 1
int group1 = 1;
menu.add(group1,1,1,"g1.item1");
menu.add(group1,2,2,"g1.item2");
//Group 2
int group2 = 2;
menu.add(group2,3,3,"g2.item1");
menu.add(group2,4,4,"g2.item2");
return true; // it is important to return true
}
```

Notice how the menu item IDs and the order IDs are independent of the groups. Android provides a set of methods on the *android.view.Menu* class that are based on group IDs. We can manipulate a group's menu items using these methods:

- removeGroup(id):- removes all menu items from that group, given the group ID.
- setGroupCheckable(id, checkable, exclusive):- We can use this method to show a check mark on a menu item when that menu item is selected.
- setGroupEnabled(id,boolean enabled):- We can enable or disable menu items in a given group
- setGroupVisible(id,visible):- we can control the visibility of a group of menu items

RESPONDING TO MENU ITEMS

There are multiple ways of responding to menu item clicks in Android. We can use the *onOptionsItemSelected()* method of the activity class; we can use stand-alone listeners, or we can use intents.

a) Through onOptionsItemSelected()

When a menu item is clicked, Android calls the onOptionsItemSelected() callback method on the Activity class.

Listing 7–4. Signature and Body of the onOptionsItemSelected Method

```
@Override
public boolean onOptionsItemSelected(MenuItem
item) {
        switch(item.getItemId()) {
        .....
        //for items handled
        return true;
        //for the rest
        ...return super.onOptionsItemSelected(item);  }  }
```

an double OS

The key pattern here is to examine the menu item ID through the *getItemId*() method of the *MenuItem* class and do what's necessary. If *onOptionsItemSelected*() handles a menu item, it returns true. The menu event will not be further propagated. For the menu item callbacks that *onOptionsItemSelected*() doesn't deal with, *onOptionsItemSelected*() should call the parent method through super.*onOptionsItemSelected*(). The default implementation of the *onOptionsItemSelected*() method returns false so that the normal processing can take place. Normal processing includes alternative means of invoking responses for a menu.

b) Through Listeners

A listener implies object creation and a registry of the listener. So this is the overhead that the performance refers to in the first sentence of this paragraph. However, we may choose to give more importance to reuse and clarity, in which case listeners provide flexibility. This approach is a two-step process. In the first step, we implement the *OnMenuClickListener* interface. Then, we take an instance of this implementation and pass it to the menu item. When the menu item is clicked, the menu item calls the *onMenuItemClick*() method of the *OnMenuClickListener* interface.

Listing 7–5. Using a Listener as a Callback for a Menu Item Click

```
//Step 1
public class MyResponse implements OnMenuClickListener
{
        //some local variable to work on
        //...
        //Some constructors
        @override
        boolean onMenuItemClick(MenuItem item)
        {
        //do our thing
        return true;
        }
}
//Step 2
MyResponse myResponse = new MyResponse(...);
menuItem.setOnMenuItemClickListener(myResponse);
...
```

The *onMenuItemClick*() method is called when the menu item has been invoked. This code executes as soon as the menu item is clicked, even before the *onOptionsItemSelected*() method is called. If *onMenuItemClick*() returns true, no other callbacks are executed—including the *onOptionsItemSelected*() callback method. This means that the listener code takes precedence over the *onOptionsItemSelected*() method.

an double os

c) Using Intent

We can also associate a menu item with an intent by using the *MenuItem's* method *setIntent(intent)*. By default, a menu item has no intent associated with it. But when an intent is associated with a menu item, and nothing else handles the menu item, then the default behavior is to invoke the intent using *startActivity*(*intent*).

ICON MENU

Android supports not only text but also images or icons as part of its menu repertoire. We can use icons to represent menu items instead of and in addition to text. Note a few limitations when it comes to using icon menus.

1) we can't use icon menus for expanded menus. This restriction may be lifted in the future, depending on device size and SDK support. Larger devices may allow this functionality, whereas smaller devices may keep the restriction.
2) Icon menu items do not support menu item check marks.
3) If the text in an icon menu item is too long, it's truncated after a certain number of characters, depending on the size of the display. (This last limitation applies to text based menu items also).

Creating an icon menu item is straightforward. We create a regular text-based menu item as before, and then we use the *setIcon*() method on the *MenuItem* class to set the image. We need to use the image's resource ID, so we must generate it first by placing the image or icon in the /res/*drawable* directory. For example, if the icon's file name is balloons, then the resource ID is *R.drawable.balloons.*

SUB MENU

A Menu object can have multiple *SubMenu* objects. Each *SubMenu* object is added to the Menu object through a call to the *Menu.addSubMenu*() method. We add menu items to a submenu the same way that we add menu items to a menu. This is because *SubMenu* is also derived from a Menu object. However, we cannot add additional submenus to a submenu.

Listing 7–7. Adding Submenus

```
private void addSubMenu(Menu menu)
{
    //Secondary items are shown just like everything else
    int base=Menu.FIRST + 100;
    SubMenu sm =
    menu.addSubMenu(base,base+1,Menu.NONE,"submenu");
    sm.add(base,base+2,base+2,"sub item1");
    sm.add(base,base+3,base+3,"sub item2");
    sm.add(base,base+4,base+4,"sub item3");
    //submenu item icons are not supported
    item1.setIcon(R.drawable.icon48x48_2);
```

androidos

```
        //the following is ok however
        sm.setIcon(R.drawable.icon48x48_1);
        //This will result in runtime exception
        //sm.addSubMenu("try this");
    }
```

NOTE: *SubMenu*, as a subclass of the Menu object, continues to carry the *addSubMenu*() method. The compiler won't complain if we add a submenu to another submenu, but we'll get a runtime exception if we try to do it.

The Android SDK documentation also suggests that submenus do not support icon menu items. When we add an icon to a menu item and then add that menu item to a submenu, the menu item ignores that icon, even if we don't see a compile-time or runtime error. However, the submenu itself can have an icon.

CONTEXT MENU

In Windows applications, for example, we can access a context menu by right-clicking a UI element. Android supports the same idea of context menus through an action called a long click. A long click is a mouse click held down slightly longer than usual on any Android view. On handheld devices such as cell phones, mouse clicks are implemented in a number of ways, depending on the navigation mechanism. If our phone has a wheel to move the cursor, a press of the wheel serves as the mouse click. Or if the device has a touch pad, a tap or a press is equivalent to a mouse click. Or we might have a set of arrow buttons for movement and a selection button in the middle; clicking that button is equivalent to clicking the mouse. Regardless of how a mouse click is implemented on our device, if we hold the mouse click a bit longer, we realize the long click.

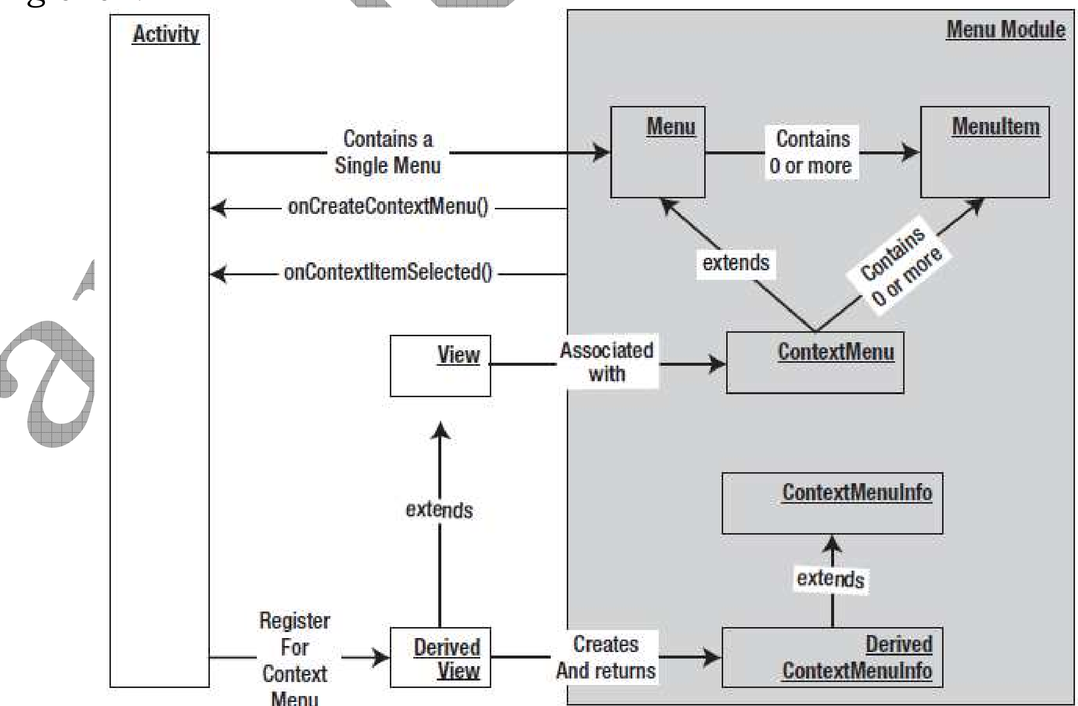


Figure 7–3. *Activities, views, and context menus*

a-double-os

Although a context menu is owned by a view, the method to populate context menus resides in the Activity class. This method is called activity.onCreateContextMenu(), and its role resembles that of the activity.onCreateOptionsMenu() method. This callback method also carries with it (as an argument to the method) the view for which the context menu items are to be populated. The steps to implement a context menu:

1) Register a view for a context menu in an activity's onCreate() method.
2) Populate the context menu using onCreateContextMenu(). We must complete step 1 before this callback method is invoked by Android.
3) Respond to context menu clicks.

DYNAMIC MENUS

If we want to create dynamic menus, use the *onPrepareOptionsMenu*() method that Android provides on an activity class. This method resembles *onCreateOptionsMenu*() except that it is called every time a menu is invoked. We should use *onPrepareOptionsMenu*() if we want to disable some menu items or menu groups based on what we are displaying. For 3.0 and above, we have to explicitly call a new provisioned method called *invalidateOptionsMenu*(), which in turn invokes the *onPrepareOptionsMenu*(). We can call this method any time something changes in our application state that would require a change to the menu.

LOADING MENU THROUGH XML

We have created all our menus programmatically. This is not the most convenient way to create menus, because for every menu, we have to provide several IDs and define constants for each of those IDs. No doubt this is tedious. Instead, we can define menus through XML files, which is possible in Android because menus are also resources. The XML approach to menu creation offers several advantages, such as

- he ability to name menus
- Order them automatically
- Give them IDs
- We can also get localization support for the menu text.

Follow these steps to work with XML-based menus:

1. Define an XML file with menu tags.
2. Place the file in the /res/menu subdirectory. The name of the file is arbitrary, and we can have as many files as we want. Android automatically generates a resource ID for this menu file.
3. Use the resource ID for the menu file to load the XML file into the menu.
4. Respond to the menu items using the resource IDs generated for each menu item.

POPUP MENUS

SDK 4.0 enhanced this slightly by adding a couple of utility methods (for example, *PopupMenu.inflate*) to the *PopupMenu* class. A pop-up menu can be invoked against any view in response to a UI event. An example of a UI event is a

an double OS

button click or a click on an image view. Figure 7–4 shows a pop-up menu invoked against a view.

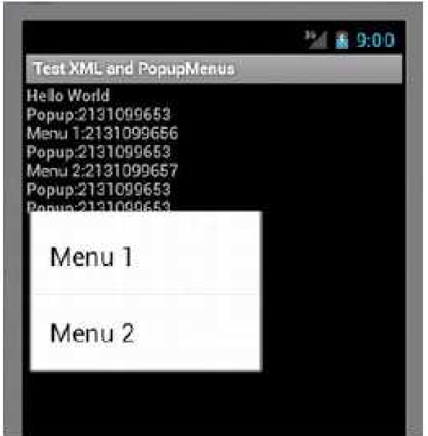
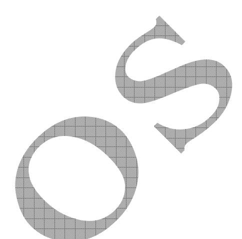


Figure 7–4. *Pop-up menu attached to a text view*



To create a pop-up menu like the one in Figure 7–4, start with a regular XML menu file as shown in Listing 7–18.

Listing 7–18. A Sample XML File for a Pop-up Menu

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- This group uses the default category. -->
    <group android:id="@+id/menuGroup_Popup">
        <item android:id="@+id/popup_menu_1"
            android:title="Menu 1" />
        <item android:id="@+id/popup_menu_2"
            android:title="Menu 2" />
    </group>
</menu>
```

Assuming the code in Listing 7–18 is in a file called popup_menu.xml. As we can see, a pop-up menu behaves much like an options menu. The key differences are as follows:

- A pop-up menu is used on demand, whereas an options menu is always available.
- A pop-up menu is anchored to a view, whereas an options menu belong to the entire activity.
- A pop-up menu uses its own menu item callback, whereas the options menu uses the onOptionsItemSelected() callback on the activity.

