B.E. [COMPUTER SCIENCE AND ENGINEERING]

VI SEMESTER

08PE603 – Professional Elective – III

[MOBILE APP DEVELOPMENT]

UNIT - I

Android: An Open Platform for Mobile Development - Native Android Applications - Android SDK features - Understanding the Android Software Stack - The Dalvik Virtual Machine - Android Application Architecture - Android Libraries - Creating the Android Application - Types of Android Applications - Android Development Tools - Externalizing the Resources - The Android Application Lifecycle.

Introduction:

Android is a software package and Linux based operating system for mobile devices such as tablet computers and smartphones. It is developed by Google and later the OHA (Open Handset Alliance). Java language is mainly used to write the android code even though other languages can be used. The goal of android project is to create a successful real-world product that improves the mobile experience for end users. Android has expanded beyond a pure mobile phone platform to provide a development platform for an increasingly wide range of hardware, including tablets and televisions. 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 applications created for it

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

- A Compatibility Definition Document (CDD) and Compatibility Test Suite (CTS) that describe 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, memory management, and process control, all optimized for mobile and embedded devices.
- Open-source libraries for application development, including SQLite, WebKit, OpenGL, and a media manager.
- A run time used to execute and host Android applications, including the Dalvik Virtual Machine (VM) and the core libraries that provide Android-specific functionality. The run time 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. **<u>Native Android Applications:</u>**

Android devices typically come with a suite of preinstalled applications that form part of the 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 also available to third-party applications. Similarly, your applications can respond to events such as incoming calls. The exact makeup of the applications available on new Android phones is likely to vary based on the hardware manufacturer and/or the phone carrier or distributor.

The open-source nature of Android means that carriers and OEMs can customize the user interface and the applications bundled with each Android device. It's important to note that for compatible devices, the underlying platform and SDK remain consistent across OEM and carrier variations. The look and feel of the user interface may vary, but your applications will function in the same way across all compatible Android devices.

2. **<u>Android SDK features:</u>**

As an application neutral platform, Android gives you the opportunity to create applications that are as much a part of the phone as anything provided out-of-the-box. The following list highlights some of the most noteworthy Android features:

- GSM, EDGE, 3G, 4G, and LTE networks for telephony or data transfer, enabling you to make 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 location detection
- 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 and microphone
- Media libraries for playing and recording a variety of audio/video or still-image formats and APIs for using sensor hardware, including accelerometers, compasses, and barometers
- Libraries for using Bluetooth and NFC hardware for peer-to-peer data transfer and IPC message passing
- Shared data stores and APIs for contacts, social networking, calendar, and multimedia
- 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 library and 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 the replacement of native applications

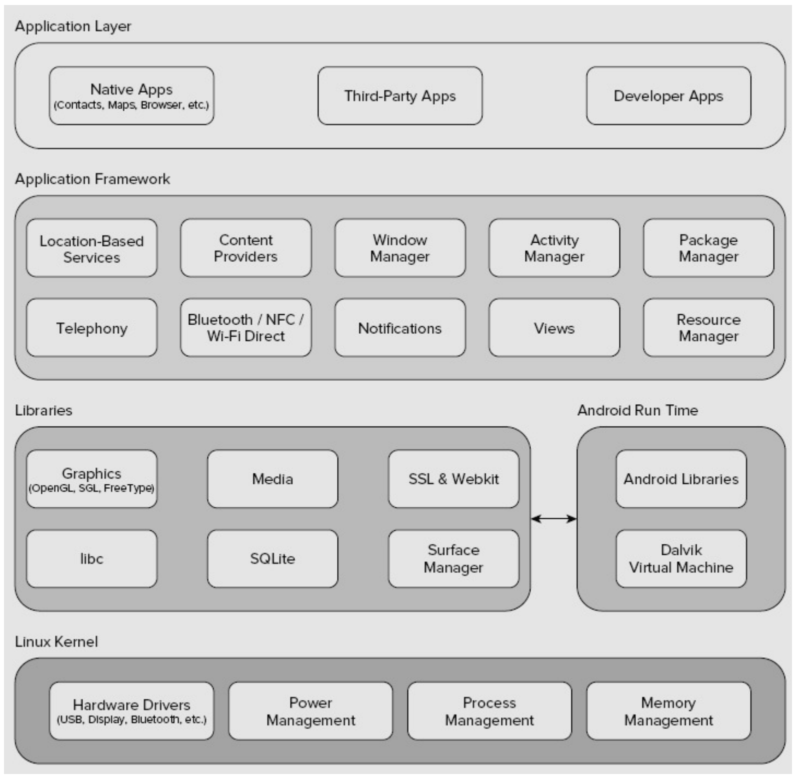
3. <u>Understanding the Android Software Stack:</u>

