SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

MOBILE APPLICATIONS USING ANDROID

UNIT-I

ANDROID APPLICATIONS

Android has become one of the most popular smartphone operating systems in the world and while out of all the smartphone users worldwide, 88 per cent are using Android OS, Google's operating system has just turned 10. In this decade-long journey, Google's Android has come a long way from being on the verge of shutting down to becoming one of Google's biggest projects.

Google names every Android version in alphabetical order which means that after Google 9.0 Pie, the next Android name with start from the alphabet Q. On the other hand, Apart from Android 1.0 and 1.1, every other Android version has been named after either sweet treats or desserts.

Android versions and their names

1. Android 1.5: Android Cupcake
2. Android 1.6: Android Donut
3. Android 2.0: Android Eclair
4. Android 2.2: Android Froyo
5. Android 2.3: Android Gingerbread
6. Android 3.0: Android Honeycomb
7. Android 4.0: Android Ice Cream Sandwich
8. Android 4.1 to 4.3.1: Android Jelly Bean
9. Android 4.4 to 4.4.4: Android KitKat
10. Android 5.0 to 5.1.1: Android Lollipop
11. Android 6.0 to 6.0.1: Android Marshmallow
12. Android 7.0 to 7.1: Android Nougat
13. Android 8.0 to Android 8.1: Android Oreo
14. Android 9.0: Android Pie

This is how Google has named all its Android versions. How many of these did you know? And since which version are you an Android user?

Android is an open source and Linux-based operating system for mobile devices such as smartphones and tablet computers. Android was developed by the Open Handset Alliance, led by Google, and other companies.

Android programming is based on Java programming language so if you have basic understanding on Java programming then it will be a fun to learn Android application development.The first beta version of the Android Software Development Kit (SDK) was released by Google in 2007 where as the first commercial version, Android 1.0, was released in September 2008.

On June 27, 2012, at the Google I/O conference, Google announced the next Android version, 4.1 **Jelly Bean**. Jelly Bean is an incremental update, with the primary aim of improving the user interface, both in terms of functionality and performance.

The source code for Android is available under free and open source software licenses. Google publishes most of the code under the Apache License version 2.0 and the rest, Linux kernel changes, under the GNU General Public License version 2.

Why Android ?

Android has powerful APIs, excellent documentation, a thriving developer community, and no development or distribution costs. As mobile devices continue to increase in popularity, and Android devices expand into exciting new form-factors, you have the opportunity to create innovative applications no matter what your development experience.

1.1 A LITTLE BACKGROUND

Historically, developers, generally coding in low-level C or C++, have needed to understand the specific hardware they were coding for, typically a single device or possibly a range of devices from asingle manufacturer. As hardware technology and mobile Internet access has advanced, this closedapproach has become outmoded.

Platforms such as Symbian were later created to provide developers with a wider target audience.These systems proved more successful in encouraging mobile developers to provide rich applicationsthat better leveraged the hardware available. Developers need to write complex C/C++ code and make heavy use of proprietary APIs that arenotoriously difficult to work with. This difficulty is amplified for applications that must work on differenthardware implementations and those that make use of a particular hardware feature, such as GPS.

In more recent years, the biggest advance in mobile phone development was the introduction of JavahostedMIDlets. MIDlets are executed on a Java virtual machine (JVM), a process that abstracts theunderlying hardware and lets developers create applications that run on the wide variety of devicesthat support the Java run time. Unfortunately, this convenience comes at the price of restrictedaccess to the device hardware.

The introduction of Java MIDlets expanded developers' audiences, but the lack of low-level hardwareaccess and sandboxed execution meant that most mobile applications were regular desktop programsor websites designed to render on a smaller screen, and didn't take advantage of the inherent mobilityof the handheld platform. Android sits alongside a new wave of modern mobile operating systems designed to support applicationdevelopment on increasingly powerful mobile hardware. Platforms like Microsoft's WindowsPhone and the Apple iPhone also provide a richer, simplified development environment for mobileapplications; however, unlike Android, they're built on proprietary operating systems.

Android offers new possibilities for mobile applications by offering an open development environmentbuilt on an open-source Linux kernel. Hardware access is available to all applications througha series of API libraries, Third-party and native Android applications arewritten with the same APIs and are executed on the same run time.

1.2 WHAT ANDROID ISN'T

Android is *not* the following:

- **A Java ME implementation**
  Android applications are written using the Java language,but they are not run within a Java ME (Mobile Edition) VM, and Java-compiled classes andexecutables will not run natively in Android.
- **Part of the Linux Phone Standards Forum (LiPS) or the Open Mobile Alliance (OMA)**
  Android runs on an open-source Linux kernel, but, while their goals are similar, Android'scomplete software stack approach goes further than the focus of these standards-definingorganizations.
- **Simply an application layer (such as UIQ or S60)**
  Although Android does include an applicationlayer, "Android'' also describes the entire software stack, encompassing the underlyingoperating system, the API libraries, and the applications themselves.
- **A mobile phone handset**
  Android includes a reference design for mobile handset manufacturers,but there is no single "Android phone." Instead, Android has been designed to supportmany alternative hardware devices.
- **Google's answer to the iPhone**
  The iPhone is a fully proprietary hardware and softwareplatform released by a single company (Apple), whereas Android is an open-sourcesoftware stack produced and supported by the Open Handset Alliance (OHA) and designedto operate on any compatible device.

1.3 ANDROID: AN OPEN PLATFORM FOR MOBILE DEVELOPMENT

More recently, Android has expanded beyond a pure mobile phone platform to provide a developmentplatform for an increasingly wide range of hardware, including tablets and televisions.Put simply, Android is an ecosystem made up of a combination of three components:

- A free, open-source operating system for embedded devices
- An open-source development platform for creating applications
- Devices, particularly mobile phones, that run the Android operating system and the applicationscreated for it.

More specifically, Android is made up of several necessary and dependent parts, including the following:

- A Compatibility Definition Document (CDD) and Compatibility Test Suite (CTS) thatdescribe the capabilities required for a device to support the software stack.
- A Linux operating system kernel that provides a low-level interface with the hardware, memorymanagement, and process control, all optimized for mobile and embedded devices.
- Open-source libraries for application development, including SQLite, WebKit, OpenGL, anda media manager.

- A run time used to execute and host Android applications, including the Dalvik VirtualMachine (VM) and the core libraries that provide Android-specific functionality. The runtime is designed to be small and efficient for use on mobile devices.
- An application framework that agnostically exposes system services to the application layer,including the window manager and location manager, databases, telephony, and sensors.
- A user interface framework used to host and launch applications.
- A set of core pre-installed applications.
- A software development kit (SDK) used to create applications, including the related tools,plug-ins, and documentation.

1.4 NATIVE ANDROID APPLICATIONS

Android devices typically come with a suite of preinstalled applications that form part ofthe Android Open Source Project (AOSP), including, but not necessarily limited to, the following:

- An e-mail client
- An SMS management application
- A full PIM (personal information management) suite, including a calendar and contacts list
- A WebKit-based web browser
- A music player and picture gallery
- A camera and video recording application
- A calculator
- A home screen
- An alarm clock

In many cases Android devices also ship with the following proprietary Google mobile applications:

- The Google Play Store for downloading third-party Android applications
- A fully featured mobile Google Maps application, including StreetView, driving directions,and turn-by-turn navigation, satellite views, and traffic conditions
- The Gmail email client
- The Google Talk instant-messaging client
- The YouTube video player
- The data stored and used by many of these native applications such as contact details are alsoavailable to third-party applications.
- Similarly, your applications can respond to events such asincoming calls.

1.5 ANDROID SDK FEATURES

The true appeal of Android as a development environment lies in its APIs.The following list highlightssome of the most noteworthy Android features:

- GSM, EDGE, 3G, 4G, and LTE networks for telephony or data transfer, enabling you tomake or receive calls or SMS messages, or to send and retrieve data across mobile networks
- Comprehensive APIs for location-based services such as GPS and network-based locationdetection

- Full support for applications that integrate map controls as part of their user interfaces
- Wi-Fi hardware access and peer-to-peer connections
- Full multimedia hardware control, including playback and recording with the camera andmicrophone
- Media libraries for playing and recording a variety of audio/video or still-image formats
- APIs for using sensor hardware, including accelerometers, compasses, and barometers
- Libraries for using Bluetooth and NFC hardware for peer-to-peer data transfer
- IPC message passing
- Shared data stores and APIs for contacts, social networking, calendar, and multi-media
- Background Services, applications, and processes
- Home-screen Widgets and Live Wallpaper
- The ability to integrate application search results into the system searches
- An integrated open-source HTML5 WebKit-based browser
- Mobile-optimized, hardware-accelerated graphics, including a path-based 2D graphics libraryand support for 3D graphics using OpenGL ES 2.0
- Localization through a dynamic resource framework
- An application framework that encourages the reuse of application components and thereplacement of native applications

❖ **Access to Hardware, Including Camera, GPS, and Sensors**

Android includes API libraries to simplify development involving the underlying device hardware.They ensure that you don't need to create specific implementations of your software for differentdevices, so you can create Android applications that work as expected on any device that supportsthe Android software stack.

❖ **Data Transfers Using Wi-Fi, Bluetooth, and NFC**

Android offers rich support for transferring data between devices, including Bluetooth, Wi-FiDirect, and Android Beam. These technologies offer a rich variety of techniques for sharing databetween paired devices, depending on the hardware available on the underlying device, allowingyou to create innovative collaborative applications. To combine maps with locations, Android includes an API for forward and reverse geocoding that lets you find map coordinates for an address, and the address of a map position.

❖ **Background Services**

Android supports applications and services designed to run in the background while your applicationisn't being actively used.Modern mobiles and tablets are by nature multifunction devices; however, their screen sizes andinteraction models mean that generally only one interactive application is visible at any time. Background services make it possible to create invisible application components that perform automaticprocessing without direct user action.*Notifications* are the standard means by which a mobile device traditionally alerts users to events thathave happened in a background application. Using the Notification Manager, you can trigger audiblealerts, cause vibration, and flash the device's LED, as well as control status bar notification icons.

❖ **SQLite Database for Data Storage and Retrieval**

Rapid and efficient data storage and retrieval are essential for a device whose storage capacity isrelatively limited. Android provides a lightweight relational database for each application via SQLite. Your applicationscan take advantage of this managed relational database engine to store data securely and efficiently.

❖ **Shared Data and Inter-Application Communication**

Android includes several techniques for making information from your applications available for useelsewhere, primarily: Intents and Content Providers.*Intents* provide a mechanism for message-passing within and between applications. Using Intents,you can broadcast a desired action (such as dialing the phone or editing a contact) systemwide forother applications to handle. You can use *Content Providers* to provide secure, managed access to your applications' private databases.The data stores for native applications, such as the contact manager, are exposed as ContentProviders so you can read or modify this data from within your own applications.

❖ **Using Widgets and Live Wallpaper to Enhance the Home Screen**

Widgets and Live Wallpaper let you create dynamic application components that provide a windowinto your applications, or offer useful and timely information, directly on the home screen.

❖ **Extensive Media Support and 2D/3D Graphics**

Bigger screens and brighter, higher-resolution displays have helped make mobiles multimediadevices. To help you make the most of the hardware available, Android provides graphics librariesfor 2D canvas drawing and 3D graphics with OpenGL.

❖ **Optimized Memory and Process Management**

Like Java and .NET, Android uses its own run time and VM to manage application memory. Unlikewith either of these other frameworks, the Android run time also manages the process lifetimes.Android ensures application responsiveness by stopping and killing processes as necessary to freeresources for higher-priority applications.

❖ **Cloud to Device Messaging**

The Android Cloud to Device Messaging (C2DM) service provides an efficient mechanism for developersto create event-driven applications based on server-side pushes.Using C2DM you can create a lightweight, always-on connection between your mobile applicationand your server, allowing you to send small amounts of data directly to your device in real time.

1.6. INTRODUCING THE OPEN HANDSET ALLIANCE

The Open Handset Alliance (OHA) is a collection of more than 80 technology companies, includinghardware manufacturers, mobile carriers, software developers, semiconductor companies, andcommercialization companies. Of particular note are the prominent mobile technology companies,including Samsung, Motorola, HTC, T-Mobile, Vodafone, ARM, and Qualcomm.

The OHA hopes to deliver a better mobile software experience for consumers by providing the platformneeded for innovative mobile development at a faster rate and with higher quality than existingplatforms, without licensing fees for either software developers or handset manufacturers.

1.7WHY DEVELOP FOR ANDROID?

The barrier to entry for new Android developers is minimal:

- No certification is required to become an Android developer.
- Google Play provides free, up-front purchase, and in-app billing options for distribution andmonetization of your applications.
- There is no approval process for application distribution.
- Developers have total control over their brands.From a commercial perspective.

1.8 What Android Has That Other Platforms Don't Have

Many of the features listed previously, such as 3D graphics and native database support, are alsoavailable in other native mobile SDKs, as well as becoming available on mobile browsers.The following noncomprehensive list details some of thefeatures available on Android that may not be available on all modern mobile development platforms:

- **Google Maps applications —** Google Maps for Mobile has been hugely popular, and Android offers a Google Map as an atomic, reusable control for use in your applications. The Map View lets you display, manipulate, and annotate a Google Map within your Activities to build map-based applications using the familiar Google Maps interface.
- **Background services and applications —** Full support for background applications and services lets you create applications based on an event-driven model, working silently while other applications are being used or while your mobile sits ignored until it rings, flashes, or vibrates to get your attention.
- **Shared data and inter-process communication —** Using Intents and Content Providers, Android lets your applications exchange messages, perform processing, and share data. You can also use these mechanisms to leverage the data and functionality provided by the native Android applications.
- **All applications are created equal —** Android doesn't differentiate between native applications and those developed by third parties.
- **Wi-Fi Direct and Android Beam —** Using these innovative new inter-device communication APIs, you can include features such as instant media sharing and streaming. Android Beam is an NFC-based API that lets you provide support for proximity-based interaction, while Wi-Fi Direct offers a wider range peer-to-peer for reliable, high-speed communication between devices.
- **Home-screen Widgets, Live Wallpaper, and the quick search box —** Using Widgets and Live Wallpaper, you can create windows into your application from the phone's home screen. The quick search box lets you integrate search results from your application directly into the phone's search functionality.

1.9THE DEVELOPMENT FRAMEWORK

SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

MOBILE APPLICATIONS USING ANDROID

Android applications normally are written using Java as the programming language but executed by means of a custom VM called *Dalvik*, rather than a traditional Java VM. Each Android application runs in a separate process within its own Dalvik instance, relinquishing all responsibility for memory and process management to the Android run time, which stops and kills processes as necessary to manage resources. Dalvik and the Android run time sit on top of a Linux kernel that handles low-level hardware interaction, including drivers and memory management, while a set of APIs provides access to all the underlying services, features, and hardware.

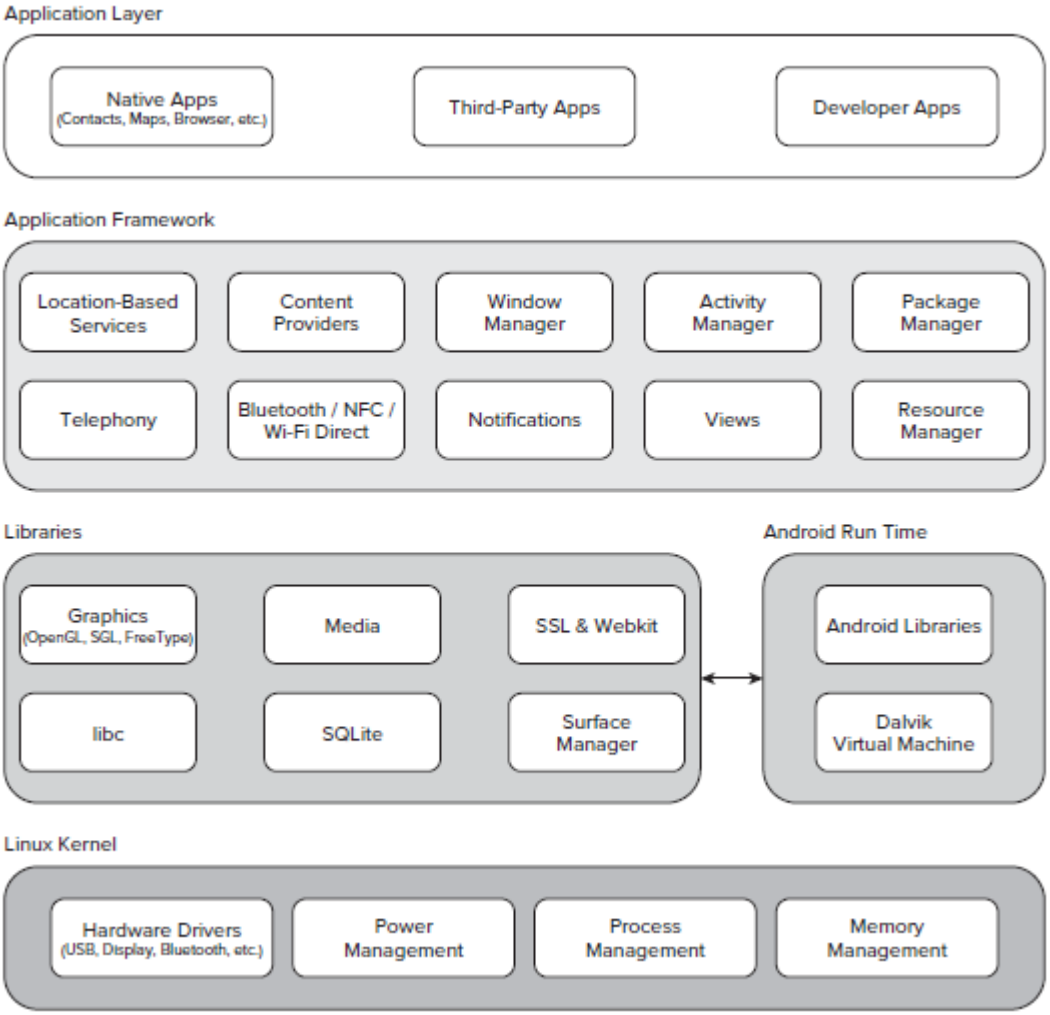
The Android SDK includes everything you need to start developing, testing, and debugging Android applications:

- **The Android APIs —** The core of the SDK is the Android API libraries that provide developer access to the Android stack. These are the same libraries that Google uses to create native Android applications.
- **Development tools —** The SDK includes several development tools that let you compile and debug your applications so that you can turn Android source code into executable applications. You will learn more about the developer tools in Chapter 2, "Getting Started."
- **The Android Virtual Device Manager and emulator —** The Android emulator is a fully interactive mobile device emulator featuring several alternative skins. The emulator runs within an Android Virtual Device (AVD) that simulates a device hardware configuration. Using the emulator you can see how your applications will look and behave on a real Android device. All Android applications run within the Dalvik VM, so the software emulator is an excellent development environment — in fact, because it's hardware-neutral, it provides a better independent test environment than any single hardware implementation.
- **Full documentation —** The SDK includes extensive code-level reference information detailing exactly what's included in each package and class and how to use them. In addition to the code documentation, Android's reference documentation and developer guide explains how to get started, gives detailed explanations of the fundamentals behind Android development, highlights best practices, and provides deep-dives into framework topics.
- **Sample code —** The Android SDK includes a selection of sample applications that demonstrate some of the possibilities available with Android, as well as simple programs that highlight how to use individual API features.
- **Online support —** Android has rapidly generated a vibrant developer community. The Google Groups are active forums of Android developers with regular input from the Android engineering and developer relations teams at Google. Stack Overflow (www.stackoverflow.com/questions/tagged/android) is also a hugely popular destination for Android questions and a great place to find answers to beginner questions.

1.10 Understanding the Android Software Stack

The Android software stack is, put simply, a Linux kernel and a collection of C/C++ librariesexposed through an application framework that provides services for, and management of, the runtime and applications. The Android software stack is composed of the elements shown in Figure.

- **Linux kernel —** Core services (including hardware drivers, process and memory management,security, network, and power management) are handled by a Linux 2.6 kernel. The kernelalso provides an abstraction layer between the hardware and the remainder of the stack.
- **Libraries —** Running on top of the kernel, Android includes various C/C++ core librariessuch as libc and SSL, as well as the following:
  - A media library for playback of audio and video media
  - A surface manager to provide display management
  - Graphics libraries that include SGL and OpenGL for 2D and 3D graphics
  - SQLite for native database support
  - SSL and WebKit for integrated web browser and Internet security
- **Android run time —** The run time is what makes an Android phone an Android phone ratherthan a mobile Linux implementation. Including the core libraries and the Dalvik VM, theAndroid run time is the engine that powers your applications and, along with the libraries,forms the basis for the application framework.
- **Core libraries —** Although most Android application development is written usingthe Java language, Dalvik is not a Java VM. The core Android libraries providemost of the functionality available in the core Java libraries, as well as the Androidspecific libraries.
- **Dalvik VM —** Dalvik is a register-based Virtual Machine that's been optimized toensure that a device can run multiple instances efficiently. It relies on the Linux kernelfor threading and low-level memory management.
- **Application framework —** The application framework provides the classes used to createAndroid applications. It also provides a generic abstraction for hardware access and managesthe user interface and application resources.
- **Application layer —** All applications, both native and third-party, are built on the applicationlayer by means of the same API libraries. The application layer runs within the Android runtime, using the classes and services made available from the application framework.



1.11 The Dalvik Virtual Machine

One of the key elements of Android is the Dalvik VM. Rather than using a traditional Java VM suchas Java ME, Android uses its own custom VM designed to ensure that multiple instances run efficiently on a single device.

The Dalvik VM uses the device's underlying Linux kernel to handle low-level functionality, includingsecurity, threading, and process and memory management. It's also possible to write C/C++applications that run closer to the underlying Linux OS. Although you *can* do this, in most casesthere's no reason you should need to.

If the speed and efficiency of C/C++ is required for your application, Android provides a nativedevelopment kit (NDK). The NDK is designed to enable you to create C++ libraries using the libcand libm libraries, along with native access to OpenGL.

1.12 Android Application Architecture

Android's architecture encourages component reuse, enabling you to publish and share Activities,Services, and data with other applications, with access managed by the security restrictions you define.The same mechanism that enables you to produce a replacement contact manager or phone dialercan let you expose your application's components in order to let other developers build on them bycreating new UI front ends or functionality extensions.The following application services are the architectural cornerstones of all Android applications,providing the framework you'll be using for your own software:

- **Activity Manager and Fragment Manager —** Control the lifecycle of your Activities andFragments, respectively, including management of the Activity stack
- **Views —** Used to construct the user interfaces for your Activities and Fragments
- **Notification Manager —** Provides a consistent and nonintrusive mechanism for signalingyour users
- **Content Providers —** Lets your applications share data
- **Resource Manager —** Enables non-code resources, such as strings an d graphics, to be externalized
- **Intents —** Provides a mechanism for transferring data between applications and their components.

1.13 Types of Android Applications

Most of the applications you create in Android will fall into one of the following categories:

- **Foreground** — An application that's useful only when it's in the foreground and is effectivelysuspended when it's not visible. Games are the most common examples. Applications have little control over their lifecycles, and a background application with no runningServices is a prime candidate for cleanup by Android's resource management. This means that youneed to save the state of the application when it leaves the foreground, and then present the samestate when it returns to the front.
- **Background** — An application with limited interaction that, apart from when being configured,spends most of its lifetime hidden. These applications are less common, but good examplesinclude call screening applications, SMS auto-responders, and alarm clocks. You can create completely invisible services, but in practice it's better to provide at least a basic levelof user control.
- **Intermittent** — Most well-designed applications fall into this category. At one extreme areapplications that expect limited interactivity but do most of their work in the background.A common example would be a media player. At the other extreme are applications that aretypically used as foreground applications but that do important work in the background.Email and news applications are great examples.Often you'll want to create an application that can accept user input and that also reacts to events when it's not the active foreground Activity. Chat and e-mail applications are typical examples.You must be particularly careful to ensure that the background processes of applications of this type are well behaved and have a minimal impact on the device's battery life.
- **Widgets and Live Wallpapers** — Some applications are represented only as a home-screenWidget or as a Live Wallpaper.Complex applications are often difficult.Such an application needs to be aware of its state when interacting with the user. This might mean updating the Activity UI when it's

visible and sending notifications to keep the user updated when it's in the background. Widget-only applications are commonly used to display dynamic information, such as battery levels, weather forecasts, or the date and time.

1.14 DEVELOPING FOR MOBILE AND EMBEDDED DEVICES

Android does a lot to simplify mobile- or embedded-device software development, but you need tounderstand the reasons behind the conventions. There are several factors to account for when writingsoftware for mobile and embedded devices, and when developing for Android in particular.

1.14.1 Hardware-Imposed Design Considerations

Small and portable, mobile devices offer exciting opportunities for software development. Their limitedscreen size and reduced memory, storage, and processor power are far less exciting, and insteadpresent some unique challenges.Compared to desktop or notebook computers, mobile devices have relatively:

- Low processing power
- Limited RAM
- Limited permanent storage capacity
- Small screens with low resolution
- High costs associated with data transfer
- Intermittent connectivity, slow data transfer rates, and high latency
- Unreliable data connections
- Limited battery life

Each new generation of phones improves many of these restrictions. In particular, newer phoneshave dramatically improved screen resolutions and significantly cheaper data costs.Be EfficientManufacturers of embedded devices, particularly mobile devices, generally value small size and longbattery life over potential improvements in processor speed. For developers, that means losing thehead start traditionally afforded thanks to Moore's law (the doubling of the number of transistorsplaced on an integrated circuit every two years). In desktop and server hardware, this usually resultsdirectly in processor performance improvements; for mobile devices, it instead means thinner, more power-efficient mobiles, with brighter, higher resolution screens. By comparison, improvements inprocessor power take a back seat.

❖ Expect Limited Capacity

Advances in flash memory and solid-state disks have led to a dramatic increase in mobile-devicestorage capacities. (MP3 collections still tend to expand to fi ll the available storage.) Although an8GB fl ash drive or SD card is no longer uncommon in mobile devices, optical disks offer more than32GB, and terabyte drives are now commonly available for PCs. Given that most of the availablestorage on a mobile device is likely to be used to store music and movies, many devices offer relativelylimited storage space for your applications. You should carefully consider how you store your application data. To make life easier, you can usethe Android databases and Content Providers to persist, reuse, and share large quantities of data,

❖ Design for Different Screens

The small size and portability of mobiles are a challenge for creating good interfaces, particularlywhen users are demanding an increasingly striking and information-rich graphical user experience.Combined with the wide range of screen sizes that make up the Android device ecosystem, creatingconsistent, intuitive, and pleasing user interfaces can be a significant challenge. Write your applications knowing that users will often only glance at the screen. Make your applicationsintuitive and easy to use by reducing the number of controls and putting the most importantinformation front and center.

Expect Low Speeds, High LatencyThe ability to incorporate some of the wealthof online information within your applicationsis incredibly powerful. Unfortunately, themobile Web isn't as fast, reliable, or readilyavailable as we would like; so, when you'redeveloping your Internet-based applications,it's best to assume that the network connectionwill be slow, intermittent, and expensive.With unlimited 4G data plans and citywideWi-Fi, this is changing, but designing for theworst case ensures that you always deliver ahigh-standard user experience. This also meansmaking sure that your applications can handle losing (or not finding) a data connection. The Android Emulator enables you to controlthe speed and latency of your network connection.

❖ At What Cost?

It's obvious why any costs associated with functionality in your applications should be minimized,and that users should be made aware when an action they perform might result in their beingcharged.It's a good approach to assume that there's a cost associated with any action involving an interactionwith the outside world. In some cases (such as with GPS and data transfer), the user can toggleAndroid settings to disable a potentially costly action. As a developer, it's important that you useand respect those settings within your application.

In any case, it's important to minimize interaction costs by doing the following:

- Transferring as little data as possible
- Caching data and geocoding results to eliminate redundant or repetitive lookups
- Stopping all data transfers and GPS updates when your Activity is not visible in the foreground(provided they're only used to update the UI)
- Keeping the refresh/update rates for data transfers (and location lookups) as low aspracticable
- Scheduling big updates or transfers at off-peak times or when connected via Wi-Fi by usingAlarms and Broadcast Receivers
- Respecting the user's preferences for background data transfersOften the best solution is to use a lower-quality option that comes at a lower cost.

1.14.2 Considering the User's Environment

Although Android has already expanded beyond its roots as a mobile phone platform, most Androiddevices are phones or tablet devices. For most people, such a device is first and foremost a phone,secondly an SMS

and email communicator, thirdly a camera, and fourthly an MP3 player. Theapplications you write will most likely be in the fifth category of "useful stuff."

That's not a bad thing — they'll be in good company with others, including Google Maps and theweb browser. That said, each user's usage model will be different; some people will never use theirdevice to listen to music, some devices don't support telephony, and some don't include cameras but the multitasking principle inherent in a device as ubiquitous as it is indispensable is an importantconsideration for usability design.

It's also important to consider when and how your users will use your applications. People use theirmobiles all the time — on the train, walking down the street, or even while driving their cars. Youcan't make people use their phones appropriately, but you can make sure that your applicationsdon't distract them any more than necessary.

Make sure that your application:

- **Is predictable and well behaved** — Start by ensuring that your Activities suspend whenthey're not in the foreground. Android fi res event handlers when your Activity is paused orresumed, so you can pause UI updates and network lookups when your application isn't visible— there's no point updating your UI if no one can see it.
- **Switches seamlessly from the background to the foreground** — With the multitasking natureof mobile devices, it's likely that your applications will regularly move into and out of thebackground. It's important that they "come to life" quickly and seamlessly. Android's nondeterministicprocess management means that if your application is in the background, there'severy chance it will get killed to free resources. This should be invisible to the user.
- **Is polite** — Your application should never steal focus or interrupt a user's current Activity.Instead, use Notifications (detailed in Chapter 10) to request your user's attention whenyour application isn't in the foreground. There are several ways to alert usersincoming calls are announced by a ringtone and/or vibration; when you have unread messages,the LED flashes; and when you have new voice mail, a small unread mail icon appearsin the status bar.
- **Presents an attractive and intuitive UI** — Your application is likely to be one of several inuse at any time, so it's important that the UI you present is easy to use. Spend the time andresources necessary to produce a UI that is as attractive as it is functional, and don't forceusers to interpret and relearn your application every time they load it. Using it should besimple, easy, and obvious — particularly given the limited screen space and distracting userenvironment.
- **Is responsive** — Responsiveness is one of the most critical design considerations on a mobiledevice. You've no doubt experienced the frustration of a "frozen" piece of software; the multifunctionalnature of a mobile makes this even more annoying. With the possibility of delayscaused by slow and unreliable data connections, it's important that your application useworker threads and background Services to keep your Activities responsive and, more important,to stop them from preventing other applications from responding promptly.

1.14.3. Developing for Android

The Android design philosophy demands that applications be designed for:

- Performance
- Responsiveness
- Freshness
- Security
- Seamlessness
- Accessibility

❖ **Being Fast and Efficient**

In a resource-constrained environment, being fast means being efficient. A lot of what you already know about writing efficient code will be applicable to Android, but the limitations of embeddedsystems and the use of the Dalvik VM mean you can't take things for granted.The smart bet for advice is to go to the source.

❖ **Being Responsive**

Android takes responsiveness very seriously. Android enforcesresponsiveness with the Activity Manager and WindowManager. If either service detects an unresponsive application,it will display an "[Application] is not responding" dialog —previously described as a *force close* error,

Android monitors two conditions to determine responsiveness:

- An application must respond to any user action, such as a key press or screen touch, within five seconds.
- A Broadcast Receiver must return from its onReceive handler within 10 seconds.Ensuring Data Freshness. The ability to multitask is a key feature in Android. One of the most important use cases for backgroundServices is to keep your application updated while it's not in use.

Where a responsive application reacts quickly to user interaction, a fresh application quickly displaysthe data users want to see and interact with. From a usability perspective, the right time toupdate your application is immediately before the user plans to use it. In practice, you need to weighthe update frequency against its effect on the battery and data usage.When designing your application, it's critical that you consider how often you will update the datait uses, minimizing the time users are waiting for refreshes or updates, while limiting the effect ofthese background updates on the battery life.

1.14.4 Ensuring Data Freshness

The ability to multitask is a key feature in Android. One of the most important use cases for backgroundServices is to keep your application updated while it's not in use.Where a responsive application reacts quickly to user interaction, a fresh application quickly displaysthe data users want to see and interact with. From a usability perspective, the right time toupdate your application is immediately before the user plans to use it. In practice, you need to weighthe update frequency against its effect on the battery and data usage.

1.14.5 Developing Secure Applications

Android applications have access to networks and hardware, can be distributed independently, andare built on an open-source platform featuring open communication, so it shouldn't be surprising that security is a significant consideration.For the most part, users need to take responsibility for the applications they install and the permissionsrequests they accept. The Android security model sandboxes each application and restrictsaccess to services and functionality by requiring applications to declare the permissions they require.During installation users are shown the application's required permissions before they commit toinstalling it. You can use several techniques to help maintain device security, and they'llbe covered in more detail as you learn the technologies involved. In particular, you should do thefollowing:

- Require permissions for any Services you publish or Intents you broadcast. Take special carewhen broadcasting an Intent that you aren't leaking secure information, such as location data.
- Take special care when accepting input to your application from external sources, such as theInternet, Bluetooth, NFC, Wi-Fi Direct, SMS messages, or instant messaging (IM). You canfind out more about using Bluetooth, NFC, Wi-Fi Direct, and SMS for application messaging.
- Be cautious when your application may expose access to lower-level hardware to third-partyapplications.
- Minimize the data your application uses and which permissions it requires.

1.14.6 Ensuring a Seamless User Experience

The idea of a seamless user experience is an important, if somewhat nebulous, concept. What dowe mean by *seamless*? The goal is a consistent user experience in which applications start, stop, andtransition instantly and without perceptible delays or jarring transitions.The speed and responsiveness of a mobile device shouldn't degrade the longer it's on. Android'sprocess management helps by acting as a silent assassin, killing background applications to freeresources as required. Knowing this, your applications should always present a consistent interface,regardless of whether they're being restarted or resumed. With an Android device typically running several third-party applications written by different developers,it's particularly important that these applications interact seamlessly.

Using Intents, applicationscan provide functionality for each other. Knowing your application may provide, or consume,third-party Activities provides additional incentive to maintain a consistent look and feel.Use a consistent and intuitive approach to usability. You can create applications that are revolutionaryand unfamiliar, but even these should integrate cleanly with the wider Android environment.Persist data between sessions, and when the application isn't visible, suspend tasks that use processorcycles, network bandwidth, or battery life. If your application has processes that need to continuerunning while your Activities are out of sight, use a Service, but hide these implementationdecisions from your users.

When your application is brought back to the front, or restarted, it should seamlessly return to itslast visible state. As far as your users are concerned, each application should be sitting silently, readyto be used but just out of sight.

1.14.7 Providing Accessibility

When designing and developing your applications, it's important not to assume that every userwill be exactly like you. This has implications for internationalization and usability but is criticalfor providing accessible support for users with disabilities that require them to interact with theirAndroid devices in different ways.

Android provides facilities to help these users navigate their devices more easily using text-to-speech,haptic feedback, and trackball or D-pad navigation.To provide a good user experience for everyone — including people with visual, physical, or agerelateddisabilities that prevent them from fully using or seeing a touchscreen — you can leverageAndroid's accessibility layer.

1.15 ANDROID DEVELOPMENT TOOLS

The Android SDK includes several tools and utilities to help you create, test, and debug your projects.A detailed examination of each developer tool is outside the scope of this book, but it's worthbriefly reviewing what's available. For additional details, check out the Android documentation athttp://developer.android.com/guide/developing/tools/index.html.

As mentioned earlier, the ADT plug-in conveniently incorporates many of these tools into theEclipse IDE, where you can access them from the DDMS perspective, including the following:

- **The Android Virtual Device and SDK Managers** — Used to create and manage AVDs and todownload SDK packages, respectively. The AVD hosts an Emulator running a particular buildof Android, letting you specify the supported SDK version, screen resolution, amount of SDcard storage available, and available hardware capabilities (such as touchscreens and GPS).
- **The Android Emulator** — An implementation of the Android VM designed to run withinan AVD on your development computer. Use the Emulator to test and debug your Androidapplications.
- **Dalvik Debug Monitoring Service (DDMS)** — Use the DDMS to monitor and control theEmulators on which you're debugging your applications.
- **Android Debug Bridge (ADB)** — A client-server application that provides a link to virtualand physical devices. It lets you copy files, install compiled application packages (.apk), andrun shell commands.
- **Logcat** — A utility used to view and filter the output of the Android logging system.
- **Android Asset Packaging Tool (AAPT)** — Constructs the distributable Android packagefi les (.apk).
- The following additional tools are also available:
- **SQLite3** — A database tool that you can use to access the SQLite database files created andused by Android.
- **Traceview** and **dmtracedump** — Graphical analysis tools for viewing the trace logs from yourAndroid application.
- **Hprof-conv** — A tool that converts HPROF profiling output files into a standard format toview in your preferred profiling tool.

- **MkSDCard** — Creates an SD card disk image that can be used by the Emulator to simulatean external storage card.
- **Dx** — Converts Java .class bytecode into Android .dex bytecode.
- **Hierarchy Viewer** — Provides both a visual representation of a layout's View hierarchy todebug and optimize your UI, and a magnified display to get your layouts pixel-perfect.
- **Lint** — A tool that analyzes your application and its resources to suggest improvements andoptimizations.
- **Draw9patch:** A handy utility to simplify the creation of NinePatch graphics using aWYSIWYG editor.
- **Monkey** and **Monkey Runner:** Monkey runs within the VM, generating pseudo-random userand system events. Monkey Runner provides an API for writing programs to control the VMfrom outside your application.
- **ProGuard** — A tool to shrink and obfuscate your code by replacing class, variable, andmethod names with semantically meaningless alternatives. This is useful to make your codemore difficult to reverse engineer.

Now take a look at some of the more important tools in more detail.

1.15.1 The Android Virtual Device Manager

The Android Virtual Device Manager is used to create and manage the virtual devices that will hostinstances of the Emulator.AVDs are used to simulate the software builds and hardware configurations available on differentphysical devices. This lets you test your application on a variety of hardware platforms withoutneeding to buy a variety of phones.

Each virtual device is configured with a name, atarget build of Android (based on the SDK version itsupports), an SD card capacity, and screen resolution,as shown in the Create new Android VirtualDevice (AVD) dialog in Fig. You can also choose to enable snapshots to savethe Emulator state when it's closed. Starting a newEmulator from a snapshot is significantly faster.Each virtual device also supports a number of specific hardware settings and restrictions that can beadded in the form of name-value pairs (NVPs) in thehardware table. Selecting one of the built-in skinswill automatically configure these additional settingscorresponding to the device the skin represents.The additional settings include the following:

- Maximum VM heap size
- Screen pixel density
- SD card support
- Existence of D-pad, touchscreen, keyboard,and trackball hardware
- Accelerometer, GPS, and proximity sensor support
- Available device memory
- Camera hardware (and resolution)
- Support for audio recording
- Existence of hardware back and home keys

Different hardware settings and screen resolutions will present alternative UI skins to represent the different hardware configurations. This simulates a variety of mobile device types. Some manufacturershave made hardware presets and virtual device skins available for their devices. Some, includingSamsung, are available as SDK packages.