FRAGMENTS IN ANDROID

A fragment is a piece of activity which enable more modular activity design. It will not be wrong if we say a fragment is a kind of sub-activity. A fragment has its own layout and behavior with life cycle. We can add or remove fragments in an activity while the activity is running. We can combine multiple fragments in a single activity to build a multi-plane UI. A fragment can be used in multiple activities. A fragment can implement a behavior that has no user

interface component. Fragments were added to the android version HoneyComb with API 11.0

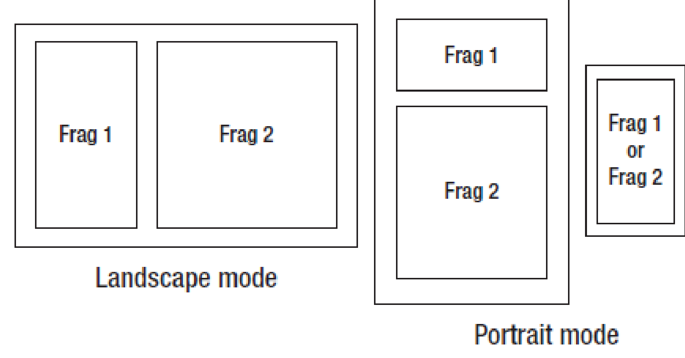


Figure 8–1. *Fragments used for a tablet UI and for a smartphone UI*

STRUCTURE OF FRAGMENT

A fragment is like a sub-activity: it has a fairly specific purpose and almost always displays a user interface. But where an activity is sub-classed below Context, a fragment is extended from Object in package android.app. A fragment is not an extension of Activity. A fragment can have a view hierarchy to engage with a user. It can be created (inflated) from an XML layout specification or created in code. A fragment has a bundle that serves as its initialization arguments.

FRAGMENT LIFE CYCLE

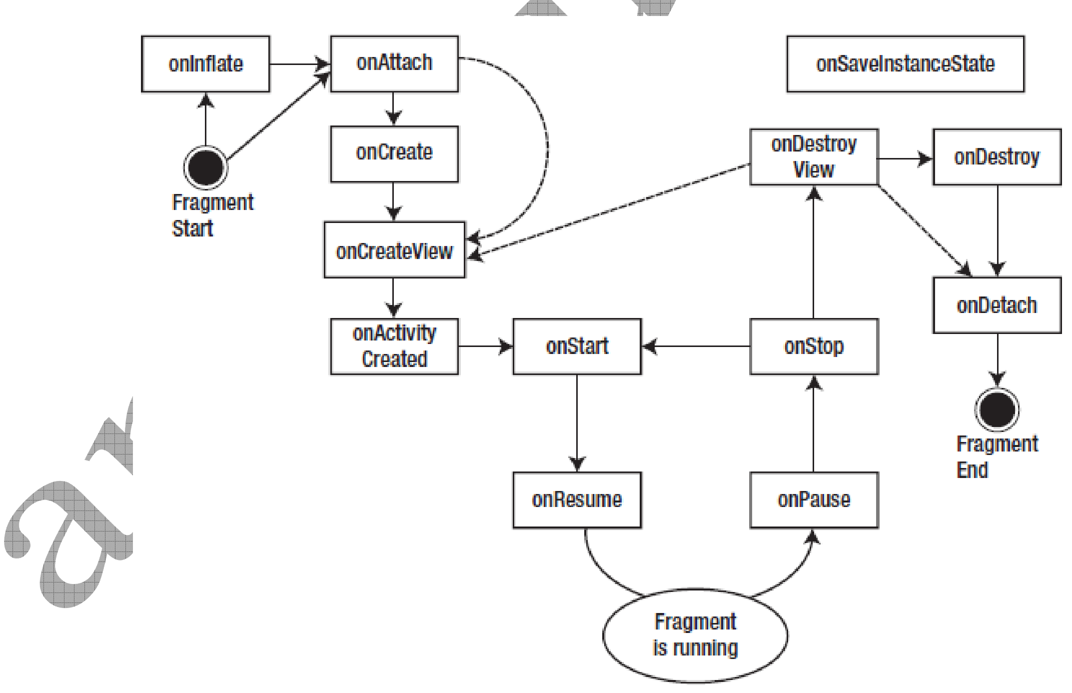


Figure 8–2. *Lifecycle of a fragment*

A fragment is very dependent on the activity in which it lives and can go through multiple steps while its activity goes through one.

Instance	Description
onAttach()	The fragment instance is associated with an activity instance. The fragment and the activity are not fully initialized. In this method a reference to the fragment for further initialization works.
onCreate()	The system calls this method when creating the fragment. The onCreate() method is called after the onCretae() method on the activity but before the onCreateView() method of the fragment. We should initialize essential component of the fragments that we want to retain when the fragment is paused or stopped then resumed.
onCreateView()	This method creates user interface. In this method we should not interactive with the activity. To draw a UI for our fragment we must turn a view from this method. If we return null, the fragment does not provide a UI.
onActivityCreated()	This is the point where host activity gets created. This is important to notice that this method is called after onCreateView() to indicate that the activity's onCreate() has completed.
onStart()	This is called once the fragment gets visible.
onResume()	From this point onwards fragment becomes active.
onPause()	The system calls this method as the first indication that the user is leaving the fragment. The fragment is visible nut becomes not active any more. If any other activity is animated on top of the activity which contains the fragment.
onStop()	The fragment becomes not visible and going to stopped
onDestroyView()	It destroys the view of the fragment. This is the counter part of onCreateView() where we set up the UI. If the fragment is recreated from the back stack then onCreateView() is called.
onDestroy()	This method does the final cleanup related to the fragment state. But is not guaranteed to get called by Android Platform
onDitach()	It is called after onDistroy() to notify that the fragment has been disassociated from its hosting activity

an double OS

FRAGMENT TRANSACTION AND BACK STACK

Fragment transaction is a sequence of steps to add, replace, remove or perform other action in response to user interaction. Each set of changes that we commit to the activity is called transaction. And we can perform one using API in fragment transition. We can also save each transaction to a back stack managed by activity allowing the user to navigate backward through the fragment changes. Each transaction is a set of changes that we want to perform at the same time. We can set up all the changes we want to perform for a given

transaction using method such as add(), replace() and remove(). Then to apply the transaction to the activity we must call commit().

Before we call commit(), how ever we might want to call addToBackStack() method in order to add the transaction to the back stack of fragment transactions. This back stack is managed by the activity and allows the user to return to the previous fragment state, by pressing the back button.

FRAGMENT MANAGER

The FragmentManager is a component that takes care of the fragments belonging to an activity. This includes fragments on the back stack and fragments that may just be hanging around. Fragments should only be created within the context of an activity. The FragmentManager class is used to access and manage these fragments for an activity. Besides getting a fragment transaction, we can also get a fragment using the fragment's ID, its tag, or a combination of bundle and key. For this, the getter methods include findFragmentById(), findFragmentByTag(), and getFragment(). The getFragment() method would be used in conjunction with putFragment(), which also takes a bundle, a key, and the fragment to be put.

The bundle is most likely going to be the savedState bundle, and putFragment() will be used in the onSaveInstanceState() callback to save the state of the current activity (or another fragment). The getFragment() method would probably be called in onCreate() to correspond to putFragment(), although for a fragment, the bundle is available to the other callback methods, as described earlier. Obviously, we can't use the getFragmentManager() method on a fragment that has not been attached to an activity yet. But it's also true that we can attach a fragment to an activity without making it visible to the user yet. If we do this, we should associate a String tag to the fragment so we can get to it in the future. We'd most likely use this method of FragmentTransaction to do this:

public FragmentTransaction add (Fragment fragment, String tag)

The fragment back stack is also the domain of the fragment manager. Whereas a fragment transaction is used to put fragments onto the back stack, the fragment manager can take fragments off the back stack. This is usually done using the fragment's ID or tag, but it can be done based on position in the back stack or just to pop the top-most fragment. Finally, the fragment manager has methods for some debugging features, such as turning on debugging messages to LogCat using enableDebugLogging() or dumping the current state of the fragment manager to a stream using dump().

SAVING FRAGMENT STATE

Another interesting class was introduced in Android 3.2: Fragment.SavedState. Using the saveFragmentInstanceState() method of FragmentManager, we can pass this method a fragment, and it returns an object

an double os

representing the state of that fragment. We can then use that object when initializing a fragment, using Fragment's setInitialSavedState() method.

PERSISTENCE OF FRAGMENTS

When we play with this sample application, make sure we rotate the device (pressing Ctrl+F11 rotates the device in the emulator). We will see that the device rotates, and the fragments rotate right along with it. If we watch the LogCat messages, we will see a lot of them for this application. During a device rotation, pay careful attention to the messages about fragments; not only does the activity get destroyed and recreated, but the fragments do also.

So far, we only wrote a tiny bit of code on the titles fragment to remember the current position in the titles list across restarts. We didn't do anything in the details fragment code to handle reconfigurations, and that's because we didn't need to. Android will take care of hanging onto the fragments that are in the fragment manager, saving them away, and then restoring them when the activity is being re-created. We should realize that the fragments we get back after the reconfiguration is complete are very likely not the same fragments in memory that we had before. These fragments have been reconstructed for we. Android saved the arguments bundle and the knowledge of which type of fragment it was, and it stored the saved-state bundles for each fragment that contain saved-state information about the fragment to use to restore it on the other side.