The Android software stack is, put simply, a Linux kernel and a collection of C/C++ libraries exposed through an application framework that provides services for, and management of, the run time and applications. The Android software stack is composed of the elements:

- **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 kernel also 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 libraries such 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 rather than a mobile Linux implementation. Including the core libraries and the Dalvik VM, the Android 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 using the Java language, Dalvik is not a Java VM. The core Android libraries provide most of the functionality available in the core Java libraries, as well as the Android-specific libraries.
  - **Dalvik VM —** Dalvik is a register-based Virtual Machine that's been optimized to ensure that a device can run multiple instances efficiently. It relies on the Linux kernel for threading and low-level memory management.
- **Application framework —** The application framework provides the classes used to create Android applications. It also provides a generic abstraction for hardware access and manages the user interface and application resources.
- **Application layer —** All applications, both native and third-party, are built on the application layer by means of the same API libraries. The application layer runs within the Android run time, using the classes and services made available from the application framework.



4. <u>The Dalvik Virtual Machine:</u>

One of the key elements of Android is the Dalvik VM. Rather than using a traditional Java VM such as 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, including security, 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 cases there's no reason you should need to.

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

All Android hardware and system service access is managed using Dalvik as a middle tier. By using a VM to host application execution, developers have an abstraction layer that ensures they should never have to worry about a particular hardware implementation.

The Dalvik VM executes Dalvik executable files, a format optimized to ensure minimal memory footprint. You create .dex executables by transforming Java language compiled classes using the tools supplied within the SDK.

5. <u>Android Application Architecture:</u>

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 dialer can let you expose your application's components in order to let other developers build on them by creating 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 and Fragments, 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 signaling your users.
- **Content Providers —** Lets your applications share data.
- **Resource Manager —** Enables non-code resources, such as strings and graphics, to be externalized.
- **Intents —** Provides a mechanism for transferring data between applications and their components.

6. <u>Android Libraries:</u>

Android offers a number of APIs for developing your applications. Android is intended to target a wide range of mobile hardware, so be aware that the suitability and implementation of some of the advanced or optional APIs may vary depending on the host device.

https://developer.android.com/reference/packages

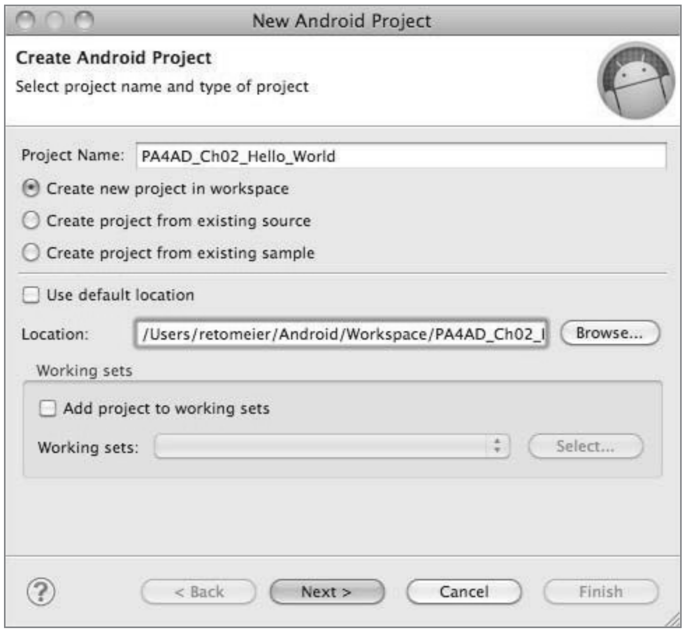
7. <u>Creating the Android Application:</u>

You've downloaded the SDK, installed Eclipse, and plugged in the plug-in. You are now ready to start programming for Android. Start by creating a new Android project and setting up your Eclipse *run* and *debug* configurations, as described in the following sections.