1.15.2 Android SDK Manager

The Android SDK Manager can be used to see which version of the SDK you have installed and toinstall new SDKs when they are released.Each platform release is displayed, along with the platform tools and a number of additional supportpackages. Each platform release includes the SDK platform, documentation, tools, and examplescorresponding to that release.

1.15.3 The Android Emulator

The Emulator is available for testing and debugging your applications.The Emulator is an implementation of the Dalvik VM, making it as valid a platform for runningAndroid applications as any Android phone. Because it's decoupled from any particular hardware,it's an excellent baseline to use for testing your applications.Full network connectivity is provided along with the ability to tweak the Internet connection speedand latency while debugging your applications.

You can also simulate placing and receiving voicecalls and SMS messages.The ADT plug-in integrates the Emulator into Eclipse so that it's launched automatically within theselected AVD when you run or debug your projects. If you aren't using the plug-in or want to use theEmulator outside of Eclipse, you can telnet into the Emulator and control it from its console. (Formore details on controlling the Emulator, check out the documentation at http://developer.android.com/guide/developing/tools/emulator.html.)To execute the Emulator, you first need to create a virtual device, as described in the previous section.The Emulator will launch the virtual device and run a Dalvik instance within it.

1.15.4 The Dalvik Debug Monitor Service

The Emulator enables you to see how your application will look, behave, and interact, but to actuallysee what's happening under the surface, you need the Dalvik Debug Monitoring Service. TheDDMS is a powerful debugging tool that lets you interrogate active processes, view the stack andheap, watch and pause active threads, and explore the filesystem of any connected Android device.The DDMS perspective in Eclipse also provides simplified access to screen captures of the Emulatorand the logs generated by LogCat.If you're using the ADT plug-in, the DDMS tool is fully integrated into Eclipse and is available fromthe DDMS perspective. If you aren't using the plug-in or Eclipse, you can run DDMS from the commandline (it's available from the tools folder of the Android SDK), and it will automatically connectto any running device or Emulator.

1.15.5 The Android Debug Bridge

The *Android Debug Bridge* (ADB) is a client-service application that lets you connect with anAndroid device (virtual or actual). It's made up of three components:

A daemon running on the device or Emulator

- A service that runs on your development computer
- Client applications (such as the DDMS) that communicate with the daemon through theservice

As a communications conduit between your development hardware and the Android device/Emulator, the ADB lets you install applications, push and pull fi les, and run shell commands on thetarget device. Using the device shell, you can change logging settings and query or modify SQLitedatabases available on the device.The ADT tool automates and simplifies a lot of the usual interaction with the ADB, includingapplication installation and updating, fi le logging, and file transfer (through the DDMSperspective).

1.15.6 The Hierarchy Viewer and Lint Tool

To build applications that are fast and responsive, you need to optimize your UI. The HierarchyViewer and Lint tools help you analyze, debug, and optimize the XML layout definitions usedwithin your application.The Hierarchy Viewer displays a visual representation of the structure of your UI layout. Starting atthe root node, the children of each nested View (including layouts) is displayed in a hierarchy. EachView node includes its name, appearance, and identifier.

To optimize performance, the performance of the layout, measure, and draw steps of creating theUI of each View at runtime is displayed. Using these values, you can learn the actual time taken tocreate each View within your hierarchy, with colored “traffic light” indicators showing the relativeperformance for each step. You can then search within your layout for Views that appear to be takinglonger to render than they should.

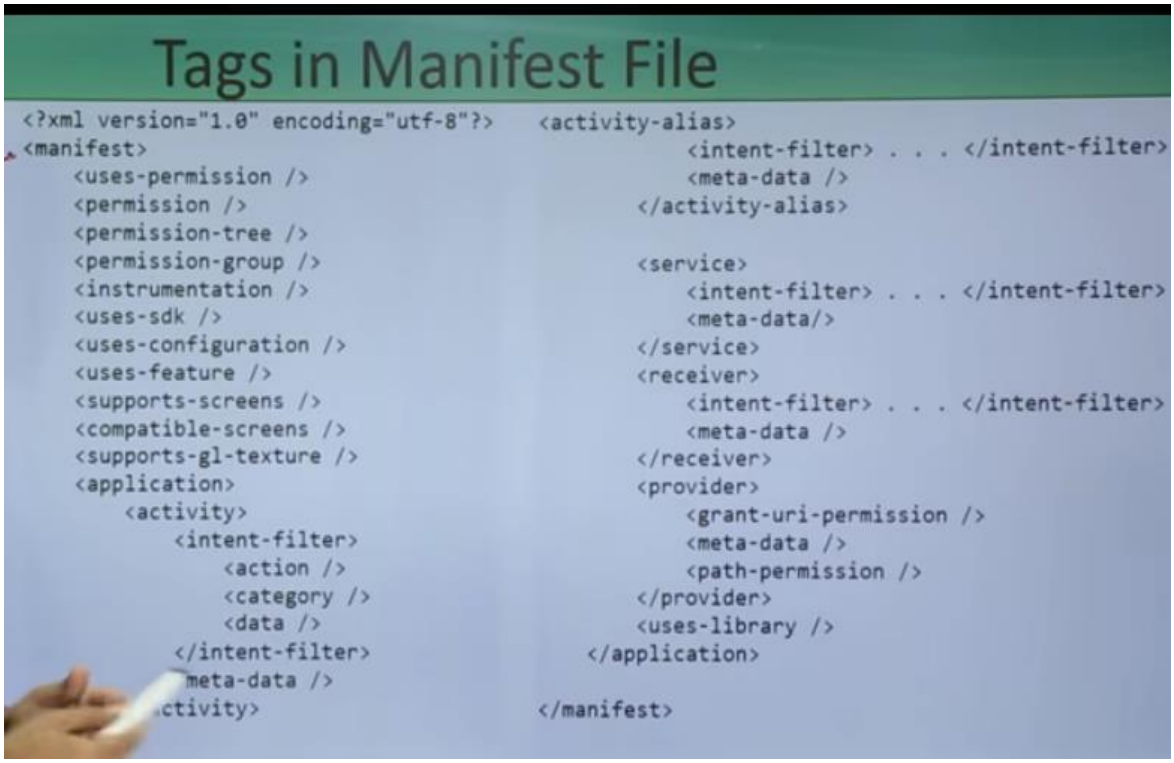
The Lint tool helps you to optimize your layouts by checking them for a series of commoninefficiencies that can have a negative impact on your application’s performance. Commonissues include a surplus of nested layouts, a surplus of Views within a layout, and unnecessaryparent Views.

1.15.7 Monkey and Monkey Runner

Monkey and Monkey Runner can be used to test your applications stability from a UI perspective.Monkey works from within the ADB shell, sending a stream of pseudo-random system and UIevents to your application. It’s particularly useful to stress test your applications to investigate edgecasesyou might not have anticipated through unconventional use of the UI.Alternatively, Monkey Runner is a Python scripting API that lets you send specific UI commands tocontrol an Emulator or device from outside the application. It’s extremely useful for performing UI,functional, and unit tests in a predictable, repeatable fashion.

UNIT - II

Application Manifest File



Each Android project includes a manifest fi le, AndroidManifest.xml, stored in the root of its project hierarchy.The manifest defines the structure and metadata of your application, its components, and its requirements.The manifest can also specify application metadata (such as its icon, version number, or theme). And additional top-level nodes can specify any required permissions, unit tests, and define hardware, screen, or platform requirements.

2.1 A Closer Look at the Application Manifest

The following XML snippet shows a typical manifest node:

<manifest xmlns: android=”http://schemas.android.com/apk/res/android”

package=“com.paad.myapp“

android:versionCode=“1“

android:versionName=“0.9 Beta“

android:installLocation=“preferExternal“>

[... manifest nodes ...]

</manifest>

The following list gives a summary of the available manifest sub-node tags and provides an XML snippet demonstrating how each tag is used:

2.1.1 uses-sdk:

This node enables you to define a minimum and maximum SDK version that must be available on a device for your application to function properly, and target SDK for which it has been designed using a combination of minSDKVersion, maxSDKVersion, and targetSDKVersion attributes, respectively.

Setting a target SDK version tells the system that there is no need to apply any forward- or backward-compatibility changes to support that particular version. To take advantage of the newest platform UI improvements, get SDK of your application to the latest platform release after you confirm it behaves as expected, even if you aren't making use of any new APIs.The maximum SDK defines an upper limit you are willing to support and your application will not be visible on the Google Play Store for devices running a higher platform release. Devices running on platforms higher than Android 2.0.1 (API level 6) will ignore any maximum SDK values at installation time.

```
<uses-sdk  android:minSdkVersion="6"
           android:targetSdkVersion="15"/>
```

2.1.2. uses-configuration:

The uses-configuration nodes specify each combination of input mechanisms are supported by your application. You shouldn't normally need to include this node, though it can be useful for games that require particular input controls.You can specify any combination of input devices that include the following:

- reqFiveWayNav — Specify true for this attribute if you require an input device capable of navigating up, down, left, and right and of clicking the current selection. This includes both trackballs and directional pads (D-pads).
- reqHardKeyboard — If your application requires a hardware keyboard, specify true.
- reqKeyboardType — Lets you specify the keyboard type as one of nokeys, qwerty, twelvekey, or undefined.
- reqNavigation — Specify the attribute value as one of nonav, dpad, trackball, wheel, or undefined as a required navigation device.
- reqTouchScreen — Select one of no touch, stylus, finger, or undefined to specify the required touch screen input.

You can specify multiple supported configurations, for example, a device with a finger touch screen, a trackball, and either a QUERTY or a twelve-key hardware keyboard, as shown here:

```
<uses-configuration
        android:reqTouchScreen="finger"
        android:reqNavigation="trackball"
```

```
android:reqHardKeyboard="true"
android:reqKeyboardType="qwerty"/>
<uses-configuration
android:reqTouchScreen="finger"
android:reqNavigation="trackball"
android:reqHardKeyboard="true"
android:reqKeyboardType="twelvekey"/>
```

2.1.3 uses-feature:

Android is available on a wide variety of hardware platforms. Use multiple uses-feature nodes to specify which hardware features your application requires.This prevents your application from being installed on a device that does not include a required piece of hardware, such as NFC hardware, as follows:

```
<uses-feature
android:name="android.hardware.nfc" />
```

You can require support for any hardware that is optional on a compatible device.

- **Audio** — For applications that requires a low-latency audio pipeline. Note that at the time of writing this book, no Android devices satisfied this requirement.
- **Bluetooth** — Where a Bluetooth radio is required.
- **Camera** — For applications that require a camera. You can also require (or set as options) autofocus, fl ash, or a front-facing camera.
- **Location** — If you require location-based services. You can also specify either network or GPS support explicitly.
- **Microphone** — For applications that require audio input.
- NFC — Requires NFC (near-fi eld communications) support.
- **Sensors** — Enables you to specify a requirement for any of the potentially available hardware sensors.
- **Telephony** — Specify that either telephony in general, or a specifi c telephony radio (GSM or CDMA) is required.
- **Touchscreen** — To specify the type of touch-screen your application requires.
- USB — For applications that require either USB host or accessory mode support.
- Wi-Fi — Where Wi-Fi networking support is required.

In particular, requesting permission to access Bluetooth, the camera, any of the location servicepermissions, audio recording, Wi-Fi, and telephony-related permissions implies the corresponding

hardware features.You can override these implied requirements by adding a required attribute and setting it to false for example, a note-taking application that supports recording an audio note:

```
<uses-feature  android:name="android.hardware.microphone"
              android:required="false" />
```

The camera hardware also represents a special case. For compatibility reasons requesting permission to use the camera, or adding a uses-feature node requiring it, implies a requirement for the camera to support autofocus. You can specify it as optional as appropriate:

```
<uses-feature  android:name="android.hardware.camera" />
<uses-feature  android:name="android.hardware.camera.autofocus
        android:required="false" />
<uses-feature  android:name="android.hardware.camera.flash"
              android:required="false" />
```

You can also use the uses-feature node to specify the minimum version of OpenGL required by your application. Use the glEsVersion attribute, specifying the OpenGL ES version as an integer. The higher 16 bits represent the major number and the lower 16 bits represent the minor number, so version 1.1 would be represented as follows:

```
<uses-feature android:glEsVersion="0x00010001" />
```

2.1.4 supports-screens:

The first Android devices were limited to 3.2" HVGA hardware.Since then, hundreds of new Android devices have been launched including tiny 2.55" QVGA phones, 10.1" tablets, and 42" HD televisions.

- **smallScreens** — Screens with a resolution smaller than traditional HVGA (typically, QVGA screens).
- **normalScreens** — Used to specify typical mobile phone screens of at least HVGA, including WVGA and WQVGA.
- **largeScreens** — Screens larger than normal. In this instance a large screen is considered to be significantly larger than a mobile phone display.
- **xlargeScreens** — Screens larger than large-typically tablet devices.
- **requiresSmallestWidthDp** — Enables you to specify a minimum supported screen width in device independent pixels.
- **compatibleWidthLimitDp** — Specifies the upper bound beyond which your application may not scale.
- **largestWidthLimitDp** — Specifies the absolute upper bound beyond which you know your application will not scale appropriately.

```
<supports-screens      android:smallScreens="false"
```

```
android:normalScreens="true"
android:largeScreens="true"
android:xlargeScreens="true"
android:requiresSmallestWidthDp="480"
android:compatibleWidthLimitDp="600"
android:largestWidthLimitDp="720"/>
```

2.1.5 supports-gl-texture— Declares that the application is capable of providing texture assets that are compressed using a particular GL texture compression format.You must use multiple supports-gl-texture elements if your application is capable of supporting multiple texture compression formats. You can fi nd the most up-to-date list of supported GL texture compression format values at

http://developer.android.com/guide/topics/manifest/supports-gl-texture-element.html.

```
<supports-gl-texture android:name ="GL_OES_compressed_ETC1_RGB8_texture" />
```

- **uses-permission** — As part of the security model, it declare the user permissions your application requires.Permissions are required for many APIs and method calls, generally those with an associated cost or security implication (such as dialing, receiving SMS, or using the location-based services).Your application components can also create permissions to restrict access to shared application components

```
<uses-permission
        android:name ="android.permission-ACCESS_FINE_LOCATION"/>
```

Your application components can then create permissions by adding an **android:permission** attribute. Then you can include a **uses-permission** tag in your manifest to use these protected components.

```
<permission   android:name="com.paad.DETONATE_DEVICE"
              android:protectionLevel="dangerous"
              android:label="Self Destruct"
              android:description="@string/
      detonate_description">
</permission>
```

2.1.6 Instrumentation:

Instrumentation classes provide a test framework for your application components at run time.They provide hooks to monitor your application and its interaction with the system resources. Create a new node for each of the test classes you've created for your application.

```
<instrumentation
        android:label="My Test"
        android:name=".MyTestClass"
```

```
        android:targetPackage="com.paad.apackage">
</instrumentation>
```

- **application** — A manifest can contain only one application node. It uses attributes to specify the metadata for your application (including its title, icon, and theme). During development you should include a **debuggable** attribute set to true to enable debugging, then be sure to disable it for your release builds.

```
<application android:icon="@drawable/icon"
        android:logo="@drawable/logo"
        android:theme="@android:style/Theme.Light"
        android:name=".MyApplicationClass"
        android:debuggable="true">
        [ ... application nodes ... ]
</application>
```

- **Activity**:

An activity tag is required for every Activity within your application.Use the **android:name** attribute to specify the Activity class name. You must include the main launch Activity and any other Activity that may be displayed.Trying to start an Activity that's not defined in the manifest will throw a runtime exception.Each Activity node supports intent-filter child tags that define the Intents that can be used to start the Activity

```
<activity android:name=".MyActivity"
    android:label="@string/app_name">
        <intent-filter>
            <action android:name
    ="android.intent.action.MAIN"/>          <category android:name
            =»android.intent.category.
    LAUNCHER» />
        </intent-filter>
</activity
```

- **Service:**As with the activity tag, add a service tag for each Service class used in your application.

```
    <service android:name=".MyService">
    </service>
```

- **Provider**:

Provider tags specify each of your application's Content Providers.Content Providers are used to manage database access and sharing.

```
<provider android:name=".MyContentProvider"
        android:authorities=  "com.paad.myapp.MyContentProvider"/>
```

- **receiver**:

By adding a receiver tag, you can register a Broadcast Receiver without having to launch your application first.Broadcast Receivers are like global event listeners that, when registered, will execute whenever a matching Intent is broadcast by the system or an application.Each receiver node supports intent-filter child tags that define the Intents that can be used to trigger the receiver:

```
<receiver android:name=".MyIntentReceiver">
        <intent-filter>
        <action android:name=
"com.paad.mybroadcastaction" />
        </intent-filter>
</receiver>
```

- **uses-library** — Used to specify a shared library that this application requires. For example "Maps, Geocoding, and Location-Based Services," are packaged as a separate library that is not automatically linked.

```
<uses-library  android:name="com.google.android.maps"
        android:required="false"/>
```

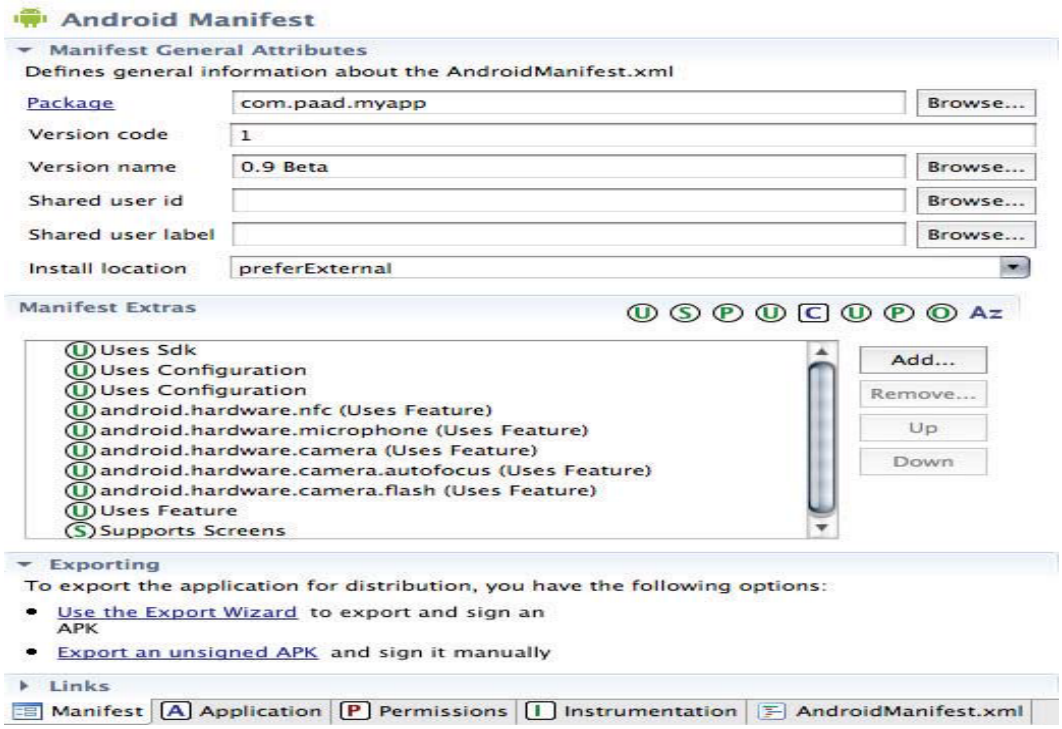
2.2 Android Manifest Editor:

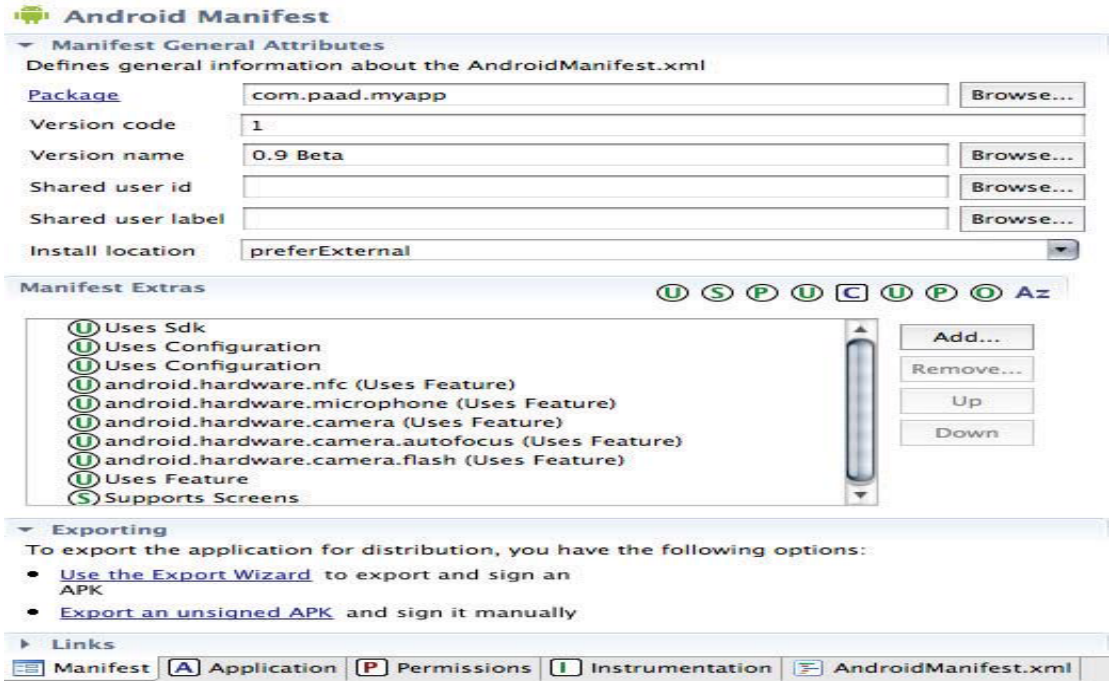
The Android Development Tools (ADT) plug-in includes a Manifest Editor, so you don't have to manipulate the underlying XML directly.right-click the AndroidManifest.xml fi le in your project folder, and select **Open With Android Manifest Editor.**This presents the Android Manifest Overview screenIt also provides shortcut links to the Application, Permissions, Instrumentation, and raw XML screens.

SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

MOBILE APPLICATIONS USING ANDROID





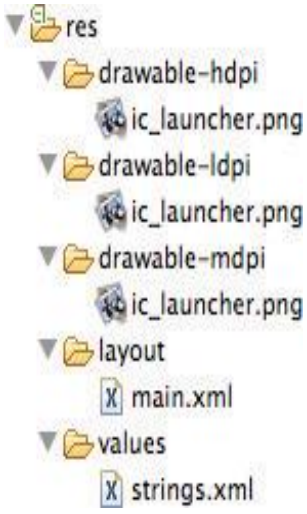
2.3 EXTERNALIZING RESOURCES

It's always good practice to keep non-code resources, such as images and string constants, external to your code. Android supports the externalization of resources, ranging from simple values such as strings and colors to more complex resources such as images (Drawables), animations, themes, and menus, most powerful resources are layouts.By externalizing resources, you make them easier to maintain, update, and manage also easily define alternative resource values to support variations in hardware — particularly, screen size and resolution.

❖ **Creating Resources:**

Application resources are stored under the res folder in your project hierarchy. Each of the available resource types is stored in subfolders, grouped by resource type.

- **drawable-ldpi, drawable-mdpi, drawable-hdpi**- Application icon
- **Layout** – layouts
- **Values**- simple values for resource definitions
  such as strings, colors, dimensions, styles,



and integer and String arrays

When your application is built, these resources will be compiled and compressed as efficiently as possible and included in your application package.This process also generates an R class file that contains references to each of the resources you include in your project with the advantage of design-time syntax checking.Within each XML file, you indicate the type of value being stored using tags

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">To Do List</string>
```

```
<plurals name=”androidPlural”>
    <item quantity=”one”>One android</item>
    <item quantity=”other”>%d androids</item>
</plurals>
<colorname=”app_background”>#FF0000FF</co lor>
<dimen name=”default_border”>5px</dimen>
<string-array name=“string_array“>
    <item>Item 1</item>
    <item>Item 2</item>
    <item>Item 3</item>
</string-array>
<array name=“integer_array“>
    <item>3</item>
    <item>2</item>
    <item>1</item>
</array>
</resources>
```

❖ Strings:

String resources are specified with the string tag, as shown in the following XML snippet:

<string name=”stop_message”>Stop.</string>

Android supports simple text styling, so you can use the HTML tags , <i>, and <u> to apply bold, italics, or underlining.

<string name=”stop_message”>Stop.</string>

❖ Colors

Use the color tag to define a new color resource with any of the following notations:

- #RGB
- #RRGGBB
- #ARGB
- #AARRGGBB

The following example shows how to specify a fully opaque blue and a partially transparent green:

<color name=”opaque_blue”>#00F</color>

<color name="transparent_green">#7700FF00</color>

❖ Dimensions

Dimensions are most commonly referenced within style and layout resources.They're useful for creating layout constants, such as borders and font heights.To specify a dimension resource, use the **dimen** tag, specifying the dimension value, followed by an identifier describing the scale of your dimension:

- px (screen pixels)
- in (physical inches)
- pt (physical points)
- mm (physical millimeters)
- dp (density-independent pixels)
- sp (scale-independent pixels)

The following XML snippet shows how to specify dimension values for a large font size and a standard border:

<dimen name="standard_border">5dp</dimen>

<dimen name="large_font_size">16sp</dimen>

❖ Styles and Themes

Style resources let your applications maintain a consistent look and feel by enabling you to specify the attribute values used by Views.To create a style, use a style tag that includes a name attribute and contains one or more item tags.Each item tag should include a name attribute used to specify the attribute (such as font size or color) being defined.

```
<?xml version="1.0" encoding="utf-8"?>
    <resources>
        <style name="base_text">
            <item name="android:textSize">14sp</item>
            <item name="android:textColor">#111</item>
        </style>
    </resources>
```

❖ Drawables

Drawable resources include bitmaps and NinePatches (stretchable PNG images).They also include complex composite Drawables, such as LevelListDrawables and StateListDrawables, that can be defined in XML.All Drawables are stored as individual files in the res/drawable folder.

❖ Layouts

Layout resources enable you to decouple your presentation layer from your business logic by designing UI layouts in XML rather than constructing them in code, which enables you to create optimized layouts for different hardware configurations, such as varying screen sizes, orientation, or the presence of keyboards and touchscreens You can use layouts to define the UI for any visual component, including Activities, Fragments and Widgets.Once defined in XML, the layout must be "inflated" into the user interface. Within an Activity this is done using setContentView (usually within the onCreate method), whereas Fragment Views are inflated using the inflate method from the Inflator object passed in to the Fragment's onCreateView handler.

❖ Animations

Android supports three types of animation:

1. **Property animations**
   - A tweened animation that can be used to potentially animate any property on the target object by applying incremental changes between two values.
   - This can be used for anything from changing the color or opacity of a View to gradually fade it in or out, to changing a font size, or increasing a character's hit points.
2. **View animations**
   - Tweened animations that can be applied to rotate, move, and stretch a View.
3. **Frame animations**
   - Frame-by-frame "cell" animations used to display a sequence of Drawable images.

❖ Property Animations:

Property animators were introduced in Android 3.0 (API level 11). It is a powerful framework that can be used to animate almost anything.Each property animation is stored in a separate XML fi le in the project's res/animator folder.

```
<?xml version="1.0" encoding="utf-8"?>
    <objectAnimator xmlns:android=    "http://schemas.android.com/apk/res/android"
    android:propertyName="alpha"
    android:duration="1000"
    android:valueFrom="0.0"
    android:valueTo="1.0"
/>
```

❖ View Animations

Each view animation is stored in a separate XML fi le in the project's res/anim folder. Table shows the valid attributes, and attribute values, supported by each animation type.

ANIMATION TYPE	ATTRIBUTES	VALID VALUES
Alpha	fromAlpha/toAlpha	Float from 0 to 1
Scale	fromXScale/toXScale	Float from 0 to 1
	fromYScale/toYScale	Float from 0 to 1
	pivotX/pivotY	String of the percentage of graphic width/height from 0% to 100%
Translate	fromX/toX	Float from 0 to 1
	fromY/toY	Float from 0 to 1
Rotate	fromDegrees/toDegrees	Float from 0 to 360
	pivotX/pivotY	String of the percentage of graphic width/height from 0% to 100%

You can create a combination of animations using the set tag. The following list shows some of the set tags available:

1. duration — Duration of the full animation in milliseconds.
2. startOffset — Millisecond delay before the animation starts.
3. fillBeforetrue — Applies the animation transformation before it begins.
4. fillAftertrue — Applies the animation transformation after it ends.
5. interpolator — Sets how the speed of this effect varies over time.

Reference the system animation resources atandroid:anim/interpolatorName.

❖ Frame-by-Frame Animations

Frame-by-frame animations produce a sequence of Drawables, each of which is displayed for a specified duration.

❖ Menus

Create menu resources to design your menu layouts in XML, rather than constructing them in code.You can use menu resources to define both Activity and context menus within your applications, and provide the same options you would have when constructing your menus in code.A menu is inflated within your application via the inflate method of the MenuInflator Service, usually

within the onCreateOptionsMenu method. Each menu definition is stored in a separate fi le, each containing a single menu, in the res/menu folder

```
<?xml version="1.0" encoding="utf-8"?>
    <menu xmlns:android= "http://schemas.android.com/apk/res/android">
        <item android:id="@+id/menu_refresh"
                android:title="@string/refresh_mi" />
        <item android:id="@+id/menu_settings"
                android:title="@string/settings_mi" />
    </menu>
```

2.3.1 Using Resources

In addition to the resources you supply, the Android platform includes several system resources that you can use in your applications. All resources can be used directly from your application code and can also be referenced from within other resources.It's important to note that when using resources, you shouldn't choose a particular specialized version. Android will automatically select the most appropriate value for a given resource identifier based on the current hardware, device, and language configurations.

❖ **Using Resources in Code**

Access resources in code using the static R class. R is a generated class based on your external resources, and created when your project is compiled.Each of the subclasses within R exposes its associated resources as variables, with the variable names matching the resource identifiers — for example, **R.string.app_name** or **R.drawable.icon**.The value of these variables is an integer that represents each resource's location in the resource table, *not an instance of the resource itself.*

setContentView, accepts a resource identifier, you can pass in the resource variable, as shown in the following code snippet:

```
// Inflate a layout resource.
setContentView(R.layout.main);
// Display a transient dialog box that displays the
// error message string resource.
    Toast.makeText(this, R.string.app_error,
    Toast.LENGTH_LONG).show();
```

When you need an instance of the resource itself, you need to use helper methods to extract them from the resource table. The resource table is represented within your application as an instance of the Resources class.

Resources myResources = getResources();

The Resources class includes getters for each of the available resource types and generally works by passing in the resource ID you'd like an instance of.

CharSequence styledText =myResources.getText(R.string.stop_message);

Drawable icon = myResources.getDrawable(R.drawable.app_icon);

int opaqueBlue = myResources.getColor(R.color.opaque_blue);

float borderWidth = myResources.getDimension(R.dimen.standard_border);

❖ Referencing Resources Within Resources

You can also use resource references as attribute values in other XML resources.To reference one resource from another, use the @ notation, as shown in the following snippet:

attribute="@[packagename:]resourcetype/resourceidentifier"

Example:

android:padding="@dimen/standard_border">

android:text="@string/stop_message"

android:textColor="@color/opaque_blue"

❖ Using System Resources

Accessing the system resources in code is similar to using your own resources. The difference is that you use the native Android resource classes available from android.R, rather than the application specific R class. The following code snippet uses the getString method available in the application context to retrieve an error message available from the system resources:

CharSequence httpError = getString(android.R.string.httpErrorBadUrl);

❖ Referring to Styles in the Current Theme

Using themes is an excellent way to ensure consistency for your application's UI. Rather than fully define each style, Android provides a shortcut to enable you to use styles from the currently applied theme.To do this, use ?android: rather than @ as a prefix to the resource you want to use. The following example shows a snippet of the preceding code but uses the current theme's text color rather than a system resource:

<EditText

android:id="@+id/myEditText"

```
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="@android:string/httpErrorBadUrl"
android:textColor="?android:textColor"
/>
```

2.3.2 Creating Resources for Different Languages and Hardware:

Using the directory structure described here, you can create different resource values for specific languages, locations, and hardware configurations.Android chooses from among these values dynamically at run time using its dynamic resource-selection mechanism.You can specify alternative resource values using a parallel directory structure within the res folder.A hyphen (-) is used to separate qualifiers that specify the conditions you provide alternatives forValues with French language and French Canadian location variations:

```
Project/
    res/
      values/
          strings.xml
      values-fr/
          strings.xml
      values-fr-rCA/
          strings.xml
```

The following list gives the qualifiers you can use to customize your resource values:

1. **Mobile Country Code and Mobile Network Code (MCC/MNC) —**The MCC is specified by mcc followed by the three-digit country code. You can optionally add the MNC using mnc and the two- or three-digit network code (for example, mcc234-mnc20 or mcc310). You can find a list of MCC/MNC codes on Wikipedia at http://en.wikipedia.org/wiki/MobileNetworkCode.
2. **Language and Region —**for example, en, en-rUS, or en-rGB
3. **Smallest Screen Width —** The lowest of the device's screen dimensions (height and width) specified in the form sw<Dimension value>dp (for example, sw600dp, sw320dp, or sw720dp).
4. **Available Screen Width —** The minimum screen width required to use the contained resources, specified in the form w<Dimension value>dp. The available screen width changes to reflect the current screen width when the device orientation changes. Android selects the largest value that doesn't exceed the currently available screen width.

5. **Available Screen Height —** The minimum screen height required to use the contained resources, specified in the form h<Dimension value>dp (for example, h720dp, h480dp, or h1280dp)
6. **Screen Size —** One of small (smaller than HVGA), medium (at least HVGA and typically smaller than VGA), large (VGA or larger), or xlarge (significantly larger than HVGA).
7. **Screen Aspect Ratio —** Specify long or notlong for resources designed specifically for wide screen. (For example, WVGA is long; QVGA is notlong.)
8. **Screen Orientation:** One of port (portrait), land (landscape), or square (square).
9. **Dock Mode —** One of car or desk. Introduced in API level 8.
10. **Screen Pixel Density** — Pixel density in dots per inch (dpi). Best practice is to supply ldpi,mdpi, hdpi, or xhdpi to specify low (120 dpi), medium (160 dpi), high (240 dpi), or extrahigh (320 dpi) pixel density assets, respectively.
11. **Keyboard Availability —** One of keys-exposed, keys-hidden, or keyssoft.
12. **Keyboard Input Type —** One of nokeys, qwerty, or 12key.
13. **Navigation Key Availability —** One of navexposed or navhidden.
14. **UI Navigation Type —** One of nonav, dpad, trackball, or wheel.
15. **Platform Version —** The target API level, specified in the form v<API Level> (for example, v7). Used for resources restricted to devices running at the specified API level or higher.

Any combination is supported; however, they must be used in the order given in the preceding listThe following example shows valid and invalid directory names for alternative layout resources.

VALID

layout-large-land

layout-xlarge-port-keyshidden

layout-long-land-notouch-nokeys

INVALID

values-rUS-en (out of order)

values-rUS-rUK (multiple values for a single qualifier)

2.3.3 Runtime Configuration Changes

Android handles runtime changes to the language, location, and hardware by terminating and restarting the active Activity. This forces the resource resolution for the Activity to be reevaluated and the most appropriate resource values for the new configuration to be selected.To have an Activity listen for runtime configuration changes, add an android:configChanges attributeto its

manifest node, specifying the configuration changes you want to handle.The following list describes some of the configuration changes you can specify:

- **mcc and mnc** — A SIM has been detected and the mobile country or network code (respectively) has changed.
- **locale** — The user has changed the device's language settings.

- **keyboardHidden** — The keyboard, d-pad, or other input mechanism has been exposed or hidden.
- **keyboard** — The type of keyboard has changed; for example, the phone may have a 12-key keypad that fl ips out to reveal a full keyboard, or an external keyboard might have been plugged in.
- **fontScale** — The user has changed the preferred font size.
- **uiMode** — The global UI mode has changed. This typically occurs if you switch between car mode, day or night mode, and so on.
- **orientation** — The screen has been rotated between portrait and landscape.
- **screenLayout** — The screen layout has changed; typically occurs if a different screen has been activated.
- **screenSize** — Introduced in Honeycomb MR2 (API level 12), occurs when the available screen size has changed, for example a change in orientation between landscape and portrait.
- **smallestScreenSize** — Introduced in Honeycomb MR2 (API level 12), occurs when the physical screen size has changed, such as when a device has been connected to an external display.