COMMUNICATIONS WITH FRAGMENTS

The fragment manager knows about all fragments attached to the current activity, the activity or any fragment in that activity can ask for any other fragment using the getter methods described earlier. Once the fragment reference has been obtained, the activity or fragment could cast the reference appropriately and then call methods directly on that activity or fragment. This would cause our fragments to have more knowledge about the other fragments than might normally be desired, but don't forget that we're running this application on a mobile device, so cutting corners can sometimes be justified.
A code snippet is provided in Listing 8–12 to show how one fragment might communicate directly with another fragment.
Listing 8–12. Direct Fragment-to-Fragment Communication

```
FragmentOther fragOther =
(FragmentOther)getFragmentManager().findFragmentByTag("other");
fragOther.callCustomMethod( arg1, arg2 );
```

In Listing 8–12, the current fragment has direct knowledge of the class of the other fragment and also which methods exist on that class. This may be okay because these fragments are part of one application, and it can be easier to simply accept the fact that some fragments will know about other fragments.

an double os

STARTACTIVITY() AND SETTARGETFRAGMENT()

A feature of fragments that is very much like activities is the ability of a fragment to stat an activity. Fragment has a startActivity() method and startActivityForResult() method. These work just like the ones for activities; when a result is passed back, it will cause the onActivityResult() callback to fire on the fragment that started the activity. There's another communication mechanism we should know about. When one fragment wants to start another fragment, there is a feature that lets the calling fragment set its identity with the called fragment. The following example of what it might look like.

Listing 8–13. Fragment-to-Target-Fragment Setup

```
mCalledFragment = new CalledFragment();
mCalledFragment.setTargetFragment(this, 0);
fm.beginTransaction().add(mCalledFragment, "work").commit();
```

With these few lines, we've created a new CalledFragment object, set the target fragment on the called fragment to the current fragment, and added the called fragment to the fragment manager and activity using a fragment transaction. When the called fragment starts to run, it will be able to call getTargetFragment(), which will return a reference to the calling fragment. With this reference, the called fragment could invoke methods on the calling fragment or even access view components directly. For example, the called fragment could set text in the UI of the calling fragment directly.

Listing 8–14. Target Fragment-to-Fragment Communication

```
TextView tv = (TextView)
getTargetFragment().getView().findViewById(R.id.text1);
tv.setText("Set from the called fragment");
```

USING DIALOGS IN ANDROID

The Android SDK offers extensive support for dialogs. A dialog is a smaller window that pops up in front of the current window to show an urgent message, to prompt the user for a piece of input, or to show some sort of status like the progress of a download. The user is generally expected to interact with the dialog and then return to the window underneath to continue with the application. Android allows a dialog fragment to also be embedded within an activity's layout. Dialogs that are explicitly supported in Android include the alert, prompt, pick-list, single-choice, multiple-choice, progress, time-picker, and date-picker dialogs.

Dialogs in Android are asynchronous, which provides flexibility. However, if we are accustomed to a programming framework where dialogs are primarily synchronous (such as Microsoft Windows, or JavaScript dialogs in web pages), we might find asynchronous dialogs a bit unintuitive. With a synchronous dialog, the line of code after the dialog is shown does not run until the dialog has been dismissed. This means the next line of code could interrogate which button was pressed, or what text was typed into the dialog. In

an double OS

Android however, dialogs are asynchronous. As soon as the dialog has been shown, the next line of code runs, even though the user hasn't touched the dialog yet. Our application has deal with this fact by implementing callbacks from the dialog, to allow the application to be notified of user interaction with the dialog.

This also means our application has the ability to dismiss the dialog from code, which is powerful. If the dialog is displaying a busy message because our application is doing something, as soon as our application has completed that task, it can dismiss the dialog from code.

DIALOG FRAGMENTS

The use of dialog fragments is to present a simple alert dialog and a custom dialog that is used to collect prompt text. Dialog-related functionality uses a class called DialogFragment. A DialogFragment is derived from the class Fragment and behaves much like a fragment. We will then use the DialogFragment as the base class for our dialogs. Once we have a derived dialog from this class such as

```
public class MyDialogFragment extends DialogFragment { ... }
```

we can then show this dialog fragment MyDialogFragment as a dialog using a fragment transaction. Following example shows a code snippet to do this.

Listing 9–1. Showing a Dialog Fragment

```
SomeActivity
{
//....other activity functions
public void showDialog()
{
//construct MyDialogFragment
MyDialogFragment mdf = MyDialogFragment.newInstance(arg1,arg2);
FragmentManager fm = getFragmentManager();
FragmentTransaction ft = fm.beginTransaction();
mdf.show(ft,"my-dialog-tag");
}
//....other activity functions
}
```

The steps to show a dialog fragment are as follows:

1. Create a dialog fragment.
2. Get a fragment transaction.
3. Show the dialog using the fragment transaction from step 2.

WORKING WITH TOAST

The alert messages are commonly used for debugging JavaScript on error pages. If we are pressed to use a similar approach for infrequent debug messages, we can use the Toast object in Android. A Toast is like an alert dialog that has a message and displays for a certain amount of time and then goes

androidOS

away. It does not have any buttons. So it can be said that it is a transient alert message. It's called Toast because it pops up like toast out of a toaster. The following example shows an example of how we can show a message using Toast.

Listing 9–10. Using Toast for Debugging

```
//Create a function to wrap a message as a toast
//show the toast
public void reportToast(String message)
{
String s = MainActivity.LOGTAG + ":" + message;
Toast.makeText(activity, s, Toast.LENGTH_SHORT).show();
}
```

The makeText() method in Listing 9–10 can take not only an activity but any context object, such as the one passed to a broadcast receiver or a service, for example. This extends the use of Toast outside of activities.

IMPLEMENTING ACTION BAR

ActionBar was introduced in the Android 3.0 SDK for tablets and is now available for phones as well in 4.0. It allows we to customize the title bar of an activity. Prior to the 3.0 SDK release, the title bar of an activity merely contained the title of an activity. Android ActionBar is modeled similar to the menu/title bar of a web browser. An action bar is owned by an activity and follows its lifecycle. An action bar can take one of three forms: tabbed action bar, list action bar, or standard action bar. We see how these various action bars look and behave in each of the modes.

Home Icon area: The icon at upper left on the action bar is sometimes called the Home icon. This is similar to a web site navigation context, where clicking the Home icon takes we to a starting point. When we transfer the user to the home activity, don't start a new home activity; instead, transfer to it by using an intent flag that clears the stack of all activities on top of the home activity. We see later that clicking this Home icon sends a callback to the option menu with menu ID android.R.id.home.

- Title area: The Title area displays the title for the action bar.
- Tabs area: The Tabs area is where the action bar paints the list of tabs specified. The content of this area is variable. If the action bar navigation mode is tabs, then tabs are shown here. If the mode is listnavigation mode, then a navigable list of drop-down items is shown. In standard mode, this area is ignored and left empty.
- Action Icon area: Following the Tabs area, the Action Icon area shows some of the option menu items as icons. We see how to choose which option menus are displayed as action icons in the example later.
- Menu Icon area: Last is the Menu Icon area. It is a single standard menu icon. When we click this menu icon, we see the expanded menu. This expanded menu looks different or shows up in a different location

android OS

depending on the size of the Android device. We can also attach a search view as if it is an action icon of the menu.

Figure 10–1 shows a typical action bar in tabbed navigation mode.

Figure 10–1. *An activity with a tabbed action bar*

TABBED NAVIGATION ACTION BAR ACTIVITY

Each of the action bar divided into separate tabs. Each tab contains their own activities. The common behavior in a base class and allow each of the derived activities, including this tabbed action bar activity, to configure the action bar. The difficult to explain these common files without the context of at least one action bar activity. Following is a list of files that are needed for this tabbed action bar:

- DebugActivity.java: Base class activity that allows for a debug text view as shown in Figure 10-1 (Listing 10-2)
- BaseActionBarActivity.java: Derived from DebugActivity and allows for common navigation (such as responding to common actions including switching between the three activities) (Listing 10-3)
- IReportBack.java: An interface that works as a communication vehicle between the debug activity and the various listeners of the action bar (Listing 10-1)
- BaseListener.java: Base listener class that works with the DebugActivity and the various actions that gets invoked from the action bar. Acts as a base class for both tab listeners and list navigation listeners (Listing 10-4)
- TabNavigationActionBarActivity.java: inherits from BaseActionBarActivity.java and configures the action bar as a tabbed

action bar. Most of the code pertaining to the tabbed action bar is in this class (Listing 10-6)

- TabListener.java: Required to add a tab to the tabbed action bar. This where we respond to tab clicks. In our case this simply logs a message to the debug view through the BaseListener (Listing 10-5)
- AndroidManifest.xml: Where activities are defined to be invoked (Listing 10-13)
- Layout/main.xml: Layout file for the DebugActivity. Because all the three status bar activities inherit this base DebugActivity, they all share this layout file (Listing 10-7)
- menu/menu.xml: A set of menu items to test the menu interaction with the action bar. The menu file is also shared across all the derived status bar activities (Listing 10-9)

IMPLEMENTING BASE ACTIVITY CLASSES

A number of the base classes use the IReportBack interface. It is introduced in

Listing 10–1. IReportBack.java

```
//IReportBack.java
package com.androidbook.actionbar;
public interface IReportBack
{
	public void reportBack(String tag, String message);
	public void reportTransient(String tag, String message);
}
```

A class that implements this interface takes a message and reports it on a screen, like a debug message. This is done through the reportBack() method. The method reportTransient() does the same thing except it uses a Toast to report that message to the user. In this example, the class that implements IReportBack is DebugActivity. This allows DebugActivity to pass itself around without exposing all of its internals. The source code for DebugActivity is presented in

Listing 10–2. DebugActivity with a Debug Text View

```
//DebugActivity.java
package com.androidbook.actionbar;
//
//Use CTRL-SHIFT-O to import dependencies
//
public abstract class DebugActivity extends Activity implements IReportBack
{
//Derived classes needs first protected abstract boolean
onMenuItemSelected(MenuItem item);
//private variables set by constructor
private static String tag=null;
private int menuId = 0;
private int layoutid = 0;
private int debugTextViewId = 0;
```

an double os

```
public DebugActivity(int inMenuId, int inLayoutId, int inDebugTextViewId,
String inTag)
{
tag = inTag;
menuId = inMenuId;
layoutid = inLayoutId;
debugTextViewId = inDebugTextViewId;
}
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(this.layoutid);
//We need the following to be able to scroll
//the text view.
TextView tv = this.getTextView();
tv.setMovementMethod(
ScrollingMovementMethod.getInstance());
}
@Override
public boolean onCreateOptionsMenu(Menu menu){
super.onCreateOptionsMenu(menu);
MenuInflater inflater = getMenuInflater();
inflater.inflate(menuId, menu);
return true;
}
@Override
public boolean onOptionsItemSelected(MenuItem item){
appendMenuItemText(item);
if (item.getItemId() == R.id.menu_da_clear){
this.emptyText();
return true;
}
boolean b = onMenuItemSelected(item);
if (b == true)
{
return true;
}
return super.onOptionsItemSelected(item);
}
protected TextView getTextView(){
return
(TextView)this.findViewById(this.debugTextViewId);
}
protected void appendMenuItemText(MenuItem menuItem){
String title = menuItem.getTitle().toString();
appendText("MenuItem:" + title);
}
```

an double os

```
protected void emptyText(){
TextView tv = getTextView();
tv.setText("");
}
protected void appendText(String s){
TextView tv = getTextView();
tv.setText(s + "\n" + tv.getText());
Log.d(tag,s);
}
public void reportBack(String tag, String message)
{
this.appendText(tag + ":" + message);
Log.d(tag,message);
}
public void reportTransient(String tag, String message)
{
String s = tag + ":" + message;
Toast mToast =
Toast.makeText(this, s, Toast.LENGTH_SHORT);
mToast.show();
reportBack(tag,message);
Log.d(tag,message);
}
}//eof-class
```

The primary goal of this base activity class is to present an activity with a debug text view in it. This text view is used to log messages coming from the reportBack() method. We use this activity as the base activity for all of the action bar activities.