Creating a New Android Project:

To create a new Android project using the Android New Project Wizard, do the following:

1. Select File Í New Í Project.
2. Select the Android Project application type from the Android folder, and click Next.
3. In the wizard that appears, enter the details for your new project. On the first page (figure), the Project Name is the name of your project file. You can also select the location your project should be saved.



New Android Project

Create Android Project

Select project name and type of project

Project Name: PA4AD_Ch02_Hello_World

(●) Create new project in workspace

( ) Create project from existing source

( ) Create project from existing sample

[ ] Use default location

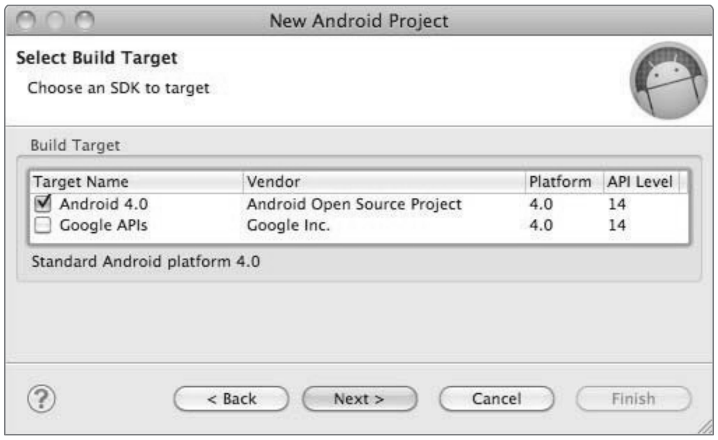
Location: /Users/retomeier/Android/Workspace/PA4AD_Ch02_I

Working sets

[ ] Add project to working sets

Working sets:

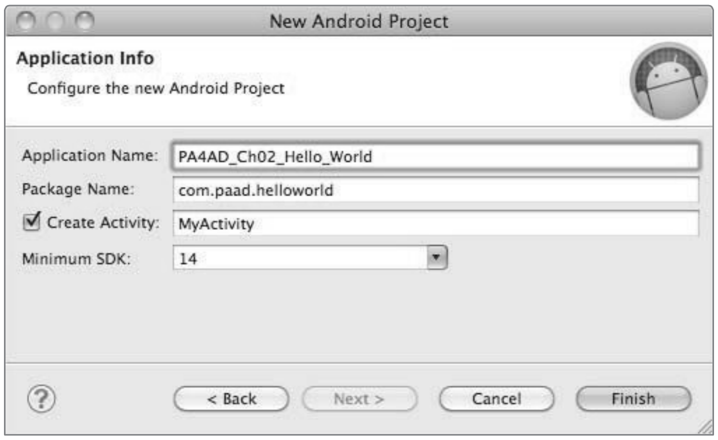
4. The next page (figure) lets you select the build target for your application. The *build ta get* is the version of the Android framework SDK that you plan to develop with. In addition to the open sourced Android SDK libraries available as part of each platform release, Google offers a set of proprietary APIs that offer additional libraries (such as Maps). If you want to use these Google-specific APIs, you must select the Google APIs package corresponding to the platform release you want to target.



Target Name	Vendor	Platform	API Level
✓ Android 4.0	Android Open Source Project	4.0	14
Google APIs	Google Inc.	4.0	14

Standard Android platform 4.0

5. The final page allows you to specify the application properties. The Application Name is the friendly name for your application; the Package Name specifies its Java pack- age; the Create Activity option lets you specify the name of a class that will be your initial Activity; and setting the Minimum SDK lets you specify the minimum version of the SDK that your application will run on.