```
android:configChanges="screenSize|orientation|keyboardHidden">
```

Adding an android:configChanges attribute suppresses the restart for the specified configuration changes, instead triggering the onConfigurationChanged handler in the associated Activity. Override this method to handle the configuration changes yourself, using the passed-in Configuration object

```
@Override
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
        // [ ... Update any UI based on resource values ... ]
        if (newConfig.orientation ==
                Configuration.ORIENTATION_LANDSCAPE){
        // [ ... React to different orientation ...
        if (newConfig.keyboardHidden ==
            Configuration.KEYBOARDHIDDEN_NO) {
        // [ ... React to changed keyboard visibility ... ]
        }
}
```

2.4 THE ANDROID APPLICATION LIFECYCLE

Unlike many traditional application platforms, Android applications have limited control over their own lifecycles.Instead, application components must listen for changes in the application state and react accordingly, taking particular care to be prepared for untimely termination.Each Android application runs in its own process, each of which is running a separate instance of Dalvik. Memory and process management is handled exclusively by the run time.Android aggressively manages its resources, doing whatever's necessary to ensure a smooth and stable user experience. In practice that means that processes will be killed, in some case without warning, to free resources for higher-priority applications.

2.5. UNDERSTANDING AN APPLICATION'S PRIORITY AND ITS PROCESS' STATES

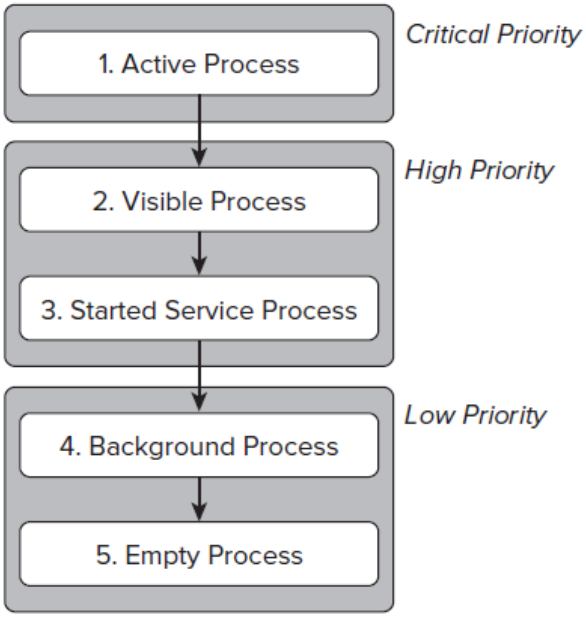
The order in which processes are killed to reclaim resources is determined by the priority of theirhosted applications. An application's priority is equal to that of its highest-priority component.If two applications have the same priority, the process that has been at that priority longest will be killed first. Process priority is also affected by interprocess dependencies; if an application has adependency on a Service or Content Provider supplied by a second application, the secondary applicationhas at least as high a priority as the application it supports.

Figure shows the priority tree used to determine the order of application termination.

1.Active processes

Active (foreground) processeshave application components the user is interacting withActive processes include the following:

- Activities in an active state
- Broadcast Receivers executing
- Services executing
- Running Services



2.Visible processes

Visible but inactive processes are those hosting "visible" Activities This happens when an Activity is only partially obscured.There are generally very few visible processes, and they'll be killed only under extreme circumstances to allow active processes to continue.

3. **Started Service processes**

Processes hosting Services that have been started. Because these Services don't interact directly with the user, they receive a slightly lower priority than visible. Activities or foreground Services. Applications with running Services are still considered foreground processes and won't be killed unless resources are needed for active or visible processes.

4. Background processes

Processes hosting Activities that aren't visible and that don't have any running Services. There will generally be a large number of background processes that Android will kill using a last-seen-fi rst-killed pattern in order to obtain resources for foreground processes.

5. Empty processes

To improve overall system performance, Android will often retain an application in memory after it has reached the end of its lifetime. Android maintains this cache to improve the start-up time of applications when they're relaunched. These processes are routinely killed, as required.

2.6 INTRODUCING THE ANDROID APPLICATION CLASS

Your application's *Application* object remains instantiated whenever your application runs.The *Application* class provides event handlers for application creation and termination, low memory conditions, and configuration changes Unlike *Activities*, the Application is not restarted as a result of

configuration changes. Extending the *Application* class with your own implementation enables you to do three things:

1. Respond to application level events broadcast by the Android run time such as low memory conditions.

2. Transfer objects between application components.

3. Manage and maintain resources used by several application components.

Of these, the latter two can be better achieved using a separate singleton class

2.6.1 Extending and Using the Application Class

```
import android.app.Application;
import android.content.res.Configuration;
public class MyApplication extends Application {
    private static MyApplication singleton;
    // Returns the application instance
    public static MyApplication getInstance() {
    return singleton;
    }
    @Override
    public final void onCreate() {
        super.onCreate();
        singleton = this;
    }
}
```

When created, you must register your new *Application* class in the manifest's application node using a name attribute, as shown in the following snippet:

```
<application android:icon="@drawable/icon"
                    android:name=".MyApplication">
                    [... Manifest nodes ...]
</application>
```

Create new state variables and global resources for access from within the application components:

```
MyObjectvalue = MyApplication.getInstance().getGlobalStateValue();
MyApplication.getInstance().setGlobalStateValue(myObjectValue);
```

Although this can be an effective technique for transferring objects between your loosely coupled application components, or for maintaining application state or shared resources, it is often better to create your own static singleton class rather than extending the Application class specifically unless you are also handling the lifecycle events described in the following section.

2.6.2 Overriding the Application Lifecycle Events

By overriding these methods, you can implement your own application-specific behavior for each of these circumstances:

- **onCreate** — Called when the application is created. Override this method to initialize your application singleton and create and initialize any application state variables or shared resources.
- **onLowMemory** — Provides an opportunity for well-behaved applications to free additional memory when the system is running low on resources.

This will generally only be called when background processes have already been terminated and the current foreground applications are still low on memory. Override this handler to clear caches or release unnecessary resources.

- **onTrimMemory** — An application specific alternative to the onLowMemory handler introduced in Android 4.0 (API level 13).
  - Called when the run time determines that the current application should attempt to trim its memory overhead – typically when it moves to the background.
  - It includes a level parameter that provides the context around the request.
- **onConfigurationChanged** — Unlike Activities Application objects are not restarted due to configuration changes.
  - If your application uses values dependent on specific configurations, override this handler to reload those values and otherwise handle confi guration changes at an application level.

You must always call through to the superclass event handlers when overriding these methods.

```
@Override
public final void onCreate() {
super.onCreate();
singleton = this;
}
@Override
public final void onLowMemory() {
```

```
super.onLowMemory();
}
@Override
public final void onTrimMemory(int level) {
super.onTrimMemory(level);
}
@Override
public final void onConfigurationChanged(Configuration newConfig) {
super.onConfigurationChanged(newConfig);
}
```

2.7 ANDROID ACTIVITIES

Each Activity represents a screen that an application can present to its users. The more complicatedyour application, the more screens you are likely to need.Typically, this includes at least a primary interface screen that handles the main UI functionalityof your application. This primary interface generally consists of a number of Fragments that make upyour UI and is generally supported by a set of secondary Activities. To move between screens you starta new Activity

2.7.1 Creating Activities

Extend Activity to create a new Activity class. Within this new class you must defi ne the UI andimplement your functionality. Listing 3-9 shows the basic skeleton code for a new Activity.

```
package com.paad.activities;
import android.app.Activity;
import android.os.Bundle;
public class MyActivity extends Activity {
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
}
}
```

The base Activity class presents an empty screen that encapsulates the window display handling.An empty Activity isn't particularly useful, so the fi rst thing you'll want to do is create the UI withFragments, layouts, and Views.

Views are the UI controls that display data and provide user interaction. Android provides severallayout classes, called *View Groups*, which can contain multiple Views to help you layoutyour UIs. Fragments are used to encapsulate segments of your UI, making it simple to createdynamic interfaces that can be rearranged to optimize your layouts for different screen sizes andorientations.

To assign a UI to an Activity, call setContentView from the onCreate method of your Activity.In this first snippet, an instance of a TextView is used as the Activity's UI:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
TextView textView = new TextView(this);
setContentView(textView);
}
```

Usually, you'll want to use a more complex UI design. You can create a layout in code using layoutView Groups, or you can use the standard Android convention of passing a resource ID for alayout defi ned in an external resource, as shown in the following snippet:

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

To use an Activity in your application, you need to register it in the manifest. Add a newactivity tag within the application node of the manifest; the activity tag includes attributesfor metadata, such as the label, icon, required permissions, and themes used by the Activity. AnActivity without a corresponding activity tag can't be displayed — attempting to do so willresult in a runtime exception.

```
<activity android:label="@string/app_name"
android:name=".MyActivity">
</activity>
```

Within the activity tag you can add intent-filter nodes that specify the Intents that can beused to start your Activity. Each Intent Filter defines one or more actions and categories that yourActivity supports. Intents and Intent Filters are covered in depth in Chapter 5, but it's worth notingthat for an Activity to be

available from the application launcher, it must include an IntentFilter listening for the MAIN action and the LAUNCHER category, as highlighted in Listing 3-10.

```
<activity android:label="@string/app_name"
android:name=".MyActivity">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

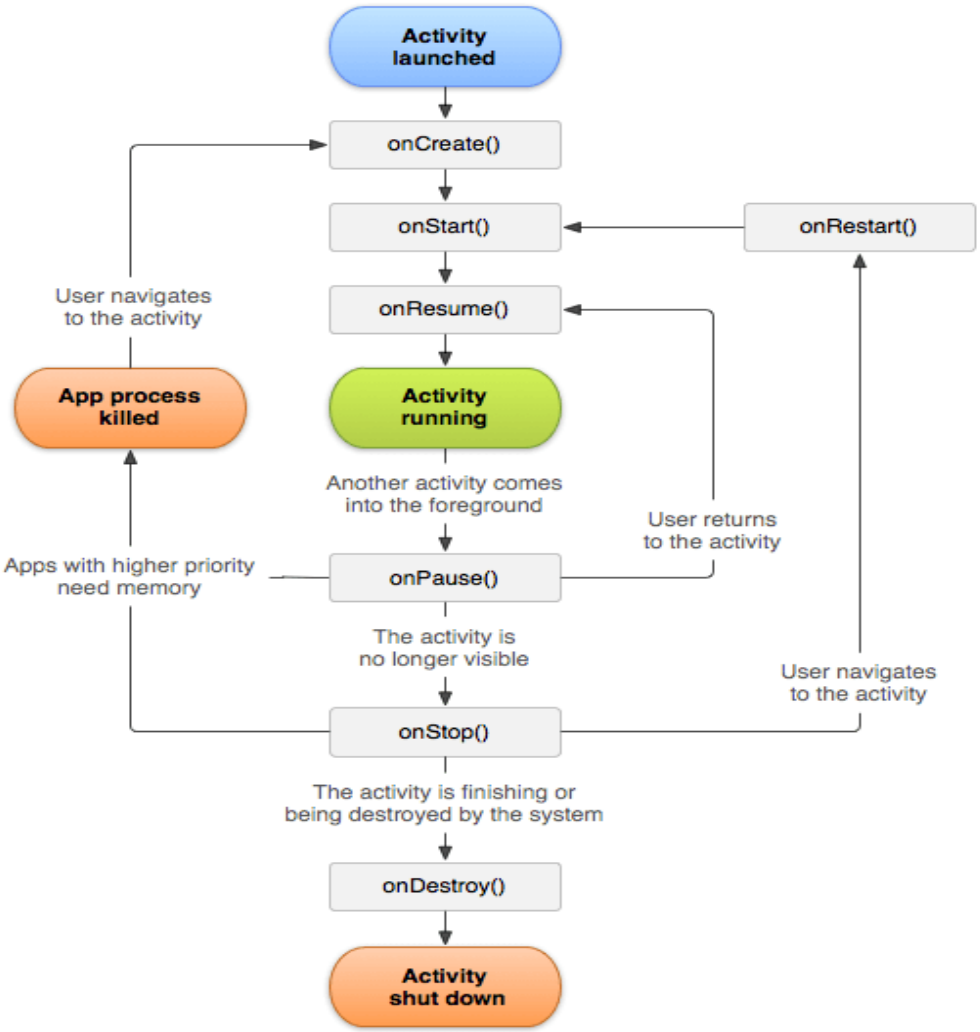
2.7.2 The Activity Lifecycle

A good understanding of the Activity lifecycle is vital to ensure that your application provides aseamless user experience and properly manages its resources.

SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

MOBILE APPLICATIONS USING ANDROID

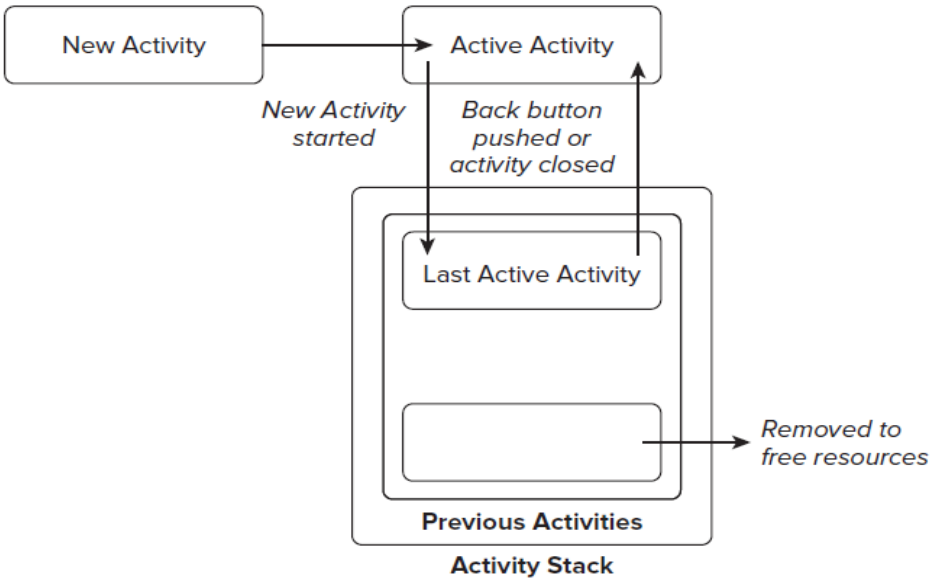


- An activity is the single screen in android. It is like window or frame of Java.
- By the help of activity, you can place all your UI components or widgets in a single screen.
- The 7 lifecycle method of Activity describes how activity will behave at different states.
- Android Activity Lifecycle is controlled by 7 methods of android.app.Activity class. The android Activity is the subclass of ContextThemeWrapper class.
- An activity represents a single screen in your app with which your user can perform a single, focused task.
- An app usually consists of multiple activities that are loosely bound to each other.
- Typically, one activity in an application is specified as the "main" activity, which is presented to the user when the app is launched.
- Each activity can then start other activities in order to perform different actions.

Method	Description
onCreate	called when activity is first created.
onStart	called when activity is becoming visible to the user.
onResume	called when activity will start interacting with the user.
onPause	called when activity is not visible to the user.
onStop	called when activity is no longer visible to the user.
onRestart	called after your activity is stopped, prior to start.
onDestroy	called before the activity is destroyed.

As explained earlier, Android applications do not control their own process lifetimes; the Androidrun time manages the process of each application, and by extension that of each Activity within it.Although the run time handles the termination and management of an Activity's process, theActivity's state helps determine the priority of its parent application. The application priority, inturn, influences the likelihood that the run time will terminate it and the Activities running within it.

Activity StacksThe state of each Activity is determined by its position on the Activity stack, a last-in–first-out collectionof all the currently running Activities. When a new Activity starts, it becomes active and ismoved to the top of the stack. If the user navigates back using the Back button, or the foregroundActivity is otherwise closed, the next Activity down on the stack moves up and becomes active.Figure illustrates this process.



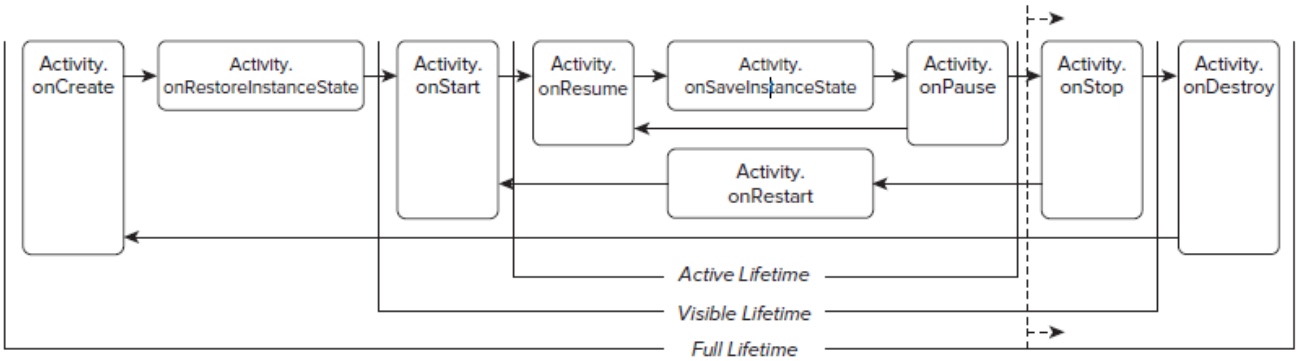
2.7.3 Activity States

As Activities are created and destroyed, they move in and out of the stack, as shown in FigureAs they do so, they transition through four possible states:

- **Active** — When an Activity is at the top of the stack it is the visible, focused, foregroundActivity that is receiving user input. Android will attempt to keep it alive at all costs, killingActivities further down the stack as needed, to ensure that it has the resources it needs. Whenanother Activity becomes active, this one will be paused.
- **Paused** — In some cases your Activity will be visible but will not have focus; at this pointit's paused. This state is reached if a transparent or non-full-screen Activity is active in frontof it. When paused, an Activity is treated as if it were active; however, it doesn't receive userinput events. In extreme cases Android will kill a paused Activity to recover resources for theactive Activity. When an Activity becomes totally obscured, it is stopped.
- **Stopped** — When an Activity isn't visible, it "stops." The Activity will remain in memory,retaining all state information; however, it is now a candidate for termination when the systemrequires memory elsewhere. When an Activity is in a stopped state, it's important to savedata and the current UI state, and to stop any non-critical operations. Once an Activity hasexited or closed, it becomes inactive.
- **Inactive** — After an Activity has been killed, and before it's been launched, it's inactive.Inactive Activities have been removed from the Activity stack and need to be restarted beforethey can be displayed and used.State transitions are nondeterministic and are handled entirely by the Android memory manager.Android will start by closing applications that contain inactive Activities, followed by those that arestopped. In extreme cases, it will remove those that are paused.

2.7.4 Monitoring State Changes

To ensure that Activities can react to state changes, Android provides a series of event handlers that are fired when an Activity transitions through its full, visible, and active lifetimes. Figure summarizesthese lifetimes in terms of the Activity states described in the previous section.



2.7.5 Understanding Activity Lifetimes

Within an Activity's full lifetime, between creation and destruction, it goes through one or moreiterations of the active and visible lifetimes. Each transition triggers the method handlers previouslydescribed. The following sections provide a closer look at each of these lifetimes and the events thatbracket them.

1. The Full Lifetime

The full lifetime of your Activity occurs between the first call to onCreate and the final call toonDestroy. It's not uncommon for an Activity's process to be terminated *without* the onDestroymethod being called.Use the onCreate method to

- initialize your Activity:
- inflate the user interface,
- get references toFragments,
- allocate references to class variables,
- bind data to controls, and start Services andTimers.

If the Activity was terminated unexpectedly by the runtime, the onCreate method ispassed a Bundle object containing the state saved in the last call to onSaveInstanceState. Youshould use this Bundle to restore the UI to its previous state, either within the onCreate method oronRestoreInstanceState.

Override onDestroy to clean up any resources created in onCreate, and ensure that all externalconnections, such as network or database links, are closed.As part of Android's guidelines for writing efficient code, it's recommended that you avoid the creationof short-term objects. The rapid creation and destruction of objects force additional garbagecollection, a process that can have a direct negative impact on the user experience. If your Activitycreates the same set of objects regularly, consider creating them in the onCreate method instead, asit's called only once in the Activity's lifetime.

2. The Visible Lifetime

An Activity's visible lifetimes are bound between calls to onStart and onStop. Between thesecalls your Activity will be visible to the user, although it may not have focus and may be partiallyobscured. Activities are likely to go through several visible lifetimes during their full lifetimebecause they move between the foreground and background. Although it's unusual, in extreme casesthe Android run time will kill an Activity during its visible lifetime without a call to onStop.

The onStop method should be used to pause or stop animations, threads, Sensor listeners, GPSlookups, Timers, Services, or other processes that are used exclusively to update the UI. There's littlevalue in consuming resources (such as CPU cycles or network bandwidth) to update the UI when itisn't visible. Use the onStart (or onRestart) methods to resume or restart these processes when theUI is visible again.

The onRestart method is called immediately prior to all but the first call to onStart. Use it toimplement special processing that you want done only when the Activity restarts within itsfull lifetime.The onStart/onStop methods are also used to register and unregister Broadcast Receivers usedexclusively to update the UI.

3. The Active Lifetime

The active lifetime starts with a call to onResume and ends with a corresponding call to onPause.An active Activity is in the foreground and is receiving user input events. Your Activity is likely togo through many active lifetimes before it's destroyed, as the active lifetime will end when a newActivity is displayed, the device goes to sleep, or the Activity loses focus.

Try to keep code in theonPause and onResume methods relatively fast and lightweight to ensure that your applicationremains responsive when moving in and out of the foreground.Immediately before onPause, a call is made to onSaveInstanceState. This method provides anopportunity to save the Activity's UI state in a Bundle that may be passed to the onCreate andonRestoreInstanceState methods. Use onSaveInstanceState to save the UI state (such ascheckbox states, user focus, and entered but uncommitted user input) to ensure that the Activity canpresent the same UI when it next becomes active. You can safely assume that during the active lifetimeonSaveInstanceState and onPause will be called before the process is terminated.

Most Activity implementations will override at least the onSaveInstanceState method to commitunsaved changes, as it marks the point beyond which an Activity may be killed without warning.Depending on your application architecture you may also choose to suspend threads, processes, orBroadcast Receivers while your Activity is not in the foreground.The onResume method can be lightweight. You do not need to reload the UI state here because thisis handled by the onCreate and onRestoreInstanceState methods when required. Use onResumeto reregister any Broadcast Receivers or other processes you may have suspended in onPause.

2.7.6 Android Activity Classes

The Android SDK includes a selection of Activity subclasses that wrap up the use of common UIwidgets. Some of the more useful ones are listed here:

- MapActivity — Encapsulates the resource handling required to support a MapView widgetwithin an Activity. Learn more about MapActivity and MapView
- ListActivity — Wrapper class for Activities that feature a ListView bound to a datasource as the primary UI metaphor, and expose event handlers for list item selection.
- ExpandableListActivity — Similar to the ListActivity but supports anExpandableListView.

UNIT-III

Building User Interface

3.1 FUNDAMENTAL ANDROID UI DESIGN

User interface (UI) design, user experience (UX), human computer interaction (HCI), and usabilityare huge topics that can't be covered in the depth they deserve within the confines of this book.Nonetheless, the importance of creating a UI that your users will understand and enjoy using can'tbe overstated.Android introduces some new terminology for familiar programming metaphors that will beexplored in detail in the following sections:**Views** — Views are the base class for all visual interface elements (commonly known as *controls*or *widgets*). All UI controls, including the layout classes, are derived from View.

- **View Groups** — View Groups are extensions of the View class that can contain multiple childViews. Extend the ViewGroup class to create compound controls made up of interconnectedchild Views. The ViewGroup class is also extended to provide the Layout Managers that helpyou lay out controls within your Activities.
- **Fragments** — Fragments, introduced in Android 3.0 (API level 11), are used to encapsulateportions of your UI. This encapsulation makes Fragments particularly useful when optimizingyour UI layouts for different screen sizes and creating reusable UI elements. EachFragment includes its own UI layout and receives the related input events but is tightlybound to the Activity into which each must be embedded. Fragments are similar to UI ViewControllers in iPhone development.
- **Activities** — Activities, described in detail in the previous chapter, represent the window,or screen, being displayed. Activities are the Android equivalent of Forms in traditionalWindows desktop development. To display a UI, you assign a View (usually a layout orFragment) to an ActivityAndroid provides several common UI controls, widgets, and Layout Managers.For most graphical applications, it's likely that you'll need to extend and modify thesestandard Views — or create composite or entirely new Views — to provide your own userexperience.

3.2 ANDROID USER INTERFACE FUNDAMENTALS

All visual components in Android descend from the View class and are referred to generically as*Views*. You'll often see Views referred to as *controls* or *widgets* (not to be confused with homescreen App Widgets described in Chapter 14, "Invading the Home Screen) — terms you're probablyfamiliar with if you've previously done any GUI development.The ViewGroup class is an extension of View designed to contain multiple Views. View Groupsare used most commonly to manage the layout of child Views, but they can also be used to createatomic reusable components. View Groups that perform the former function are generally referredto as *layouts*.

In the following sections you'll learn how to put together increasingly complex UIs, before beingintroduced to Fragments, the Views available in the SDK, how to extend these Views, build yourown compound controls, and create your own custom Views from scratch.

3.2.1 Assigning User Interfaces to Activities

A new Activity starts with a temptingly empty screen onto which you place your UI. To do so,call setContentView, passing in the View instance, or layout resource, to display. Because emptyscreens aren't particularly inspiring, you will almost always use setContentView to assign anActivity's UI when overriding its onCreate handler.The setContentView method accepts either a layout's resource ID or a single View instance. Thislets you define your UI either in code or using the preferred technique of external layout resources.

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

Using layout resources decouples your presentation layer from the application logic, providing the flexibility to change the presentation without changing code. This makes it possible to specify differentlayouts optimized for different hardware configurations, even changing them at run time basedon hardware changes (such as screen orientation changes).You can obtain a reference to each of the Views within a layout using the findViewById method:

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

If you prefer the more *traditional* approach, you can construct the UI in code:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
TextView myTextView = new TextView(this);
setContentView(myTextView);
myTextView.setText("Hello, Android");
}
```

The setContentView method accepts a single View instance; as a result, you use layouts to addmultiple controls to your Activity.If you're using Fragments to encapsulate portions of your Activity's

UI, the View inflated withinyour Activity's onCreate handler will be a layout that describes the relative position of each of yourFragments (or their containers). The UI used for each Fragment is defined in its own layout and inflated within the Fragment itself, as described later in this chapter.Note that once a Fragment has been inflated into an Activity, the Views it contains become part ofthat Activity's View hierarchy. As a result you can find any of its child Views from within the parentActivity, using findViewById as described previously.

3.3 INTRODUCING LAYOUTS

Layout Managers(or simply *layouts*) are extensions of the ViewGroup class and are used to positionchild Views within your UI. Layouts can be nested, letting you create arbitrarily complex UIs usinga combination of layouts.The Android SDK includes a number of layout classes. You can use these, modify them, or createyour own to construct the UI for your Views, Fragments, and Activities. It's up to you to select anduse the right combination of layouts to make your UI aesthetically pleasing, easy to use, and efficientto display.The following list includes some of the most commonly used layout classes available in theAndroid SDK:

- **FrameLayout** — The simplest of the Layout Managers, the Frame Layout pins each childview within its frame. The default position is the top-left corner, though you can use thegravity attribute to alter its location. Adding multiple children stacks each new child on topof the one before, with each new View potentially obscuring the previous ones.
- **LinearLayout —** A Linear Layout aligns each child View in either a vertical or a horizontalline. A vertical layout has a column of Views, whereas a horizontal layout has a row ofViews. The Linear Layout supports a weight attribute for each child View that can controlthe relative size of each child View within the available space.
- **RelativeLayout**— One of the most flexible of the native layouts, the Relative Layoutlets you define the positions of each child View relative to the others and to the screenboundaries.
- **GridLayout**— Introduced in Android 4.0 (API level 14), the Grid Layout uses a rectangular grid of infinitely thin lines to lay out Views in a series of rows and columns. The Grid Layoutis incredibly flexible and can be used to greatly simplify layouts and reduce or eliminate thecomplex nesting often required to construct UIs using the layouts described above. It's goodpractice to use the Layout Editor to construct your Grid Layouts rather than relying ontweaking the XML manually.

Each of these layouts is designed to scale to suit the host device's screen size by avoiding the use ofabsolute positions or predetermined pixel values. This makes them particularly useful when designingapplications that work well on a diverse set of Android hardware.The Android documentation describes the features and properties of each layout class in detail; so,rather than repeat that information here, I'll refer you to http://developer.android.com/guide/topics/ui/layout-objects.html.

You'll see practical example of how these layouts should be used as they're introduced in the examplesthroughout this book. Later in this chapter you'll also learn how to create compound controlsby using and/or extending these layout classes.

3.3.1 Defining Layouts

The preferred way to define a layout is by using XML external resources.Each layout XML must contain a single root element. This root node can contain as many nestedlayouts and Views as necessary to construct an arbitrarily complex UI.The following snippet shows a simple layout that places a TextView above an EditText controlusing a vertical LinearLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Enter Text Below"
/>
<EditText
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Text Goes Here!"
/>
</LinearLayout>
```

For each of the layout elements, the constants wrap_content and match_parent are used ratherthan an exact height or width in pixels. These constants, combined with layouts that scale (such asthe Linear Layout, Relative Layout, and Grid Layout) offer the simplest, and most powerful, techniquefor ensuring your layouts are screen-size and resolution independent.

The wrap_content constant sets the size of a View to the minimum required to contain the contentsit displays (such as the height required to display a wrapped text string). The match_parent constantexpands the View to match the available space within the parent View, Fragment, or Activity.Later in this chapter you'll learn how to set the minimum height and width for your own controls,as well as further best practices for resolution independence.Implementing layouts in XML decouples the presentation layer from the View, Fragment, andActivity controller code and business logic. It also lets you create hardware configuration-specificvariations that are dynamically loaded without requiring code changes.When preferred, or required,

you can implement layouts in code. When assigning Views to layoutsin code, it's important to apply LayoutParameters using the setLayoutParams method, or by passingthem in to the addView call:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);
TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);
myTextView.setText("Enter Text Below");
myEditText.setText("Text Goes Here!");
int lHeight = LinearLayout.LayoutParams.MATCH_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;
ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myEditText, new LinearLayout.LayoutParams(lHeight, lWidth));
setContentView(ll);
```

3.3.2 Using Layouts to Create Device Independent User Interfaces

A defining feature of the layout classes described previously, and the techniques described for usingthem within your apps, is their ability to scale and adapt to a wide range of screen sizes, resolutions,and orientations.The variety of Android devices is a critical part of its success. For developers, this diversity introducesa challenge for designing UIs to ensure that they provide the best possible experience for users,regardless of which Android device they own.Using a Linear Layout.

The Linear Layout is one of the simplest layout classes. It allows you to create simple UIs (or UI elements)that align a sequence of child Views in either a vertical or a horizontal line.The simplicity of the Linear Layout makes it easy to use but limits its flexibility. In most cases youwill use Linear Layouts to construct UI elements that will be nested within other layouts, such as theRelative Layout.

The following code shows two nested Linear Layouts — a horizontal layout of two equally sized buttonswithin a vertical layout that places the buttons above a List View.

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

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="5dp">
    <Button
        android:text="@string/cancel_button_text"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"/>
    <Button
        android:text="@string/ok_button_text"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"/>
    </LinearLayout>
    <ListView
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    </LinearLayout>
```

If you find yourself creating increasingly complex nesting patterns of Linear Layouts, you will likelybe better served using a more flexible Layout Manager.Using a Relative LayoutThe Relative Layout provides a great deal of flexibility for your layouts, allowing you to define theposition of each element within the layout in terms of its parent and the other Views.

The following code modifies the layout described in the following code to move the buttons below the List View.

- **Relative Layout:**

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

```
<LinearLayout
        android:id="@+id/button_bar"
        android:layout_alignParentBottom="true"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="5dp">
<Button
        android:text="@string/cancel_button_text"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"/>
<Button
        android:text="@string/ok_button_text"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"/>
</LinearLayout>
<ListView
        android:layout_above="@id/button_bar"
        android:layout_alignParentLeft="true"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
</ListView>
</RelativeLayout>
```

❖ Using a Grid Layout

The Grid Layout was introduced in Android 3.0 (API level 11) and provides the most flexibility ofany of the Layout Managers.The Grid Layout uses an arbitrary grid to position Views. By using row and column spanning, theSpace View, and Gravity attributes, you can create complex without resorting to the often complexnesting required to construct UIs using the Relative Layout described previously.The Grid Layout is particularly useful for constructing layouts that require alignment in two directions— for example, a form

whose rows and columns must be aligned but which also includes elementsthat don't fi t neatly into a standard grid pattern.It's also possible to replicate all the functionality provided by the Relative Layout by using the GridLayout and Linear Layout in combination. For performance reasons it's good practice to use theGrid Layout in preference to creating the same UI using a combination of nested layouts.

The following code shows the same layout as described in previous code using a Grid Layout to replace the Relative Layout.

- **Grid Layout:**

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
    xmlns:android=http://schemas.android.com/apk/res/android
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
<ListView
    android:background="#FF444444"
    android:layout_gravity="fill">
</ListView>
<LinearLayout
    android:layout_gravity="fill_horizontal"
    android:orientation="horizontal"
    android:padding="5dp">
<Button
    android:text="Cancel"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"/>
<Button
    android:text="OK"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
```

```
        android:layout_weight="1"/>
</LinearLayout>
</GridLayout>
```

Note that the Grid Layout elements do not require width and height parameters to be set. Instead,each element wraps its content by default, and the layout_gravity attribute is used to determine inwhich directions each element should expand.

3.3.3 Optimizing Layouts

Inflating layouts is an expensive process; each additional nested layout and included View directlyimpacts on the performance and responsiveness of your application.To keep your applications smooth and responsive, it's important to keep your layouts as simple as possible and to avoid inflating entirely new layouts for relatively small UI changes.Redundant Layout Containers Are Redundant

A Linear Layout within a Frame Layout, both of which are set to MATCH_PARENT, does nothing butadd extra time to inflate. Look for redundant layouts, particularly if you've been making significantchanges to an existing layout or are adding child layouts to an existing layout.Layouts can be arbitrarily nested, so it's easy to create complex, deeply nested hierarchies. Althoughthere is no hard limit, it's good practice to restrict nesting to fewer than 10 levels.One common example of unnecessary nesting is a Frame Layout used to create the single root noderequired for a layout, as shown in the following snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
<ImageView
        android:id="@+id/myImageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/myimage"
        />
<TextView
        android:id="@+id/myTextView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
```

```
        android:gravity="center_horizontal"
        android:layout_gravity="bottom"
        />
    </FrameLayout>
```

In this example, when the Frame Layout is added to a parent, it will become redundant. A betteralternative is to use the Merge tag:

```
<?xml version="1.0" encoding="utf-8"?>
<merge
xmlns:android="http://schemas.android.com/apk/res/android">
<ImageView
        android:id="@+id/myImageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/myimage"
/>
<TextView
        android:id="@+id/myTextView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        android:gravity="center_horizontal"
        android:layout_gravity="bottom"
/>
</merge>
```

When a layout containing a merge tag is added to another layout, the merge node is removed and itschild Views are added directly to the new parent.The merge tag is particularly useful in conjunction with the include tag, which is used to insert thecontents of one layout into another:

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

```
<include android:id="@+id/my_action_bar"
        layout="@layout/actionbar"/>
<include android:id="@+id/my_image_text_layout"
        layout="@layout/image_text_layout"/>
</LinearLayout>
```

Combining the merge and include tags enables you to create flexible, reusable layout definitionsthat don't create deeply nested layout hierarchies. You'll learn more about creating and using simpleand reusable layouts later in this chapter.Avoid Using Excessive ViewsEach additional View takes time and resources to inflate. To maximize the speed and responsivenessof your application, none of its layouts should include more than 80 Views. When you exceed thislimit, the time taken to inflate the layout becomes significant.To minimize the number of Views inflated within a complex layout, you can use a ViewStub.A View Stub works like a lazy include — a stub that represents the specified child Views within theparent layout — but the stub is only inflated explicitly via the inflate method or when it's madevisible.