TABBED LISTENER

Before we are able to work with a tabbed action bar, we need a tabbed listener. A tabbed listener allows we to respond to the click events on the tabs. We derive our tabbed listener from a base listener that allows we to log tab actions. Listing 10–4 shows the base listener that uses the IReportBack for logging.

Listing 10–4. A Common Listener for Action Bar Enabled Activities

```
//BaseListener.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class BaseListener
{
protected IReportBack mReportTo;
protected Context mContext;
public BaseListener(Context ctx, IReportBack target)
{
mReportTo = target;
mContext = ctx;  }   }
```

an double os

This base class holds a reference to an implementation of IReportBack and also the activity that can be used as a context. This tabbed listener documents the callbacks from the action bar tabs to the debug text. In this case, the DebugActivity from Listing 10–2 is the implementer of IReportBack and also plays the role of the context. Now that we have a base listener, Listing 10–5 shows the tabbed listener.

Listing 10–5. Tab Listener to Respond to Tab Actions

```
// TabListener.java
package com.androidbook.actionbar;
//
//Use CTRL-SHIFT-O to import dependencies
//
public class TabListener extends BaseListener
implements ActionBar.TabListener
{
private static String tag = "tc>";
public TabListener(Context ctx,
IReportBack target)
{
super(ctx, target);
}
public void onTabReselected(Tab tab,
FragmentTransaction ft)
{
this.mReportTo.reportBack(tag,
"ontab re selected:" + tab.getText());
}
public void onTabSelected(Tab tab,
FragmentTransaction ft)
{
this.mReportTo.reportBack(tag,
"ontab selected:" + tab.getText());
}
public void onTabUnselected(Tab tab,
FragmentTransaction ft)
{
this.mReportTo.reportBack(tag,
"ontab un selected:" + tab.getText());
}
}
```

This tabbed listener documents the callbacks from the action bar tabs to the debug text view of Figure 10–1.

TABBED ACTION BAR

With the tabbed listener in place, we can finally construct the tabbed navigation activity. This is presented in Listing 10–6.

an double os

Listing 10–6. Tab-Navigation Enabled Action Bar Activity

```
// TabNavigationActionBarActivity.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class TabNavigationActionBarActivity
extends BaseActionBarActivity
{
private static String tag =
"Tab Navigation ActionBarActivity";
public TabNavigationActionBarActivity()
{
super(tag);
}
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
workwithTabbedActionBar();
}
public void workwithTabbedActionBar()
{
ActionBar bar = this.getActionBar();
bar.setTitle(tag);
bar.setNavigationMode(
ActionBar.NAVIGATION_MODE_TABS);
TabListener tl = new TabListener(this,this);
Tab tab1 = bar.newTab();
tab1.setText("Tab1");
tab1.setTabListener(tl);
bar.addTab(tab1);
Tab tab2 = bar.newTab();
tab2.setText("Tab2");
tab2.setTabListener(tl);
bar.addTab(tab2);
}
}//eof-class
```

We now look at the code for this activity (Listing 10–6) in the following subsections, which draw attention to each aspect of working with a tabbed action bar. We start with getting access to the action bar belonging to an activity.

DEBUG TEXT VIEW LAYOUT

As the tabs of the action bar are clicked, the tab listeners are set up in such a way that debug messages are sent to the debug text view. Listing 10–7 shows the layout file for the DebugActivity, which in turn contains the debug text view.

Listing 10–7. Debug Activity Text View Layout File

an double os

```
<?xml version="1.0" encoding="utf-8"?>
<!-- /res/layout/main.xml -->
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:gravity="fill"
        >
<TextView android:id="@+id/textViewId"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="@android:color/white"
        android:text="Initial Text Message"
        android:textColor="@android:color/black"
        android:textSize="25sp"

        android:scrollbars="vertical"
        android:scrollbarStyle="insideOverlay"
        android:scrollbarSize="25dip"
        android:scrollbarFadeDuration="0"
        />
</LinearLayout>
```

There are a few things worth noting about this layout. We set the background color of the text view to white. This lets we capture screens in brighter light. The text size is also set to a large font to aid screen capture. We also set up the text view so that it is enabled for scrolling. Although typically layouts use ScrollView, a text view is already enabled for scrolling by itself. In addition to enabling the scrolling properties in the XML file for the text view, we need to call the setMovementMethod() method on the text view as shown in

Listing 10–8. Enabling Text View for Scrolling

```
TextView tv = this.getTextView();
tv.setMovementMethod(
ScrollingMovementMethod.getInstance());
```

This code is extracted from the DebugActivity (Listing 10–2).

As the text view is scrolled, notice that the scrollbar appears and then fades away. This is not a good indicator if there is text beyond visible range. We can tell the scrollbar to stay by setting the fade duration to 0. See Listing 10–7 for how to set this parameter.

an double OS

ACTION BAR AND MENU INTERACTION

This example also demonstrates how menus interact with the action bar. So, we need to set up a menu file. This file is presented in Listing 10–9.

Listing 10–9. Menu XML File for This Project

```
<!-- /res/menu/menu.xml -->
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
<!-- This group uses the default category. -->
<group android:id="@+id/menuGroup_Main">
<item android:id="@+id/menu_action_icon1"
        android:title="Action Icon1"
        android:icon="@drawable/creep001"
        android:showAsAction="ifRoom"/>
<item android:id="@+id/menu_action_icon2"
        android:title="Action Icon2"
        android:icon="@drawable/creep002"
        android:showAsAction="ifRoom"/>
<item android:id="@+id/menu_icon_test"
        android:title="Icon Test"
        android:icon="@drawable/creep003"/>
<item android:id="@+id/menu_invoke_listnav"
        android:title="Invoke List Nav"
        />
<item android:id="@+id/menu_invoke_standardnav"
        android:title="Invoke Standard Nav"
        />
<item android:id="@+id/menu_invoke_tabnav"
        android:title="Invoke Tab Nav"
        />
<item android:id="@+id/menu_da_clear"
        android:title="clear" />
</group>
</menu>
```

an double os

Figure 10–2. *An activity with a tabbed action bar and expanded menu*

LIST NAVIGATION ACTION BAR ACTIVITY

Because our base classes are carrying the most of the work, it is fairly easy to implement and test the list action bar navigation activity. We need the following additional files to implement this activity:

- SimpleSpinnerArrayAdapter.java: Needed to set up the list navigation bar along with the listener. This class provides the rows required by a drop-down navigation list (Listing 10–12).
- ListListener.java: Acts as a listener to the list navigation activity. This class needs to be passed to the action bar when setting it up as a list action bar (Listing 10–13).
- ListNavigationActionBarActivity.java: Implements the list navigation action bar activity (Listing 10–14).

Once we have these three new files, we need to update the following two files:

- BaseActionBarActivity.java: Uncomment the invocation of the list action bar activity (Listing 10–3).
- AndroidManifest.xml: Define the new list navigation action bar activity in the manifest file (Listing 10–11).

SPINNER ADAPTER

To be able to initialize the action bar with list navigation mode, we need the following two things:

- A spinner adapter that can tell the list navigation what the list of navigation text is
- A list navigation listener so that when one of the list items is picked we can get a call back

Listing 10–12 presents the SimpleSpinnerArrayAdapter that implements the SpinnerAdapter interface. the goal of this class is to give a list of items to show.

Listing 10–12. Creating a Spinner Adapter for List Navigation

```
//SimpleSpinnerArrayAdapter.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class SimpleSpinnerArrayAdapter
extends ArrayAdapter<String>
implements SpinnerAdapter
{
public SimpleSpinnerArrayAdapter(Context ctx)
{
super(ctx,
android.R.layout.simple_spinner_item,
new String[]{"one","two"});
this.setDropDownViewResource(
android.R.layout.simple_spinner_dropdown_item);
}
public View getDropDownView(
int position, View convertView, ViewGroup parent)
{
return super.getDropDownView(
position, convertView, parent);
}
}
```

There is no SDK class that directly implements the SpinnerAdapter interface required by list navigation. So, we derive this class from an ArrayAdapter and provide a simple implementation for the SpinnerAdapter. At the end of the chapter is a reference URL on spinner adapters for further reading. Let's move on now to the list navigation listener.

LIST LISTENER

This is a simple implementing the ActionBar.OnNavigationListener. Listing 10–13 shows the code for this class.

Listing 10–13. Creating a List Listener for List Navigation

```
//ListListener.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class ListListener
extends BaseListener
```

ali double os

```
implements ActionBar.OnNavigationListener
{
public ListListener(
Context ctx, IReportBack target)
{
super(ctx, target);
}
public boolean onNavigationItemSelected(
int itemPosition, long itemId)
{
this.mReportTo.reportBack(
"list listener","ItemPostion:" + itemPosition);
return true;
}
}
```

Like the tabbed listener in Listing 10–5, we inherit from our BaseListener so that we can log events to the debug text view through the IReportBack interface.

LIST ACTION BAR

We now have what we require to set up a list navigation action bar. The source code for the list navigation action bar activity is shown in Listing 10–14. This class is very similar to the tabbed activity we coded earlier.

Figure 10–3. *An activity with a list navigation action bar*

Listing 10–14. List Navigation Action Bar Activity

```
// ListNavigationActionBarActivity.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class ListNavigationActionBarActivity
extends BaseActionBarActivity
{
private static String tag=
"List Navigation ActionBarActivity";
public ListNavigationActionBarActivity()
{
super(tag);
}
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
workwithListActionBar();
}
public void workwithListActionBar()
{
ActionBar bar = this.getActionBar();
bar.setTitle(tag);
bar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
bar.setListNavigationCallbacks(
new SimpleSpinnerArrayAdapter(this),
new ListListener(this,this));
}
}//eof-class
```

The important code is highlighted in Listing 10–14. The code is quite simple: we take a spinner adapter and a list listener and set them as list navigation callbacks on the action bar.