Application Name: PA4AD_Ch02_Hello_World
Package Name: com.paad.helloworld
✓ Create Activity: MyActivity
Minimum SDK: 14

6. When you've entered the details, click Finish. If you selected Create Activity, the ADT plug-in will create a new project that includes a class that extends Activity. Rather than being completely empty, the default template implements Hello World. Before modifying the project, take this opportunity to configure launch configurations for running and debugging.

Creating an Android Virtual Device:

AVDs are used to simulate the hardware and software configurations of different Android devices, allowing you test your applications on a variety of hardware platforms. There are no prebuilt AVDs in the Android SDK, so without a physical device, you need to create at least one before you can run and debug your applications.

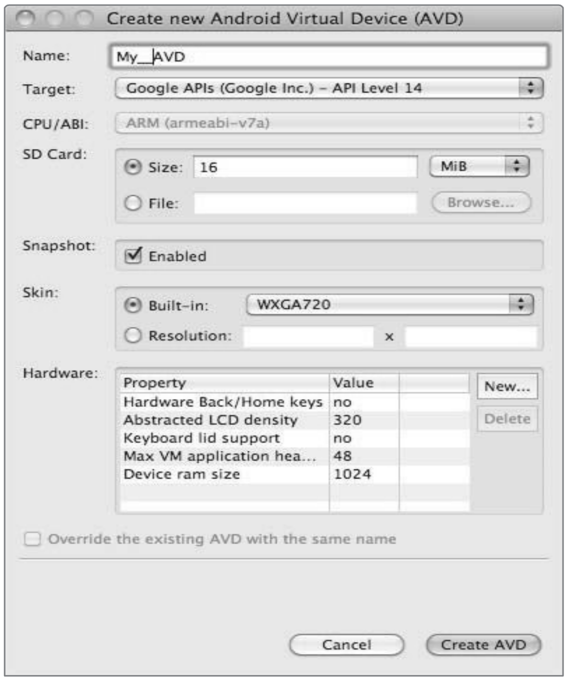
1. Select Window → AVD Manager (or select the AVD Manager icon on the Eclipse toolbar).
2. Select the New... button. The resulting Create new Android Virtual Device (AVD) dialog allows you to configure a name, a target build of Android, an SD card capacity, and device skin.

3. Create a new AVD called "My_AVD" that targets Android 4.0.3, includes a 16MB SD Card, and uses the Galaxy Nexus skin, as shown in figure.
4. Click Create AVD and your new AVD will be created and ready to use.

Creating Launch Configurations:

Launch configurations let you specify runtime options for running and debugging applications. Using a launch configuration you can specify the following:

- The Project and Activity to launch
- The deployment target (virtual or physical device)
- The Emulator's launch parameters
- Input/output settings (including console defaults)



Name: My_AVD
Target: Google APIs (Google Inc.) – API Level 14
CPU/ABI: ARM (armeabi-v7a)
SD Card: Size: 16 MiB
File:
Snapshot: ✓ Enabled
Skin: Built-in: WXGA720
Resolution: x
Hardware:

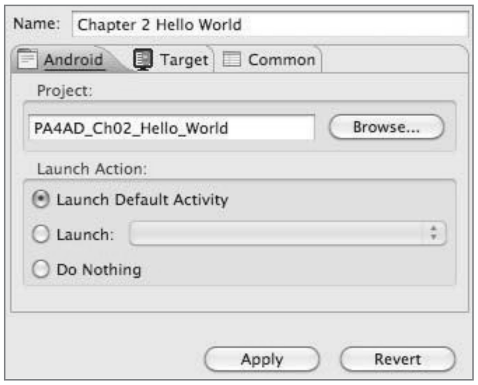
Property	Value
Hardware Back/Home keys	no
Abstracted LCD density	320
Keyboard lid support	no
Max VM application hea...	48
Device ram size	1024

Override the existing AVD with the same name

You can specify different launch configurations for running and debugging applications. The following steps show how to create a launch configuration for an Android application:

1. Select Run Configurations… or Debug Configurations… from the Run menu.
2. Select your application from beneath the Android Application node on the project type list, or right-click the Android Application node and select New.
3. Enter a name for the configuration. You can create multiple configurations for each project, so create a descriptive title that will help you identify this particular setup.
4. Choose your start-up options. The first (Android) tab lets you select the project to run and the Activity that you want to start when you run (or debug) the application. figure shows the settings for the project you created earlier.