```
// Find the stub
View stub = findViewById(R.id. download_progress_panel_stub);
// Make it visible, causing it to inflate the child layout
stub.setVisibility(View.VISIBLE);
// Find the root node of the inflated stub layout
View downloadProgressPanel = findViewById(R.id.download_progress_panel);
```

As a result, the Views contained within the child layout aren't created until they arerequired — minimizing the time and resource cost of inflating complex UIs.When adding a View Stub to your layout, you can override the id and layout parameters of theroot View of the layout it represents:

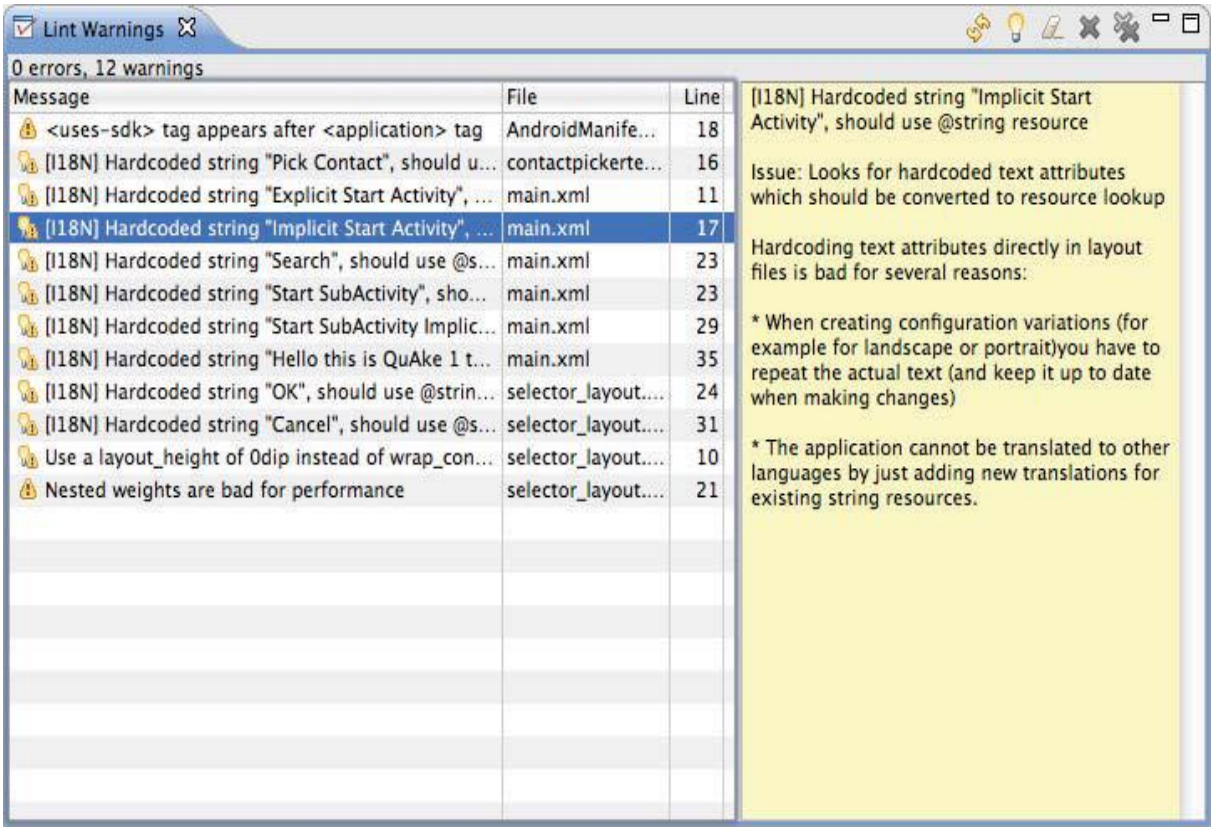
```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout "xmlns:android=http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
<ListView
        android:id="@+id/myListView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />
<ViewStub
```

```
android:id="@+id/download_progress_panel_stub"
android:layout="@layout/progress_overlay_panel"
android:inflatedId="@+id/download_progress_panel"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_gravity="bottom"
/>
</FrameLayout>
```

This snippet modifies the width, height, and gravity of the imported layout to suit the requirements of the parent layout. This flexibility makes it possible to create and reuse the same generic child layoutsin a variety of parent layouts. An ID has been specified for both the stub and the View Group it will become when inflated usingthe id and inflatedId attribute, respectively.

❖ Using Lint to Analyze Your Layouts

To assist you in optimizing your layout hierarchies, the Android SDK includes lint — a powerfultool that can be used to detect problems within you application, including layout performanceissues.The lint tool is available as a command-line tool or as a window within Eclipse supplied as part ofthe ADT plug-in, as shown in the figure.



In addition to using Lint to detect each optimization issue described previously in this section, youcan also use Lint to detect missing translations, unused resources, inconsistent array sizes, accessibilityand internationalization problems, missing or duplicated image assets, usability problems, andmanifest errors.Lint is a constantly evolving tool, with new rules added regularly. A full list of the tests performedby the Lint tool can be found at http://tools.android.com/tips/lint-checks.

3.4 INTRODUCING FRAGMENTS

Fragments enable you to divide your Activities into fully encapsulated reusable components, eachwith its own lifecycle and UI.The primary advantage of Fragments is the ease with which you can create dynamic and flexible UIdesigns that can be adapted to suite a range of screen sizes — from small-screen smartphones to tablets.Each Fragment is an independent module that is tightly bound to the Activity into which it is placed.Fragments can be reused within multiple activities, as well as laid out in a variety of combinations tosuit multipane tablet UIs and added to, removed from, and exchanged within a running Activity tohelp build dynamic UIs.

Fragments provide a way to present a consistent UI optimized for a wide variety of Android devicetypes, screen sizes, and device densities.Although it is not necessary to divide your Activities (and their corresponding layouts) intoFragments, doing so will drastically improve the flexibility of your UI and make it easier for you toadapt your user experience for new device configurations.

3.4.1 Creating New Fragments

Extend the Fragment class to create a new Fragment, (optionally) defining the UI and implementingthe functionality it encapsulates.In most circumstances you'll want to assign a UI to your Fragment. It is possible to create aFragment that *doesn't* include a UI but instead provides background behavior for an Activity. This isexplored in more detail later in this chapter.If your Fragment does require a UI, override the onCreateView handler to inflate and return therequired View hierarchy, as shown in the Fragment skeleton code in

```
package com.paad.fragments;
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
public class MySkeletonFragment extends Fragment {
@Override
public View onCreateView(LayoutInflater inflater,
ViewGroup container,
```

Bundle savedInstanceState) {

// Create, or inflate the Fragment's UI, and return it.

// If this Fragment has no UI then return null.

return inflater.inflate(R.layout.my_fragment, container, false);

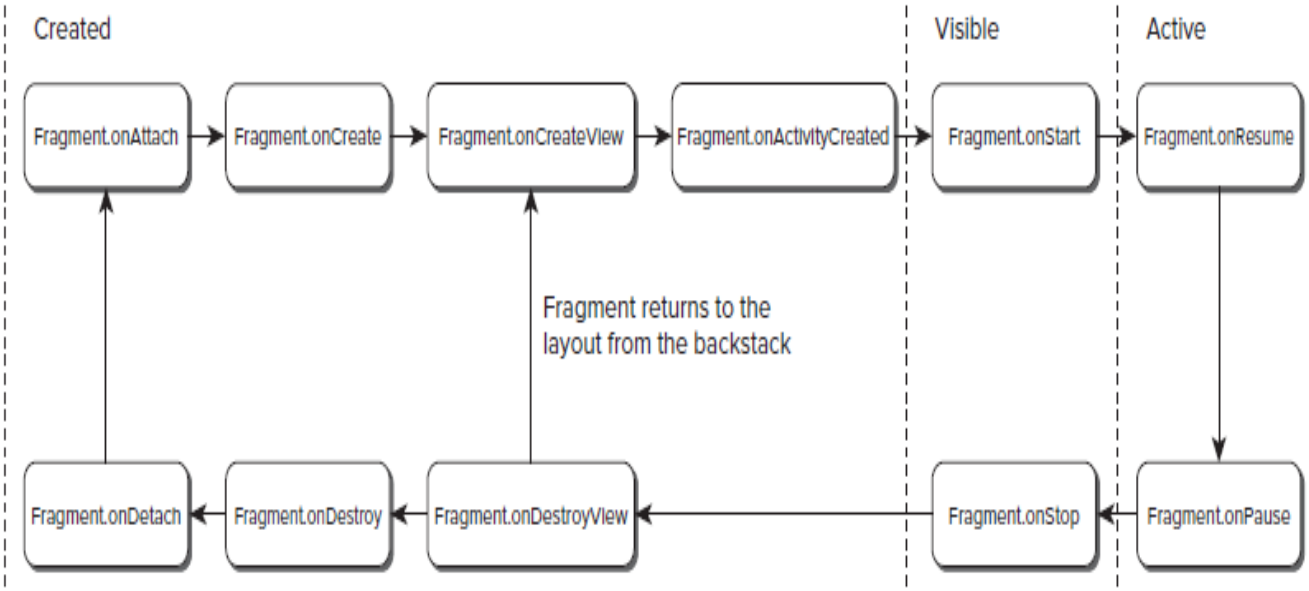
}

}

You can create a layout in code using layout View Groups; however, as with Activities, the preferredway to design Fragment UI layouts is by inflating an XML resource.Unlike Activities, Fragments don't need to be registered in your manifest. This is because Fragmentscan exist only when embedded into an Activity, with their lifecycles dependent on that of theActivity to which they've been added.

3.4.2 The Fragment Lifecycle

The lifecycle events of a Fragment mirror those of its parent Activity; however, after the containingActivity is in its active — resumed — state adding or removing a Fragment will affect its lifecycleindependently.Fragments include a series of event handlers that mirror those in the Activity class. They are triggeredas the Fragment is created, started, resumed, paused, stopped, and destroyed. Fragments alsoinclude a number of additional callbacks that signal binding and unbinding the Fragment from itsparent Activity, creation (and destruction) of the Fragment's View hierarchy, and the completion ofthe creation of the parent Activity.



❖ Fragment-Specific Lifecycle Events

Most of the Fragment lifecycle events correspond to their equivalents in the Activity class. Those that remain are specific to Fragments and the way inwhich they're inserted into their parent Activity.

- **Attaching and Detaching Fragments from the Parent Activity**

  The full lifetime of your Fragment begins when it's bound to its parent Activity and ends when it'sbeen detached. These events are represented by the calls to onAttach and onDetach, respectively.As with any handler called after a Fragment/Activity has become paused, it's possible that onDetachwill not be called if the parent Activity's process is terminated *without* completing its full lifecycle.The onAttach event is triggered before the Fragment's UI has been created, before the Fragmentitself or its parent Activity have finished their initialization. Typically, the onAttach event is used togain a reference to the parent Activity in preparation for further initialization tasks.

- **Creating and Destroying Fragments**

  The created lifetime of your Fragment occurs between the first call to onCreate and the final callto onDestroy. As it's not uncommon for an Activity's process to be terminated *without* the correspondingonDestroy method being called, so a Fragment can't rely on its onDestroy handler beingtriggered.As with Activities, you should use the onCreate method to initialize your Fragment. It's good practiceto create any class scoped objects here to ensure they're created only once in the Fragment'slifetime.

- **Creating and Destroying User Interfaces**

  A Fragment's UI is initialized (and destroyed) within a new set of event handlers: **onCreateView** and **onDestroyView**, respectively.

  - ✓ Use the onCreateView method to initialize your Fragment
  - ✓ Inflate the UI
  - ✓ get references (and bind data to) the Views it contains
  - ✓ and then create any required Services and Timers.

  Once you have inflated your View hierarchy, it should be returned from this handler:

  return inflater.inflate(R.layout.my_fragment, container, false);

If your Fragment needs to interact with the UI of its parent Activity, wait until the **onActivityCreated** event has been triggered. This signifies that the containing Activity has completed its initialization and its UI has been fully constructed.

❖ Fragment States

Fragment state transitions are closely related to the corresponding Activity state transitions. Like Activities, Fragments are active when they belong to an Activity that is focused and in the foreground. When an Activity is paused or stopped, the Fragments it contains are also paused and stopped, and the Fragments contained by an inactive Activity are also inactive. When an Activity is finally destroyed, each Fragment it contains is likewise destroyed.As the Android memory manager nondeterministically closes applications to free resources, the Fragments within those Activities are also destroyed.

While Activities and their Fragments are tightly bound, one of the advantages of using Fragments is the flexibility to dynamically add or remove Fragments from an active Activity. As a result, each Fragment can progress through its full, visible, and active lifecycle several times within the active lifetime of its parent Activity. There should be no difference in a Fragment moving from a paused, stopped, or inactive state back

to active, so it's important to save all UI state and persist all data when a Fragment is paused or stopped. Like an Activity, when a Fragment becomes active again, it should restore that saved state.

3.4.3. Introducing the Fragment Manager

Each Activity includes a Fragment Manager to manage the Fragments it contains. You can access the Fragment Manager using the **getFragmentManager** method:

```
FragmentManager fragmentManager = getFragmentManager();
```

The Fragment Manager provides the methods used to access the Fragments currently added to the Activity, and to perform Fragment Transaction to add, remove, and replace Fragments.

3.4.4. Adding Fragments to Activities

The simplest way to add a Fragment to an Activity is by including it within the Activity's layout using the fragment tag

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.paad.weatherstation.MyListFragment"
        android:id="@+id/my_list_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1" />
    <fragment android:name="com.paad.weatherstation.DetailsFragment"
        android:id="@+id/details_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="3" />
</LinearLayout>
```

This technique works well when you use Fragments to define a set of static layouts based on various screen sizes. If you plan to dynamically modify your layouts by adding, removing, and replacing Fragments

at run time, a better approach is to create layouts that use container Views into which Fragments can be placed at runtime, based on the current application state.

❖ Using Fragment Transactions

Fragment Transactions can be used to add, remove, and replace Fragments within an Activity at run time. Using Fragment Transactions, you can make your layouts dynamic — that is, they will adapt and change based on user interactions and application state.Each Fragment Transaction can include any combination of supported actions, including adding, removing, or replacing Fragments.

They also support the specification of the transition animations to display and whether to include the Transaction on the back stack. A new Fragment Transaction is created using the **beginTransaction** method from the Activity's Fragment Manager. Modify the layout using the add, remove, and replace methods, as required, before setting the animations to display, and setting the appropriate back-stack behavior. When you are ready to execute the change, call commit to add the transaction to the UI queue.

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
// Add, remove, and/or replace Fragments Specify animations Add to back stack if required.
fragmentTransaction.commit();
```

❖ Adding, Removing, and Replacing Fragments

When adding a new UI Fragment, specify the Fragment instance to add, along with the container View into which the Fragment will be placed. Optionally, you can specify a tag that can later be used to find the Fragment by using the **findFragmentByTag** method:

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
fragmentTransaction.add(R.id.ui_container, new MyListFragment());
fragmentTransaction.commit();
```

To remove a Fragment, you fi rst need to fi nd a reference to it, usually using either the Fragment Manager's **findFragmentById** or **findFragmentByTag** methods. Then pass the found Fragment instance as a parameter to the remove method of a Fragment Transaction:

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
Fragment fragment = fragmentManager.findFragmentById(R.id.details_fragment);
fragmentTransaction.remove(fragment);
fragmentTransaction.commit();
```

You can also replace one Fragment with another. Using the replace method, specify the container ID containing the Fragment to be replaced, the Fragment with which to replace it, and (optionally) a tag to identify the newly inserted Fragment.

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

*fragmentTransaction.**replace**(R.id.details_fragment,new DetailFragment(selected_index));*
fragmentTransaction.commit();

❖ Using the Fragment Manager to Find Fragments

To find Fragments within your Activity, use the Fragment Manager's **findFragmentById** method. If you have added your Fragment to the Activity layout in XML, you can use the Fragment's resource identifier:

MyFragment myFragment = (MyFragment)fragmentManager.findFragmentById(R.id.MyFragment);

Alternatively, you can use the **findFragmentByTag** method to search for the Fragment using the tag you specified in the Fragment Transaction:

MyFragment myFragment =

MyFragment)fragmentManager.findFragmentByTag(MY_FRAGMENT_TAG);

3.4.5. Interfacing Between Fragments and Activities

Use the **getActivity** method within any Fragment to return a reference to the Activity within which it's embedded. This is particularly useful for finding the current Context, accessing other Fragments using the Fragment Manager, and finding Views within the Activity's View hierarchy.

TextView textView = (TextView)getActivity().findViewById(R.id.textview);

Although it's possible for Fragments to communicate directly using the host Activity's Fragment Manager, it's generally considered better practice to use the Activity as an intermediary. This allows the Fragments to be as independent and loosely coupled as possible, with the responsibility for deciding how an event in one Fragment should affect the overall UI falling to the host Activity. Where your Fragment needs to share events with its host Activity (such as signaling UI selections).

```
public interface OnSeasonSelectedListener {
    public void onSeasonSelected(Season season);
}
private OnSeasonSelectedListener onSeasonSelectedListener;
private Season currentSeason;
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    try{
```

```
            onSeasonSelectedListener = (OnSeasonSelectedListener)activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString() +" must implement
            OnSeasonSelectedListener");
        }
    }
    private void setSeason(Season season) {
        currentSeason = season;  onSeasonSelectedListener.onSeasonSelected(season);
    }
```

3.4.6. Fragments Without User Interfaces

This is particularly well suited to background tasks that regularly touch the UI or where it's important to maintain state across Activity restarts caused by configuration changes. You can choose to have an active Fragment retain its current instance when its parent Activity is recreated using the **setRetainInstance** method. After you call this method, the Fragment's lifecycle will change. Rather than being destroyed and re-created with its parent Activity, the same Fragment instance is retained when the Activity restarts. It will receive the **onDetach** event when the parent Activity is destroyed, followed by the **onAttach, onCreateView,** and **onActivityCreated** events as the new parent Activity is instantiated.

The following snippet shows the skeleton code for a Fragment without a UI:

```
public class NewItemFragment extends Fragment {
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        // Get a type-safe reference to the parent Activity.
    }
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Create background worker threads and tasks.
}
@Override
public void onActivityCreated(Bundle savedInstanceState) {
super.onActivityCreated(savedInstanceState);
   // Initiate worker threads and tasks.
 }
}
```

To add this Fragment to your Activity, create a new Fragment Transaction, specifying a tag to use to identify it. Because the Fragment has no UI, it should not be associated with a container View and generally shouldn't be added to the back stack.

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

fragmentTransaction.add(workerFragment, MY_FRAGMENT_TAG);

fragmentTransaction.commit();

Use the **findFragmentByTag** from the Fragment Manager to fi nd a reference to it later

MyFragment myFragment = (MyFragment)fragmentManager.findFragmentByTag(MY_FRAGMENT_TAG);

3.4.7Android Fragment Classes

The Android SDK includes a number of Fragment subclasses that encapsulate some of the most common Fragment implementations. Some of the more useful ones are listed here:

- **DialogFragment** — A Fragment that you can use to display a floating Dialog over the parent Activity. You can customize the Dialog's UI and control its visibility directly via the Fragment API. Dialog Fragments are covered in more detail in Chapter 10, "Expanding the User Experience."
- **ListFragment** — A wrapper class for Fragments that feature a ListView bound to a data source as the primary UI metaphor. It provides methods to set the Adapter to use and exposes the event handlers for list item selection. The List Fragment is used as part of the To-Do List example in the next section.
- **WebViewFragment** — A wrapper class that encapsulates a WebView within a Fragment. The child WebView will be paused and resumed when the Fragment is paused and resumed.

3.5 THE ANDROID WIDGET TOOLBOX

Android supplies a toolbox of standard Views to help you create your UIs.

- **TextView** — A standard read-only text label that supports multiline display, string formatting, and automatic word wrapping.
- **EditText** — An editable text entry box that accepts multiline entry, word-wrapping, and hint text.
- **Chronometer** — A Text View extension that implements a simple count-up timer.
- **ListView** — A View Group that creates and manages a vertical list of Views, displaying them as rows within the list.
- **Spinner** — A composite control that displays a Text View and an associated List View that lets you select an item from a list to display in the textbox.
- **Button** — A standard push button.
- **ToggleButton** — A two-state button that can be used as an alternative to a check box. It's particularly appropriate where pressing the button will initiate an action as well as changing a state
- **ImageButton** — A push button for which you can specify a customized background image (Drawable).
- **CheckBox** — A two-state button represented by a checked or unchecked box.
- **RadioButton** — A two-state grouped button. A group of these presents the user with a number of possible options, of which only one can be enabled at a time.
- **ViewFlipper** — A View Group that lets you define a collection of Views as a horizontal row in which only one View is visible at a time, and in which transitions between visible views can be animated.

- **VideoView** — Handles all state management and display Surface configuration for playing videos more simply from within your Activity.
- **QuickContactBadge** — Displays a badge showing the image icon assigned to a contact you specify using a phone number, name, email address, or URI. Clicking the image will display the quick contact bar, which provides shortcuts for contacting the selected contact.
- **ViewPager** — Released as part of the Compatibility Package, the View Pager implements a horizontally scrolling set of Views similar to the UI used in Google Play and Calendar. The View Pager allows users to swipe or drag left or right to switch between different Views.

3.6 CREATING NEW VIEWS

It is possible to implement beautiful UIs optimized for your application's workflow.The best approach to use when creating a new View depends on what you want to achieve:

- Modify or extend the appearance and/or behavior of an existing View when it supplies the basic functionality you want. By overriding the event handlers and/or onDraw, but still calling back to the superclass's methods, you can customize a View without having to re-implement its functionality.
- Combine Views to create atomic, reusable controls that leverage the functionality of several interconnected Views. For example, you could create a stopwatch timer by combining a TextView and a Button that resets the counter when clicked.
- Create an entirely new control when you need a completely different interface that you can't get by changing or combining existing controls.

3.6.1. Modifying Existing Views

To create a new View based on an existing control, create a new class that extends it, as shown with the TextView derived class

```
Import ...
public class MyTextView extends TextView {
public MyTextView (Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
 }
public MyTextView (Context context) {
        super(context);
}
public MyTextView (Context context, AttributeSet attrs) {
        super(context, attrs);
}
 }
```

To override the appearance or behavior of your new View, override and extend the event handlers associated with the behavior you want to change.

```
@Override
public void onDraw(Canvas canvas) {
[ ... Draw things on the canvas under the text ... ]
        // Render the text as usual using the TextView base class.    super.onDraw(canvas);
        [ ... Draw things on the canvas over the text ... ]
}
@Override
```

```
public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
        [ ... Perform some special processing ... ]
        [ ... based on a particular key press ... ]
        // Use the existing functionality implemented by
        // the base class to respond to a key press event.
        return super.onKeyDown(keyCode, keyEvent);
}
 }
```

3.6.2 Creating Compound Controls

Compound controls are atomic, self-contained View Groups that contain multiple child Views laid out and connected together. When you create a compound control, you define the layout, appearance, and interaction of the Views it contains.

```
public class MyCompoundView extends LinearLayout {
public MyCompoundView(Context context) {
super(context);
 }
public MyCompoundView(Context context, AttributeSet attrs) {
 super(context, attrs);
 }
 }
```

As with Activities, the preferred way to design compound View UI layouts is by using an external resource.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <EditText
                android:id="@+id/editText"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
        />
        <Button
                android:id="@+id/clearButton"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="Clear"
        />
</LinearLayout>
```

To use this layout in your new compound View, override its constructor to inflate the layout resource using the inflate method from the LayoutInflater system service. The inflate method takes the layout resource and returns the inflated View.

```
public class ClearableEditText extends LinearLayout {
        EditText editText;
        Button clearButton;
        public ClearableEditText(Context context) {
```

```
super(context);
// Inflate the view from the layout resource.
String infService = Context.LAYOUT_INFLATER_SERVICE;
LayoutInflater li;
li = (LayoutInflater)getContext().getSystemService(infService);
li.inflate(R.layout.clearable_edit_text, this, true);
// Get references to the child controls.
editText = (EditText)findViewById(R.id.editText);
clearButton = (Button)findViewById(R.id.clearButton);
// Hook up the functionality
hookupButton();
}
}
}
```

You can create a reusable layout by creating an XML resource that encapsulates the UI pattern you want to reuse. You can then import these layout patterns when creating the UI for Activities or Fragments by using the include tag within their layout resource definitions.

<include layout="@layout/clearable_edit_text"/>

The include tag also enables you to override the id and layout parameters of the root node of the included layout:

```
<include layout="@layout/clearable_edit_text"
android:id="@+id/add_new_entry_input"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_gravity="top"/>
```

3.6.3 Creating Custom Views

Creating new Views gives you the power to fundamentally shape the way your applications look and feel. By creating your own controls, you can create UIs that are uniquely suited to your needs. You extend either the View or SurfaceView class. The View class provides a Canvas object with a series of draw methods and Paint classes. Use them to create a visual interface with bitmaps and raster graphics. You can then override user events, including screen touches or key presses to provide interactivity. The SurfaceView class provides a Surface object that supports drawing from a background thread and optionally using openGL to implement your graphics.

❖ Handling User Interaction Events

For your new View to be interactive, it will need to respond to user-initiated events such as key presses, screen touches, and button clicks.

- **onKeyDown** — Called when any device key is pressed; includes the D-pad, keyboard, hang-up, call, back, and camera buttons
- **onKeyUp** — Called when a user releases a pressed key
- **onTrackballEvent** — Called when the device's trackball is moved
- **onTouchEvent** — Called when the touchscreen is pressed or released, or when it detects movement

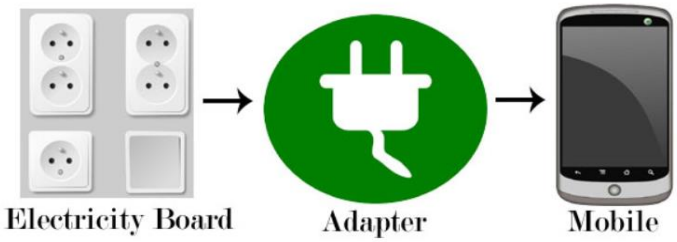
Skeleton class that overrides each of the user interaction handlers in a View is shown in the next slide.

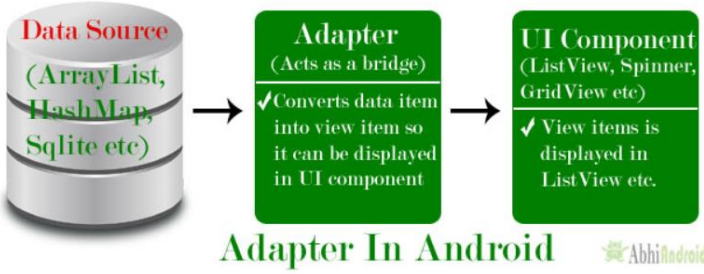
```
@Override
public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
```

```
// Return true if the event was handled. return true;
}
@Override
public boolean onKeyUp(int keyCode, KeyEvent keyEvent) {
 // Return true if the event was handled. return true;
}
@Override
public boolean onTrackballEvent(MotionEvent event ) {
// Get the type of action this event represents
int actionPerformed = event.getAction();
// Return true if the event was handled.
return true;
}
@Override
public boolean onTouchEvent(MotionEvent event) {
 // Get the type of action this event represents
int actionPerformed = event.getAction();
// Return true if the event was handled.
return true;
}
```

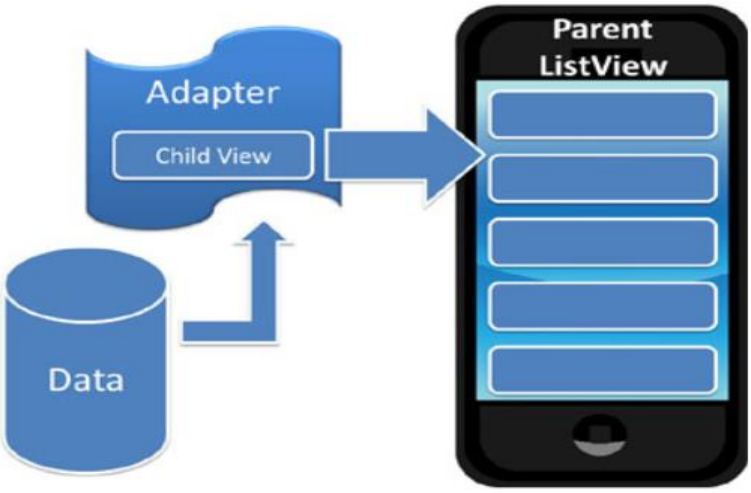
3.7 INTRODUCING ADAPTERS

Adapters are the link between a set of data and the AdapterView that displays the data.Adapters are used to bind data to View Groups that extend the AdapterView class.





Adapters are responsible for creating child Views that represent the underlying data within the bound parent View.It is incapable of displaying any data on its own. Its contents are always determined by another object, an adapter.



An adapter is an object of a class that implements the Adapter interface. It acts as a link between a data set and an adapter view, an object of a class that extends the abstract AdapterView class. The data set can be anything that presents data in a structured manner. Arrays, List objects, and Cursor objects are commonly used data sets.

3.7.1 Introducing Some Native Adapters:

In most cases you won't have to create your own Adapters from scratch. Android supplies a set of Adapters that can pump data from common data sources (including arrays and Cursors) into the native controls that extend Adapter View.The following list highlights two of the most useful and versatile native Adapters:

1. **ArrayAdapter** — The Array Adapter uses generics to bind an Adapter View to an array of objects of the specified class.
   - By default, the Array Adapter uses the toString value of each object in the array to create and populate Text Views.
   - Alternative constructors enable you to use more complex layouts, or you can extend the class (as shown in the next section) to bind data to more complicated layouts.

2. **SimpleCursorAdapter** — The Simple Cursor Adapter enables you to bind the Views within a layout to specific columns contained within a Cursor (typically returned from a Content Provider query).
   - You specify an XML layout to inflate and populate to display each child, and then bind each column in the Cursor to a particular View within that layout.
   - The adapter will create a new View for each Cursor entry and inflate the layout into it, populating each View within the layout using the Cursor's corresponding column value.

You can create your own Adapter classes and build your own AdapterView-derived controls.

❖ ArrayAdapter

ArrayAdapter consists of two elements- Array and Adapter.Array is a list or array of Java objects and objects can be anything like- string, integer etc. Adapter is a collection handler that returns each item in the collection as a view. It is used for managing the items in the list (the data model or data source).

ArrayAdapter class can handle a list or array of Java objects as input. Every Java object is mapped to one row. By default it maps the toString() method of the object to a view in the row layout.

3.7.2 Customizing the Array Adapter

ArrayAdapter(Context context, int resource, int textViewResourceId, T[] objects)

is the implementation of a ArrayAdapter.

Parameters used in ArrayAdapter Class:

- **context:**

The first parameter is used to pass the context means the reference of current class. Here this is a keyword used to show the current class reference. We can also use getApplicationContext(), getActivity() in the place of this keyword.getApplicationContext() is used in a Activity and getActivity() is used in a Fragment.Below is the example code in which we set the current class reference in a adapter.

ArrayAdapter arrayAdapter = new ArrayAdapter(this, int resource, int textViewResourceId, T[] objects);

- **resource:**

The second parameter is resource id used to set the layout(xml file) for list items in which you have a text view.

ArrayAdapter arrayAdapter = new ArrayAdapter(this, R.layout.list_view_items, int textViewResourceId, T[] objects);

- **textViewResourceId:**

The third parameter is textViewResourceId which is used to set the id of TextView where you want to display the actual text.

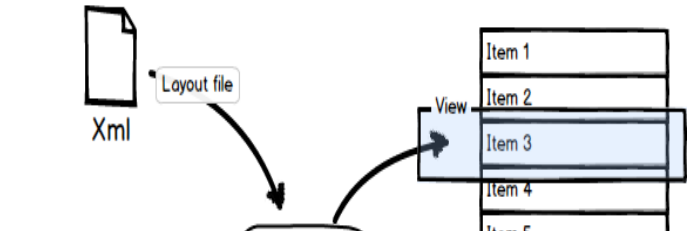
ArrayAdapter arrayAdapter = new ArrayAdapter(this, R.layout.list_view_items, R.id.textView, T[] objects);

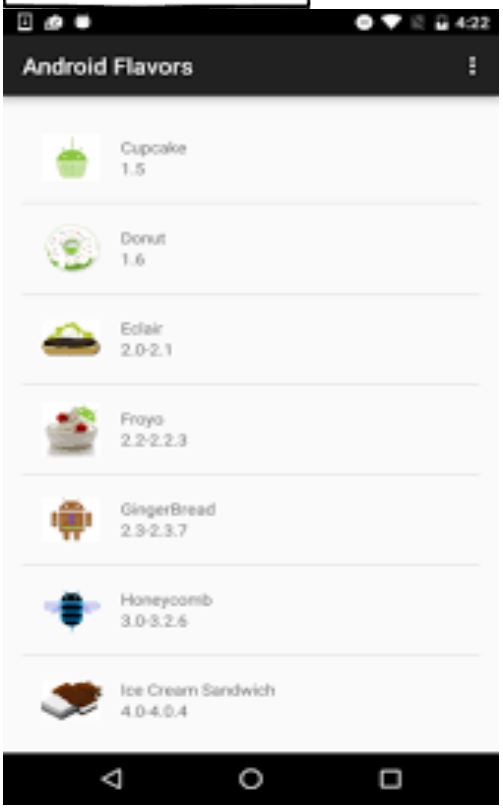
- **objects:**

The fourth parameter is an array of objects, used to set the array of elements in the textView. We can set the object of array or array list here.

String animalList[] = {"Lion","Tiger","Monkey","Elephant","Dog","Cat","Camel"};

ArrayAdapter arrayAdapter = new ArrayAdapter(this, R.layout.list_view_items, R.id.textView, animalList);





3.7.3 Using the Simple Cursor Adapter

The SimpleCursorAdapter is used to bind a Cursor to an Adapter View using a layout to define the UI of each row/item. The content of each row's View is populated using the column values of the corresponding row in the underlying Cursor.Construct a Simple Cursor Adapter by passing in the current

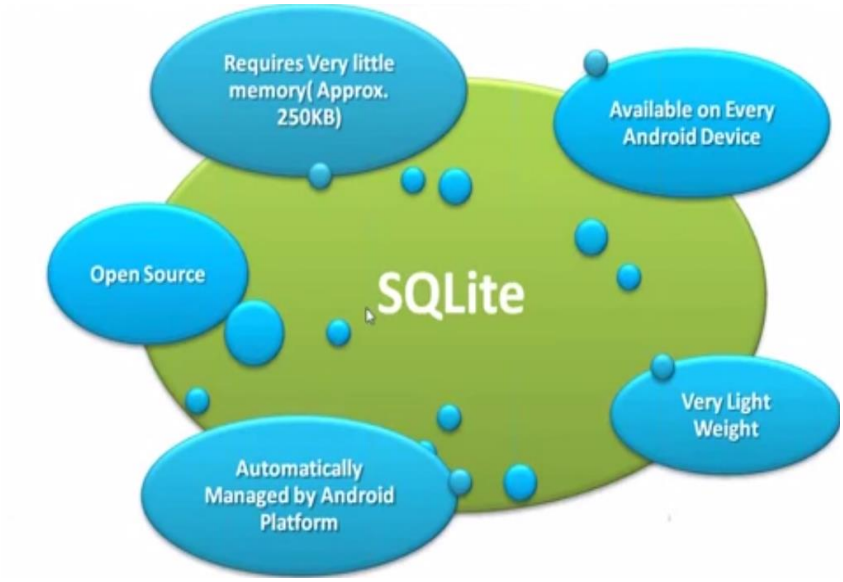
context, a layout resource to use for each item, a Cursor that represents the data to display, and two integer arrays: one that contains the indexes of the columns from which to source the data, and a second (equally sized) array that contains resource IDs to specify which Views within the layout should be used to display the contents of the corresponding columns.

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
        String[] fromColumns = new String[]{
        CallLog.Calls.CACHED_NAME,
                CallLog.Calls.NUMBER};
                int[] toLayoutIDs = new int[] {
        R.id.nameTextView, R.id.numberTextView
                };
        SimpleCursorAdapter myAdapter;
        myAdapter = new SimpleCursorAdapter(MyActivity.this,
                                R.layout.mysimplecursorlayout,
                                cursor,fromColumns,toLayoutIDs);
        myListView.setAdapter(myAdapter);
}
```

3.8 INTRODUCING ANDROID DATABASES

Android provides structured data persistence through a combination of SQLite relational databases and Content Providers.SQLite databases can be used to store application data using a managed, structured approach.Every application can create its own databases over which it has complete control.

Android databases are stored in the **/data/data/<package_name>/databases** folder on your device (or emulator).Content Providers offer a generic, well-defined interface for using and sharing data that provides a consistent abstraction from the underlying data source.Content Providers provide an interface for publishing and consuming data, based around a simple URI addressing model using the **content:// schema**



3.8.1. SQLite Databases

SQLite is a well-regarded relational database management system (RDBMS). It is:

Open-source
Standards-compliant
Lightweight
Single-tier

It has been implemented as a compact C library that's included as part of the Android software stack implemented as a library, rather than running as a separate ongoing process.This reduces external dependencies, minimizes latency, and simplifies transaction locking and synchronization.Lightweight and powerful, SQLite differs from many conventional database engines by loosely typing each column, meaning that column values are not required to conform to a single type;Each value is typed individually in each row. As a result, type checking isn't necessary when assigning or extracting values from each column within a row.

3.8.2 CONTENT VALUES AND CURSORS

Content Values are used to insert new rows into tables. Each ContentValues object represents a single table row as a map of column names to values.Database queries are returned as Cursor objects. Rather than extracting and returning a copy of the result values Cursors are pointers to the result set within the underlying data. The Cursor class includes a number of navigation functions, including, but not limited to, the following:

- moveToFirst — Moves the cursor to the first row in the query result
- moveToNext — Moves the cursor to the next row
- moveToPrevious — Moves the cursor to the previous row
- getCount — Returns the number of rows in the result set
- getColumnIndexOrThrow — Returns the zero-based index for the column with the specified name (throwing an exception if no column exists with that name)
- getColumnName — Returns the name of the specified column index
- getColumnNames — Returns a string array of all the column names in the current Cursor
- moveToPosition — Moves the cursor to the specified row
- getPosition — Returns the current cursor position

3.8.3 WORKING WITH SQLITE DATABASES

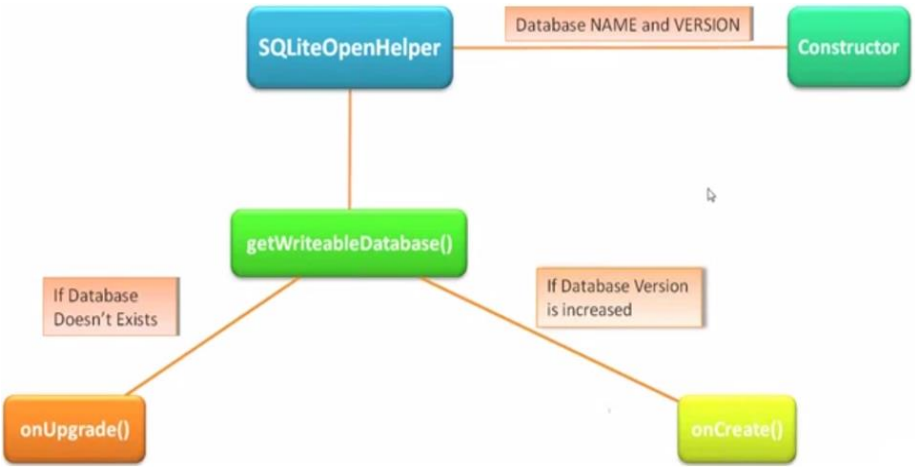
❖ Introducing the SQLiteOpenHelper:

SQLiteOpenHelper is an abstract class used to implement the best practice pattern for creating, opening, and upgrading databases.By implementing an SQLite Open Helper, you hide the logic used to decide if a database needs to be created or upgraded before it's opened, as well as ensure that each operation is completed efficiently.The SQLite Open Helper caches database instances after they've been successfully opened, so you can make requests to open the database immediately prior to performing a query or transaction. There is no need to close the database manually unless you no longer need to use it again.

```
private static class HoardDBOpenHelper extends SQLiteOpenHelper {
private static final String DATABASE_NAME = "myDatabase.db";
private static final String DATABASE_TABLE = "GoldHoards";
private static final int DATABASE_VERSION = 1;
private static final String DATABASE_CREATE = "create table " +DATABASE_TABLE + " (" +
KEY_ID +"integer primary key autoincrement, "  +KEY_GOLD_HOARD_NAME_COLUMN + " text not
null, " +KEY_GOLD_HOARDED_COLUMN        + " float, "
+KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + " integer);";
```

```
public HoardDBOpenHelper(Context context, String name, CursorFactory factory, int version) {
        super(context, name, factory, version);
}
@Override
public void onCreate(SQLiteDatabase db) {
        db.execSQL(DATABASE_CREATE);
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,int newVersion) {
        Log.w("TaskDBAdapter", "Upgrading from version " +oldVersion + " to " +newVersion +
",which will destroy all old data");
db.execSQL("DROP TABLE IF IT EXISTS " + DATABASE_TABLE);
onCreate(db);
}}
```

To access a database using the SQLite Open Helper, call getWritableDatabase or getReadableDatabase to open and obtain a writable or read-only instance of the underlying database, respectively.If the database version has changed, the onUpgrade handler will fire. In either case, the get<read/ writ>ableDatabase call will return the cached, newly opened, newly created, or upgraded database, as appropriate.When a database has been successfully opened, the SQLite Open Helper will cache it, so you can(and should) use these methods each time you query or perform a transaction on the database, rather than caching the open database within your application.



❖ Opening and Creating Databases Without the SQLiteOpen Helper

If you would prefer to manage the creation, opening, and version control of your databases directly, rather than using the SQLite Open Helper, you can use the application Context's openOrCreateDatabase method to create the database itself:

```
SQLiteDatabase db = context.openOrCreateDatabase(DATABASE_NAME,
Context.MODE_PRIVATE,null);
```

After you have created the database, you must handle the creation and upgrade logic handled within the onCreate and onUpgrade handlers of the SQLite Open Helper — typically using the database's execSQL method to create and drop tables, as required.It's good practice to defer creating and opening databases until they're needed, and to cache database instances after they're successfully opened to limit the associated efficiency costs.

Sqlitedb=openOrCreateDatabase("EmployeeDB", context.MODEPRIVATE, null);
Sqlitedb.execSQL("CREATE TABLE IF NOT EXISTS EmpRegistration(EmpID integer PRIMARY KEY AUTOINCREMENT, EmpName VARCHAR(225), EmpMail VARCHAR(225), EmpAge VARCHAR(25));");

❖ Android Database Design Considerations

Files (such as bitmaps or audio files) are not usually stored within database tables. Use a string to store a path to the file, preferably a fully qualified URI.Although not strictly a requirement, it's strongly recommended that all tables include an auto-increment key field as a unique index field for each row.
If you plan to share your table using a Content Provider, a unique ID field is required.

❖ Querying a Database

Each database query is returned as a Cursor. This lets Android manage resources more efficiently by retrieving and releasing row and column values on demand.To execute a query on a Database object, use the query method, passing in the following:

- An optional Boolean that specifies if the result set should contain only unique values.
- The name of the table to query.
- A projection, as an array of strings, that lists the columns to include in the result set.
- A where clause that defines the rows to be returned. You can include ? wildcards that will bereplaced by the values passed in through the selection argument parameter.
- An array of selection argument strings that will replace the ? wildcards in the where clause.
- A group by clause that defines how the resulting rows will be grouped.
- A having clause that defines which row groups to include if you specified a group by clause.
- A string that describes the order of the returned rows.
- A string that defines the maximum number of rows in the result set.

To return a selection of rows from within an SQLite database table

```
String[] result_columns = new String[] {
KEY_ID,                                KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
KEY_GOLD_HOARDED_COLUMN };
String where = KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + "=" + 1;
String whereArgs[] = null;
String groupBy = null;
String having = null;
String order = null;
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
Cursor cursor = db.query(HoardDBOpenHelper.DATABASE_TABLE,result_columns,
                                    where, whereArgs, groupBy, having, order);
```

❖ Extracting Values from a Cursor

To extract values from a Cursor, first use the moveTo<location> methods to position the cursor at the correct row of the result Cursor, and then use the type-safe get<type> methods to return the value stored at the current row for the specified column. To find the column index of a particular column within a result Cursor, use its getColumnIndexOrThrow and getColumnIndex methods.

float totalHoard = 0f;

```
float averageHoard = 0f;
int GOLD_HOARDED_COLUMN_INDEX =cursor.getColumnIndexOrThrow(
                    KEY_GOLD_HOARDED_COLUMN);
while (cursor.moveToNext()) {
        float hoard = cursor.getFloat(GOLD_HOARDED_COLUMN_INDEX);
        totalHoard += hoard;
}
float cursorCount = cursor.getCount();
averageHoard = cursorCount > 0 ?(totalHoard / cursorCount) : Float.NaN;
cursor.close();
```

❖ Adding, Updating, and Removing Rows

➢ Inserting Rows

To create a new row, construct a ContentValues object and use its put methods to add name/value pairs representing each column name and its associated value.