an double os

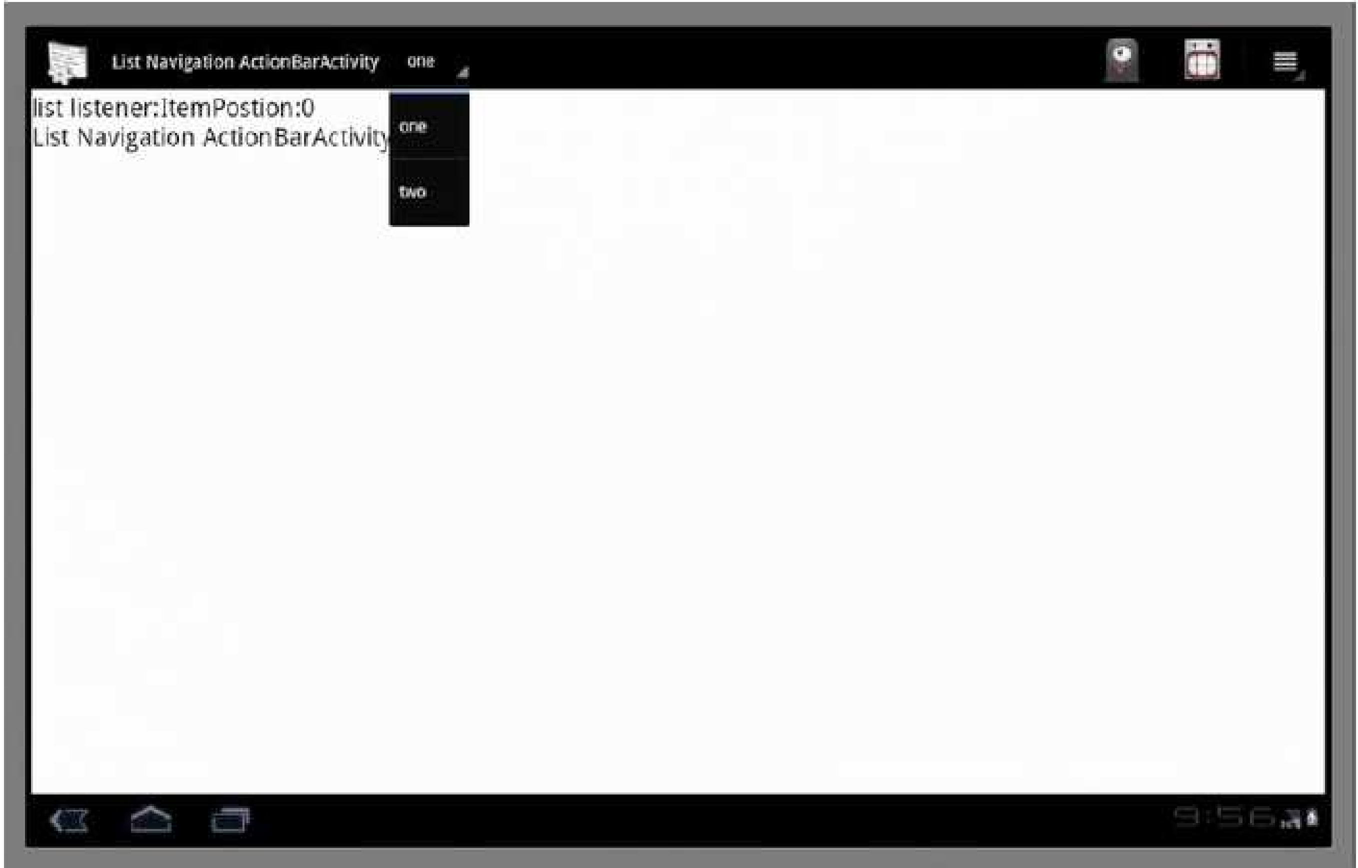


Figure 10–4. *An activity with an opened navigation list*

STANDARD NAVIGATION ACTION BAR ACTIVITY

The nature of a standard navigation action bar. We set up an activity and set its action bar navigation mode as standard. We then see what the standard navigation looks like and examine its behavior. As in the case of ListNavigationActionBarActivity, because our base classes are carrying most of the work, it is easy to implement and test the standard action bar navigation activity. We need the following additional file:

- StandardNavigationActionBarActivity.java: This is the implementation file for configuring the action bar as a standard navigation mode action bar (Listing 10–17).

We also need to update the following two files:

- BaseActionBarActivity.java: Uncomment the invocation of the standard action bar activity in response to a menu item (see Listing 10–18 for changes and Listing 10–3 for the original file).
- AndroidManifest.xml: Define this new activity in the manifest file (see Listing 10–19 for this activity's definition so we can add this to the main Android manifest file in Listing 10–11).

We used tabbed listeners while setting up the tabbed action bar and list listeners for setting up the list navigation action bar. For a standard action bar, there are no listeners other than the menu callbacks. The menu callbacks don't need to be specially set up because they are hooked up automatically by the SDK. As a result, it is quite easy to set up the action bar in the standard

navigation mode. Listing 10–17 presents the source code for the standard navigation action bar activity.

Listing 10–17. Standard Navigation Action Bar Activity

```
//StandardNavigationActionBarActivity.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class StandardNavigationActionBarActivity
extends BaseActionBarActivity
{
private static String tag=
"Standard Navigation ActionBarActivity";
public StandardNavigationActionBarActivity()
{
super(tag);
}
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
workwithStandardActionBar();
}
public void workwithStandardActionBar()
{
ActionBar bar = this.getActionBar();
bar.setTitle(tag);
bar.setNavigationMode(ActionBar.NAVIGATION_MODE_STANDARD);
//test to see what happens if we were to attach tabs
attachTabs(bar);
}
public void attachTabs(ActionBar bar)
{
TabListener tl = new TabListener(this,this);
Tab tab1 = bar.newTab();
tab1.setText("Tab1");
tab1.setTabListener(tl);
bar.addTab(tab1);
Tab tab2 = bar.newTab();
tab2.setText("Tab2");
tab2.setTabListener(tl);
bar.addTab(tab2);
}
}//eof-class
```

The only thing necessary to set up an action bar as a standard navigation action bar is to set its navigation mode as such. That portion of the code is highlighted in Listing 10– 17.

an double OS

Figure 10–5. *An activity with a standard navigation action bar*

ACTION BAR AND SEARCH VIEW

In Android 4.0, because the action bar is available on phones, there is an increasing interest in using it as a search facility. This section shows how to use a search widget in the action bar. We will provide code snippets that we can use to modify the project we have seen so far to include a search widget. Although we present only snippets, we can see the full code in the downloadable project for this chapter. A search view widget is a search box that fits between our tabs and the menu icons in the action bar, as shown in Figure 10–7.

Figure 10–7. *An action bar search view*

We need to do the following to use search in our action bar:

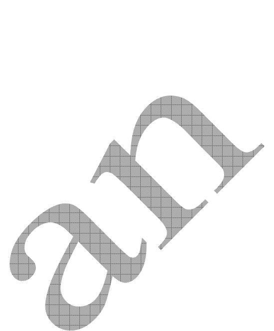
1. Define a menu item pointing to a search view provided by the SDK. We also need an activity into which we can load this menu. This is often called the search invoker activity.
2. Create another activity that can take the query from the search view in step 1 and provide results. This is often called the search results activity.
3. Create an XML file that allows we to customize the search view widget.

4. This file is often called searchable.xml and resides in the res/xml subdirectory.
5. Declare the search results activity in the manifest file. This definition needs to point to the XML file defined in step 3.
6. In our menu setup for the search invoker activity, indicate that the search view needs to target the search results activity from step 2.

We will provide code snippets for each of these steps. As mentioned earlier, the complete code is available in the downloadable project. In fact, when we run the project for this chapter, the search view is visible on all the action bars presented in the previous sections of this chapter: tab, list, and standard.

ACTION BAR AND FRAGMENTS.

The action bar is generally recommended for use with fragments when we’re dealing with tablets. Because fragments are inside an activity, and an activity owns the action bar, we don’t need the abstraction of a base class to ensure the same action bar for each activity. All fragments share the same activity, so they also share the same action bar. The solution is simpler.



an double os

MODULE V

FILES, SAVING STATE AND PREFERENCES

The simplest and most versatile data-persistence techniques in Android: Shared Preferences, instance-state Bundles, and local files. Saving and loading data is essential for most applications. At a minimum, an Activity should save its user interface (UI) state before it becomes inactive to ensure the same UI is presented when it restarts. It's also likely that we'll need to save user preferences and UI selections.

Android offers several alternatives for saving application data, each optimized to fulfill a particular need. Shared Preferences are a simple, lightweight name/value pair (NVP) mechanism for saving primitive application data, most commonly a user's application preferences. Android also offers a mechanism for recording application state within the Activity lifecycle handlers, as well as for providing access to the local file system, through both specialized methods and the java.io classes. Android also offers a rich framework for user preferences, allowing we to create settings screens consistent with the system settings.

SAVING APPLICATION DATA

The data-persistence techniques in Android provide options for balancing speed, efficiency, and robustness.

- Shared Preferences — When storing UI state, user preferences, or application settings, we want a lightweight mechanism to store a known set of values. Shared Preferences let we save groups of name/value pairs of primitive data as named preferences.
- Saved application UI state — Activities and Fragments include specialized event handlers to record the current UI state when our application is moved to the background.
- Files — It's not pretty, but sometimes writing to and reading from files is the only way to go. Android lets we create and load files on the device's internal or external media, providing support for temporary caches and storing files in publicly accessible folders.

There are two lightweight techniques for saving simple application data for Android applications: Shared Preferences and a set of event handlers used for saving Activity instance state. Both mechanisms use an NVP mechanism to store simple primitive values. Both techniques support primitive types Boolean, string, fl oat, long, and integer, making them ideal means of quickly storing default values, class instance variables, the current UI state, and user preferences.

CREATING AND SAVING SHARED PREFERENCES

Using the SharedPreferences class, we can create named maps of name/value pairs that can be persisted across sessions and shared among application components running within the same application sandbox.

andoubleos

To create or modify a Shared Preference, call getSharedPreferences on the current Context, passing in the name of the Shared Preference to change.

```
SharedPreferences mySharedPreferences =
getSharedPreferences(MY_PREFS,
Activity.MODE_PRIVATE);
```

Shared Preferences are stored within the application's sandbox, so they can be shared between an application's components but aren't available to other applications. To modify a Shared Preference, use the SharedPreferences.Editor class. Get the Editor object by calling edit on the Shared Preferences object we want to change.

```
SharedPreferences.Editor editor = mySharedPreferences.edit();
```

Use the put<type> methods to insert or update the values associated with the specified name:

```
// Store new primitive types in the shared preferences object.
editor.putBoolean("isTrue", true);
editor.putFloat("lastFloat", 1f);
editor.putInt("wholeNumber", 2);
editor.putLong("aNumber", 3l);
editor.putString("textEntryValue", "Not Empty");
```

To save edits, call apply or commit on the Editor object to save the changes asynchronously or synchronously, respectively.

```
// Commit the changes.
editor.apply();
```

RETRIEVING SHARED PREFERENCES

Accessing Shared Preferences, like editing and saving them, is done using the getSharedPreferences method.

Use the type-safe get<type> methods to extract saved values. Each getter takes a key and a default value (used when no value has yet been saved for that key.)

```
// Retrieve the saved values.
boolean isTrue = mySharedPreferences.getBoolean("isTrue", false);
float lastFloat = mySharedPreferences.getFloat("lastFloat", 0f);
int wholeNumber = mySharedPreferences.getInt("wholeNumber", 1);
long aNumber = mySharedPreferences.getLong("aNumber", 0);
String stringPreference =
mySharedPreferences.getString("textEntryValue", "");
```

We can return a map of all the available Shared Preferences keys values by calling getAll, and check for the existence of a particular key by calling the contains method.

```
Map<String, ?> allPreferences = mySharedPreferences.getAll();
boolean containsLastFloat = mySharedPreferences.contains("lastFloat");
```

PREFERENCE FRAMEWORK AND PREFERENCE ACTIVITY

Android offers an XML-driven framework to create system-style Preference Screens for our applications. By using this framework we can create Preference Activities that are consistent with those used in both native and other third-party applications. This has two distinct advantages:

- Users will be familiar with the lawet and use of our settings screens.
- We can integrate settings screens from other applications (including system settings such as location settings) into our application's preferences.