5. Use the Target tab, as shown in figure, to select the default virtual device to launch on, or select Manual to select a physical or virtual device each time you run the application. You can also configure the Emulator's network connection settings and optionally wipe the user data and disable the boot animation when launching a virtual device.

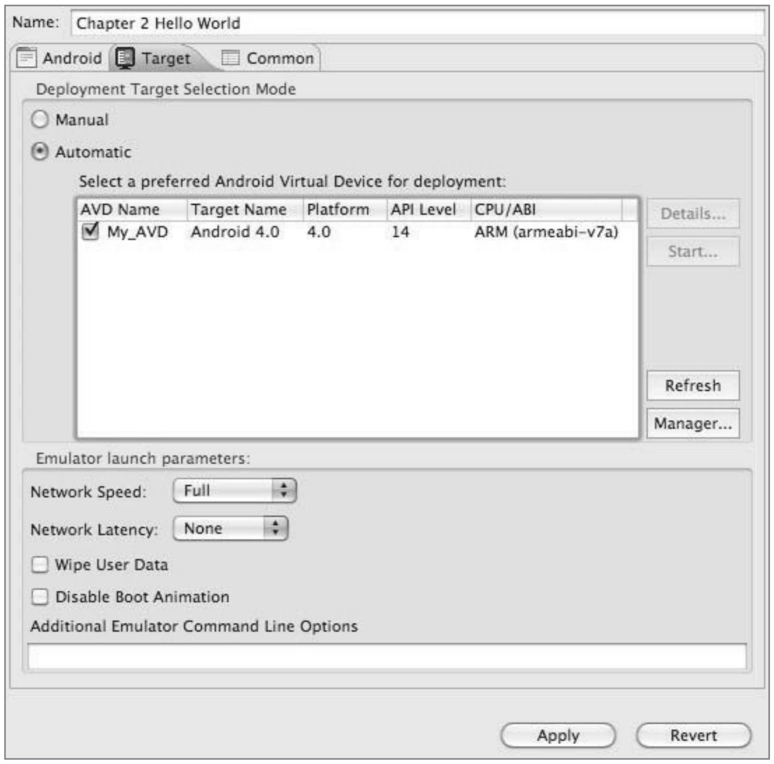


Name: Chapter 2 Hello World

Project: PA4AD_Ch02_Hello_World

Launch Action:
- [✓ Launch Default Activity]
- Launch:
- Do Nothing

6. Set any additional properties in the Common tab.
7. Click Apply, and your launch configuration will be saved.



Name: Chapter 2 Hello World

Deployment Target Selection Mode
- Manual
- [✓ Automatic]

Select a preferred Android Virtual Device for deployment:

AVD Name	Target Name	Platform	API Level	CPU/ABI
✓ My_AVD	Android 4.0	4.0	14	ARM (armeabi-v7a)

Emulator launch parameters:

Network Speed: Full

Network Latency: None

Wipe User Data

Disable Boot Animation

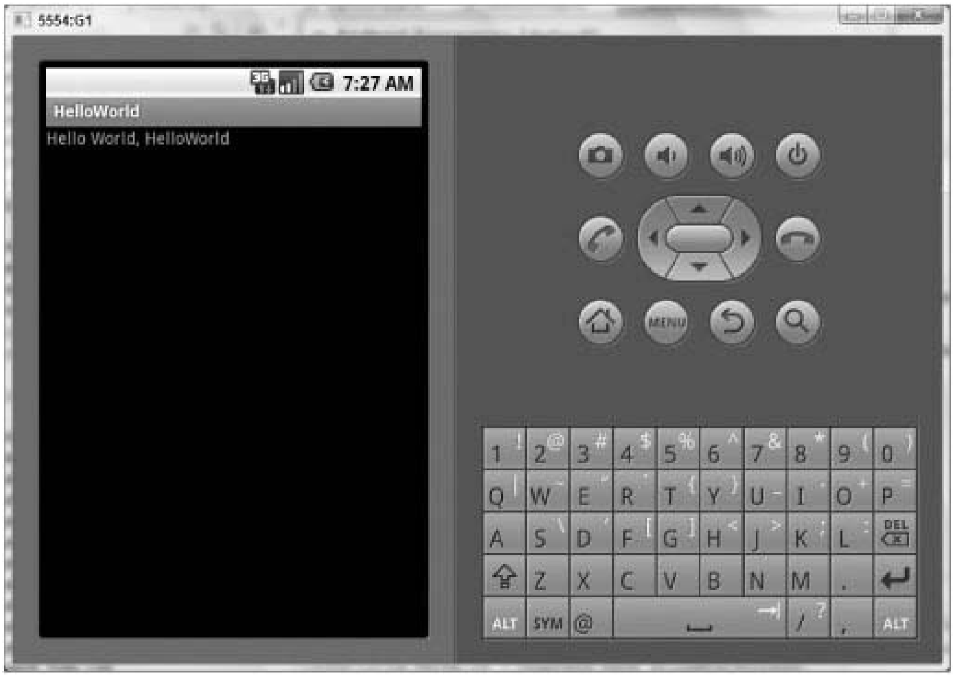
Additional Emulator Command Line Options

Running and Debugging Your Android Application:

You've created your first project and created the run and debug configurations for it. Before making any changes, test your installation and configurations by running and debugging the Hello World project. From the Run menu, select Run or Debug to launch the most recently selected configuration, or select Run Configurations… or Debug Configurations… to select a specific configuration. If you're using the ADT plug-in, running or debugging your application does the following:

- Compiles the current project and converts it to an Android executable (.dex)
- Packages the executable and your project's resources into an Android package (.apk)
- Starts the virtual device (if you've targeted one and it's not already running)
- Installs your application onto the target device
- Starts your application

If you're debugging, the Eclipse debugger will then be attached, allowing you to set breakpoints and debug your code. If everything is working correctly, you'll see a new Activity running on the device or in the Emulator, as shown in figure.



8. <u>Types of Android Applications:</u>

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 effectively suspended when it's not visible. Games are the most common examples.
- **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 examples include call screening applications, SMS auto-responders, and alarm clocks.
- **Intermittent:** Most well-designed applications fall into this category. At one extreme are applications 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 are typically used as foreground applications but that do important work in the background. Email and news applications are great examples.

- **Widgets and Live Wallpapers:** Some applications are represented only as a home-screen Widget or as a Live Wallpaper.

Complex applications are often difficult to pigeonhole into a single category and usually include elements of each of these types. When creating your application, you need to consider how it's likely to be used and then design it accordingly. The following sections look more closely at some of the design considerations for each application type.

1. **Foreground Applications**

When creating foreground applications, you need to consider carefully the Activity lifecycle so that the Activity switches seamlessly between the background and the foreground. Applications have little control over their lifecycles, and a background application with no running Services is a prime candidate for cleanup by Android's resource management. This means that you need to save the state of the application when it leaves the foreground, and then present the same state when it returns to the front. It's also particularly important for foreground applications to present a slick and intuitive user experience.

2. **Background Applications**

These applications run silently in the background with little user input. They often listen for messages or actions caused by the hardware, system, or other applications, rather than relying on user interaction. You can create completely invisible services, but in practice it's better to provide at least a basic level of user control. At a minimum you should let users confirm that the service is running and let them configure, pause, or terminate it, as needed.

3. **Intermittent Applications**

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. These applications are generally a union of visible Activities and invisible background Services and Broadcast Receivers. 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. 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.

4. **Widgets and Live Wallpapers**

In some circumstances your application may consist entirely of a Widget or Live Wallpaper. By creating Widgets and Live Wallpapers, you provide interactive visual components that can add functionality to user's home screens. Widget-only applications are commonly used to display dynamic information, such as battery levels, weather forecasts, or the date and time.