```
ContentValues newValues = new ContentValues();
newValues.put(KEY_GOLD_HOARD_NAME_COLUMN, hoardName);
newValues.put(KEY_GOLD_HOARDED_COLUMN, hoardValue);
newValues.put(KEY_GOLD_HOARD_ACCESSIBLE_COLUMN, hoardAccessible);
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
db.insert(HoardDBOpenHelper.DATABASE_TABLE, null, newValues);
```

➢ Updating Rows

Updating rows is also done with Content Values. Create a new ContentValues object, using the put methods to assign new values to each column you want to update.

```
ContentValues updatedValues = new ContentValues();
updatedValues.put(KEY_GOLD_HOARDED_COLUMN, newHoardValue);
String where = KEY_ID + "=" + hoardId; String whereArgs[] = null;
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
db.update(HoardDBOpenHelper.DATABASE_TABLE, updatedValues, here, whereArgs);
```

➢ Deleting Rows

To delete a row, simply call the delete method on a database, specifying the table name and a where clause that returns the rows you want to delete

```
String where = KEY_GOLD_HOARDED_COLUMN + "=" + 0;
String whereArgs[] = null;
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
db.delete(HoardDBOpenHelper.DATABASE_TABLE, where, whereArgs);
```

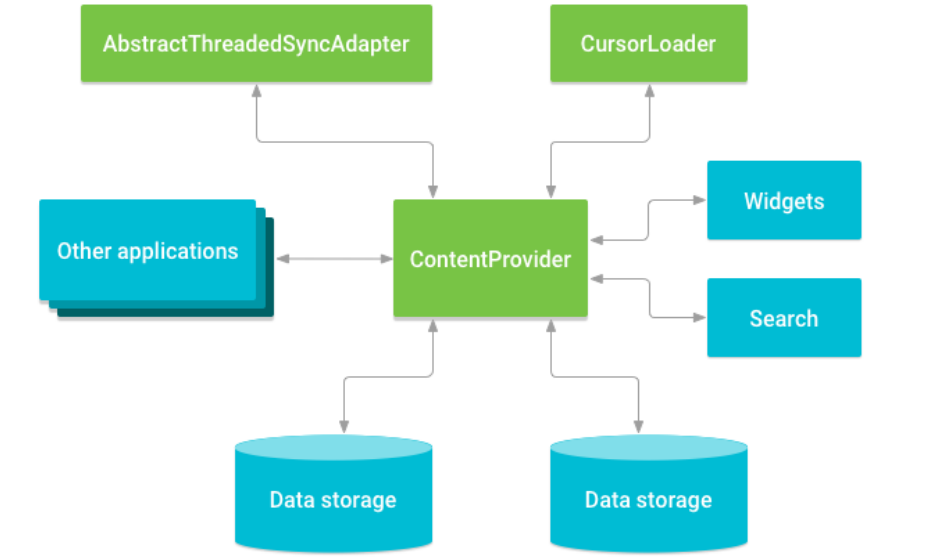
UNIT-IV

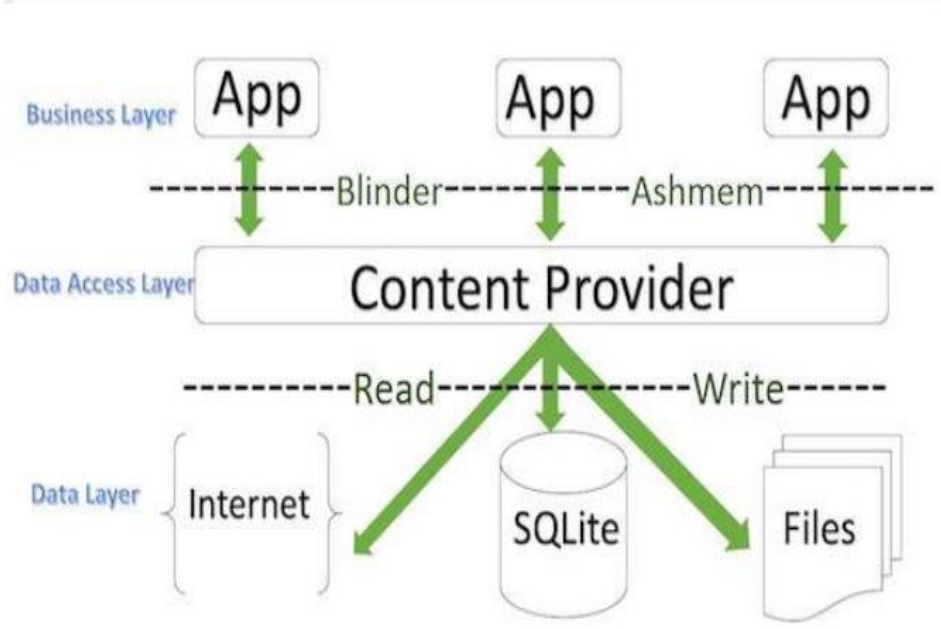
4.1Content Providers

4.1.1 Creating and Using Content Provider

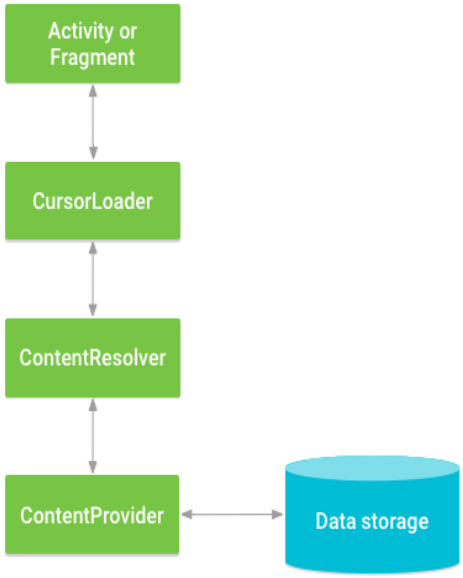
A Content provider coordinates access to the data storage layer in your application for a number of different APIs and components as illustrated in the figure

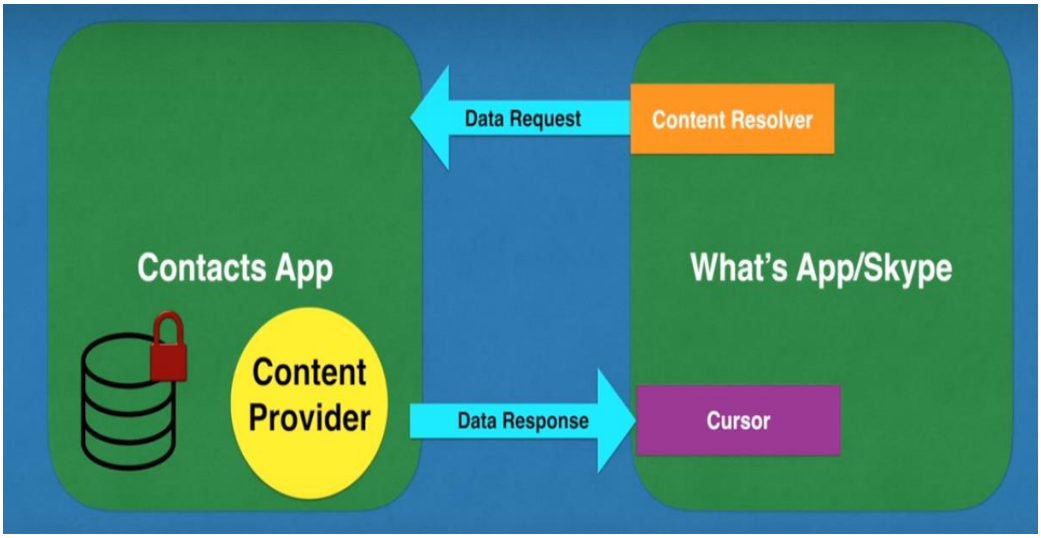
- Sharing access to your application data with other applications
- Sending data to a widget
- Returning custom search suggestions for your application through the search framework using SearchRecentSuggestionsProvider
- Synchronizing application data with your server using an implementation of AbstractThreadedSyncAdapter
- Loading data in your UI using a CursorLoader





To access data in a content provider,use the ContentResolver object in your application's Context to communicate with the provider as a client. The ContentResolver object 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. To query a content provider, you specify the query string in the form of a URI which has following format
<prefix>://<authority>/<data_type>/<id>





1. **prefix**

This is always set to content://

2. **authority**

This specifies the name of the content provider, for example *contacts*, *browser* etc. For third-party content providers, this could be the fully qualified name, such as *com.tutorialspoint.statusprovider*.

3. **data_type**

This indicates the type of data that this particular provider provides. For example, if you are getting all the contacts from the *Contacts* content provider, then the data path would be *people* and URI would look like this *content://contacts/people*

4**. id**

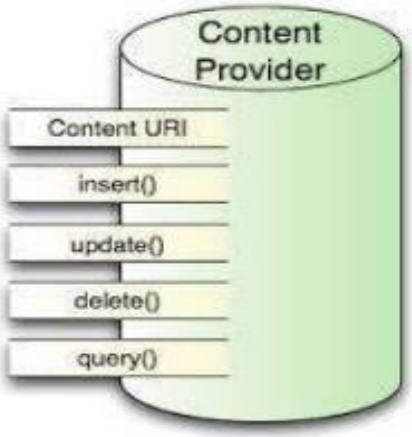
This specifies the specific record requested. For example, if you are looking for contact number 5 in the Contacts content provider then URI would look like this *content://contacts/people/5*.

4.1.2 CREATING CONTENT PROVIDERS

This involves number of simple steps to create your own content provider.

1. First of all you need to create a Content Provider class that extends the *ContentProviderbaseclass.*
2. Second, you need to define your content provider URI address which will be used to access the content.
3. Next you will need to create your own database to keep the content. Usually, Android uses SQLite database and framework needs to override *onCreate()* method which will use SQLite Open Helper method to create or open the provider's database.
4. When your application is launched, the *onCreate()* handler of each of its Content Providers is called on the main application thread.
5. Next you will have to implement Content Provider queries to perform different database specific operations.
6. Finally register your Content Provider in your activity file using <provider> tag.

Here is the list of methods which you need to override in Content Provider class to have your Content Provider working



❖ Registering Content Providers

To create a new Content Provider, extend the abstract ContentProvider class:

public class MyContentProvider extends ContentProvider

Like Activities and Services, Content Providers must be registered in your application manifest before the Content Resolver can discover them.This is done using a provider tag that includes a name attribute describing the Provider's class name and an authorities tag.The general form for defining a Content Provider's authority is as follows:

com.<CompanyName>.provider.<ApplicationName>

The completed provider tag should follow the format shown in the following XML snippet:

<provider android:name=".MyContentProvider"
android:authorities="com.paad.skeletondatabaseprovider"/>

❖ Publishing Your Content Provider's URI Address

Each Content Provider should expose its authority using a public static CONTENT_URI property to make it more easily discoverable. This should include a data path to the primary content — for example:

```
public static final Uri CONTENT_URI =
Uri.parse("content://com.paad.skeletondatabaseprovider/ elements");
```

The following snippet, represents a request for a single record:

content://com.paad.skeletondatabaseprovider/elements/5

❖ Creating the Content Provider's Database

To initialize the data source you plan to access through the Content Provider, override the onCreate method.

```
MySQLiteOpenHelper myOpenHelper;
@Override
public boolean onCreate() {
        myOpenHelper = new MySQLiteOpenHelper(getContext(),
                MySQLiteOpenHelper.DATABASE_NAME, null,
                 MySQLiteOpenHelper.DATABASE_VERSION);
        return true;
}
```

❖ Implementing Content Provider Queries

To support queries with your Content Provider, you must implement the query and getType methods.

Content Resolvers use these methods to access the underlying data, without knowing its structure or implementation.These methods enable applications to share data across application boundaries without having to publish a specific interface for each data source.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[]
selectionArgs, String ,sortOrder) {
        SQLiteDatabase db;
        try {
                db = myOpenHelper . getWritableDatabase();
                } catch (SQLiteException ex) {
                        db = myOpenHelper . getReadableDatabase();
                }
String groupBy = null;
String having = null;
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
switch (uriMatcher . match(uri)) {
        case SINGLE_ROW :
                String rowID = uri.getPathSegments().get(1);
                queryBuilder . appendWhere(KEY_ID + "=" + rowID);
        default: break;
}
queryBuilder . setTables(MySQLiteOpenHelper . DATABASE_TABLE);
```

```
Cursor cursor = queryBuilder .query(db, projection, selection, selectionArgs, groupBy,
having, sortOrder);
return cursor;
}
```

❖ Content Provider Transactions

To expose delete, insert, and update transactions on your Content Provider, implement the corresponding delete, insert, and update methods.Like the query method, these methods are used by Content Resolvers to perform transactions on the underlying data without knowing its implementation — allowing applications to update data across application boundaries.

Storing Files in a Content Provider

Rather than store large files within your Content Provider, you should represent them within a table as fully qualified URIs to a file stored somewhere else on the file system.To support files within your table, you must include a column labeled _data that will contain the path to the file represented by that record. This column should not be used by client applications. Override the openFile handler to provide a ParcelFileDescriptor when the Content Resolver requests the file associated with that record.

```
public ParcelFileDescriptor openFile(Uri uri, String mode)
-----------------------------------------
if (mode.contains("w"))
fileMode |= ParcelFileDescriptor.MODE_WRITE_ONLY;
if (mode.contains("r"))
fileMode |= ParcelFileDescriptor.MODE_READ_ONLY;
if (mode.contains("+"))
fileMode |= ParcelFileDescriptor.MODE_APPEND;
```

4.1.3. USING CONTENT PROVIDERS

When Content Providers are used to expose data, Content Resolvers are the corresponding class used to query and perform transactions on those Content Provider. Whereas Content Providers provide an abstraction from the underlying data, Content Resolvers provide an abstraction from the Content Provider being queried or transacted.

❖ Introducing the Content Resolver

Each application includes a ContentResolver instance, accessible using the getContentResolver method, as follows:

```
ContentResolver cr = getContentResolver();
```

The Content Resolver includes query and transaction methods corresponding to those defined within your Content Providers. The Content Resolver does not need to know the implementation of the Content Providers it is interacting with — each query and transaction method simply accepts a URI that specifies the Content Provider to interact with.Content Providers usually accept two forms of URI, one for requests against all data and another that specifies only a single row.

❖ Querying Content Providers

Content Provider queries take a form very similar to that of database queries. Query results are returned as Cursors over a result set.To extract values from a result Cursor, use the moveTo<location> methods in combination with the get<type> methods to extract values from the specified row and column.

❖ Querying for Content Asynchronously Using the Cursor Loader

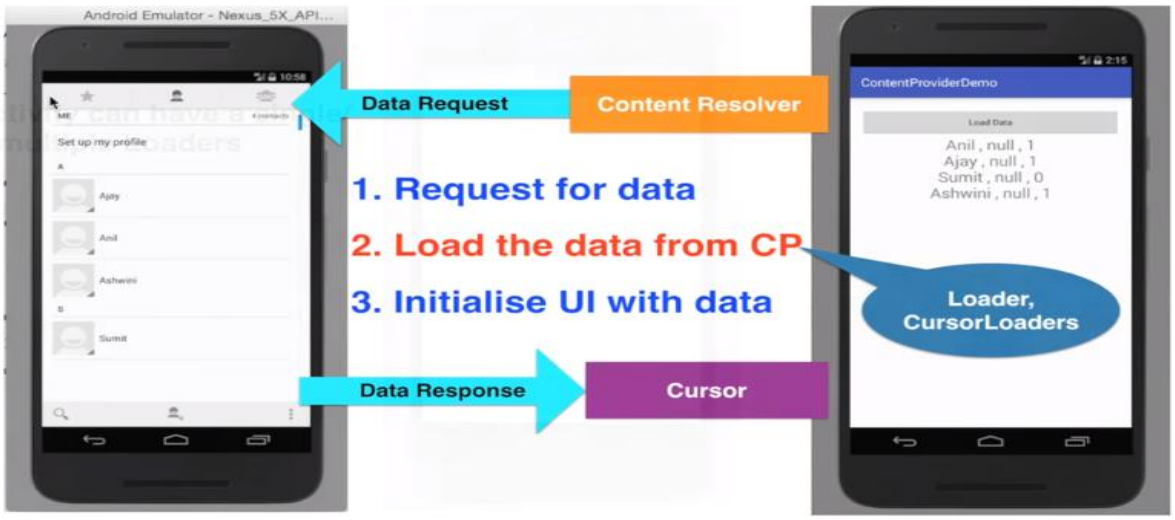
Database operations can be time-consuming, so it's particularly important that any database and Content Provider queries are not performed on the main application thread.It can be difficult to manage Cursors, synchronize correctly with the UI thread, and ensure all queries occur on a background.To help simplify the process, Android 3.0 introduced the Loader class.

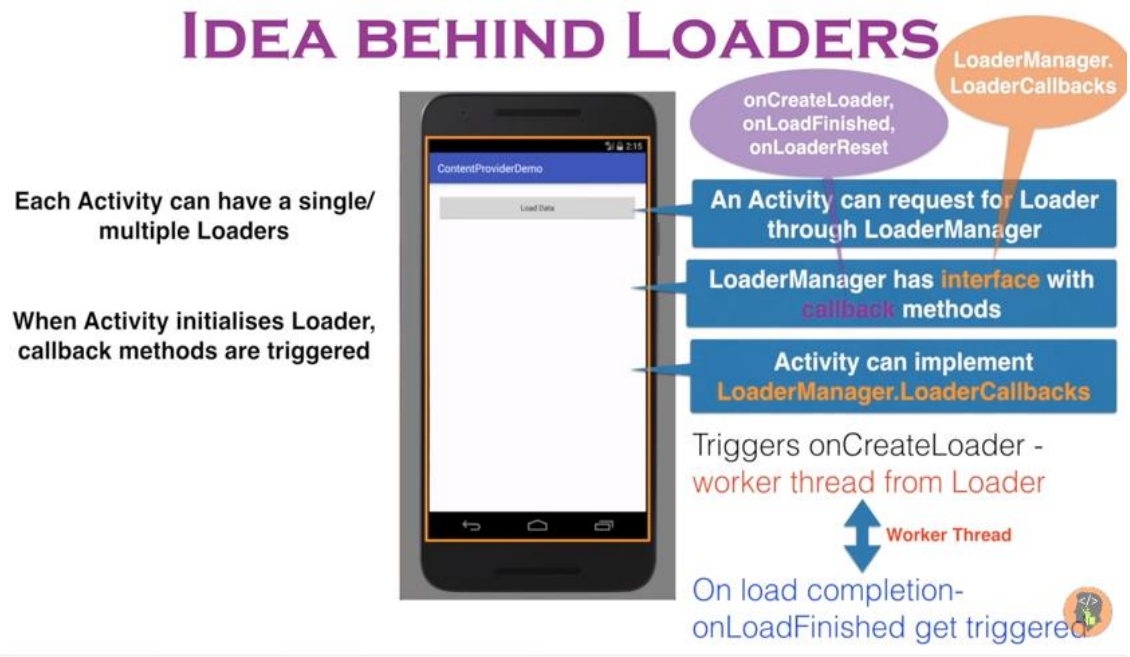
➢ Introducing Loaders

Loaders are available within every Activity and Fragment via the LoaderManager. They are designed to asynchronously load data and monitor the underlying data source for changes.While loaders can be implemented to load any kind of data from any data source, of particular interest is the CursorLoader class. The Cursor Loader allows you to perform asynchronous queries against Content Providers, returning a result Cursor and notifications of any updates to the underlying provider.

➢ Using the Cursor Loader

The Cursor Loader handles all the management tasks required to use a Cursor within an Activity or Fragment. This includes managing the Cursor lifecycle to ensure Cursors are closed when the Activity is terminated.





- Implementing Cursor Loader Callbacks

The Loader Callbacks consist of three handlers:

1. **onCreateLoader** — Called when the loader is initialized, this handler should create and return new Cursor Loader object. Accordingly, when this handler is executed, the query parameters you specify will be used to perform a query using the Content Resolver.
2. **onLoadFinished** — When the Loader Manager has completed the asynchronous query, the onLoadFinished handler is called, with the result Cursor passed in as a parameter. Use this Cursor to update adapters and other UI elements.
3. **onLoaderReset** — When the Loader Manager resets your Cursor Loader, onLoaderReset is called. Within this handler you should release any references to data returned by the query and reset the UI accordingly. The Cursor will be closed by the Loader Manager, so you shouldn't attempt to close it.Each Activity and Fragment provides access to its Loader Manager through a call to getLoaderManager.

LoaderManager loaderManager = getLoaderManager();

To initialize a new Loader, call the Loader Manager's initLoader method, passing in a reference to your Loader Callback implementation, an optional arguments Bundle, and a loader identifier.

Bundle args = null;
loaderManager .initLoader(LOADER_ID, args, myLoaderCallbacks);

This is generally done within the onCreate method of the host Activity.To discard the previous Loader and re-create it, use the restartLoader method.

loaderManager.restartLoader(LOADER_ID, args,myLoaderCallbacks);

- ❖ Adding, Deleting, and Updating Content
- Inserting Content

The Content Resolver offers two methods for inserting new records into a Content Provider: insert and bulkInsert. Both methods accept the URI of the Content Provider into which you're inserting; the insert method takes a single new ContentValues object, and the bulkInsert method takes an array.The insert method returns a URI to the newly added record, whereas the bulkInsert method returns the number of successfully added rows.

```
ContentValues  newValues = new ContentValues();
newValues .put(MyHoardContentProvider.KEY_GOLD_
        HOARD_NAME_COLUMN, hoardName);
newValues.put(MyHoardContentProvider.KEY_GOLD_                HOARDED_COLUMN,
hoardValue);
newValues.put(MyHoardContentProvider.KEY_GOLD_
        HOARD_ACCESSIBLE_COLUMN, hoardAccessible);
ContentResolver   cr = getContentResolver();
Uri  myRowUri = cr .insert    (MyHoardContentProvider.CONTENT_URI, newValues);
```

- Deleting Content

To delete a single record, call delete on the Content Resolver, passing in the URI of the row you want to remove. Alternatively, you can specify a where clause to remove multiple rows.

```
String where = MyHoardContentProvider.
        KEY_GOLD_HOARDED_COLUMN +"=" + 0;
String whereArgs[] = null;
ContentResolver   cr = getContentResolver();
int   deletedRowCount =cr .delete (MyHoardContentProvider.CONTENT_URI, where,
        whereArgs);
```

- Updating Content

You can update rows by using the Content Resolver's update method.The update method takes the URI of the target Content Provider, a ContentValues object that maps column names to updated values, and a where clause that indicates which rows to update.When the update is executed, every row matched by the where clause is updated using the specified Content Values, and the number of successful updates is returned.

```
ContentValues   updatedValues = new ContentValues();
updatedValues. put(MyHoardContentProvider .
        KEY_GOLD_HOARDED_COLUMN, newHoardValue);
Uri   rowURI = ContentUris. withAppendedId(MyHoardContentProvider.CONTENT_URI,
                hoardId);
String where = null;
String whereArgs[] = null;
```

- Accessing Files Stored in Content Providers

Content Providers represent large files as fully qualified URIs rather than raw file blobs; however, this is abstracted away when using the Content Resolver.To access a file stored in, or to insert a new file into, a Content Provider, simply use the Content Resolver's openOutputStream or openInputStream methods, respectively, passing in the URI to the Content Provider row containing the file you require.
The Content Provider will interpret your request and return an input or output stream to the requested file.

4.2. ADDING SEARCH TO YOUR APPLICATION

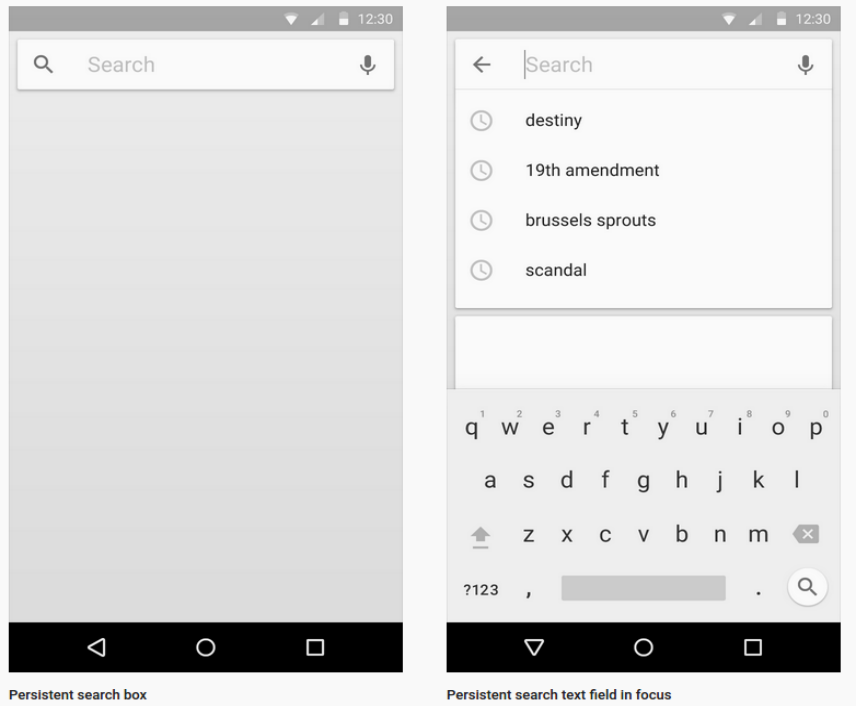
Surfacing your application's content through search is a simple and powerful way to make your content more discoverable, increase user engagement, and improve the visibility of your application.On

mobile devices speed is everything, and search provides a mechanism for users to quickly find the content they need.

Android includes a framework that simplifies the process of making your Content Providers searchable, adding search functionality to your Activities, and surfacing application search results on the home screen.Until Android 3.0 (API level 11) most Android devices featured a hardware search key. In more recent releases this has been replaced with on-screen widgets, typically placed on your application's Action Bar.

You can provide search capabilities for your application in three ways:

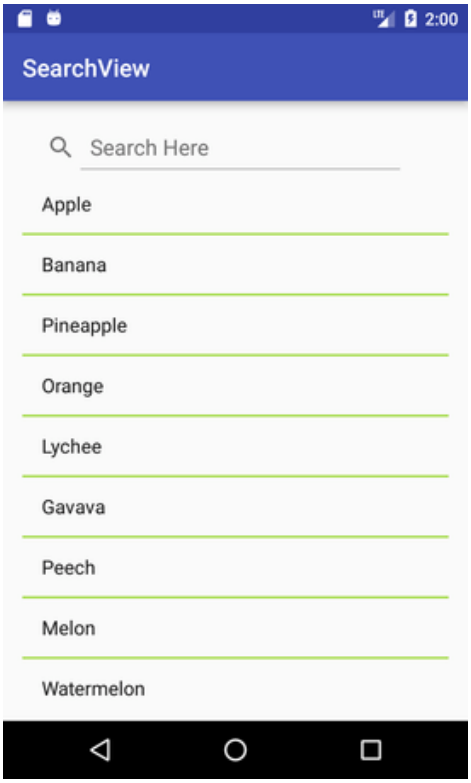
1. **Search bar** — When activated, the search bar *is displayed over the title* bar of your Activity. The search bar is activated when the user presses the hardware search button, or it can be initiated programmatically with a call to your Activity's onSearchRequested method.
2. **Search View** — Introduced in Android 3.0 (API level 11), the Search View is a search widget that can be placed anywhere within your Activity. Typically represented as an icon in the Action Bar.
3. **Quick Search Box** — The Quick Search Box is a home screen search Widget that performs searches across all supported applications. You can configure your application's search results to be surfaced for searches initiated through the Quick Search Box.

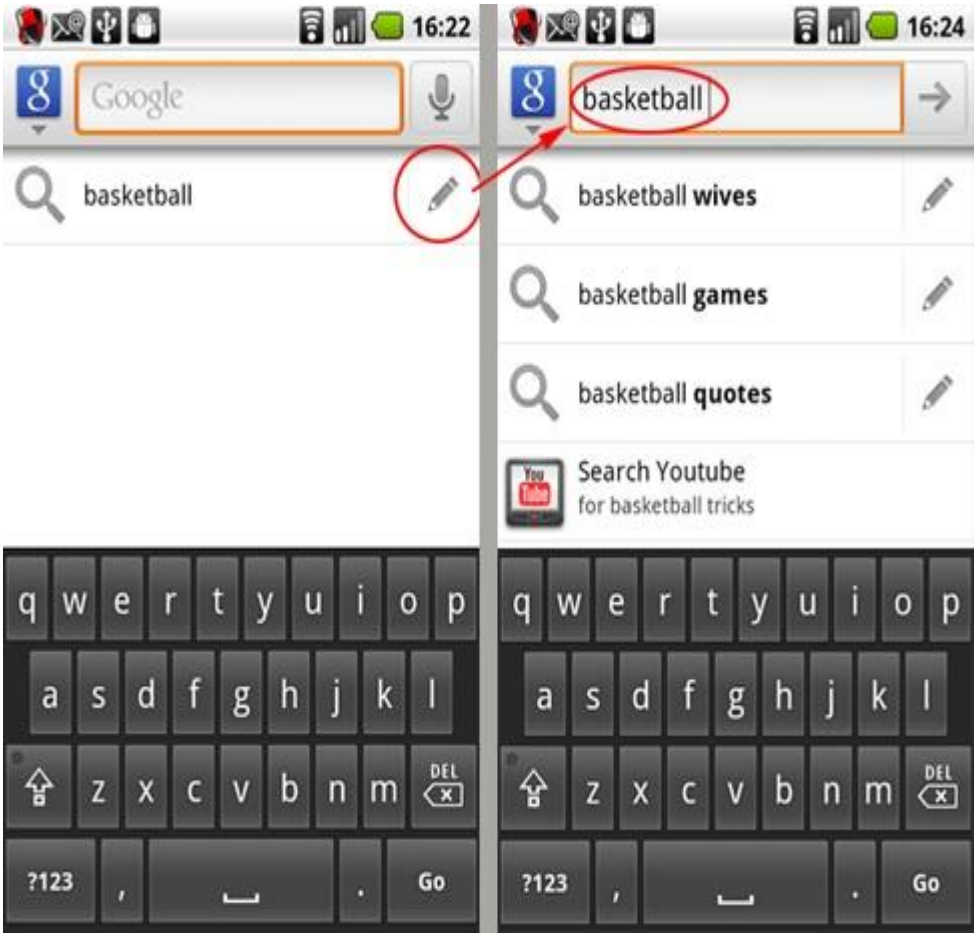


SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

MOBILE APPLICATIONS USING ANDROID





4.2.1 Making Your Content Provider Searchable

❖ Creating searchable configuration

Before you can enable the search dialog or use a Search View widget within your application, you need to define what is searchable.The first step is to create a new searchable metadata XML resource in your project's res/xml folder.Best practice suggests you also include an android:hint attribute to help users understand what they can search for.

```
<?xml version="1.0" encoding="utf-8"?>
<searchable
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:label="@string/app_name"
        android:hint="@string/search_hint">
</searchable>
```

4.2.2 Creating a Search Activity for Your Application

After defining the Content Provider to search, you must now create an Activity that will be used to display search results. This will most commonly be a simple List View-based Activity, but you can use any user interface, provided that it has a mechanism for displaying search results.To indicate that an Activity can be used to provide search results, include an Intent Filter registered for the android.intent.action.SEARCH action andandroid.intent.category.DEFAULT category.You must also include a meta-data tag that includes a name attribute that specifies android.app .searchable, and a resource attribute that specifies a searchable XML resource.

```
<activity android:name=".DatabaseSkeletonSearchActivity"
        android:label="Element Search"
        android:launchMode="singleTop">
        <intent-filter>
            <action  android:name="android.intent.action
                    .SEARCH" />
            <category  android:name= "android.intent.category
                    .DEFAULT" />
        </intent-filter>
        <meta-data
            android:name="android.app.searchable"
            android:resource="@xml/searchable"/>
</activity>
```

❖ Extracting the search query

Searches initiated from within the search results Activity will result in new Intents being received — you can capture those Intents and extract the new queries from the onNewIntent handler.

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        parseIntent(getIntent());
}
@Override
protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        parseIntent(getIntent());
}
private void parseIntent(Intent intent) {


        String searchQuery = intent.getStringExtra(SearchManager.QUERY);
        performSearch(searchQuery);
}
}
```

4.2.3 Making Your Search Activity the Default Search Provider for Your Application

It is good practice to use the same search results form for your entire application. To set a search Activity as the default search result provider for all Activities within your application, add a *meta-data* tag within the application manifest node. Set the name attribute to android. *app.default_searchable* and specify your search Activity using the value attribute.

```
<meta-data
android:name="android.app.default_searchable"
android:value=".DatabaseSkeletonSearchActivity"
/>
```

❖ Performing a Search and Displaying the Results

When the search Activity receives a new search query, it must execute the search and display the results within the Activity.If you are searching a Content Provider, it's good practice to use a Cursor Loader to execute a query whose result Cursor is bound to a List View.In most circumstances you'll need to provide some functionality beyond simply displaying the search results. If you are using a List Activity or List Fragment, you can override the *onListItemClick* handler to react to user's selecting a search result, such as displaying the result details

```
public class DBSkeletonSearchActivity extends ListActivity implements LoaderManager.
LoaderCallbacks<Cursor> {
        public void onCreate(Bundle savedInstanceState) {
            // Create a new adapter and bind it to the List View
setListAdapter(adapter);
// Initiate the Cursor Loader
getLoaderManager().initLoader(0, null, this);
        }
    private void performSearch(String query) {
            Bundle args = new Bundle();
// Pass the search query as an argument to the Cursor Loader
            args.putString(QUERY_EXTRA_KEY, query);
            getLoaderManager(). restartLoader(0, args, this);
        }
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
// Extract the search query from the arguments.
// Construct the new query in the form of a Cursor Loader.
            }
    public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
        // Replace the result Cursor displayed by the Cursor Adapter with the new
result set.
adapter.swapCursor(cursor);
        }
    public void onLoaderReset(Loader<Cursor> loader) {
// Remove the existing result Cursor from the List Adapter.
            adapter.swapCursor(null);
}  }
```

- ❖ Providing actions for search result selection

In most circumstances you'll need to provide some functionality beyond simply displaying the search results.

```
@Override
protected void onListItemClick(ListView listView, View view, int position, long id) {
    super.onListItemClick(listView, view, position, id);
    // Create a URI to the selected item.
    Uri selectedUri =ContentUris. withAppendedId
    (MyContentProvider.CONTENT_URI, id);
    // Create an Intent to view the selected item.
    Intent intent = new Intent(Intent.ACTION_VIEW);
```

```
        intent.setData(selectedUri);
        // Start an Activity to view the selected item.
        startActivity(intent);
    }
```

4.2.4 Using the Search View Widget

Android 3.0 (API level 11) introduced the *SearchView* widget as an alternative to the Activity search bar. The Search View appears and behaves as an *Edit Text View*, but it can be configured to offer search suggestions and to initiate search queries within your application in the same way as the search bar in earlier versions of Android.To connect your Search View to your search Activity, you must first extract a reference to its *SearchableInfo* using the Search Manager's *getSearchableInfo* method.

Use the Search View's *setSearchableInfo* method to bind this Activities searchable object to your Search View.

```
SearchManager searchManager =
        (SearchManager)getSystemService(Context.SEARCH_SERVICE);
SearchableInfo searchableInfo =
                searchManager. getSearchableInfo(getComponentName());
                // Bind the Activity's SearchableInfo to the Search View
SearchView searchView = (SearchView)findViewById(R.id.searchView);
searchView. setSearchableInfo(searchableInfo);
```

By default, the Search View will be displayed as an icon that, when clicked, expands to the search edit box. You can use its setIconifiedByDefault method to disable this and have it always display as an edit box.

```
searchView.setIconifiedByDefault(false);
```

By default a Search View query is initiated when the user presses Enter. You can choose to also display a button to submit a search using the setSubmitButtonEnabled method.

```
searchView.setSubmitButtonEnabled(true);
```

4.2.5 Supporting Search Suggestions from a Content Provider

One of the most engaging innovations in search is the provision of real-time search suggestions as users type their queries, allowing them to bypass the search result Activity and jump directly to the search result.To support search suggestions, you need to configure your Content Provider to recognize specific URI paths as search queries.Although your search Activity can structure its query and display the results Cursor data in any way, if you want to provide search suggestions, you need to create (or modify) a Content Provider to receive search queries and return suggestions using the expected projection.