The preference framework consists of four parts:

- Preference Screen layout — An XML file that defines the hierarchy of items displayed in our Preference screens. It specifies the text and associated controls to display, the allowed values, and the Shared Preference keys to use for each control.
- Preference Activity and Preference Fragment — Extensions of PreferenceActivity and PreferenceFragment respectively, that are used to host the Preference Screens. Prior to Android 3.0, Preference Activities hosted the Preference Screen directly; since then, Preference Screens are hosted by Preference Fragments, which, in turn, are hosted by Preference Activities.
- Preference Header definition — An XML file that defines the Preference Fragments for our application and the hierarchy that should be used to display them.
- Shared Preference Change Listener — An implementation of the OnSharedPreferenceChangeListener class used to listen for changes to Shared Preferences.

PREFERENCE LAYOUT IN XML

Unlike in the standard UI layout, preference definitions are stored in the res/xml resources folder. Although conceptually they are similar to the UI layout resources. Preference Screen layouts use a specialized set of controls designed specifically for preferences. Each preference layout is defined as a hierarchy, beginning with a single PreferenceScreen element:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">
</PreferenceScreen>
```

We can include additional Preference Screen elements, each of which will be represented as a selectable element that will display a new screen when clicked.
Within each Preference Screen we can include any combination of PreferenceCategory and Preference<control> elements. Preference Category

an double OS

elements, as shown in the following snippet, are used to break each Preference Screen into subcategories using a title bar separator:

```
<PreferenceCategory
android:title="My Preference Category"/>
```

Figure 7-2 shows the SIM card lock, device administration, and credential storage Preference Categories used on the Security Preference Screen. All that remains is to add the preference controls that will be used to set the preferences. Although the specific attributes available for each preference control vary, each of them includes at least the following four:

- android:key — The Shared Preference key against which the selected value will be recorded.
- android:title — The text displayed to represent the preference.
- android:summary — The longer text description displayed in a smaller font below the title text.
- android:defaultValue — The default value that will be displayed (and selected) if no preference value has been assigned to the associated preference key.

FIGURE 7-2

Listing 7-1 shows a sample Preference Screen that includes a Preference Category and CheckBox Preference.

Listing 7-1: A simple Shared Preferences screen

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">
<PreferenceCategory
android:title="My Preference
Category">
<CheckBoxPreference
android:key="PREF_CHECK_BOX"
android:title="Check Box Preference"
android:summary="Check Box
Preference Description"
android:defaultValue="true"
/>
</PreferenceCategory>
</PreferenceScreen>
```

FIGURE 7-3

When displayed, this Preference Screen will appear as shown in Figure 7-3.

NATIVE PREFERENCE CONTROLS

Android includes several preference controls to build our Preference Screens:

- CheckBoxPreference — A standard preference check box control used to set preferences to true or false.
- EditTextPreference — Allows users to enter a string value as a preference. Selecting the preference text at run time will display a text-entry dialog.
- ListPreference — The preference equivalent of a spinner. Selecting this preference will display a dialog box containing a list of values from which to select. We can specify different arrays to contain the display text and selection values.
- MultiSelectListPreference — Introduced in Android 3.0 (API level 11), this is the preference equivalent of a check box list.
- RingtonePreference — A specialized List Preference that presents the list of available ringtones for user selection. This is particularly useful when we're constructing a screen to configure notification settings.

We can use each preference control to construct our Preference Screen hierarchy. Alternatively, we can create our own specialized preference controls by extending the Preference class (or any of the subclasses listed above).

PREFERENCE FRAGMENTS

Since Android 3.0, the PreferenceFragment class has been used to host the preference screens defined by Preferences Screen resources. To create a new Preference Fragment, extend the PreferenceFragment class, as follows:

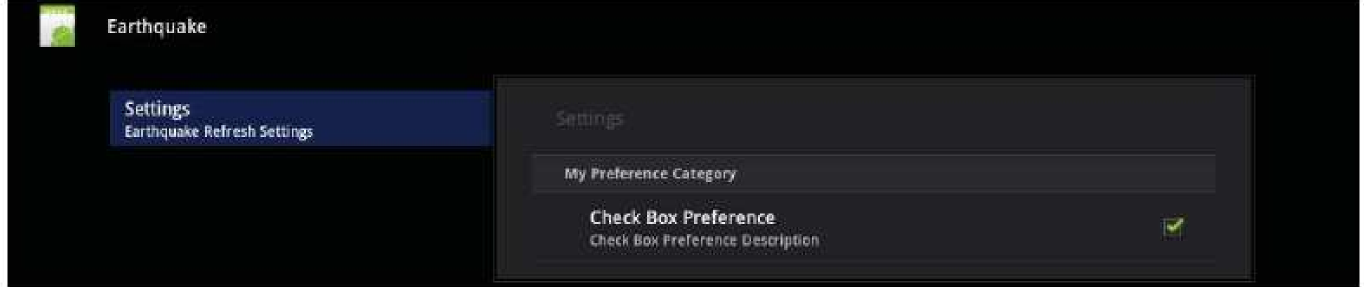
```
public class MyPreferenceFragment extends PreferenceFragment
```

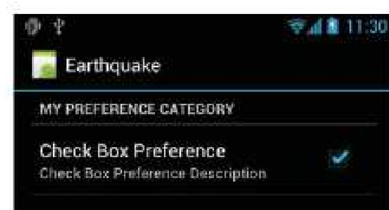
To inflate the preferences, override the onCreate handler and call addPreferencesFromResource, as shown here:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
addPreferencesFromResource(R.xml.userpreferences);
}
```

Our application can include several different Preference Fragments, which will be grouped according to the Preference Header hierarchy and displayed within a Preference Activity.

an double os





PREFERENCE ACTIVITY

The PreferenceActivity class is used to host the Preference Fragment hierarchy defined by a preference headers resource. Prior to Android 3.0, the Preference Activity was used to host Preference Screens directly. For applications that target devices prior to Android 3.0, we may still need to use the Preference Activity in this way. To create a new Preference Activity, extend the PreferenceActivity class as follows:

```
public class MyFragmentPreferenceActivity extends PreferenceActivity
```

When using Preference Fragments and headers, override the onBuildHeaders handler, calling load- HeadersFromResource and specifying our preference headers resource file:

```
public void onBuildHeaders(List<Header> target) {
loadHeadersFromResource(R.xml.userpreferenceheaders, target);
}
```

For legacy applications, we can infl ate the Preference Screen directly in the same way as we would from a Preference Fragment — by overriding the onCreate handler and calling add PreferencesFromResource, specifying the Preference Screen layout XML resource to display within that Activity:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
addPreferencesFromResource(R.xml.userpreferences);
}
```

Like all Activities, the Preference Activity must be included in the application manifest:

```
<activity android:name=".MyPreferenceActivity"
android:label="My Preferences">
</activity>
```

To display the application settings hosted in this Activity, open it by calling startActivity or startActivityForResult:

```
Intent i = new Intent(this, MyPreferenceActivity.class);
startActivityForResult(i, SHOW_PREFERENCES);
```

PERSISTING THE APPLICATION STATE

To save Activity instance variables, Android offers two specialized variations of Shared Preferences. The first uses a Shared Preference named specifically for our Activity, whereas the other relies on a series of lifecycle event handlers.

1. Using Shared Preferences

If we want to save Activity information that doesn't need to be shared with other components (e.g., class instance variables), we can call Activity.getPreferences()without specifying a Shared Preferences name. This returns a Shared Preference using the calling Activity's class name as the Shared Preference name.

```
// Create or retrieve the activity preference object.
SharedPreferences activityPreferences =
getPreferences(Activity.MODE_PRIVATE);

// Retrieve an editor to modify the shared preferences.
SharedPreferences.Editor editor = activityPreferences.edit();

// Retrieve the View
TextView myTextView = (TextView)findViewById(R.id.myTextView);

// Store new primitive types in the shared preferences object.
editor.putString("currentTextValue",
myTextView.getText().toString());

// Commit changes.
editor.apply();
```

2. Using the Lifecycle Handlers

Activities offer the onSaveInstanceState handler to persist data associated with UI state across sessions. It's designed specifically to persist UI state should an Activity be terminated by the run time, either in an effort to free resources for foreground applications or to accommodate restarts caused by hardware configuration changes.

If an Activity is closed by the user (by pressing the Back button), or programmatically with a call to finish, the instance state bundle will not be passed in to onCreate or onRestoreInstanceState when the Activity is next created. Data that should be persisted across user sessions should be stored using Shared Preferences, as described in the previous sections. By overriding an Activity's onSaveInstanceState event handler, we can use its Bundle parameter to save UI instance values. Store values using the same put methods as shown for Shared Preferences, before passing the modified Bundle into the super class's handler:

androdouble os

INCLUDING STATIC FILES AS RESOURCES

If our application requires extern al file resources, we can include them in our distribution package by placing them in the res/raw folder of our project hierarchy. To access these read-only file resources, call the openRawResource method from our application's Resource object to receive an InputStream based on the specified file. Pass in the filename (without the extension) as the variable name from the R.raw class, as shown in the following skeleton code:

```
Resources myResources = getResources();
InputStream myFile = myResources.openRawResource(R.raw.myfilename);
```

Adding raw files to our resources hierarchy is an excellent alternative for large, preexisting data sources (such as dictionaries) for which it's not desirable (or even possible) to convert them into Android databases. Android's resource mechanism lets we specify alternative resource files for different languages, locations, and hardware configurations. For example, we could create an application that loads a different dictionary resource based on the user's language settings.

WORKING WITH FILE SYSTEM

It's good practice to use Shared Preferences or a database to store our application data, but there may still be times when we'll want to use files directly rather than rely on Android's managed mechanisms — particularly when working with multimedia files.

File-Management Tools

Android supplies some basic file-management tools to help we deal with the file system. Many of these utilities are located within the java.io.file package. Complete coverage of Java file-management utilities is beyond the scope of this book, but Android does supply some specialized utilities for file management that are available from the application Context.

- deleteFile — Enables we to remove files created by the current application
- fileList — Returns a string array that includes all the files created by the current application

These methods are particularly useful for cleaning up temporary files left behind if our application crashes or is killed unexpectedly.

Using Application-Specific Folders to Store Files

Many applications will create or download files that are specific to the application. There are two options for storing these application-specific files: internally or externally.

Note: When referring to the external storage, we refer to the shared/media storage that is accessible by all applications and can typically be mounted to a computer file system when the device is connected via USB. Although it is typically located on the SD Card, some devices implement this as a separate partition on the internal storage. The most important thing to remember when

storing files on external storage is that no security is enforced on files stored here. Any application can access, overwrite, or delete files stored on the external storage. It's also important to remember that files stored on external storage may not always be available. If the SD Card is ejected, or the device is mounted for access via a computer, our application will be unable to read (or create) files on the external storage.

Android offers two corresponding methods via the application Context, getDir and getExternalFilesDir, both of which return a File object that contains the path to the internal and external application file storage directory, respectively. All files stored in these directories or the subfolders will be erased when our application is uninstalled.

Both of these methods accept a string parameter that can be used to specify the subdirectory into which we want to place our files. In Android 2.2 (API level 8) the Environment class introduced a number of DIRECTORY_[Category] string constants that represent standard directory names, including downloads, images, movies, music, and camera files.

Files stored in the application folders should be specific to the parent application and are typically not detected by the media-scanner, and therefore won't be added to the Media Library automatically. If our application downloads or creates files that should be added to the Media Library or otherwise made available to other applications, consider putting them in the public external storage directory.