9. <u>Android Development Tools:</u>

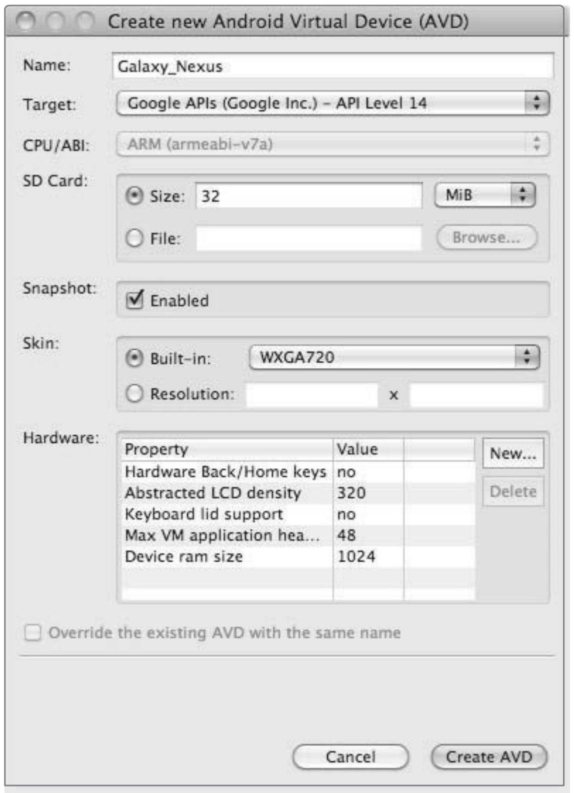
The Android SDK includes several tools and utilities to help you create, test, and debug your projects. As mentioned earlier, the ADT plug-in conveniently incorporates many of these tools into the Eclipse IDE, where you can access them from the DDMS perspective, including the following:

- **The Android Virtual Device and SDK Managers:** Used to create and manage AVDs and to download SDK packages, respectively. The AVD hosts an Emulator running a particular build of Android, letting you specify the supported SDK version, screen resolution, amount of SD card storage available, and available hardware capabilities (such as touchscreens and GPS).
- **The Android Emulator:** An implementation of the Android VM designed to run within an AVD on your development computer. Use the Emulator to test and debug your Android applications.
- **Dalvik Debug Monitoring Service (DDMS):** Use the DDMS to monitor and control the Emulators on which you're debugging your applications.
- **Android Debug Bridge (ADB):** A client-server application that provides a link to virtual and physical devices. It lets you copy files, install compiled application packages (.apk), and run 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 package files (.apk).
- **SQLite3:** A database tool that you can use to access the SQLite database files created and used by Android.
- **Traceview** and **dmtracedump:** Graphical analysis tools for viewing the trace logs from your Android application.
- **Hprof-conv:** A tool that converts HPROF profiling output files into a standard format to view in your preferred profiling tool.
- **MkSDCard:** Creates an SD card disk image that can be used by the Emulator to simulate an 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 to debug 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 and optimizations.
- **Draw9patch:** A handy utility to simplify the creation of NinePatch graphics using a WYSIWYG editor.
- **Monkey** and **Monkey Runner:** Monkey runs within the VM, generating pseudo-random user and system events. Monkey Runner provides an API for writing programs to control the VM from outside your application.

- **ProGuard:** A tool to shrink and obfuscate your code by replacing class, variable, and method names with semantically meaningless alternatives. This is useful to make your code more difficult to reverse engineer.

The Android Virtual Device Manager

The Android Virtual Device Manager is used to create and manage the virtual devices that will host instances of the Emulator. AVDs are used to simulate the software builds and hardware configurations available on different physical devices. This lets you test your application on a variety of hardware platforms without needing to buy a variety of phones. Each virtual device is configured with a name, a target build of Android (based on the SDK version it supports), an SD card capacity, and screen resolution, as shown in the Create new Android Virtual Device (AVD) dialog in figure. You can also choose to enable snapshots to save the Emulator state when it's closed. Starting a new Emulator from a snapshot is significantly faster. Each virtual device also supports a number of specific hardware settings and restrictions that can be added in the form of name-value pairs (NVPs) in the hardware table. Selecting one of the built-in skins will automatically configure these additional settings corresponding to the device the skin represents.