```
private static final int ALLROWS = 1;
private static final int SINGLE_ROW = 2;
private static final int SEARCH = 3;
private static final UriMatcher uriMatcher;
static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("com.paad.skeletondatabaseprovider", "elements",
                                ALLROWS);
        uriMatcher.addURI("com.paad.skeletondatabaseprovider","elements/ #",
                                        SINGLE_ROW);
uriMatcher.addURI("com.paad.skeletondatabaseprovider",
```

```
                SearchManager. SUGGEST_URI_PATH_QUERY, SEARCH);
uriMatcher.addURI("com.paad.skeletondatabaseprovider",
        SearchManager.SUGGEST_URI_PATH_QUERY + "/*", SEARCH);
uriMatcher.addURI("com.paad.skeletondatabaseprovider",
                SearchManager.SUGGEST_URI_PATH_SHORTCUT, SEARCH);
uriMatcher.addURI("com.paad.skeletondatabaseprovider",
                        SearchManager.SUGGEST_URI_PATH_SHORTCUT +
                        "/*", SEARCH);
}
```

The Search Manager requests your search suggestions by initiating a query on your Content Provider, passing in the query value as the last element in the URI path.There are two required columns, SUGGEST_COLUMN_TEXT_1, which displays the search result text, and _id, which indicates the unique row ID.It's also useful to include a SUGGEST_COLUMN_INTENT_DATA_ID column. The value returned in this column can be appended to a specified URI path and used to populate an Intent that will be fired if the suggestion is selected.As speed is critical for real-time search results, in many cases it's good practice to create a separate table specifically to store and provide them.

4.3Native Android Content Providers

Android exposes several native Content Providers, which you can access directlythe most useful Content Providers, including the following:

1. **Media Store** — Provides centralized, managed access to the multimedia on your device, including audio, video, and images. You can store your own multimedia within the Media Store and make it globally available.
2. **Browser** — Reads or modifies browser and browser search history.
3. **Contacts Contract —** Retrieves, modifies, or stores contact details and associated social stream updates.
4. **Calendar** — Creates new events, and deletes or updates existing calendar entries. That includes modifying the attendee lists and setting reminders.
5. **Call Log** — Views or updates the call history, including incoming and outgoing calls, missed calls, and call details, including caller IDs and call durations.

4.3.1 Using the Media Store Content Provider

The Android Media Store is a managed repository of audio, video, and image files.Whenever you add a new multimedia file to the file system, it should also be added to the Media Store using the Content Scanner, this will expose it to other applications, including media players.the MediaStore class includes Audio, Video, and Images subclasses, which in turn contain subclasses that are used to provide the column names and content URIs for the corresponding media providers.The Media Store segregates media kept on the internal and external volumes of the host device.Each Media Store subclass provides a URI for either the internally or externally stored media using the forms:

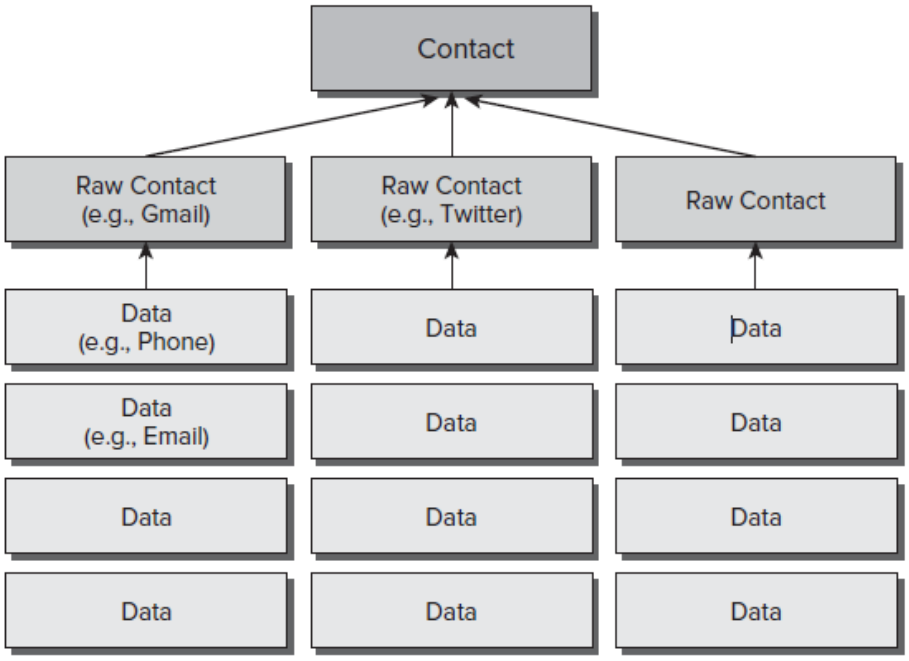
MediaStore.<mediatype>.Media.EXTERNAL_CONTENT_URI
MediaStore.<mediatype>.Media.INTERNAL_CONTENT_URI

4.3.2 Using the Contacts Contract Content Provider

Android makes the full database of contact information available to any application that has been granted the READ_CONTACTS permission.The Contacts Contract Provider provides an extensible database of contact-related information.This allows users to specify multiple sources for their contact information. More importantly, it allows developers to arbitrarily extend the data stored against each contact, or even become an alternative provider for contacts and contact details.

❖ Introducing the Contacts Contract Content Provider

The contract between the contacts provider and applications. Contains definitions for the supported URIs and columns.This allows users to specify multiple sources for their contact information. More importantly, it allows developers to arbitrarily extend the data stored against each contact. Rather than providing a single, fully defined table of contact detail columns, the Contacts Contract provider uses a three-tier data model to store data, associate it with a contact, and aggregate it to a single person using the following ContactsContract subclasses



- **Data** — Each row in the underlying table defines a set of personal data (phone numbers, email addresses, and so on)
  - Although there is a predefined set of common column names for each personal data-type available, this table can be used to store *any value.*
  - When adding new data to the Data table, you specify a Raw Contact to which a set of data will be associated.
- **RawContacts** — From Android 2.0 (API level 5) forward, users can add multiple contact account providers to their device. Each row in the Raw Contacts table defines an account to which a set of Data values is associated.
- **Contacts** — The Contacts table aggregates rows from Raw Contacts that all describe the same person

❖ Reading Contact Details

To access any contact details, you need to include the READ_CONTACTS uses-permission in your application manifest:

<uses-permission android:name="android.permission.READ_CONTACTS"/>

Use the Content Resolver to query any of the three Contact Contracts Providers using their respective CONTENT_URI static constants. The *ContactsContract.Data* Content Provider is used to store all the contact details, such as addresses, phone numbers, and email addresses.To simplify this lookup, Android provides the query URI ContactsContract.Contacts.CONTENT_FILTER_URI. To extract the associated contact details, find the _ID value from the returned Cursor, and use it to create a query on the Data table.Use ContactsContract.PhoneLookup.CONTENT_FILTER_URI, appending the number to look up as an additional path segment.

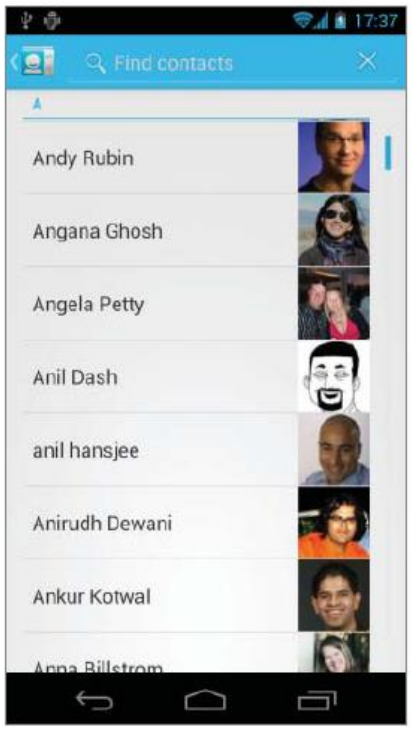
Uri lookupUri =Uri.withAppendedPath (ContactsContract.PhoneLookup. CONTENT_FILTER_URI, incomingNumber);

❖ Creating and Picking Contacts Using Intents

The Contacts Contract Content Provider includes an Intent-based mechanism that can be used to view, insert, or select a contact using an existing contact application.To display a list of contacts for your users to select from, you can use the Intent.ACTION_PICK action along with the ContactsContract.Contacts.CONTENT_URI.

```
private static int PICK_CONTACT = 0;
private void pickContact() {
        Intent intent = new Intent(Intent.ACTION_PICK,
        ContactsContract.Contacts.CONTENT_URI);
        startActivityForResult(intent, PICK_CONTACT);
}
```

This will display a List View of the contacts available as shown in the picture. When the user selects a contact, it will be returned as a URI within the data property of the returned Intent as



```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
super.onActivityResult(requestCode, resultCode, data);
if ((requestCode == PICK_CONTACT) && (resultCode == RESULT_OK)) {
resultTextView.setText(data.getData().toString());
}
}
```

There are two alternatives to insert a new contact, both of which will prepopulate the new contact for using the values you specify as extras in your Intent. The ContactsContract.Intents.SHOW_OR_CREATE_CONTACT action will search the contacts Provider for a particular email address or telephone number URI, offering to insert a new entry only if a contact with the specified contact address doesn't exist.Use the constants in the ContactsContract.Intents.Insert class to include Intent extras that can be used to prepopulate contact details, including the name, company, email, phone number, notes, and postal address of the new contact.

```
Intent intent =
        new Intent(ContactsContract.Intents.SHOW_OR_CREATE_CONTACT,
                                ContactsContract.Contacts.CONTENT_URI);
intent.setData(Uri.parse("tel:(650)253-0000"));
intent.putExtra(ContactsContract.Intents.Insert.COMPANY, "Google");
intent.putExtra(ContactsContract.Intents.Insert.POSTAL,
                "1600 Amphitheatre Parkway, Mountain View, California");
startActivity(intent);
```

Modifying and Augmenting Contact Details DirectlyYou can use the contact Content Providers to modify, delete, or insert contact records after adding the WRITE_CONTACTS uses-permission to your application manifest.

```
<uses-permission android:name="android.permission.WRITE_CONTACTS"/>
```

4.3.3 Using the Calendar Content Provider

Android 4.0 (API level 14) introduced a supported API for accessing the Calendar Content Provider.The Calendar API allows you to insert, view, and edit the complete Calendar database, providing access to calendars, events, attendees, and event reminders using either Intents or through direct manipulation of the Calendar Content Providers.Like the Contacts Contract Content Provider, the Calendar Content Provider is designed to support multiple synchronized accounts.

❖ Querying the Calendar

To access the Calendar Content Provider, you must include the READ_CALENDAR uses-permission in your application manifest:

```
<uses-permission android:name="android.permission.READ_CALENDAR"/>
```

Use the Content Resolver to query any of the Calendar Provider tables using their CONTENT_URI static constant. Each table is exposed from within the CalendarContract class, including:

1. **Calendars** — The Calendar application can display multiple calendars associated with multiple accounts. This table holds each calendar that can be displayed, as well as details such as the calendar's display name, time zone, and color.
2. **Events** — The Events table includes an entry for each scheduled calendar event, including the name, description, location, and start/end times.

3. **Instances** — Each event has one or (in the case of recurring events) multiple instances. The Instances table is populated with entries generated by the contents of the Events table and includes a reference to the event that generated it.
4. **Attendees** — Each entry in the Attendees table represents a single attendee of a given event.Each attendee can include a name, email address, and attendance status, and if they are optional or required guests.
5. **Reminders** — Event reminders are represented within the Reminders table, with each row representing one reminder for a particular event.Each class includes its column names as static properties.

❖ Creating and Editing Calendar Entries Using Intents

The Calendar Content Provider includes an Intent-based mechanism. Using Intents, you can open the Calendar application to a specific time, view event details, insert a new event, or edit an existing event.
Like the Contacts API, using Intents is the best practice to direct manipulation of the underlying tables whenever possible.

❖ Creating New Calendar Events

Using the Intent.ACTION_INSERT action, specifying the CalendarContract. Events. CONTENT_URI, you can add new events to an existing calendar without requiring any special permissions.When triggered, the Intent will be received by the Calendar application, which will create a new entry prepopulated with the data provided.

.

```
Intent intent = new Intent(Intent.ACTION_INSERT, CalendarContract. Events. CONTENT_URI);
intent.putExtra(CalendarContract.Events.TITLE, “Launch!”);
intent.putExtra(CalendarContract.Events.DESCRIPTION, “Professional Android 4 “ +
                                                “Application Development release!”);
intent.putExtra(CalendarContract.Events.EVENT_LOCATION, “Wrox.com”);
Calendar startTime = Calendar.getInstance();
startTime.set(2012, 2, 13, 0, 30);
intent.putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME, startTime. getTimeInMillis());
intent.putExtra(CalendarContract.EXTRA_EVENT_ALL_DAY, true);
startActivity(intent);
```

❖ Editing Calendar Events

To edit a calendar event, you must first know its row ID. To find this, you need to query the Events Content Provider. When you have the ID of the event you want to edit, create a new Intent using the Intent.ACTION_EDIT action and a URI that appends the event’s row ID to the end of the Events table’s CONTENT_URI,

```
long rowID = 760;
Uri uri = ContentUris.withAppendedId( CalendarContract.Events.
                        CONTENT_URI, rowID);
Intent intent = new Intent(Intent.ACTION_EDIT, uri);
// Modify the calendar event details
Calendar startTime = Calendar.getInstance();
startTime.set(2012, 2, 13, 0, 30);
intent.putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME,
                        startTime.getTimeInMillis());
```

```
intent.putExtra(CalendarContract.EXTRA_EVENT_ALL_DAY, true);
// Use the Calendar app to edit the event.
startActivity(intent);
```

❖ Displaying the Calendar and Calendar Events

You can also use Intents to display a particular event or to open the Calendar application to display a specific date and time using the Intent.ACTION_VIEW action.To view an existing event, specify the Intent's URI using a row ID, as you would when editing an event. To view a specific date and time, the URI should be of the form

```
content://com.android.calendar/time/[milliseconds since epoch]
 as shown below
// Create a URI that specifies a particular time to view.
Calendar startTime = Calendar.getInstance();
startTime.set(2012, 2, 13, 0, 30);
Uri uri = Uri.parse("content://com.android.calendar/time/" +
                                    String.valueOf(startTime.getTimeInMillis()));
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
// Use the Calendar app to view the time.
startActivity(intent);
```

❖ Modifying Calendar Entries Directly

If you are building your own contacts application, or want to build a Sync Adapter to integrate events from your own cloud-based calendar service, you can use the Calendar Content Providers to modify, delete, or insert contact records after adding the WRITE_CONTACTS uses-permission to your application manifest.

```
<uses-permission android:name="android.permission.WRITE_CALENDAR"/>
```

WORKING IN THE BACKGROUND

4.4INTRODUCING SERVICES

Android offers the Service class to create application components that handle long-lived operations and include functionality that doesn't require a user interface.Services run invisibly — doing Internet lookups, processing data, updating your Content Providers, firing Intents, and triggering Notifications.Running Services have a higher priority than inactive or invisible (stopped) Activities, making them less likely to be terminated by the run time's resource management. The only reason Android will stop a Service prematurely is to provide additional resources for a foreground component (usually an Activity). When that happens, your Service can be configured to restart automatically when resources become available.

4.4.1 Creating and Controlling Services

To define a Service, create a new class that extends Service. You'll need to override the onCreate and onBind methods.

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
public class MyService extends Service {
@Override
        public void onCreate() {
```

```
        super.onCreate();
        // TODO: Actions to perform when service is created.
    }
@Override
    public IBinder onBind(Intent intent) {
        // TODO: Replace with service binding implementation.
        return null;
    }
}
```

After you've constructed a new Service, you must register it in the application manifest.

```
<service android:enabled="true"    android:name= ".MyService"/>
```

To ensure your Service can be started and stopped only by your own application, add a permission attribute to its Service node.

```
<service android:enabled="true"
        android:name=".MyService"
        android:permission="com.paad.
MY_SERVICE_PERMISSION"/>
```

This will require any third-party applications to include a uses-permission in their manifests in order to access this Service.

❖ Executing a Service and Controlling Its Restart Behavior

Override the onStartCommand event handler to execute the task (or begin the ongoing operation) encapsulated by your Service. You can also specify your Service's restart behavior within this handler.The onStartCommand method is called whenever the Service is started using startService, so it may be executed several times within a Service's lifetime. You should ensure that your Service accounts for this.Services are launched on the main Application Thread, meaning that any processing done in the onStartCommand handler will happen on the main GUI Thread. The standard pattern for implementing a Service is to create and run a new Thread from onStartCommand to perform the processing in the background, and then stop the Service when it's been completed.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
startBackgroundTask(intent, startId);
return Service.START_STICKY;
}
```

This pattern lets onStartCommand complete quickly, and it enables you to control the restart behavior by returning one of the following Service constants:

1. **START_STICKY**— Describes the standard behavior, which is similar to the way in which onStart was implemented prior to Android 2.0. If you return this value, onStartCommand will be called any time your Service restarts after being terminated by the run time. Note that on a restart the Intent parameter passed in to onStartCommand will be null. This mode typically is used for Services that handle their own states and that are explicitly started and stopped as required (via startService and stopService). This includes Services that play music or handle other ongoing background tasks.
2. **START_NOT_STICKY**— This mode is used for Services that are started to process specific actions or commands. Typically, they will use stopSelf to terminate once that command has been completed.

Following termination by the run time, Services set to this mode restart only if there are pending start calls. If no startService calls have been made since the Service was terminated, the Service will be stopped without a call being made to onStartCommand.Rather than restarting the Service during a period of resource contention, it's often more prudent to let the Service stop and retry at the next scheduled interval.

3. **START_REDELIVER_INTENT**— In some circumstances you will want to ensure that the commands you have requested from your Service are completed — for example when timeliness is important.This mode is a combination of the first two; if the Service is terminated by the run time, it will restart only if there are pending start calls *or the process was killed prior to its calling* stopSelf. In the latter case, a call to onStartCommand will be made, passing in the initial Intent whose processing did not properly complete.The flag parameter can be used to discover how the Service was started. In particular, you determine if either of the following cases is true:
4. **START_FLAG_REDELIVERY** — Indicates that the Intent parameter is a redelivery caused by the system run time's having terminated the Service before it was explicitly stopped by a call to stopSelf.
5. **START_FLAG_RETRY**— Indicates that the Service has been restarted after an abnormal termination. It is passed in when the Service was previously set to START_STICKY.

❖ Starting and Stopping Services

To start a Service, call startService. Much like Activities, you can either use an action to implicitly start a Service with the appropriate Intent Receiver registered, or you can explicitly specify the Service using its class.If the Service requires permissions that your application does not have, the call to startService will throw a SecurityException.

```
private void explicitStart() {
// Explicitly start My Service
Intent intent = new Intent(this, MyService.class);
// TODO Add extras if required.
startService(intent);
}
private void implicitStart() {
// Implicitly start a music Service
Intent intent = new Intent(MyMusicService.PLAY_ALBUM);
intent.putExtra(MyMusicService.ALBUM_NAME_EXTRA, "United");
intent.putExtra(MyMusicService.ARTIST_NAME_EXTRA, "Pheonix");
startService(intent);
}
```

To stop a Service, use stopService, specifying an Intent that defines the Service to stop

```
// Stop a service explicitly.
stopService(new Intent(this, MyService.class));
// Stop a service implicitly.
        Intent intent = new Intent(MyMusicService.PLAY_ALBUM);
stopService(intent);
```

❖ Self-Terminating Services

Due to the high priority of Services, they are not commonly killed by the run time, so self-termination can significantly improve the resource footprint of your application.By explicitly stopping the Service when your processing is complete by making a call to stopSelf, you allow the system to recover the resources

otherwise required to keep it running.You can call stopSelf either without a parameter to force an immediate stop, or by passing in a startId value to ensure processing has been completed for each instance of startService called so far.

4.4.2 Binding Services to Activities

Binding is useful for Activities that would benefit from a more detailed interface with a Service. To support binding for a Service, implement the onBind method, returning the current instance of the Service being bound.

```
@Override
public IBinder onBind(Intent intent) {
        return binder;
}
public class MyBinder extends Binder {
        MyMusicService getService() {
        return MyMusicService.this;
}
}
private final IBinder binder = new MyBinder();
```

The connection between the Service and another component is represented as a ServiceConnection.To bind a Service to another application component, you need to implement a new ServiceConnection, overriding the onServiceConnected and onServiceDisconnected methods to get a reference to the Service instance after a connection has been established.

```
// Reference to the service
private MyMusicService serviceRef;
// Handles the connection between the service and activity
private ServiceConnection mConnection = new ServiceConnection() {
                public void onServiceConnected(ComponentName className,
                                                IBinder service) {
                // Called when the connection is made.
                serviceRef = ((MyMusicService.MyBinder)service).getService();
}
public void onServiceDisconnected(ComponentName className) {
                // Received when the service unexpectedly disconnects.
                serviceRef = null;
}
};
```

To perform the binding, call bindService within your Activity, passing in an Intent (either explicit or implicit) that selects the Service to bind to, and an instance of a ServiceConnection implementation.You can also specify a number of binding flags,

```
// Bind to the service
Intent  bindIntent = new Intent(MyActivity.this,
        MyMusicService.class);
bindService(bindIntent, mConnection,
        Context.BIND_AUTO_CREATE);
```

Android 4.0 (API level 14) introduced a number of new flags that can be used and combined when binding a Service to an application:

- BIND_ADJUST_WITH_ACTIVITY — Causes the Service's priority to be adjusted based on the relative importance of the Activity to which it is bound.
- BIND_ABOVE_CLIENT and BIND_IMPORTANT — Specify that the bound Service is so important to the binding client that it should become a foreground process when the client is in the foreground — in the case of BIND_ABOVE_CLIENT, you are specifying that the run time should terminate the Activity before the bound Service in cases of low memory.
- BIND_NOT_FOREGROUND — Ensures the bound Service is never brought to foreground priority. By default, the act of binding a Service increases its relative priority.
- BIND_WAIVE_PRIORITY — Indicates that binding the specified Service shouldn't alter its priority.

Android applications do not (normally) share memory, but in some cases your application may want to interact with (and bind to) Services running in different application processes.You can communicate with a Service running in a different process by using broadcast Intents or through the extras Bundle in the Intent used to start the Service. If you need a more tightly coupled connection, you can make a Service available for binding across application boundaries by using Android Interface Definition Language (AIDL). AIDL defines the Service's interface in terms of OS-level primitives, allowing Android to transmit objects across process boundaries.

4.4.3 Creating Foreground Services

Android uses a dynamic approach to managing resources that can result in your application's components being terminated with little or no warning.When calculating which applications and application components should be killed, Android assigns running Services the second-highest priority.Only active, foreground Activities are considered a higher priority.In cases where your Service is interacting directly with the user, it may be appropriate to lift its priority to the equivalent of a foreground Activity's.

You can do this by setting your Service to run in the foreground by calling its startForeground method.

Because foreground Services are expected to be interacting directly with the user (for example, by playing music), calls to startForeground must specify an ongoing Notification.This notification will be displayed for as long as your Service is running in the foreground.

```
private void startPlayback(String album, String artist) {
        int NOTIFICATION_ID = 1;
        // Create an Intent that will open the main Activity
        // if the notification is clicked.
        Intent intent = new Intent(this, MyActivity.class);
        PendingIntent  pi = PendingIntent.getActivity(this, 1, intent, 0);
        // Set the Notification UI parameters
        Notification notification = new Notification(R.drawable.icon,
                        "Starting Playback", System.currentTimeMillis());
        notification.setLatestEventInfo(this, album, artist, pi);
        // Set the Notification as ongoing
        notification.flags = notification.flags |  Notification.
                FLAG_ONGOING_EVENT;
        // Move the Service to the Foreground
        startForeground(NOTIFICATION_ID, notification);
}
```

It's good practice to provide a simple way for users to disable a foreground Service. When your Service no longer requires foreground priority, you can move it back to the background, and optionally remove the ongoing notification using the stopForeground method.

```
public void pausePlayback() {
        // Move to the background and remove the Notification
        stopForeground(true);
}
```

4.5USING BACKGROUND THREADS

Responsiveness is one of the most critical attributes of a good Android application.To ensure that your application responds quickly to any user interaction or system event, it's vital that you move all processing and I/O operations off the main application Thread and into a child Thread.In Android, Activities that don't respond to an input event (such as a key press) within 5 seconds, and **Broadcast Receivers** that don't complete their ***onReceive*** handlers within 10 seconds, areconsidered unresponsive.In practice, users will notice input delays and UI pauses of more than a couple of 100 ms.It's important to use background Threads for any nontrivial processing that doesn't directly interact with the user interface.

It's particularly important to schedule

- ✓ file operations,
- ✓ network lookups,
- ✓ database transactions,
- ✓ and complex calculations

on a background Thread.Android offers a number of alternatives for moving your processing to the background. You can implement your own Threads and use the **Handler** class to synchronize with the **GUI Thread** before updating the UI. Alternatively, the ***AsyncTask*** **class** lets you define an operation to be performed in the background and provides event handlers that enable you to monitor progress and post the results on the GUI Thread.

4.5.1 Using *AsyncTask* to Run Asynchronous Tasks

The **AsyncTask** class implements a best practice pattern for moving your time-consuming operations onto a **background Thread** and synchronizing with the **UI Thread** for updates and when the processing is complete. **AsyncTask** handles all the Thread creation, management. **AsyncTasks** are a good solution for short-lived background processing whose progress and results need to be reflected on the UI. However, they aren't persisted across Activity restarts. For longer running background processes, such as downloading data from the Internet, a **Service component** is a better approach.

❖ Creating New Asynchronous Tasks

Each **AsyncTask** implementation can specify parameter types that will be used for input parameters, the progress-reporting values, and result values. If you don't need or want to take input parameters, **update**

progress, or report a final result, simply specify **Void** for any or all the types required.To create a new asynchronous task, extend the **AsyncTask class**, specifying the parameter types to use.

```
private class MyAsyncTask extends AsyncTask<String, Integer, String> {
@Override
protected String   doInBackground(String... parameter) {
                String  result = "";
                int   myProgress = 0;
                int   inputLength = parameter[0].length();
                for (int i = 1; i <= inputLength; i++) {
                        myProgress = i;
                        result = result + parameter[0].charAt(inputLength-i);
                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) { }
                        publishProgress(myProgress);
                }
                return result;
}
@Override
protected void onProgressUpdate(Integer... progress) {
                asyncProgress.setProgress(progress[0]);
}
@Override
protected void onPostExecute(String result) {
                asyncTextView.setText(result);
}
}
```

Your subclass should also override the following event handlers:

1. ***doInBackground***— This method will be executed on the background Thread, so place long-running code here, and don't attempt to interact with UI objects from within this handler. **publishProgress** method can be used from within this handler to pass parameter values to the **onProgressUpdate handler**, and when your background task is complete, you can return the final result as a parameter to the **onPostExecute** handler, which can update the UI accordingly.

2. ***onProgressUpdate***— Override this handler to update the UI with interim progress updates. This handler receives the set of parameters passed in to **publishProgress** (typically from within the **doInBackground handler**).This handler is synchronized with the GUI Thread when executed, so you can safely modify UI elements.

3. ***onPostExecute***— When **doInBackground** has completed, the return value from that method is passed in to this event handler.Use this handler to update the UI when your asynchronous task has completed. This handler is synchronized with the GUI Thread when executed, so you can safely modify UI elements.

❖ Running Asynchronous Tasks

After you've implemented an asynchronous task, execute it by creating a new instance and calling execute. You can pass in a number of parameters, each of the type specified in your implementation.

String input = "redrum ... redrum";

new MyAsyncTask().execute(input);

4.5.2 Introducing the Intent Service

The **Intent Service** is a convenient **wrapper class** that implements the best practice pattern for background Services that perform set tasks on demand, such as recurring Internet updates or data processing.The **Intent Service** queues request Intents as they are received and processes them consecutively on an asynchronous background Thread. After every received Intent has been processed, the Intent Service will terminate itself.The Intent Service handles all the complexities around queuing multiple requests, background Thread creation, and **UI Thread** synchronization.To implement a Service as an Intent Service, extend **IntentService** and override the **onHandleIntent** handler

The **onHandleIntent** handler will be executed on a **worker Thread**, once for each Intent received. The **Intent Service** is the best-practice approach to creating Services that perform set tasks either on-demand or at regular intervals.

```
import android.app.IntentService;
import android.content.Intent;
public class MyIntentService extends IntentService {
        public MyIntentService(String name) {
                super(name);
                // TODO Complete any required constructor tasks.
        }
        @Override
        public void onCreate() {
                super.onCreate();
                // TODO: Actions to perform when service is created.
        }
        @Override
        protected void onHandleIntent(Intent intent) {
                // This handler occurs on a background thread.
                // TODO The time consuming task should be implemented here.
                // Each Intent supplied to this IntentService will be
                // processed consecutively here. When all incoming Intents
                // have been processed the Service will terminate itself.
        }
}
```

4.5.3 Introducing Loaders

The **abstract Loader class** was introduced in **Android 3.0** (API level 11) to encapsulate the best practice technique for asynchronous data loading within UI elements, such as Activities and Fragments.

Loaders are also available within the **Android Support Library**.When creating your own Loader implementation, it is typically best practice to extend the **AsyncTaskLoader** class rather than the **Loader** class directly.

In general **custom Loaders** should:

- Asynchronously load data.
- Monitor the source of the loaded data
- and automatically provide updated results

4.5.4 Manual Thread Creation and GUI Thread Synchronization

Although using **Intent Services** and creating **AsyncTasks** are useful shortcuts, there are times when you want to create and manage your own Threads to perform background processing. This is often the case when you have **long-running or inter-related Threads** that require more subtle or complex management than is provided by the two techniques described so far.You can create and manage **child Thread**s using **Android's Handler class** and the **Threading classes** available within **java.lang.Thread**.

Simple skeleton code for moving processing onto a child Thread.

```
// This method is called on the main GUI thread.
private void backgroundExecution() {
        // This moves the time consuming operation to a child thread.
        Thread thread = new Thread(null,
        doBackgroundThreadProcessing,"Background");
        thread.start();
}
// Runnable that executes the background processing method.
private Runnable doBackgroundThreadProcessing = new Runnable() {
                public void run() {
                        backgroundThreadProcessing();
                }
};
// Method which does some processing in the background.
private void backgroundThreadProcessing() {
                // [ ... Time consuming operations ... ]
}
```

Whenever you're using **background Threads** in a GUI environment, it's important to synchronize **child Threads** with the main application **(GUI) Thread** before attempting to create or modify UI elements.Within your application components, **Notifications** and **Intents** are always received and handled on the **GUI Thread**. In all other cases, operations that explicitly interact with objects created on

the GUI Thread (such as **Views**) or that display messages (such as **Toasts**) must be invoked on the **main Thread**.

If you are running within an Activity, you can also use the **runOnUiThread** method, which lets you force a method to execute on the same Thread as the Activity UI, as shown in the following code snippet:

```
runOnUiThread(new Runnable() {
        public void run() {
        // Update a View or other Activity UI element.
        }
});
```

You can also use the **Handler class** to post methods onto the Thread in which the Handler was created.Using the **Handler class**, you can post updates to the user interface from a background Thread using the Post method.The outline for using the Handler to update the GUI Thread.

```
private void backgroundExecution() {
    Thread thread = new Thread(null, doBackgroundThreadProcessing, "Background");
    thread.start();
}
private Runnable doBackgroundThreadProcessing = new Runnable() {
        public void run() {
            backgroundThreadProcessing();
        }
};
private void backgroundThreadProcessing() {
        // [ ... Time consuming operations ... ]
        handler.post(doUpdateGUI);
}
private Handler handler = new Handler();
private Runnable doUpdateGUI = new Runnable() {
            public void run() {
            updateGUI();
        }
};
private void updateGUI() {
// [ ... Open a dialog or modify a GUI element ... ]
}
```

The **Handler class** also enables you to delay posts or execute them at a specific time, using the **postDelayed** and **postAtTime** methods, respectively:

```
// Post a method on the UI thread after 1sec.
handler.postDelayed(doUpdateGUI, 1000);
// Post a method on the UI thread after the device has been in
// use for 5mins.
int upTime = 1000*60*5;
handler.postAtTime(doUpdateGUI, SystemClock.
        uptimeMillis() + upTime);
```

4.6 USING ALARMS

Alarms are a means of firing Intents at predetermined times or intervals. Unlike **Timers**, Alarms operate outside the scope of your application, so you can use them to trigger application events or actions even after your application has been closed. Alarms are particularly powerful when used in combination with **Broadcast Receivers**, enabling you to set Alarms that fire **broadcast Intents**, start **Services**, or even open **Activities**, without your application need to be open or running.

Alarms are an effective means to reducing your application's resource requirements, by enabling you to stop Services and eliminate timers while maintaining the ability to perform scheduled actions. You can use Alarms to schedule regular updates based on network lookups, to schedule time-consuming or cost-bound operations at "off-peak" times, or to schedule retries for failed operations.Alarm operations are handled through the **AlarmManager**, a system Service accessed via **getSystemService**, as follows:

```
AlarmManager  alarmManager = (AlarmManager)
                getSystemService(Context.ALARM_SERVICE);
```

4.6.1 Creating, Setting, and Canceling Alarms

To create a new one-shot Alarm, use the set method and specify an alarm type, a trigger time, and a **Pending Intent** to fire when the Alarm triggers. If the trigger time you specify for the Alarm occurs in the past, the Alarm will be triggered immediately.

The following four alarm types are available:

1. **RTC_WAKEUP** — Wakes the device from sleep to fire the Pending Intent at the clock time specified.
2. **RTC** — Fires the Pending Intent at the time specified but does not wake the device.
3. **ELAPSED_REALTIME** — Fires the Pending Intent based on the amount of time elapsed since the device was booted but does not wake the device. The elapsed time includes any period of time the device was asleep.

4. **ELAPSED_REALTIME_WAKEUP —** Wakes the device from sleep and fires the Pending Intent after a specified length of time has passed since device boot.

❖ **Creating a waking Alarm that triggers in 10 seconds**

```
// Get a reference to the Alarm Manager
AlarmManager alarmManager = (AlarmManager) getSystemService (Context.ALARM_SERVICE);
// Set the alarm to wake the device if sleeping.
int alarmType = AlarmManager.ELAPSED_REALTIME_WAKEUP;
// Trigger the device in 10 seconds.
long timeOrLengthofWait = 10000;
// Create a Pending Intent that will broadcast and action
String ALARM_ACTION = "ALARM_ACTION";
Intent intentToFire = new Intent(ALARM_ACTION);
PendingIntent alarmIntent = PendingIntent.getBroadcast(this, 0,  intentToFire, 0);
// Set the alarm
alarmManager.set(alarmType, timeOrLengthofWait, alarmIntent);
```

Setting a second Alarm using the same **Pending Intent** replaces the preexisting Alarm.To cancel an Alarm, call cancel on the **Alarm Manager**, passing in the **Pending Intent** you no longer want to trigger,

alarmManager.cancel(alarmIntent);

4.6.2 Setting Repeating Alarms

Repeating alarms work in the same way as the one-shot alarms but will trigger repeatedly at the specified interval.Because alarms are set outside your Application lifecycle, they are perfect for scheduling regular updates or data lookups so that they don't require a Service to be constantly running in the background.To set a repeating alarm, use the **setRepeating** or **setInexactRepeating** method on the **Alarm Manager**. Both methods support an alarm type, an initial trigger time, and a Pending Intent to fire when the alarm triggers. The interval value passed in to this method lets you specify an exact interval for your alarm,down to the millisecond.

The **setInexactRepeating** method helps to reduce the battery drain associated with waking the device on a regular schedule to perform updates.At run time Android will synchronize multiple inexact repeating alarms and trigger them simultaneously.Rather than specifying an exact interval, the **setInexactRepeating** method accepts one of the following **Alarm Manager** constants:

INTERVAL_FIFTEEN_MINUTES

INTERVAL_HALF_HOUR

INTERVAL_HOUR

INTERVAL_HALF_DAY

INTERVAL_DAY

Using an inexact repeating alarm, prevents each application from separately waking the device in a similar but nonoverlapping period. By synchronizing these alarms, the system is able to limit the impact of regularly repeating events on battery resources.

```
// Get a reference to the Alarm Manager
AlarmManager alarmManager = (AlarmManager) getSystemService
(Context.ALARM_SERVICE);
// Set the alarm to wake the device if sleeping.
int alarmType = AlarmManager.
        ELAPSED_REALTIME_WAKEUP;
// Schedule the alarm to repeat every half hour.
long timeOrLengthofWait = AlarmManager
        .INTERVAL_HALF_HOUR;
// Create a Pending Intent that will broadcast and action
String ALARM_ACTION = "ALARM_ACTION";
Intent intentToFire = new Intent(ALARM_ACTION);
PendingIntent alarmIntent = PendingIntent. getBroadcast (this,
0,intentToFire, 0);
// Wake up the device to fire an alarm in half an hour, and every half-hour after that.
alarmManager.setInexactRepeating(alarmType,
        timeOrLengthofWait, timeOrLengthofWait, alarmIntent);
```

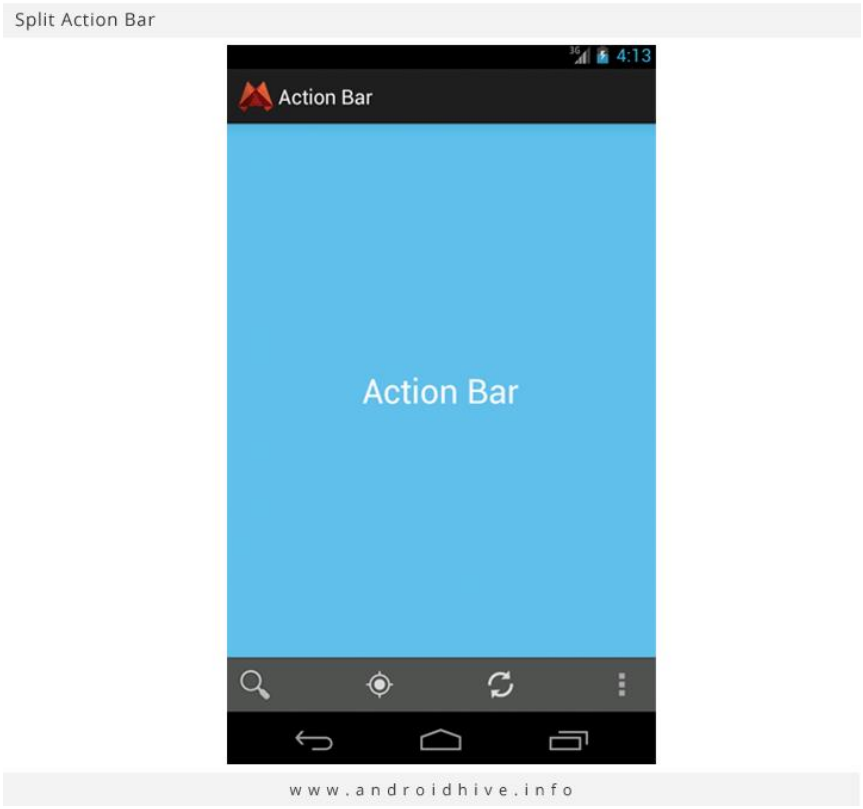
UNIT- V

5.1ACTION BAR

The **Action Bar** component was introduced in Android 3.0 (API level 11).It's a navigation panel that replaces the title bar at the top of every Activity or within a particular application's Activities and that formalizes a common Android design pattern.It's possible to hide the Action Bar, but best practice is to keep it and customize it to suit the style and navigation requirements of your application.

Action Bar provides a framework for presenting branding, navigation, and surfacing the key actions to be performed within an Activity consistently across applications.The Action Bar is enabled by default in any Activity that uses the (default) **Theme.Holo** theme and whose application has a target (or minimum) SDK version of 11 or higher. To enable the Action Bar by setting the target SDK to Android 4.0.3 (API level 15) and not modifying the default theme.

<uses-sdk android:targetSdkVersion="15" />



To toggle the visibility of the Action Bar at run time, you can use its show and hide methods:

```
ActionBar actionBar = getActionBar();
// Hide the Action Bar
actionBar.hide();
```

// Show the Action Bar

actionBar.show();

Alternatively, you can apply a theme that doesn't include the Action Bar, such as the **Theme.Holo .NoActionBar** theme

<activityandroid:name=".MyNonActionBarActivity"
android:theme="@android:style/Theme.Holo.NoActionBar">

You can create or customize your own theme that removes the Action Bar by setting the **android:windowActionBar** style property to false:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="NoActionBar" parent="@style/ActivityTheme">
        <item name="android:windowActionBar">false</item>
    </style>
</resources>
```

When you apply a theme that excludes the Action Bar from an Activity, you can't programmatically display it at run time. A call to **getActionBar** will return null**.**

5.1.1Customizing the Action Bar

One of the primary purposes of the Action Bar is to provide a consistent UI between applications.As such, the customization options are purposefully limited, though you can customize the **Action Bar** to provide your own application branding and identity. You can control your branding by specifying the image to appear (if any) at the far left, the application title to display, and the background Drawable to use.