Creating Private Application Files

Android offers the openFileInput and openFileOutput methods to simplify reading and writing streams from and to files stored in the application's sandbox.

```
String FILE_NAME = "tempfile.tmp";
// Create a new output file stream that's private to this application.
FileOutputStream fos = openFileOutput(FILE_NAME,
Context.MODE_PRIVATE);
// Create a new file input stream.
FileInputStream fis = openFileInput(FILE_NAME);
```

These methods support only those files in the current application folder; specifying path separators will cause an exception to be thrown. If the filename we specify when creating a FileOutputStream does not exist, Android will create it for we. The default behavior for existing files is to overwrite them; to append an existing file, specify the mode as Context.MODE_APPEND.

By default, files created using the openFileOutput method are private to the calling application — a different application will be denied access. The standard way to share a file between applications is to use a Content Provider. Alternatively, we can specify either Context.MODE_WORLD_READABLE or

Context.MODE_WORLD_WRITEABLE when creating the output file, to make it available in other applications, as shown in the following snippet:

```
String OUTPUT_FILE = "publicCopy.txt";
FileOutputStream fos = openFileOutput(OUTPUT_FILE,
Context.MODE_WORLD_WRITEABLE);
```

We can find the location of files stored in our sandbox by calling getFilesDir. This method will return the absolute path to the files created using openFileOutput:

```
File file = getFilesDir();
Log.d("OUTPUT_PATH_", file.getAbsolutePath());
```

Using the Application File Cache

Should our application need to cache temporary files, Android offers both a managed internal cache, and (since Android API level 8) an unmanaged external cache. We can access them by calling the getCacheDir and getExternalCacheDir methods, respectively, from the current Context. Files stored in either cache location will be erased when the application is uninstalled. Files stored in the internal cache will potentially be erased by the system when it is running low on available storage; files stored on the external cache will not be erased, as the system does not track available storage on external media. In either case its good form to monitor and manage the size and age of our cache, deleting files when a reasonable maximum cache size is exceeded.

Storing Publicly Readable Files

Android 2.2 (API level 8) also includes a convenience method, Environment.getExternal StoragePublicDirectory, that can be used to find a path in which to store our application files. The returned location is where users will typically place and manage their own files of each type. This is particularly useful for applications that provide functionality that replaces or augments system applications, such as the camera, that store files in standard locations. The getExternalStoragePublicDirectory method accepts a String parameter that determines which subdirectory we want to access using a series of Environment static constants:

- DIRECTORY_ALARMS — Audio files that should be available as user-selectable alarm sounds
- DIRECTORY_DCIM — Pictures and videos taken by the device
- DIRECTORY_DOWNLOADS — Files downloaded by the user
- DIRECTORY_MOVIES — Movies
- DIRECTORY_MUSIC — Audio files that represent music
- DIRECTORY_NOTIFICATIONS — Audio files that should be available as user-selectable notification sounds
- DIRECTORY_PICTURES — Pictures
- DIRECTORY_PODCASTS — Audio files that represent podcasts
- DIRECTORY_RINGTONES — Audio files that should be available as user-selectable ringtones

an doubleOS

Note that if the returned directory doesn't exit, we must create it before writing files to the directory, as shown in the following snippet:

```
String FILE_NAME = "MyMusic.mp3";
File path = Environment.getExternalStoragePublicDirectory(
Environment.DIRECTORY_MUSIC);
File file = new File(path, FILE_NAME);
try {
path.mkdirs();
[... Write Files ...]
} catch (IOException e) {
Log.d(TAG, "Error writing " + FILE_NAME, e);
}
```

PERSISTING DATA

To accomplish many of the activities offered by modern mobile phones, such as tracking contacts, events, and tasks, a mobile operating system and its applications must be adept at storing and keeping track of large quantities of data. This data is usually structured in rows and columns, like a spreadsheet or a very simple database. Beyond a traditional application's requirements for storing data, the Android application life cycle demands rapid and consistent persistence of data for it to survive the volatility of the mobile environment, where devices can suddenly lose power or the Android operating system can arbitrarily decide to remove our application from memory. Android provides the light-weight but powerful SQLite relational database engine for persisting data.

Relational Database Overview

A relational database provides an efficient, structured, and generic system for managing persistent information. With a database, applications use structured queries to modify information in persistent two-dimensional matrices called tables (or in the original theoretical papers, relations). Developers write queries in a high-level language called the Standard Query Language, or more commonly, SQL. SQL is the common language for relational database management systems (RDBMSs) that have been a popular tool for data management since the late 1970s. SQL became an industry-wide standard when it was adopted by NIST in 1986 and ISO in 1987. It is used for everything from terabyte Oracle and SQL Server installations to, as we shall see, storing email on our phone. Database tables are a natural fit for data that includes many instances of the same kind of thing—a typical occurrence in software development. For example, a contact list has many contacts, all of which potentially have the same type of information (i.e., address, phone number, etc.). Each "row" of data in a table stores information about a different person, while each "column" stores a specific attribute of each person: names in one column, address in another column, and home phone number in a third. When someone is related to multiple things (such as multiple addresses), relational databases have ways of handling that too.

an double OS

SQLITE

Android uses the SQLite database engine, a self-contained, transactional database engine that requires no separate server process. Many applications and environments beyond Android make use of it, and a large open source community actively develops SQLite. In contrast to desktop-oriented or enterprise databases, which provide a plethora of features related to fault tolerance and concurrent access to data, SQLite aggressively strips out features that are not absolutely necessary in order to achieve a small footprint. For example, many database systems use static typing, but SQLite does not store database type information. Instead, it pushes the responsibility of keeping type information into high-level languages, such as Java, that map database structures into high-level types.

SQLite is not a Google project, although Google has contributed to it. SQLite has an international team of software developers who are dedicated to enhancing the software's capabilities and reliability. Reliability is a key feature of SQLite. More than half of the code in the project is devoted to testing the library. The library is designed to handle many kinds of system failures, such as low memory, disk errors, and power failures. The database should never be left in an unrecoverable state, as this would be a showstopper on a mobile phone where critical data is often stored in a database. Fortunately, the SQLite database is not susceptible to easy corruption—if it were, an inopportune battery failure could turn a mobile phone into an expensive paperweight.

The SQL Language

Writing Android applications usually requires a basic ability to program in the SQL language, although higher-level classes are provided for the most common data-related activities. We will provide we with enough detail about Android-oriented SQL to let we implement data persistence in a wide variety of Android applications. We'll use simple SQL commands to explain the SQLite language, and along the way, we'll demonstrate how to use the sqlite3 command to see the effects those queries have on the tables they modify.

With SQLite, the database is a simple file in the Android file system, which could reside in flash or external card memory, but we will find that most applications' databases reside in a directory called /data/data/com.example.ourAppPackage/databases. We can issue the ls command in the adb shell to list the databases that Android has created for us in that directory.

The database takes care of persistence—that is, it updates the SQLite file in the way specified by each SQL statement issued by an application. In the following text, we describe SQLite commands as they are used inside the sqlite3 command-line utility. Later we will show ways to achieve the same effects using the Android API. Although command-line SQL will not be part of the application we ship, it can certainly help to debug applications as we're developing them. We will find that writing database code in Android is usually

an double os

an iterative process of writing Java code to manipulate tables, and then peeking at created data using the command line.

SQL Data Definition Commands

Statements in the SQL language fall into two distinct categories: those used to create and modify tables—the locations where data is stored—and those used to create, read, update, and delete the data in those tables. In this section we'll look at the former, the data definition commands:

CREATE TABLE

Developers start working with SQL by creating a table to store data. The CREATE TABLE command creates a new table in a SQLite database. It specifies a name, which must be unique among the tables in the database, and various columns to hold the data. Each column has a unique name within the table and a type (the types are defined by SQL, such as a date or text string). The column may also specify other attributes, such as whether values have to be unique, whether there is a default value when a row is inserted without specifying a value, and whether NULL is allowed in the column.

A table is similar to a spreadsheet. Returning to the example of a contact database, each row in the table contains the information for one contact. The columns in the table are the various bits of information we collect about each individual contact: first name, last name, birthday, and so on.

NOTE: The tables created by SQL CREATE TABLE statements and the attributes they contain are called a database schema.

DROP TABLE

This removes a table added with the CREATE TABLE statement. It takes the name of the table to be deleted. On completion, any data that was stored in the table may not be retrieved.

Here is some SQL code that will create and then delete a simple table for storing contacts:

```
CREATE TABLE contacts (
    first_name TEXT,
    last_name TEXT,
    phone_number TEXT,
    height_in_meters REAL);
DROP TABLE contacts;
```

When entering commands through sqlite3, we must terminate each command with a semicolon. We may change the database schema after we create tables (which we may want to do to add a column or change the default value of a column) by entering the ALTER TABLE command.

an double os

SQLITE TYPES

We must specify a type for each column that we create in all tables that we define; SQLite supports the following data types:

- **TEXT:-**A text string, stored using the database encoding (UTF-8, UTF-16BE, or UTF-16LE). We will find that the TEXT type is the most common.
- **REAL:-**A floating-point value, stored as an 8-byte IEEE floating-point number.
- **BLOB:-**Arbitrary binary data, stored exactly as if it was input. We can use the BLOB data type to store any kind of variable-length data, such as an executable file, or a downloaded image. Generally, blobs can add a large performance overhead to a mobile database and we should usually avoid using them.
- **INTEGER:-**A signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.

Database Constraints

Database constraints mark a column with particular attributes. Some constraints enforce data-oriented limitations, such as requiring all values in a column to be unique (e.g., a column containing Social Security numbers). Other constraints exhibit more functional uses. Relational constraints, PRIMARY KEY and FOREIGN KEY, form the basis of enterable relationships.

Most tables should have a particular column that uniquely identifies each given row. Designated in SQL as a PRIMARY KEY, this column tends to be used only as an identifier for each row and (unlike a Social Security number) has no meaning to the rest of the world. Thus, we do not need to specify values for the column. Instead, we can let SQLite assign incrementing integer values as new rows are added. Other databases typically require we to specially mark the column as auto incrementing to achieve this result. SQLite also offers an explicit AUTOINCREMENT constraint, but auto increments primary keys by default. The incrementing values in the column take on a role similar to an opaque object pointer in a high-level language such as Java or C: other database tables and code in a high-level language can use the column to reference that particular row.

When database rows have a unique primary key, it is possible to start thinking about dependencies between tables. For example, a table used as an employee database could define an integer column called employer_id that would contain the primary key values of rows in a different table called employers. If we perform a query and select one or more rows from the employers table, we can use grab their IDs and to look up employees in an employees table through the table's employer_id column. This allows a program to find the employees of a given employer. The two tables (stripped down to a few columns relevant to this example) might look like this:

```
CREATE TABLE employers (
        _id INTEGER PRIMARY KEY,
        company_name TEXT);
```

an double OS

```
CREATE TABLE employees (
    name TEXT,
    annual_salary REAL NOT NULL CHECK (annual_salary > 0),
    employer_id REFERENCES employers(_id));
```

The idea of a table referring to another table's primary key has formal support in SQL as the FOREIGN KEY column constraint, which enforces the validity of cross-table references. This constraint tells the database that integers in a column with a foreign key constraint must refer to valid primary keys of database rows in another table. Thus, if we insert a row into the employees table with an employer_id for a row that does not exist in the employers table, many flavors of SQL will raise a constraint violation. This may help we to avoid orphaned references, also known as enforcement of foreign keys. However, the foreign key constraint in SQLite is optional, and is turned off in Android. As of Android 2.2, we cannot rely on a foreign key constraint to catch incorrect foreign key references, so we will need to take care when creating database schemas that use foreign keys. There are several other constraints with less far-reaching effects:

- **UNIQUE:-**Forces the value of the given column to be different from the values in that column in all existing rows, whenever a row is inserted or updated. Any insert or update operation that attempts to insert a duplicate value will result in an SQLite constraint violation.
- **NOT NULL:-**Requires a value in the column; NULL cannot be assigned. Note that a primary key is both UNIQUE and NOT NULL.
- **CHECK:-**Takes a Boolean-valued expression and requires that the expression return true for any value inserted in the column. An example is the CHECK (annual_salary > 0), attribute shown earlier in the employees table.