❖ Modifying the Icon and Title Text

By default, the Action Bar displays the Drawable you specify using your application or Activity's **android:icon** attribute, alongside the corresponding **android:label** attribute on a black background.You can specify an alternative graphic using the **android:logo** attribute. Unlike the square icon, there is no limit to the width of the logo graphic — though it's good practice to limit it to approximately double the width of the icon image.The logo image typically is used to provide the top-level branding for your application, so it's good practice to hide the title label when using a logo image.

ActionBar actionBar = getActionBar();

actionBar.setDisplayShowTitleEnabled(false);

Where both an icon and logo image are supplied, you can switch between them at run time by using the setDisplayUseLogoEnabled method:

actionBar.setDisplayUseLogoEnabled(displayLogo);

If you choose to hide the icon and logo, you can do so by setting the **setDisplayShowHomeEnabled** method to false:

actionBar.setDisplayShowHomeEnabled(false);

You also can use the icon and title text to provide navigation and context cues.

actionBar.setSubtitle("Inbox");

actionBar.setTitle("Label:important");

❖ Customizing the Background

The default background color of the Action Bar depends on the underlying theme. The native Android Action Bar background is transparent, with the **Holo theme** background set to black. You can specify any Drawable as the background image for your Action Bar by using the **setBackgroundDrawable** method.

```
ActionBar actionBar = getActionBar();
Resources r = getResources();
Drawable myDrawable = r.getDrawable(R.drawable.gradient_header);
actionBar.setBackgroundDrawable(myDrawable);
```

Under normal circumstances the Action Bar will reserve space at the top of your Activity, with your layout being inflated into the remaining space. Alternatively, you can choose to overlay the Action Bar above your Activity layout by requesting the FEATURE_ACTION_BAR_OVERLAY window feature.

```
@Override
public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);
    getWindow().requestFeature(Window.FEATURE_ACTION_BAR_OVERLAY);
    setContentView(R.layout.main);
}
```

❖ Enabling the Split Action Bar Mode

SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

MOBILE APPLICATIONS USING ANDROID

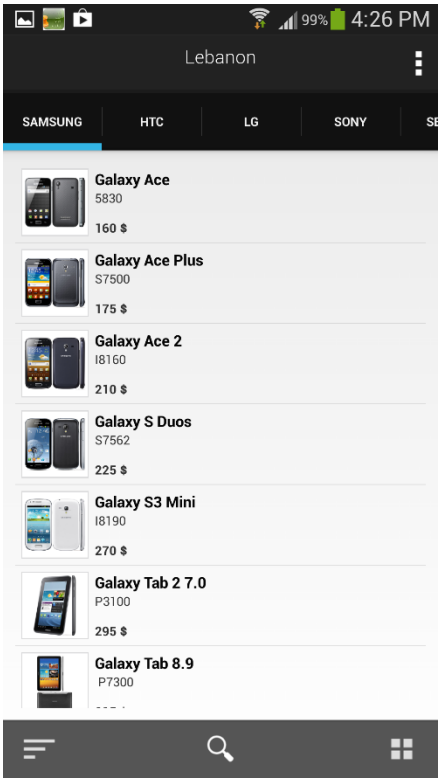
Android 4.0 (API level 14) sought to optimize many of the features initially designed for tablets for use on smaller, smartphone devices. You can enable the split Action Bar by setting the **android:uiOptions** attribute within your application or Activity manifest nodes to **splitActionBarWhenNarrow**

<activity android:label="My Activity"

android:name=".ActionBarActivity"

android:logo="@drawable/ic_launcher"
android:uiOptions="splitActionBarWhenNarrow">

On supported devices with narrow screens (such as a smartphone in portrait mode), enabling the split Action Bar mode will allow the system to split the Action Bar into separate sections.The layout is calculated and performed by the run time and may change depending on the orientation of the host device and any Action Bar configuration changes you make at run time.



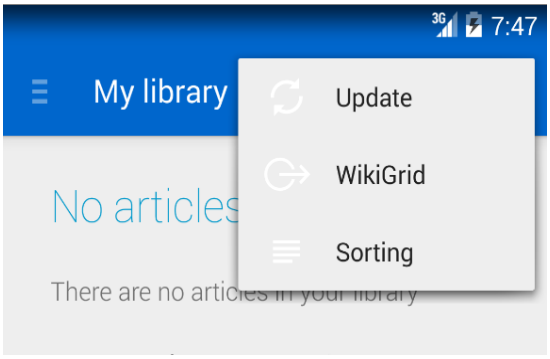
5.1.2Customizing the Action Bar to Control Application Navigation Behavior

The Action Bar introduces a several options for providing consistent and predictable navigation within your application, they are.

- **Application icons** — The application icon or logo is used to provide a consistent navigation path, typically by resetting the application to its home Activity. You can also configure the icon to represent moving "up" one level of context.
- **Tabs and drop-downs** — The Action Bar supports built-in tabs or drop-down lists that can be used to replace the visible Fragments within an Activity.

Icon navigation can be considered a way to navigate the Activity stack, whereas tabs and dropdowns are used for Fragment transitions within an Activity. In practice, the actions you perform when the application icon is clicked or a tab is changed will depend on the way you've implemented your UI.

Selecting the application icon should change the overall context of your UI in the same way that an Activity switch might do, whereas changing a tab or selecting a drop-down should change the data being displayed



❖ Configuring Action Bar Icon Navigation Behavior

In most cases, the application icon should act as a shortcut to return to the "home" Activity, typically the root of your Activity stack. To make the application icon clickable, you must call the Action Bar's **setHomeButtonEnabled** method:

actionBar.setHomeButtonEnabled(true);

Clicking the application icon/logo is broadcast by the system as a special Menu Item click. Menu Item selections are handled within the **onOptionsItemSelected** handler of your Activity, with the ID of the Menu Item parameter set to **android.R.id.home**.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
                case (android.R.id.home) :
                        Intent intent = new Intent(this, ActionBarActivity.class);
                        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
                        startActivity(intent);
                        return true;
                default:
                        return super.onOptionsItemSelected(item);
        }
 }
```

Traditionally, Android applications have started a new Activity to transition between different contexts. In turn, pressing the back button closes the active Activity and returns to the previous one. To supplement this behavior, you can configure the application icon to offer "up" navigation.To enable up navigation for your application icon, call the Action Bar's s**etDisplayHomeAsUpEnabled** method:

```
actionBar.setDisplayUseLogoEnabled(false);

actionBar.setDisplayHomeAsUpEnabled(true);
```

❖ Using Navigation Tabs

In addition to the application icon navigation, the Action Bar also offers navigation tabs and dropdown lists. These navigation options are designed to work closely with Fragments, providing a mechanism for altering the context of the current Activity by replacing the visible Fragments.

To configure your Action Bar to display tabs, call its **setNavigationMode** method, specifying **ActionBar.NAVIGATION_MODE_TABS**. The tabs should provide the application context, so it's good practice to disable the title text as well.

```
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
actionBar.setDisplayShowTitleEnabled(false);
```

Tabs are added to the Action Bar using its addTab method. Start by creating a new Tab and using its **setText** and **setIcon** methods to determine the title and image to be displayed, respectively.Alternatively, you can use the **setCustomView** method to replace the standard text and image layout with your own custom View.

```
Tab tabOne = actionBar.newTab();

tabOne.setText("First Tab") .setIcon(R.drawable.ic_launcher)
        .setContentDescription("Tab the First").setTabListener (new
        TabListener<MyFragment> (this, R.id.fragmentContainer, MyFragment.class));

actionBar.addTab(tabOne);
```

The tab switching is handled using a **TabListener**, allowing you to create **Fragment Transactions** in response to tabs being selected, unselected, and reselected.Note that you are not required to execute the Fragment Transaction created in each handler — the Action Bar will execute it for you when necessary.

The basic workflow used within this **Tab Listener** is to instantiate, configure, and then add a new Fragment to your layout within the **onTabSelected** handler. The Fragment associated with the unselected tab should be detached from your layout and recycled if its tab is reselected.

❖ Using Drop-Down Lists for Navigation

To configure your Action Bar to display a drop-down list, call its **setNavigationMode** method, specifying **ActionBar.NAVIGATION_MODE_LIST**:

```
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
```

Populate the drop-down list by creating a new Adapter that implements the **SpinnerAdapter** interface, such as an **Array Adapter** or **Simple Cursor Adapter:**

```
ArrayList<CharSequence> al = new ArrayList<CharSequence>();
```

```
al.add("Item 1");
al.add("Item 2");
ArrayAdapter<CharSequence> dropDownAdapter = new ArrayAdapter<CharSequence>(this,
                                    android.R.layout.simple_list_item_1,  al);
```

To assign the Adapter to your Action Bar, and handle selections, call the Action Bar's **setListNavigationCallbacks**, passing in your adapter and an **OnNavigationListener**

```
// Select the drop-down navigation mode.
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
// Create a new Spinner Adapter that contains the values to be displayed in the drop down.
ArrayAdapter dropDownAdapter =
ArrayAdapter.createFromResource(this,R.array.my_dropdown_values,android.
                                    R.layout.simple_list_item_1);
// Assign the callbacks to handle drop-down selections.
actionBar.setListNavigationCallbacks(dropDownAdapter,  new OnNavigationListener() {
    public boolean onNavigationItemSelected(int itemPosition, long itemId) {
     // TODO Modify your UI based on the position
     // of the drop down item selected.   return true;
}
});
```

When a user selects an item from the drop-down list, the **onNavigationItemSelected** handler will be triggered.Use the **itemPosition** and **itemId** parameters to determine how your UI should be adapted based on the new selection.

❖ Using Custom Navigation Views

For situations where neither tabs nor drop-down lists are appropriate, the Action Bar allows you to add your own custom View (including layouts) using the **setCustomView** method:

```
actionBar.setDisplayShowCustomEnabled(true);
actionBar.setCustomView(R.layout.my_custom_navigation);
```

Custom Views will appear in the same place as the tabs or drop-down lists — to the right of the application icon but to the left of any actions. To ensure consistency between applications, it's generally good form to use the standard navigation modes.

5.1.3 Introducing Action Bar Actions

The right side of the Action Bar is used to display "actions" and the associated overflow menu.

Action items should be menu options that are either most frequently used, most important for users to discover, or most expected based on the actions available in similar applications. Generic and seldom used options, such as settings, help, or "about this app" options, should never be presented as action items.

Action Bar actions and the overflow menu were introduced, along with the Action Bar itself, in Android 3.0 (API level 11) as an alternative to the hardware menu button and its associated options menu. As such, they are populated using the same APIs that were previously used to create and manage the options menu.

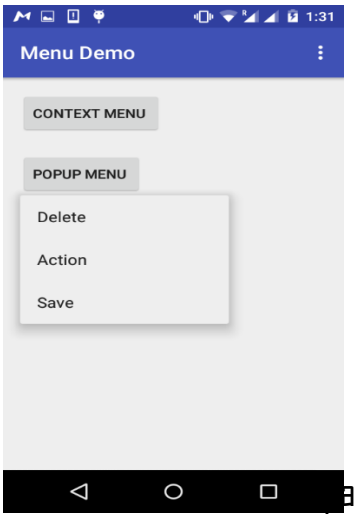
Action items should be menu options that are either most frequently used, most important for users to discover, or most expected based on the actions available in similar applications. Generic and seldom used options, such as settings, help, or "about this app" options, should never be presented as action items. Generally speaking, action items should be global actions that don't depend on the current context. Navigating within an Activity — for example, changing tabs or selecting an item from a navigation drop-down — should not alter the available action items.

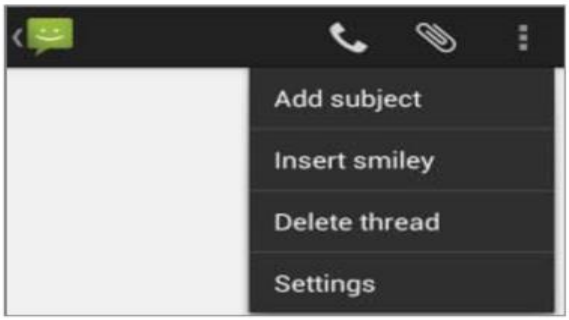
5.2 MENU

5.2.1 CREATING AND USING MENUS AND ACTION BAR ACTION ITEMS

Menus offer a way to expose application functions without sacrificing valuable screen space.Each Activity can specify its own menu that's displayed when the device's hardware menu button is pressed.In Android 3.0 (API level 11) the hardware menu button was made optional, and the Activity menu was deprecated. To replace them, Action Bar actions and the overflow menu were introduced.

Android also supports **Context Menus** and **Popup Menus** that can be assigned to any View. Context Menus are normally triggered when a user holds the middle D-pad button, depresses the trackball, or long-presses the touch screen for approximately 3 seconds when the View has focus.





❖ Introducing the Android Menu System

To improve the usability of application menus, Android uses a three-stage menu system optimized for small screens

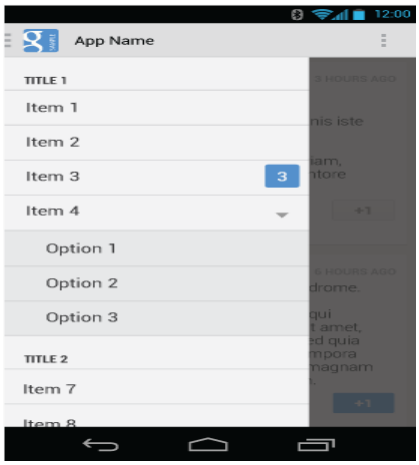
- **The icon menu and Action Bar actions** — The icon menu is a compact display that appears along the bottom of the screen when the menu button is pressed on Android devices earlier than Android 3.0.

It displays the icons and text for a limited number of Menu Items (typically six), selected based on the order in which they were added to the menu.Menu Items in the icon menu and Action Bar do not display check boxes, radio buttons, or shortcut keys, so it's generally good practice not to depend on check boxes or radio buttons in icon Menu Items or actions because they will not be visible.

- **The expanded menu and overflow menu** — Prior to Android 3.0, the expanded menu was triggered when a user selected the More Menu Item from the icon menu. The **expanded menu** displays a scrollable list of only the Menu Items that weren't visible in the icon menu.

The extended and overflow menus display the full Menu Item text, along with any associated check boxes and radio buttons, They do not display icons. Pressing back from the expanded menu returns you to the icon menu.

- **Submenus** — The Android alternative is to display each submenu in a floating window. Because Android does not support nested submenus, you can't add a submenu to a submenu nor can you specify a submenu to be an action.



❖ Creating a Menu

To add a menu to an Activity, override its **onCreateOptionsMenu** handler, it receives a Menu object as a parameter. You can store a reference to, and continue to use, the Menu reference elsewhere in your code until the next time **onCreateOptionsMenu** is called. You should always call through to the superclass implementation because it can add additional system menu options where appropriate. Use the **add method** on the Menu object to populate your menu. For each new Menu Item, you must specify the following:

A **group value** to separate Menu Items for batch processing and ordering.A **unique identifier** for each Menu Item. For efficiency reasons, Menu Item selections normally are handled using the **onOptionsItemSelected** event handler, so this unique identifier makes it possible for you to determine which Menu Item was selected. It is conventional to declare each **menu ID** as a private static variable within the Activity class. You can use the **Menu.FIRST** static constant and simply increment that value for each subsequent item.

An **order value** that defines the order in which the Menu Items are displayed. The **Menu Item display text**, either as a character string or as a string resource.When you have finished populating the Menu, **return true** to indicate that you have handled the Menu creation.

static final private int MENU_ITEM = Menu.FIRST;

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
int groupId = 0;
int menuItemId = MENU_ITEM;
int menuItemOrder = Menu.NONE;
  int menuItemText = R.string.menu_item;
MenuItem menuItem = menu.add(groupId, menuItemId,menuItemOrder,menuItemText);
  return true;
 }
```

Like the Menu object, each **MenuItem** returned by an add call is valid until the next call to **onCreateOptionsMenu.** Rather than maintaining a reference to each item, you can find a particular Menu Item by passing its ID into the Menu's **findItem** method:

```
MenuItem menuItem = menu.findItem(MENU_ITEM);
```

❖ Specifying Action Bar Actions

To specify a Menu Item as an Action Bar action, use its **setShowAsActionFlags** method, passing in one of the following options:

- **SHOW_AS_ACTION** — Forces the Menu Item to always be displayed as an action.
- **SHOW_AS_IF_SPACE** — Specifies that the Menu Item should be available as an action provided there is enough space in the Action Bar to display it.

It is good practice to always use this option to provide the system with maximum flexibility on how to layout the available actions.By default, actions display only their associated icon. You can optionally OR either of the preceding flags with **SHOW_AS_ACTION_WITH_TEXT** to make the Menu Item's text visible as well.

```
menuItem.setShowAsAction(MenuItem.SHOW_AS_ACTION_IF_ROOM.
 MenuItem. SHOW_AS_ACTION_WITH_TEXT);
```

❖ Menu Item Options

Android supports

- **Check boxes** — Check boxes on Menu Items are visible in the overflow and expanded menus, as well as within submenus. To set a Menu Item as a check box, use the **setCheckable** method. The state of that check box is controlled via **setChecked**.

```
menu.add(0, CHECKBOX_ITEM, Menu.NONE, "CheckBox").setCheckable(true);
```

- **Radio buttons** — A radio button group is a group of items displaying circular buttons, in which only one item can be selected at any given time, assign the same group identifier to each item and then call **Menu.setGroupCheckable**, passing in that group identifier and setting the exclusive parameter to true:

```
menu.add(RB_GROUP, RADIOBUTTON_1, Menu.NONE, "Radiobutton 1");
    menu.add(RB_GROUP, RADIOBUTTON_2, Menu.NONE, "Radiobutton 2");
menu.add(RB_GROUP, RADIOBUTTON_3, Menu.NONE,"Radiobutton 3"). setChecked(true);
    menu.setGroupCheckable(RB_GROUP, true, true);
```

- **Shortcut keys** — You can specify a keyboard shortcut for a Menu Item by using the **setShortcut** method. Each call to setShortcut requires two shortcut keys — one for use with the numeric keypad and another to support a full keyboard. Neither key is case-sensitive.

```
// Add a shortcut to this Menu Item, '0' if using the numeric keypad
        // or 'b' if using the full keyboard.
        menuItem.setShortcut('0', 'b');
```

- **Condensed titles —** The **setTitleCondensed** method lets you specify text to be displayed only in the icon menu or as Action Bar actions. The normal title will only be used when the Menu Item is displayed in the overflow or extended menu.

```
menuItem.setTitleCondensed("Short Title");
```

- **Icons** — Icons are displayed only in the icon menu or as an action; they are not visible in the extended menu or submenus. You can specify any Drawable resource as a menu icon.

```
menuItem.setIcon(R.drawable.menu_item_icon);
```

- **Menu item click listener** —Menu Item selections should be handled by the **onOptionsItemSelected**handler.

```
menuItem.setOnMenuItemClickListener(newOnMenuItemClickListener() {
        public boolean onMenuItemClick(MenuItem _menuItem) {
                [ ... execute click handling, return true if handled ... ]
                return true;
}
});
```

- Intents — An Intent assigned to a Menu Item is triggered when a Menu Item click isn't handled by a **MenuItemClickListener** or the Activity's **onOptionsItemSelected** handler. When the Intent is triggered, Android will execute **startActivity**, passing in the specified Intent.

```
menuItem.setIntent(new Intent(this, MyOtherActivity.class));
```

❖ Adding Action Views and Action Providers

The introduction of actions and the **Action Bar** in Android 3.0 (API level 11) made it possible to add richer interaction modes to the menu system. In particular, it's now possible to add interactive Views to your Action Bar actions in the form of action **Views** and **Action Providers**.

You can replace the icon/text used to represent an action with any **View** or layout using the Menu Item's **setActionView** method, passing in either a View instance or a layout resource.

```
menuItem.setActionView(R.layout.my_action_view)  .setShowAsActionFlags
(MenuItem.SHOW_AS_ACTION_IF_ROOM|MenuItem.
SHOW_AS_ACTION_COLLAPSE_ACTION_VIEW);
```

Once added, the action View will be displayed whenever the associated Menu Item is displayed as an action in the Action Bar, but will never be displayed if the Menu Item is relegated to the overflow menu.A better alternative, introduced in Android 4.0 (API level 14), is to add the **MenuItem.SHOW_AS_ ACTION_COLLAPSE_ACTION_VIEW flag**. When this flag is set, the Menu Item will be represented using its standard icon and/or text properties until it's pressed.

In either case, it's up to you to wire the View's interaction handlers. Typically, this is done within the **onCreateMenuOptions** handler:

```
View myView = menuItem.getActionView();
Button button = (Button)myView.findViewById(R.id.goButton);
button.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
  // TODO React to the button press.
}
});
```

Android 4.0 (API level 14) introduced a new alternative to manually creating and configuring action Views within each Activity.Instead, you can create Action Providers by extending the **ActionProvider** class.Action Providers are similar to action Views but encapsulate both the appearance and interaction models associated with the View.For example, Android 4.0 includes the **ShareActionProvider** to encapsulate the "share" action. To assign an Action Provider to a Menu Item, use the **setActionProvider** method, assigning it an Intent to use to perform the sharing action.

❖ Adding Menu Items from Fragments.

With most of your UI encapsulated within Fragments, it makes sense to encapsulate any related Activity Menu Items and Action Bar actions within those Fragments.To register your Fragment as a contributor to the options Menu, call **setHasOptionsMenu** within its **onCreate** hander:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
 setHasOptionsMenu(true);
}
```

You can then override the **onCreateOptionsMenu** handler. Defining Menu Hierarchies in XMLRather than constructing your Menus in code, it's best practice to define your Menu hierarchies as XML resources.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<item    android:id="@+id/action_item"
          android:icon="@drawable/action_item_icon"
          android:title="@string/action_item_title"
          android:showAsAction="ifRoom">
</item>
<item    android:id="@+id/action_view_item"
          android:icon="@drawable/action_view_icon"
          android:title="@string/action_view_title"
          android:showAsAction="ifRoom|collapseActionView"
         android:actionLayout="@layout/my_action_view">
</item>
<item    android:id="@+id/action_provider_item"
          android:title="Share"
          android:showAsAction="always"
    android:actionProviderClass="
          android.widget.ShareActionProvider">
</item>

<item    android:id="@+id/item02"
           android:checkable="true"
```

```
            android:title="@string/menu_item_two">
    </item>
    <item   android:id="@+id/item03"
    android:numericShortcut="3"
    android:alphabeticShortcut="3"
    android:title="@string/menu_item_three">
    </item>
    <item   android:id="@+id/item04"
            android:title="@string/submenu_title">
    <menu>
    <item       android:id="@+id/item05"
        android:title="@string/submenu_item">
            </item>
    </menu>
    </item>
    </menu>
```

To use your Menu resource, use the **MenuInflater class** within your **onCreateOptionsMenu** or **onCreateContextMenu** event handlers.

```
public boolean onCreateOptionsMenu(Menu menu) {        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater();   inflater.inflate(R.menu.my_menu, menu);
                return true;
}
```

❖ Updating Menu Items Dynamically

By overriding your Activity's **onPrepareOptionsMenu** method, you can modify a Menu based on an application's current state immediately before the Menu is displayed. This lets you dynamically disable/enable Menu Items, set visibility, and modify text.Note that the **onPrepareOptionsMenu** method is triggered whenever the menu button is clicked, the overflow menu displayed, or the Action Bar is created.

To modify Menu Items dynamically, you can either record a reference to them from within the **onCreateOptionsMenu** method when they're created, or you can use the **findItem** method on the Menu object, here **onPrepareOptionsMenu** is overridden.

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {      super.onPrepareOptionsMenu(menu);
MenuItem menuItem = menu.findItem(MENU_ITEM);
        [ ... modify Menu Items ... ]
                return true;
}
```

❖ Handling Menu Selections

Android handles the Action Bar actions, overflow menu, and Activity menu selections using a single event handler, **onOptionsItemSelected**.The Menu Item selected is passed in to this method as the **MenuItem** parameter. To react to the menu selection, compare the **item.getItemId** value to the Menu Item identifiers you used when populating the Menu and perform the corresponding action.

```
public boolean onOptionsItemSelected(MenuItem item) {    super.onOptionsItemSelected(item);
        switch (item.getItemId()) {
                case (MENU_ITEM):
                        [ ... Perform menu handler actions ... ]
                        return true;
                default: return false;
               }
}
```

If you have supplied Menu Items from within a Fragment, you can choose to handle them within the **onOptionsItemSelected** handler of either the Activity or the Fragment.

❖ Introducing Submenus and Context Menus

Submenus are displayed as regular Menu Items that, when selected, reveal more items.Traditionally, submenus are displayed in a hierarchical tree layout. Rather than a tree structure, selecting a submenu presents either a single floating window (in the case of the legacy Menu system) or replaces the overflow menu (in the case of Android 3.0 and above), both of which display all its Menu Items. You can add submenus by using the **addSubMenu** method.

It supports the same parameters as the **add method** used to add normal Menu Items, enabling you to specify a group, unique identifier, and text string for each submenu. You can also use the **setHeaderIcon** and **setIcon** methods to specify an icon to display in the submenu's header bar or icon menu, respectively. The Menu Items within a submenu support the same options as those assigned to the icon or extended menus, with the exception of the Action Bar action-related properties.

```
SubMenu sub = menu.addSubMenu(0, 0, Menu.NONE, "Submenu");
sub.setHeaderIcon(R.drawable.icon);

sub.setIcon(R.drawable.icon);

MenuItem submenuItem = sub.add(0, 0, Menu.NONE, "Submenu Item");
```

➢ Using Context Menus and Popup Menus

Context Menus are contextualized by the currently focused View and are triggered when a user long-presses the trackball, middle D-pad button, or a View (typically for approximately 3 seconds). A Context Menu is displayed as a floating window above your Activity.Android 3.0 (API level 11) introduced the **PopupMenu** class, a lighter-weight alternative to the **ContextMenu** that anchors itself to a specific View.

You define and populate Context Menus and Popup Menus as you define and populate Activity Menus.There are two options available for creating Context Menus for a particular View.

➢ Creating Context Menus

One option is to create a generic **ContextMenu** object for a View class by overriding a View's **onCreateContextMenu** handler, as shown here:

```
@Override public void onCreateContextMenu(ContextMenu menu) {
super.onCreateContextMenu(menu);  menu.add("ContextMenuItem1");
 }
```

The **Context Menu** created here will be available within any Activity that includes this View class. The more common alternative is to create Activity-specific Context Menus by overriding the Activity's **onCreateContextMenu** method, and registering the Views that should use it using the **registerForContextMenu** method,

Once a View has been registered, the **onCreateContextMenu** handler will be triggered the fi rst time a **Context Menu** is displayed for that View. To populate the Context Menu parameter with the appropriate Menu Items, override **onCreate ContextMenu** and check which View has triggered the menu creation.

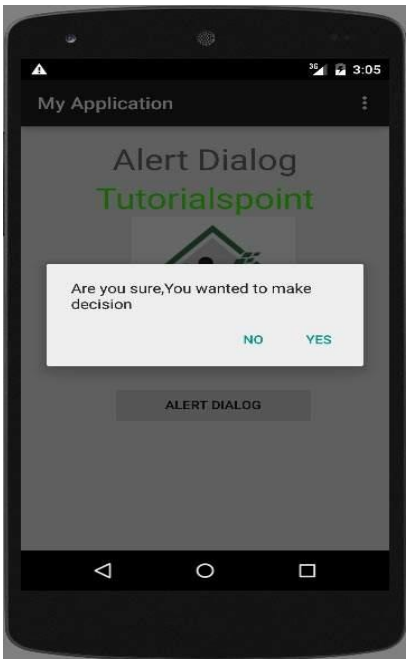
```
@Override public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {

    super.onCreateContextMenu(menu, v, menuInfo);

    menu.setHeaderTitle("Context Menu");

    menu.add(0, Menu.FIRST, Menu.NONE, "Item 1").setIcon(R.drawable.menu_item);

    menu.add(0, Menu.FIRST+1, Menu.NONE, "Item 2").setCheckable(true);

    menu.add(0, Menu.FIRST+2, Menu.NONE, "Item 3").setShortcut('3', '3');

    SubMenu sub = menu.addSubMenu("Submenu");

    sub.add("Submenu Item");
```

}

As shown in the preceding code, the ContextMenu class supports the same add method as the Menu class, so you can populate a Context Menu in the same way that you populate Activity menus using the add method. This includes using the add method to add submenus to your Context Menus. Note that icons will never be displayed; however, you can specify the title and icon to display in the Context Menu's header bar.

5.3 DIALOGS

Used to help users answer questions, make selections, and confirm actions, and to display warning or error messages.Dialogs also can be used to represent system-level events, such as displaying errors or supporting account selection. It is good practice to limit the use of Dialogs within your applications, and when using them to limit the degree of customization.



There are three ways to implement a dialog in Android:

1. Using the Dialog class (or its extensions) — In addition to the general-purpose AlertDialog class, Android includes a number of classes that extend Dialog. Each is designed to provide specific dialog-box functionality.

A Dialog class-based screen is constructed and controlled entirely within its calling Activity, so it doesn't need to be registered in the manifest.

2. Dialog-themed Activities — You can apply the dialog theme to a regular Activity to give it the appearance of a standard dialog box.

3. Toasts — Toasts are special nonmodal transient message boxes, often used by Broadcast Receivers and Services to notify users of events occurring in the background.

5.3.1 Introducing Dialogs

❖ Creating a Dialog

To create a new Dialog, instantiate a new Dialog instance and set the title and layout, using the **setTitle** and **setContentView** methods, respectively. The **setContentView** method accepts a resource identifier for a layout that will be inflated to display the Dialog's UI.Once the Dialog is configured to your liking, use the show method to display it.

```
Dialog dialog = new Dialog(MyActivity.this);
dialog.setTitle("Dialog Title");
dialog.setContentView(R.layout.dialog_view);
TextView text = (TextView)dialog.findViewById(R.id.dialog_text_view);
text.setText("This is the text in my dialog");
dialog.show();
```

Using the Alert Dialog Class

The **AlertDialog** class is one of the most versatile Dialog-class implementations.It offers a number of options that let you construct dialogs for some of the most common use cases, including the following:

- Presenting a message to users offering them one to three options in the form of buttons, usually a combination of OK, Cancel, Yes, and No.
- Offering a list of options in the form of check boxes or radio buttons.
- Providing a text entry box for user input.

To construct the Alert Dialog UI, create a new **AlertDialog.Builder** object:

```
AlertDialog.Builder  ad = new AlertDialog.Builder(context);
```

You can then assign values for the title and message to display, and optionally assign values to be used for any buttons, selection items, and text input boxes you want to display.Use the **setCancelable** method to determine if the user should be able to close the dialog by pressing the back button without making a selection.

If you choose to make the Dialog cancelable, you can use the **setOnCancelListener** method to attach an On Cancel Listener to react to this event:

```
ad.setCancelable(true);
ad.setOnCancelListener(new DialogInterface.OnCancelListener() {
public void onCancel(DialogInterface dialog) {
        eatenByGrue();
        }
    }
);
```

❖ Using Specialized Input Dialogs

Android includes several specialized dialog boxes that encapsulate controls designed to facilitate common user-input requests. They include the following:

- **CharacterPickerDialog** — Lets users select an accented character based on a regular character source.
- **DatePickerDialog** — Lets users select a date from a **DatePicker View.** The constructor includes a callback listener to alert your calling Activity when the date has been set.
- **TimePickerDialog** — Similar to the Date Picker Dialog, this dialog lets users select a time from a **TimePicker View**.
- **ProgressDialog** — Displays a progress bar beneath a message text box.

In each case, to use the specialist Dialog, construct a new instance of it, setting its properties and event handlers, before displaying the Dialog:

```
DatePickerDialog datePickerDialog =new DatePickerDialog(
MyActivity.this,new OnDateSetListener() {
public void onDateSet(DatePicker view, int year, int monthOfYear, int dayOfMonth) {
        // TODO Use the selected date.
    }
},1978, 6, 19);
datePickerDialog.show();
```

❖ Managing and Displaying Dialogs Using Dialog Fragments

You can use the **show** method of each Dialog instance to display it, but a better alternative is to use Dialog Fragments. A Dialog Fragment is a Fragment that contains a Dialog.Dialog Fragments are included as part of the Android Support Package, making it possible to use them for projects targeting all Android platforms down to Android 1.6 (API level 4).To use a Dialog Fragment, extend the **DialogFragment** class, Override the **onCreateDialog** handler to return a Dialog constructed.You can display the **Dialog Fragment** using the **Fragment Manager** and **Fragment Transactions** the same way as you would any other Fragment.

```
String tag = "my_dialog";
DialogFragment  myFragment = MyDialogFragment.newInstance(dateString);
myFragment.show(getFragmentManager(), tag);
```

Alternatively, you can override the **onCreateView** handler to inflate a custom Dialog layout within your Dialog Fragment, just as you would your custom Dialog class.

```
@Override
```

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,Bundle savedInstanceState) {
// Inflate the Dialog's UI.
View view = inflater.inflate(R.layout.dialog_view, container, false);
// Update the Dialog's contents.
TextView text = (TextView)view.findViewById(R.id.dialog_text_view);
text.setText("This is the text in my dialog");
return view;
}
```

Note that in must circumstances you can gain the same level of control over the appearance and lifecycle of your Dialog by using a Dialog Fragment.The easiest way to make an Activity look like a Dialog is to apply the **android:style/Theme**.Dialog theme when you add the Activity to your manifest, as shown in the following XML snippet:

```
<activity android:name="MyDialogActivity"
android:theme="@android:style/Theme.Dialog">
</activity>
```

This will cause your Activity to behave as a Dialog, floating on top of, and partially obscuring, the Activity beneath it.

5.3.2 LET'S MAKE A TOAST

They provide an ideal mechanism for alerting users to events occurring in background Services without interrupting foreground applications.The Toast class includes a static **makeText** method that creates a standard Toast display window.To construct a new Toast, pass the current Context, the text message to display, and the length of time to display it (LENGTH_SHORT or LENGTH_LONG) into the **makeText** method. After creating a Toast, you can display it by calling **show**

```
Context context = this;
String msg = "To health and happiness!";
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context, msg, duration);
toast.show();
```

❖ Customizing Toasts

You can modify a Toast by setting its display position and assigning it alternative Views or layouts.

Context context = this;

String msg = "To the bride and groom!";

int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, msg, duration);

int offsetX = 0;

int offsetY = 0;

toast.setGravity(Gravity.BOTTOM, offsetX, offsetY);

toast.show();

Using **setView** on a Toast object, you can specify any View (including a layout) to display using the Toast mechanism.For example the following code will assigns a layout, containing the **CompassView** Widget.

Context context = getApplicationContext();

String msg = "Cheers!";

int duration = Toast.LENGTH_LONG;

Toast toast = Toast.makeText(context, msg, duration);

toast.setGravity(Gravity.TOP, 0, 0);

LinearLayout ll = new LinearLayout(context);

```
ll.setOrientation(LinearLayout.VERTICAL);
TextView myTextView = new TextView(context);
CompassView cv = new CompassView(context);
myTextView.setText(msg);
int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;
ll.addView(cv, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.setPadding(40, 50, 0, 50);
toast.setView(ll);
toast.show();
```

❖ Using Toasts in Worker Threads

As GUI components, Toasts must be created and shown on the GUI thread; otherwise, you risk throwing a cross-thread exception.A Handler is used to ensure that the Toast is opened on the GUI thread.

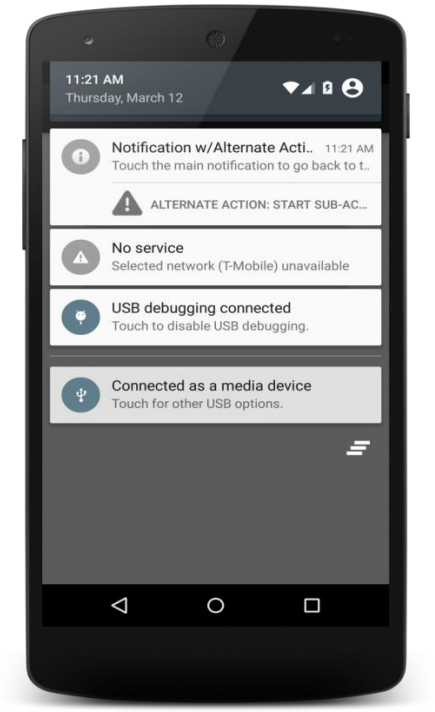
```
Handler handler = new Handler();
private void mainProcessing() {
    Thread thread = new Thread(null, doBackgroundThreadProcessing,"Background");
    thread.start();
}
private Runnable doBackgroundThreadProcessing = new Runnable() {
    public void run() {
    backgroundThreadProcessing();
}
};
```

5.4. INTRODUCING NOTIFICATIONS

Your application can use Notifications to alert users of events that may require their attention without one of its Activity's being visible.Notifications are handled by the **Notification Manager** and currently have the ability to

- Display a status bar icon
- Flash the lights/LEDs
- Vibrate the phone

- Sound audible alerts (ringtones, Media Store audio)
- Display additional information within the notification tray
- Broadcast Intents using interactive controls from within the notification tray



Notifications are the preferred mechanism for invisible application components (Broadcast Receivers, Services, and inactive Activities) to alert users that events have occurred that may require attention. They are also used to indicate ongoing background Services — and are required to indicate a Service that has foreground priority. It's likely that your users will have their phones with them at all times but quite unlikely that they will be paying attention to them, or your application, at any given time.

Generally, users will have several applications open in the background, and they won't be paying attention to any of them.Notifications can be persisted through insistent repetition, being marked ongoing, or simply by displaying an icon on the status bar. Status bar icons can be updated regularly or expanded to show additional information using the expanded notification tray,

5.4.1 Introducing the Notification Manager

The **NotificationManager** is a system Service used to manage Notifications. Get a reference to it using the **getSystemService** method,

```
String svcName = Context.NOTIFICATION_SERVICE;
NotificationManager notificationManager;
notificationManager = (NotificationManager)getSystemService(svcName);
```

Using the Notification Manager, you can trigger new Notifications, modify existing ones, or cancel those that are no longer required.

5.4.2 Creating Notifications

Android offers a number of ways to convey information to users using Notifications:

- Status bar icon
- Sounds, lights, and vibrations
- Details displayed within the extended notification tray

❖ **Creating a Notification and Configuring the Status Bar Display**

Start by creating a new Notification object, passing in an icon to display on the status bar along with the ticker text to display on the status bar when the Notification is triggered

```
int icon = R.drawable.icon;
String tickerText = "Notification";
long when = System.currentTimeMillis();
Notification notification = new Notification(icon, tickerText, when);
```

The ticker text should be a short summary that describes what you are notifying the user of (for example, an SMS message or email subject line).You also need to specify the timestamp of the Notification; the Notification Manager will sort Notifications in this order.You can also set the Notification object's number property to display the number of events a status bar icon represents.Setting this value to a number greater than 1, as shown in the following line of code, overlays the values as a small number over the status bar icon:

```
notification.number++;
```

Using the Default Notification Sounds, Lights, and Vibrations. The simplest and most consistent way to add sounds, lights, and vibrations to your Notifications is to use the default settings. Using the defaults property, you can combine the following constants:

- Notification.DEFAULT_LIGHTS
- Notification.DEFAULT_SOUND
- Notification.DEFAULT_VIBRATE

For example, the following code snippet assigns the default sound and vibration settings to a Notification:

```
notification.defaults = Notification.DEFAULT_SOUND |
```

Notification.DEFAULT_VIBRATE;

If you want to use all the default values, you can use the **Notification.DEFAULT_ALL** constant.

- **Making Sounds**

Most native phone events, from incoming calls to new messages and low battery, are announced by audible ringtones.Android lets you assign any available audio file to signal a Notification. Assign a new sound to a Notification using its sound property, specifying a URI to the audio fi le, as shown in the following snippet:

Uri ringURI = RingtoneManager. getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);

notification.sound = ringURI;

To use your own custom audio, push the file onto your device or include it as a raw resource.

- **Vibrating the Device**

You can use the device's vibrator to execute a vibration pattern specific to your Notification.Android lets you control the pattern of a vibration; you can use vibration to alert the user to new information being available, or use a specific pattern to convey the information directly.Before you can use vibration in your application, you need to request the VIBRATE uses-permission in your manifest:

<uses-permission android:name="android.permission.VIBRATE"/>

To set a vibration pattern, assign a long[] to the Notification's vibrate property. Construct the array so that values representing the length of time (in milliseconds) to vibrate alternate with values representing the length of time to pause.The following example shows how to modify a Notification to vibrate in a repeating pattern of 1 second on and 1 second off, for 5 seconds total:

long[] vibrate = new long[] { 1000, 1000, 1000, 1000, 1000 };

notification.vibrate = vibrate;

You can take advantage of this fine-grained control to pass contextual information to your users.

- **Flashing the Lights**

Notifications also include properties to configure the color and flash frequency of the device's LED.The **ledARGB** property can be used to set the LED's color, whereas the **ledOffMS** and **ledOnMS** properties let you set the frequency and pattern of the flashing LED. You can turn on the LED by setting the **ledOnMS** property to 1 and the **ledOffMS** property to 0, or turn it off by setting both properties to 0.After configuring the LED settings, you must also add the **FLAG_SHOW_LIGHTS** flag to the Notification's flags property. The following code snippet shows how to turn on the red device LED:

notification.ledARGB = Color.RED;

notification.ledOffMS = 0;

notification.ledOnMS = 1;

notification.flags = notification.flags | Notification.FLAG_SHOW_LIGHTS;

Controlling the color and fl ash frequency gives you another opportunity to pass additional information to users.

- **Using the Notification Builder**

The Notification Builder, introduced in Android 3.0 (API level 11) to simplify the process of configuring the flags, options, content, and layout of Notifications, is the preferred alternative when constructing Notifications for newer Android platforms.

```
Notification.Builder builder =new Notification.Builder(MyActivity.this);
        builder.setSmallIcon(R.drawable.ic_launcher).setTicker("Notification")
                .setWhen(System.currentTimeMillis())
                .setDefaults(Notification.DEFAULT_SOUND |
                    Notification.DEFAULT_VIBRATE)
                .setSound( RingtoneManager.getDefaultUri(
                    RingtoneManager.TYPE_NOTIFICATION))
                .setVibrate(new long[] { 1000, 1000, 1000, 1000, 1000 })
                .setLights(Color.RED, 0, 1);
    Notification notification = builder.getNotification();
```

5.4.3 Setting and Customizing the Notification Tray UI

You can configure the appearance of the Notification within the extended notification tray in a number of ways:

- Use the **setLatestEventInfo** method to update the details displayed in the standard notification tray display.
- Use the **Notification Builder** to create and control one of several alternative notification tray UIs.
- Set the **contentView** and **contentIntent** properties to assign a custom UI for the extended status display using a Remote Views object.
- From Android 3.0 (API level 11) onward, you can assign Broadcast Intents to each View within the Remote Views object that describes your custom UI to make them fully interactive.

- **Using the Standard Notification UI**

The simplest approach is to use the **setLatestEventInfo** method to specify the title and text fields used to populate the default notification tray layout.

```
notification.setLatestEventInfo(context, expandedTitle, expandedText, launchIntent);
```

The **PendingIntent** you specify will be fired if a user clicks the **Notification** item. In most cases that Intent should open your application and navigate to the Activity that provides context for the notification.The Notification Builder also provides support for displaying a progress bar within your Notification. Using the **setProgress** method, you can specify the current progress in relation to a maximum value.

```
builder.setSmallIcon(R.drawable.ic_launcher)
                    .setTicker("Notification")
                    .setWhen(System.currentTimeMillis())
                    .setContentTitle("Progress")
                    .setProgress(100, 50, false)
                    .setContentIntent(pendingIntent);
```

- **Creating a Custom Notification UI**

If the details available in the standard Notification display are insufficient (or unsuitable) for your needs. You can create your own layout and assign it to your Notification using a Remote Views object.To use this layout within a Notification, you must package it within a Remote Views object:

```
RemoteViews myView =new RemoteViews(this.getPackageName(),R.layout.
                    my_status_window_layout);
```

If you are using the Notification Builder, you can assign your custom View using the **setContent** method.

```
RemoteViews myRemoteView =new RemoteViews(this.getPackageName(),
                                        R.layout.my_notification_layout);
  builder.setSmallIcon(R.drawable.notification_icon)
                    .setTicker("Notification")
                    .setWhen(System.currentTimeMillis())
                    .setContentTitle("Progress")
                    .setProgress(100, 50, false)
                    .setContent(myRemoteView);
```

You will also need to assign a Pending Intent to the **contentIntent** property:

```
Intent intent = new Intent(this, MyActivity.class);
PendingIntent pendingIntent= PendingIntent.getActivity(this, 0, intent, 0);
notification.contentView = new RemoteViews(this.getPackageName(),
```

R.layout.my_status_window_layout);

notification.contentIntent = pendingIntent;

- **Creating a Custom Notification UI**

If the details available in the standard Notification display are insufficient (or unsuitable) for your needs, you can create your own layout and assign it to your Notification using a Remote Views object.To use the layout within a Notification, you must package it within a Remote Views object:

RemoteViews myView = new RemoteViews(this.getPackageName(), R.layout.my_status_window_layout);

Remote Views are a mechanism that enables you to embed and control a layout embedded within a separate application, most commonly when creating home screen Widgets. There are strict limits on the Views you can use when creating a layout to be used for Remote Views. If you are using the Notification Builder, you can assign your custom View using the setContent method.

RemoteViews myRemoteView =new RemoteViews (this.getPackageName(),

R.layout.my_notification_layout);

builder.setSmallIcon(R.drawable.notification_icon).setTicker("Notification")

.setWhen(System.currentTimeMillis())

.setContentTitle("Progress")

.setProgress(100, 50, false)

.setContent(myRemoteView);

You can modify the properties and appearance of the Views used in your Notification layout using the set* methods on the Remote Views object.

notification.contentView.setImageViewResource(R.id.status_icon, R.drawable.icon);

notification.contentView.setTextViewText(R.id.status_text,"Current Progress:");

notification.contentView.setProgressBar(R.id.status_progress,100, 50, false);

Android 4.0 (API level 14) introduced the ability to attach click listeners to the Views contained within your custom Notification layout.To assign a click listener to a View within your Remote Views layout, use the **setOnClickPendingIntent** method, passing in the resource id of the View to bind to, and a Pending Intent to broadcast when the View is clicked

Intent newIntent = new Intent(BUTTON_CLICK); PendingIntent newPendingIntent = PendingIntent.getBroadcast(MyActivity.this, 2,newIntent, 0);

notification.contentView.setOnClickPendingIntent(R.id.status_progress, newPendingIntent);

Using this mechanism, you can make your notifications fully interactive — effectively allowing you to embed a home screen Widget within the Notification tray

- **Customizing the Ticker View**

On some devices, particularly tablet devices, you can specify a **Remote Views** object that will be displayed within the system bar instead of the Notification ticker text.Use the **setTicker** method on the Notification Builder to specify a Remote Views object to display.

```
RemoteViews myTickerView =   new RemoteViews(this.getPackageName(),
                                                R.layout.my_ticker_layout);
builder.setSmallIcon(R.drawable.notification_icon)
                .setTicker("Notification", myTickerView)
                .setWhen(System.currentTimeMillis())
                .setContent(myRemoteView);
```

5.4.4 Configuring Ongoing and Insistent Notifications

You can configure Notifications as insistent and/or ongoing by setting the FLAG_INSISTENT and FLAG_ONGOING_EVENT flags, respectively.Notifications flagged as ongoing are used to represent events that are currently in progress.Using the Notification Builder, you can mark a Notification as ongoing using the **setOngoing** method.

```
builder.setSmallIcon(R.drawable.notification_icon).setTicker("Notification")
        .setWhen(System.currentTimeMillis())
.setContentTitle("Progress")
.setProgress(100, 50, false)
                .setContent(myRemoteView)
        .setOngoing(true);
```

If you aren't using the Notification Builder, you can apply the **Notification.FLAG_ONGOING_EVENT** flag directly to the Notification's flags property:

```
notification.flags = notification.flags|Notification.FLAG_ONGOING_EVENT;
```

An ongoing Notification is a requirement for a foreground Service.Insistent Notifications repeat their audio, vibration, and light settings continuously until canceled. These Notifications should be used only for events that require immediate and timely attention — such as an incoming call or the ringing of a user-set alarm clock.To make a Notification insistent, apply the **Notification.FLAG_INSISTENT** flag directly to the Notification's flags property:

```
notification.flags = notification.flags |  Notification.FLAG_INSISTENT;
```

Note that insistent Notifications are particularly intrusive and should seldom be used within thirdparty applications.This is reflected in the lack of a corresponding Notification Builder method for setting this flag.

5.4.5 Triggering, Updating, and Canceling Notifications

To fire a Notification, pass it in to the notify method of the **NotificationManager** along with an integer reference ID.If you've used a Notification Builder to construct the Notification, use its **getNotification** method to obtain the Notification to broadcast.

```
String svc = Context.NOTIFICATION_SERVICE;
NotificationManager notificationManager  = (NotificationManager)getSystemService(svc);
int NOTIFICATION_REF = 1; Notification notification = builder.getNotification();
notificationManager.notify(NOTIFICATION_REF, notification);
```

To update a Notification that's already been fired, including updating the UI of any attached content View, retrigger it using the Notification Manager, passing the notify method the same reference ID.You can pass in either the same Notification object or an entirely new one. As long as the ID values are the same, the new Notification will be used to replace the status icon and extended status window details.To update a Notification without triggering any of the associated lights, audio, or vibration settings, use the Notification Builder's **setOnlyAlertOnce** method.

```
builder.setSmallIcon(R.drawable.notification_icon).setTicker("Updated Notification")
        .setWhen(System.currentTimeMillis())        .setContentTitle("More Progress")
.setProgress(100, 75, false)
                                .setContent(myRemoteView)
.setOngoing(true)
.setOnlyAlertOnce(true);
Notification notification = builder.getNotification();
notificationManager.notify(NOTIFICATION_REF, notification);
```

Alternatively, you can apply the FLAG_ONLY_ALERT_ONCE fl ag directly to the Notification:

```
notification.flags = notification.flags | Notification.FLAG_ONLY_ALERT_ONCE;
```

Canceling a Notification removes its icon from the status bar and its extended details from the Notification tray. It's good practice to cancel a Notifi cation after the user has acted upon it.

```
builder.setSmallIcon(R.drawable.ic_launcher) .setTicker("Notification")
                .setWhen(System.currentTimeMillis())
```

```
.setContentTitle("Title")
.setContentText("Subtitle")                                    .setContentInfo("Info")
                                                    .setLargeIcon(myIconBitmap)
                                          .setContentIntent(pendingIntent)
.setAutoCancel(true);
```

Alternatively, apply the FLAG_AUTO_CANCEL flag when not using the Notification Builder:

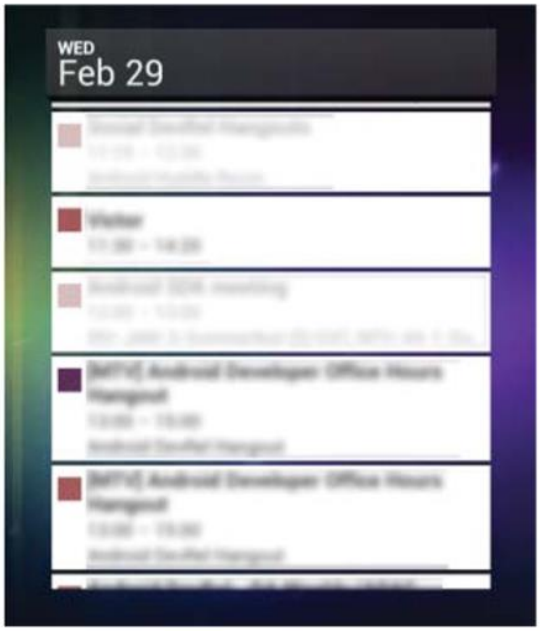
notification.flags = notification.flags | Notification.FLAG_AUTO_CANCEL;

To cancel the Notification, use the Notification Manager's cancel method, passing in the reference id of the Notification you want to cancel

notificationManager.cancel(NOTIFICATION_REF);

5.5. COLLECTION VIEW WIDGETS

Android 3.0 (API level 11) introduced Collection View Widgets, a new style of Widgets designed to display collections of data as lists, grids, or stacks.



As the name suggests, Collection View Widgets are designed to add support for collection-based Views specifically as follows:

- **StackView** — A flip-card style View that displays its child Views as a stack. The stack will automatically rotate through its collection, moving the topmost item to the back to reveal the one beneath it. Users can manually transition between items by swiping up or down to reveal the previous or next items, respectively.
- **ListView** — Each item in the bound collection is displayed as a row on a vertical list.

- **GridView** — A two-dimensional scrolling grid where each item is displayed within a cell. You can control the number of columns, their width, and relevant spacing.

Each of these controls extends the **Adapter View** class.Depending on the View used to display the collection, the specified layout will represent each row in a **list**, each card in a **stack**, or each cell in a **grid**. The UI used to represent each item is restricted to the same Views and layouts supported by App Widgets:

- FrameLayout
- LinearLayout
- RelativeLayout
- AnalogClock
- Button
- Chronometer
- ImageButton
- ImageView
- ProgressBar
- TextView
- ViewFlipper

Collection View Widgets can be used to display any collection of data, but they're particularly useful for creating dynamic Widgets that surface data held within your application's Content Providers.Collection View Widgets are implemented in much the same way as regular App Widgets — using App Widget Provider Info files to configure the Widget settings, **BroadcastReceivers** to define their behavior, and **RemoteViews** to modify the Widgets at run time.

In addition, collection-based App Widgets require the following components:

- An additional layout resource that defines the UI for each item displayed within the Widget.
- A **RemoteViewsFactory** that acts as a de facto Adapter for your Widget by supplying the Views that will be displayed within your collection View. It creates the Remote Views using the item layout definition and populates its elements using the underlying data you want to display.
- A **RemoteViewsService** that instantiates and manages the Remote Views Factory.

With these components complete, you can use the **Remote Views Factory** to create and update each of the Views that will represent the items in your collection. You can automate this process by creating a Remote View and using the **setRemoteAdapter** method to assign the Remote Views Service to it. When the **Remote View** is applied to the collection Widget, the Remote Views Service will create and update each item, as necessary.

5.5.1 Creating Collection View Widget Layouts

Collection View Widgets require two layout definitions — one that includes the **Stack**, **List**, or **Grid View**, and another that describes the layout to be used by each item within the stack, list, or grid.As with regular App Widgets, it's best practice to define your layouts as external XML layout resources

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout  xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
         android:layout_height="match_parent"
         android:padding="5dp">
<StackView    android:id="@+id/widget_stack_view"
            android:layout_width="match_parent"
            android:layout_height="match_parent"  />
</FrameLayout>
```

The Widget layout is used within the **AppWidgetProviderInfo** resource as it would be for any **App Widget**. The item layout is used by a **Remote Views Factory** to create the Views used to represent each item in the underlying collection.

5.5.2 Creating the Remote Views Service

The **Remote Views Service** is used as a wrapper that instantiates and manages a Remote Views Factory, which, in turn, is used to supply each of the Views displayed within the **Collection View Widget**. To create a Remote Views Service, extend the **RemoteViewsService** class and override the **onGetViewFactory** handler to return a new instance of a **Remote Views Factory**

```
import java.util.ArrayList;
import android.appwidget.AppWidgetManager;
import android.content.Context;
import android.content.Intent;
import android.widget.RemoteViews;
import android.widget.RemoteViewsService;
public class MyRemoteViewsService extends RemoteViewsService {
@Override
public RemoteViewsFactory onGetViewFactory(Intent intent) {
            return new MyRemoteViewsFactory(getApplicationContext(), intent);
}
}
```

As with any Service, you'll need to add your **Remote Views Service** to your application manifest using a service tag. To prevent other applications from accessing your Widgets, you must specify the **android.permission.BIND_REMOTEVIEWS** permission.

```
<service android:name=".MyRemoteViewsService"
        android:permission="android.permission.BIND_REMOTEVIEWS">
</service>
```

Creating a Remote Views FactoryThe **RemoteViewsFactory** acts as a thin wrapper around the **Adapter class**. It is where you create and populate the Views that will be displayed in the Collection View Widget — effectively binding them to the underlying data collection. To implement your **Remote Views Factory**, extend the **RemoteViewsFactory** class. This is normally done within the enclosing **Remote Views Service class**.Your implementation should mirror that of a **custom Adapter** that will populate the **Stack, List, or Grid View**. Note that the **Remote Views Factory** doesn't need to know what kind of Collection View Widget will be used to display each item.

5.5.3 Populating Collection View Widgets Using a Remote Views Service

With your **Remote Views Factory** complete, all that remains is to bind the List, Grid, or Stack View within your App Widget Layout to the **Remote Views Service**.This is done using a **Remote View**, typically within the **onUpdate** handler of your App Widget implementation. Create a new **Remote View** instance as you would when updating the UI of a standard App Widget. Use the **setRemoteAdapter** method to bind your **Remote Views Service** to the particular List, Grid, or Stack View within the Widget layout.

The **Remote View Service** is specified using an Intent of the following form:

```
Intent intent = new Intent(context, MyRemoteViewsService.class);
```

This Intent is received by the **onGetViewFactory** handler within the **Remote Views Service**, enabling you to pass additional parameters into the Service and the Factory it contains.You also specify the ID of the Widget you are binding to, allowing you to specify a different Service for different Widget instances.

The **setEmptyView** method provides a means of specifying a View that should be displayed if (and only if) the underlying data collection is empty.After completing the binding process, use the App Widget Manager's **updateAppWidget** method to apply the binding to the specified Widget

5.5.4 Adding Interactivity to the Items Within a Collection View Widget

For efficiency reasons, it's not possible to assign a unique **onClickPendingIntent** to each item displayed as part of a Collection View Widget. Instead, use the **setPendingIntentTemplate** to assign a template Intent to your Widget.

```
Intent templateIntent = new Intent(Intent.ACTION_VIEW);
templateIntent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);

PendingIntent  templatePendingIntent = PendingIntent.getActivity(  context, 0,
                                                templateIntent,
```

PendingIntent.FLAG_UPDATE_CURRENT);
views.setPendingIntentTemplate(R.id.widget_stack_view,templatePendingIntent);

This Pending Intent can then be "**filled-in**" within the **getViewAt** handlerof your **Remote Views Service** implementation using the **setOnClickFillInIntent** method of your **Remote Views object.**

// Create the item-specific fill-in Intent that will populate

// the Pending Intent template created in the App Widget Provider.

Intent fillInIntent = new Intent();

fillInIntent.putExtra(Intent.EXTRA_TEXT, myWidgetText.get(index));

rv.setOnClickFillInIntent(R.id.widget_title_text, fillInIntent);

The fill-in Intent is applied to the template Intent using the **Intent.fillIn** method. It copies the contents of the fill-in Intent into the template Intent, replacing any undefined fields with those defined by the fill-in Intent. Fields with existing data will not be overridden.One of the most powerful uses of Collection View Widgets is to surface data from your Content Providers to the home screen.

5.5.5 Refreshing Your Collection View Widgets

The **App Widget Manager** includes the **notifyAppWidgetViewDataChanged** method, which allows you to specify a **Widget ID** (or array of IDs) to update, along with the resource identifier for the collection View within that Widget whose underlying data source has changed:

appWidgetManager.notifyAppWidgetViewDataChanged(appWidgetIds,R.id.widget_stack_view);

This will cause the **onDataSetChanged** handler within the associated **Remote Views Factory** to be executed, followed by the meta-data calls, including **getCount**, before each of the Views is re-created. Alternatively, the techniques used to update App Widgets — altering the minimum refresh rate, using Intents, and setting Alarms — can also be used to update Collection View Widgets; however, they will cause the entire Widget to be re-created, meaning that refreshing the collection-based Views based on changes to the underlying data is more efficient.

5.6 LIVE FOLDERS

Live Folders provided a similar functionality for earlier versions of Android — a means by which your application can expose data from its **Content Providers** directly on to the home screen. Although **Collection View Widgets** are the supported alternative for devices running Android 3.0 or above, you can still use **Live Folders** to provide dynamic home screen shortcuts to information stored in your application for devices running earlier version of Android. When added, a **Live Folder** is represented on the home screen as a shortcut icon. Selecting the icon will open the Live Folder.You could also think of it as a folder of shortcuts into your application.

For example, a music application might allow the user to create **live folders** for favorite music. Similarly, a to-do list application might include support for a live folder of the day's tasks. Finally, a game might have a live folder for saved game points.

When the user chooses to create a LiveFolder, the Android system provides a list of all activities that respond to the ACTION_CREATE_LIVE_FOLDER Intent. If the user chooses your Activity, that Activity creates the LiveFolder and passes it back to the system using the setResult() method.

The LiveFolder consists of the following components:

- Folder name
- Folder icon
- Display mode (grid or list)
- Content provider URI for the folder contents

The first task when enabling a content provider to serve up data to a LiveFolder is to provide an <intent-filter> for an Activity that handles enabling the LiveFolder.This is done within the AndroidManifest.xml file as follows:

```
<intent-filter>
<action android:name=
"android.intent.action.CREATE_LIVE_FOLDER" />
<category
android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

5.6.1 Creating Live Folders

Live Folders are a combination of a Content Provider and an Activity. To create a new Live Folder, you need to define the following:

1. **An Activity** responsible for creating and configuring the Live Folder by generating and returning a specially formatted Intent
2. **A Content Provider** that provides the items to be displayed using the required column names Unlike Collection View Widgets, each Live Folder item can display up to only three pieces of information: an icon, a title, and a description.

❖ The Live Folder Content Provider

Any Content Provider can provide the data displayed within a Live Folder. Live Folders use a standard set of column names:

1. **LiveFolders._ID** — A unique identifier used to indicate which item was selected if a user clicks the Live Folder list.

2. **LiveFolders.NAME** — The primary text, displayed in a large font. This is the only required column.

3. **LiveFolders.DESCRIPTION** — A longer descriptive field in a smaller font, displayed beneath the name column.

4. **LiveFolders.ICON_BITMAP** — An image bitmap to be displayed at the left of each item.

Alternatively, you can use a combination of **LiveFolders.ICON_PACKAGE** and **LiveFolder.ICON_RESOURCE** to specify a Drawable resource to use from a particular package. Rather than renaming the columns within your **Content Provider** to suit the requirements of Live Folders, you should apply a **projection** that maps your existing column names to those required by a **Live Folder**

```
private static final HashMap<String, String> LIVE_FOLDER_PROJECTION;

static {

        // Create the projection map.

LIVE_FOLDER_PROJECTION = new HashMap<String, String>();

// Map existing column names to those required by a Live Folder.
LIVE_FOLDER_PROJECTION.put(LiveFolders._ID,KEY_ID + " AS " + LiveFolders._ID);
LIVE_FOLDER_PROJECTION.put(LiveFolders.NAME,KEY_NAME+" AS "+
                                                    LiveFolders.NAME);

LIVE_FOLDER_PROJECTION.put(LiveFolders.DESCRIPTION, KEY_DESCRIPTION + " AS " +
                                                    LiveFolders.DESCRIPTION);

LIVE_FOLDER_PROJECTION.put(LiveFolders.ICON_BITMAP,KEY_IMAGE + " AS " +
                                    LiveFolders.ICON_BITMAP);

}
```

Only the ID and name columns are required; the bitmap and description columns can be used or left unmapped, as required. The projection typically will be applied within the query method of your **Content Provider** when the query request URI matches the pattern you specify for Live Folder request

```
public static Uri LIVE_FOLDER_URI   = Uri.parse("com.paad.provider.MyLiveFolder");
public Cursor query(Uri uri, String[] projection, String
                        selection,String[],selectionArgs, String sortOrder) {
SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
   switch (URI_MATCHER.match(uri)) {
            case LIVE_FOLDER:
                        qb.setTables(MYTABLE);
qb.setProjectionMap(LIVE_FOLDER_PROJECTION);
                        break;
default:
                        throw new IllegalArgumentException("Unknown URI " + uri);
                      }
  Cursor c = qb.query(null, projection, selection, selectionArgs,null, null, null);
  c.setNotificationUri(getContext().getContentResolver(), uri);
  return c;
}
```

❖ The Live Folder Activity

The **Live Folder** is defined using an Intent returned as the result of an Activity (typically from within the **onCreate** handler). Use the Intent's **setData** method to specify the URI of the Content Provider supplying the data (with the appropriate projection applied). You can configure the Intent further by using a series of extras as follows:

1. **LiveFolders.EXTRA_LIVE_FOLDER_DISPLAY_MODE** — Specifies the display mode to use. This can be either LiveFolders.DISPLAY_MODE_LIST or LiveFolders.DISPLAY_MODE_ GRID to display your Live Folder as a list or grid, respectively.

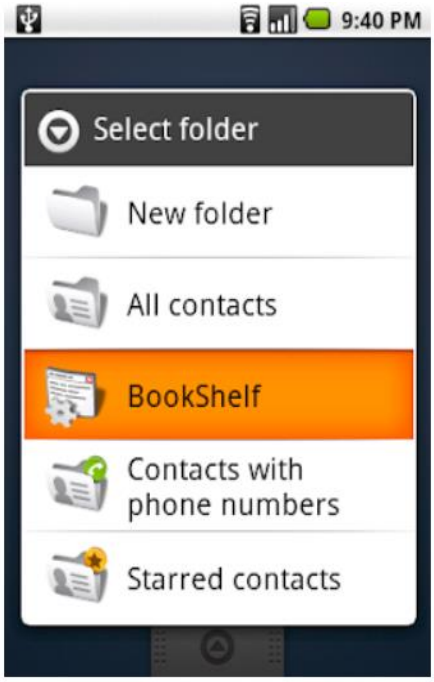
2.**LiveFolders.EXTRA_LIVE_FOLDER_ICON** — Provides a Drawable resource that will be used as the home screen icon that represents the Live Folder when it hasn't been opened.

3. **LiveFolders.EXTRA_LIVE_FOLDER_NAME** — Provides a descriptive name to use with the icon described above to represent the Live Folder on the home screen when it hasn't been opened.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
screen  String action = getIntent().getAction();
if (LiveFolders.ACTION_CREATE_LIVE_FOLDER.equals(action)) {
                        Intent intent = new Intent();
                        intent.setData(MyContentProvider.LIVE_FOLDER_URI);
                        intent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_DISPLAY_MODE,

                                                        LiveFolders.DISPLAY_MODE_LIST);
intent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_ICON,Intent.ShortcutIconResource.
                                                fromContext(this,R.drawable.icon));
intent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_NAME, "Earthquakes");
                        setResult(RESULT_OK, intent);
                    } else
                        setResult(RESULT_CANCELED);
                        finish();
}
```

By adding a **LiveFolders.EXTRA_LIVE_FOLDER_BASE_INTENT** extra to the returned Intent, you can specify a base Intent to fire when a Live Folder item is selected. When this value is set, selecting a Live Folder item will result in **startActivity** being called with the specified base Intent used as the Intent parameter. Best practice is to set the data parameter of this Intent to the base URI of the Content Provider that's supplying the Live Folder's data. In such cases the Live Folder will automatically append the value stored in the selected item's _id column to the Intent's data value.In order for the system to identify an Activity as a Live Folder, you must include an Intent Filter for the **CREATE_LIVE_FOLDER** action when adding the Live Folder Activity to your application manifest

```
<activity android:name=".MyLiveFolder"
android:label="My Live Folder">
    <intent-filter>
        <action android:name="android.intent.action.CREATE_LIVE_FOLDER"/>
    </intent-filter>
</activity>
```



5.7 QUICK SEARCH BOX

The QSB is positioned prominently on the home screen. The user can launch it at any time by clicking it or pressing the hardware search key.Android 1.6 (API level 4) introduced the ability to serve your application search results through the universal QSB. By surfacing search results from your application through this mechanism, you provide users with an additional access point to your application through live search results.

- **Local vs. global search**

There are essentially two kinds of search in Android:

- local search and
- global search

Local search enables users to search content of the currently visible app. Local search is appropriate for nearly every type of app. A recipe app could offer users to search for words in the title of the recipes or within the list of ingredients. Local search is strictly limited to the app offering it and it's results are not visible outside of the app.

Global search on the other hand makes the content also accessible from within the Quick Search Box on the home screen. Android uses multiple data source for global search and your app can be one of it. In the next

screen taken on a tablet you can see the results of a Google search on the left and some results from apps or for app titles on the right.

The user experience for local and global search is a bit different since global search uses multiple data sources. By default it provides search results from Google, searches for installed apps or contacts on the device and it also might include results from apps that enable global search. The following screenshot of a phone shows all content that is included in the list of search results.

❖ Enabling search for your app

You always need to execute at least three steps to make your app searchable. If you want to provide search suggestions you have to add a fourth step:

1. You have to write an xml file with the search configuration
2. You have to write a search activity
3. You have to tie the former two together using the AndroidManifest.xml
4. You have to add a content provider for search suggestions - if needed

I am going to cover the first three steps in the following sections. And I am going to write about the content provider for search suggestions in the next part of this tutorial.

❖ Configuring your search

You have to tell Android how you want your app to interact with Android's search capabilities. This is done in an xml file. Following established practices you should name the file searchable.xml and store it in the res/xml folder of your app.

The least you have to configure is the label for the search box - even though the label is only relevant for global search. You also should always add a hint as Android displays this text within the form field. You can use this to guide the user.

❖ Adding a SearchActivity

Whenever a users performs a search, Android calls an activity to process the search query. The intent used to call the activity contains the query string as an intent extra and uses the value of the static field Intent.ANDROID_SEARCH. Since you configure which activity to use for search (see next section), Android calls your activity with an explicit intent.

To make search work you have to configure it within your project's AndroidManifest.xml file. These are the things you have to put in there:

- Your search activity
- The intent used for search
- The launch mode for your activity
- The meta-data pointing to your searchable.xml
- Further meta-data to define which activity to use for search.

5.8 Surfacing Search Results to the Quick Search Box

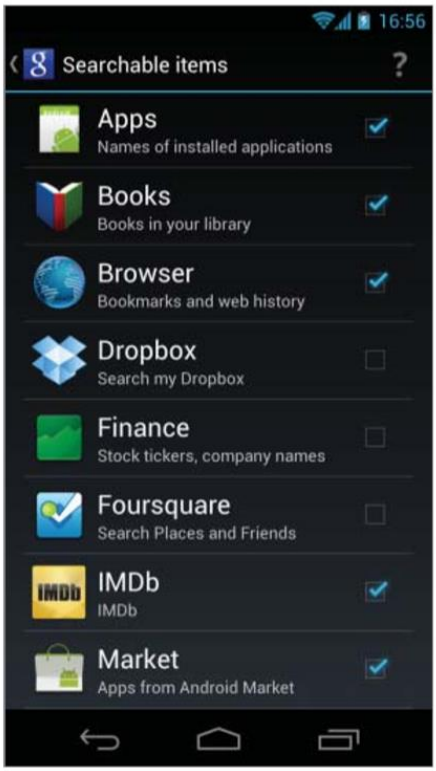
To serve your search results to the QSB, you must first implement search functionality within your application. To make your results available globally, modify the **searchable.xml** file that describes the application search **meta data** and add two new attributes:

1. **searchSettingsDescription** — Used to describe your search results in the Settings menu. This is what the users will see when browsing to include application results in their searches.
2. **includeInGlobalSearch** — Set this to true to surface these results to the QSB.

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
android:label="@string/search_label"

android:searchSuggestAuthority="com.paad.provider.mysearch"
android:searchSuggestIntentAction="android.intent.action.VIEW"
android:searchSettingsDescription="@string/search_description"
android:includeInGlobalSearch="true">

</searchable>
```

To avoid the possibility of misuse, adding new search providers requires users to opt-in, so your search results will not be automatically surfaced directly to the QSB. For users to add your application's search

results to their QSB search, they must opt-in using the system settings. From the QSB Activity, they must select Menu Í Settings Í Searchable Items and tick the check boxes alongside each Provider they want to enable.



5.9. CREATING LIVE WALLPAPER

Live Wallpaper was introduced in Android 2.1 (API level 7) as a way to create dynamic, interactive home screen backgrounds. They offer an exciting alternative for displaying information to your users directly on their home screen. Live Wallpapers use a Surface View to render a dynamic display that can be interacted with in real time. Your Live Wallpaper can listen for, and reach to, screen touch events — letting users engage directly with the background of their home screen.

Where to Locate Live Wallpapers on Your Mobile Device

On your Android device screen, you can do a long press on the empty space, and a list of icons will be displayed at the bottom. For example, we have "Wallpaper," "Widgets," "Settings," and so on. Yours could look slightly different due to a variety of Android platforms and devices in the market. Mine is running Oreo OS on a Nexus 6P. "Wallpapers" is our target interest. Once you click that icon, you will be presented with a list of static wallpapers, followed by live wallpapers ready for use on your device. Figure 1 shows the result on my Android phone.

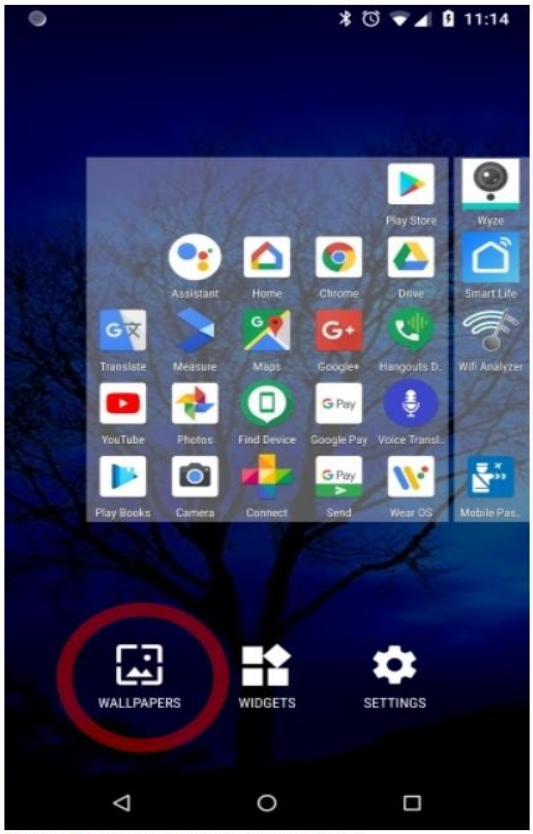


Figure 1: Live Wallpapers on the Device

To create a new Live Wallpaper, you need the following three components:

1. An **XML resource** that describes the metadata associated with the Live Wallpaper — specifically its author, description, and a thumbnail used to represent it from the Live Wallpaper picker.
2. A **Wallpaper Service** implementation that will wrap, instantiate, and manage your Wallpaper Engine.
3. A **Wallpaper Service Engine** implementation (returned through the Wallpaper Service) that defines the UI and interactive behavior of your Live Wallpaper. The Wallpaper Service Engine represents where the bulk of your Live Wallpaper implementation will live.

5.9.1 Creating a Live Wallpaper Definition Resource

The Live Wallpaper resource definition is an XML file stored in the **res/xml** folder. Its resource identifier is its filename without the XML extension. Use attributes within a wallpaper tag to define the author name, description, and thumbnail to display in the Live Wallpaper gallery.

<wallpaper xmlns:android=”http://schemas.android.com/apk/res/android”
android:author=”@string/author”

android:description=”@string/description”

android:thumbnail="@drawable/wallpapericon" />

Note that you must use references to existing string resources for the author and description attribute values. String literals are not valid. You can also use the **settingsActivity** tag to specify an Activity that should be launched to configure the Live Wallpaper's settings, much like the configuration Activity used to configure Widget settings:

```
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"
    android:author="@string/author"
android:description="@string/description"
android:thumbnail="@drawable/wallpapericon"
android:settingsActivity="com.paad.mylivewallpaper.WallpaperSettings" />
```

This Activity will be launched immediately before the Live Wallpaper is added to the home screen, allowing the user to configure the Wallpaper.

5.9.2 Creating a Wallpaper Service

Extend the **WallpaperService** class to create a wrapper Service that instantiates and manages the **Wallpaper Service Engine** class. All the drawing and interaction for Live Wallpaper is handled in the **Wallpaper Service Engine** class. Override the **onCreateEngine** handler to return a new instance of your custom **Wallpaper Service Engine**,

```
import android.service.wallpaper.WallpaperService;
import android.service.wallpaper.WallpaperService.Engine;
public class MyWallpaperService extends WallpaperService {
        @Override
public Engine onCreateEngine() {
            return new MyWallpaperServiceEngine();
        }
 }
```

After creating the Wallpaper Service, add it to your application manifest using a **service tag**. A Wallpaper Service must include an **Intent Filter** to listen for the **android.service.wallpaper .WallpaperService** action and a meta-data node that specifies **android.service.wallpaper** as the name attribute and associates it with the resource file described in the previous section using a resource attribute.An application that includes a Wallpaper Service must also require the **android.permission.BIND_ WALLPAPER** permission.

```
<application  android:icon="@drawable/icon"
   android:label="@string/app_name"
android.permission="android.permission.BIND_WALLPAPER">
```

```
<service android:name=".MyWallpaperService">
        <intent-filter>
            <action android:name="android.service.wallpaper.WallpaperService" />
        </intent-filter>
        <meta-data  android:name="android.service.wallpaper"
                    android:resource="@xml/mylivewallpaper"  />
</service>
</application>
```

5.9.3 Creating a Wallpaper Service Engine

The **WallpaperService.Engine** class is where you define the behavior of the Live Wallpaper. The **Wallpaper Service Engine** includes the **Surface View** onto which you will draw your Live Wallpaper and handlers notifying you of touch events and home screen offset changes and is where you should implement your redraw loop. The **Surface View**,is a specialized drawing canvas that supports updates from background threads, making it ideal for creating smooth, dynamic, and interactive graphics.

To implement your own **Wallpaper Service engine**, extend the **WallpaperService.Engine** class within an enclosing **Wallpaper Service** class. You must wait for the Surface to complete its initialization — indicated by the **onSurfaceCreated** handler being called — before you can begin drawing on it. After the Surface has been created, you can begin the drawing loop that updates the Live Wallpaper's UI, is done by scheduling a new frame to be drawn at the completion of the drawing of the previous frame. The rate of redraws in this example is determined by the desired frame rate. You can use the **onTouchEvent** and the **onOffsetsChanged** handlers to add interactivity to your Live Wallpapers