DATABASE MANIPULATION USING SQLITE

Once we have defined tables using data definition commands, we can then insert our data and query the database. The following data manipulation commands are the most commonly used SQL statements:

1) SELECT

This statement provides the main tool for querying the database. The result of this statement is zero or more rows of data, where each row has a fixed number of columns. We can think of the SELECT statement as producing a new table with only the rows and columns that we choose in the statement. The SELECT statement is the most complicated command in the SQL language, and supports a broad number of ways to build relationships between data across one or more database tables. Clauses for SQL's SELECT command, which are all supported by the Android API, include the following:

- FROM, which specifies the tables from which data will be pulled to fulfill the query.
- WHERE, which specifies conditions that selected rows in the tables must match to be returned by the query.

- GROUP BY, which orders results in clusters according to column name.
- HAVING, which further limits results by evaluating groups against expressions. We might remove groups from our query that do not have a minimum number of elements.
- ORDER BY, which sets the sort order of query results by specifying a column name that will define the sort, and a function (e.g., ASC for ascending, DSC for descending) that will sort the rows by elements in the specified column.
- LIMIT, which limits the number of rows in a query to the specified value (e.g., five rows).

Here are a few examples of SELECT statements:

*SELECT * FROM contacts;*
SELECT first_name, height_in_meters FROM contacts
WHERE last_name = "Smith";
SELECT employees.name, employers.name
FROM employees, employers WHERE employee.employer_id = employer._id ORDER BY employer.company_name ASC;

The first statement retrieves all the rows in the contacts table, because no WHERE clause filters results. All columns (indicated by the asterisk, *) of the rows are returned. The second statement gets the names and heights of the members of the Smith family. The last statement prints a list of employees and their employers, sorted by company name.

2) INSERT

This statement adds a new data row to a specified database table along with a set of specified values of the proper SQLite type for each column (e.g., 5 for an integer). The insert may specify a list of columns affected by the insert, which may be less than the number of columns in the table. If we don't specify values for all columns, SQLite will fill in a default value for each unspecified column, if we defined one for that column in our CREATE TABLE statement. If we don't provide a default, SQLite uses a default of NULL.
Here are a few examples of INSERT statements:

INSERT INTO contacts(first_name) VALUES("Thomas");
INSERT INTO employers VALUES(1, "Acme Balloons");
INSERT INTO employees VALUES("Wile E. Coyote", 100000.000, 1);

The first adds a new row to the contacts for someone whose first name is Thomas and whose last name, phone number, and height are unknown (NULL). The second adds Acme Balloons as a new employer, and the third adds Wile E. Coyote as an employee there.

3) UPDATE

This statement modifies some rows in a given table with new values. Each assignment specifies a table name and a given function that should provide a new value for the column. Like SELECT, we can specify a WHERE

clause that will identify the rows that should be updated during an invocation of the UPDATE command. Like INSERT, we can also specify a list of columns to be updated during command execution. The list of columns works in the same manner as it does with INSERT. The WHERE clause is critical; if it matches no rows, the UPDATE command will have no effect, but if the clause is omitted, the statement will affect every row in the table. Here are a few examples of UPDATE statements:

```
UPDATE contacts
    SET height_in_meters = 10, last_name = "Jones"
UPDATE employees
    SET annual_salary = 200000.00
    WHERE employer_id = (
        SELECT _id
            FROM employers
            WHERE company_name = "Acme Balloons");
```

The first claims that all our friends are giants with the last name Jones. The second is a more complex query. It gives a substantial raise to all the employees of Acme Balloons.

Additional Database Concepts

We now know enough simple SQL to be able to start working with databases in Android. As the applications we write grow in sophistication, we are likely to make use of the following SQL constructs that we won't cover in detail in this book:

- **Inner join:-** An inner join selects data across two or more tables where data is related by a foreign key. This type of query is useful for assembling objects that need to be distributed across one or more tables. The employee/employer example earlier demonstrated an inner join. As we've noted, since Android does not enforce foreign keys, we can get into trouble here if a key for a join does not exist as a valid crossable reference—that is, a foreign key column actually points to a primary key of a row in another table that actually exists.

- **Compound query:-** SQLite supports complex database manipulations through combinations of statements. One of the update examples shown earlier was a compound query with a SELECT embedded in an UPDATE.

- **Triggers:-** A database trigger allows a developer to write SQL statements that will receive a callback when particular database conditions occur.

Database Transactions

Database transactions make sequences of SQL statements atomic: either all statements succeed or none of them have any effect on the database. This can be important, for instance, if our app encounters an unfortunate occurrence such as a system crash. A transaction will guarantee that if the

an double OS

device fails partway through a given sequence of operations, none of the operations will affect the database. In database jargon, SQLite transactions support the widely recited ACID transaction properties:

With SQLite, every database operation that modifies a database runs in its own database transaction, which means a developer can be assured that all values of an insert will be written if the statement succeeds at all. We can also explicitly start and end a transaction so that it encompasses multiple statements. For a given transaction, SQLite does not modify the database until all statements in the transaction have completed successfully.

Given the volatility of the Android mobile environment, we recommend that in addition to meeting the needs for consistency in our app, we also make liberal use of transactions to support fault tolerance in our application.

SQL AND DATABASE CENTRIC DATA MODEL FOR ANDROID

Now that we have some basic SQL programming knowledge, we can start thinking about how to put it to use in an Android application. Our goal is to create robust applications based on the popular Model-View-Controller (MVC) pattern that underlies well-written UI programs, specifically in a way that works well for Android.

One fundamental difference between mobile phone apps and desktop apps is how they handle persistence. Traditional desktop-based applications—word processors, text editors, drawing programs, presentation programs, and so on—often use a document centric form of the MVC pattern. They open a document, read it into memory, and turn it into objects in memory that form the data model. Such programs will make views for the data model, process user input through their controller, and then modify the data model. The key consequence of this design is that we explicitly open and save documents in order to make the data model persist between program invocations. We've seen how user interface components work in Android. Next we'll explore the Android APIs for database manipulation, which will prepare we to implement an application data model that works in a new way.

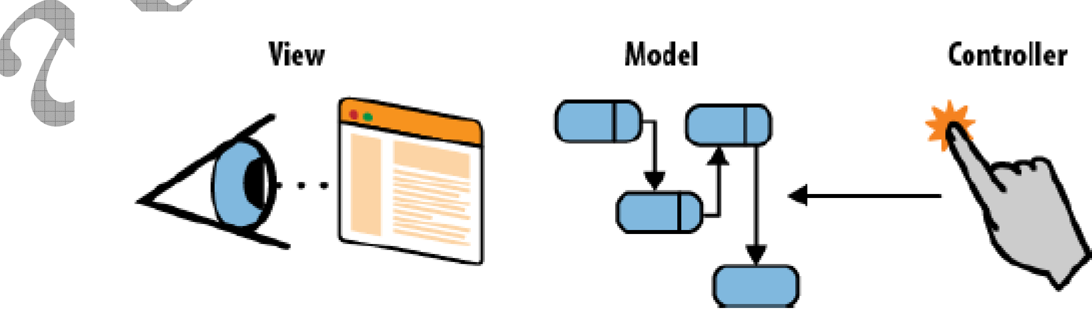


Figure 10-1. Document-centric applications, which implement a data model with in-memory objects

Robust use of Android combines data models and user interface elements in a different manner. Apps run on mobile devices with limited memory, which can run out of battery power at unpredictable and possibly inopportune times. Small mobile devices also place a premium on reducing the interactive burden on the user: reminding a user he ought to save a document when he is trying to answer a phone call is not a good user experience. The whole concept of a document is absent in Android. The user should always have the right data at hand and be confident her data is safe.

To make it easy to store and use application data incrementally, item by item, and always have it in persistent memory without explicitly saving the whole data model, Android provides support in its database, view, and activity classes for database-centric data. We'll explain how to use Android database classes to implement this kind of model.

ANDROID DATABASE CLASSES.

This section introduces the Java classes that give we access to the SQLite functions, with the data-centric model we just described in mind:

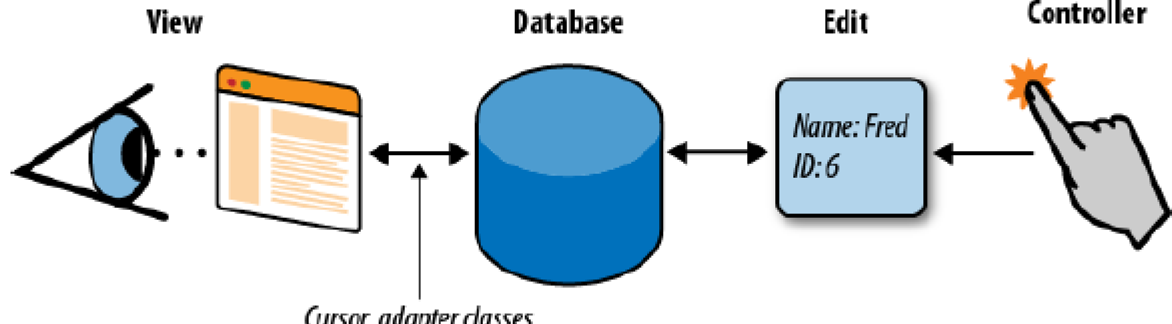


Figure 10-2. Android support for a data model that mostly resides in a database

- **SQLiteDatabase:-**Android's Java interface to its relational database, SQLite. It supports an SQL implementation rich enough for anything we're likely to need in a mobile application, including a cursor facility.
- **Cursor:-**A container for the results of a database query that supports an MVC-style observation system. Cursors are similar to JDBC result sets and are the return value of a database query in Android. A cursor can represent many objects without requiring an instance for each one. With a cursor, we can move to the start of query results and access each row one at a time as needed. To access cursor data, we call methods named as Cursor.getAs*(int columnNumber) (e.g., getAsString). The values the cursor will return depend on the current cursor index, which we can increment by calling Cursor.moveToNext, or decrement by calling Cursor.moveTo Previous, as needed. We can think of the current index of the cursor as a pointer to a result object. Cursors are at the heart of the basis for Android MVC.
- **SQLiteOpenHelper:-**Provides a life cycle framework for creating and upgrading our application database. It's quite helpful to use this class to assist with the critical task of transitioning the data from one version of an

application to a possible new set of database tables in a new version of an application.

- **SQLiteQueryBuilder**:- Provides a high-level abstraction for creating SQLite queries for use in Android applications. Using this class can simplify the task of writing a query since it saves we from having to fiddle with SQL syntax ourselves.

an double os