Create new Android Virtual Device (AVD)

Name: Galaxy_Nexus

Target: Google APIs (Google Inc.) – API Level 14

CPU/ABI: ARM (armeabi-v7a)

SD Card: [✓ Size:] 32 MiB File:

Snapshot: ✓ Enabled

Skin: [✓ Built-in:] WXGA720 Resolution: x

Hardware:

Property	Value
Hardware Back/Home keys	no
Abstracted LCD density	320
Keyboard lid support	no
Max VM application hea...	48
Device ram size	1024

Override the existing AVD with the same name

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 manufacturers have made hardware presets and virtual device skins available for their devices. Some, including Samsung, are available as SDK packages.

Android SDK Manager

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

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 running Android 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 speed and latency while debugging your applications. You can also simulate placing and receiving voice calls and SMS messages. The ADT plug-in integrates the Emulator into Eclipse so that it's launched automatically within the selected AVD when you run or debug your projects. If you aren't using the plug-in or want to use the Emulator outside of Eclipse, you can telnet into the Emulator and control it from its console. To execute the Emulator, you first need to create a virtual device, as described in the previous sec

The Dalvik Debug Monitor Service

The Emulator enables you to see how your application will look, behave, and interact, but to actually see what's happening under the surface, you need the Dalvik Debug Monitoring Service. The DDMS is a powerful debugging tool that lets you interrogate active processes, view the stack and heap, watch and pause active threads, and explore the file system of any connected Android device. The DDMS perspective in Eclipse also provides simplified access to screen captures of the Emulator and the logs generated by LogCat.

If you're using the ADT plug-in, the DDMS tool is fully integrated into Eclipse and is available from the DDMS perspective. If you aren't using the plug-in or Eclipse, you can run DDMS from the command line (it's available from the tools folder of the Android SDK), and it will automatically connect to any running device or Emulator. The Emulator will launch the virtual device and run a Dalvik instance within it.

The Android Debug Bridge

The *Android Debug Bridge* (ADB) is a client-service application that lets you connect with an Android 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 the service

As a communications conduit between your development hardware and the Android device/ Emulator, the ADB lets you install applications, push and pull files, and run shell commands on the target device. Using the device shell, you can change logging settings and query or modify SQLite databases available on the device. The ADT tool automates and simplifies a lot of the usual interaction with the ADB, including application installation and updating, file logging, and file transfer (through the DDMS perspective).

The Hierarchy Viewer and Lint Tool

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

To optimize performance, the performance of the layout, measure, and draw steps of creating the UI of each View at runtime is displayed. Using these values, you can learn the actual time taken to create each View within your hierarchy, with colored "traffic light" indicators showing the relative performance for each step. You can then search within your layout for Views that appear to be taking longer to render than they should. The Lint tool helps you to optimize your layouts by checking them for a series of common inefficiencies that can have a negative impact on your application's performance. Common issues include a surplus of nested layouts, a surplus of Views within a layout, and unnecessary parent Views.

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 UI events to your application. It's particularly useful to stress test your applications to investigate edge cases you 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 to control an Emulator or device from outside the application. It's extremely useful for performing UI, functional, and unit tests in a predictable, repeatable fashion.

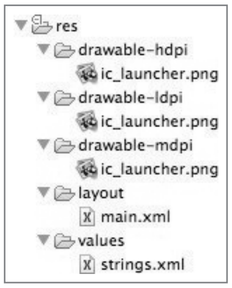
10. Externalizing the Resources:

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. Perhaps the most powerful externalizable resources are layouts.

By externalizing resources, you make them easier to maintain, update, and manage. This also lets you easily define alternative resource values for internationalization and to include different resources to support variations in hardware particularly, screen size and resolution. How Android dynamically selects resources from resource trees that contain different values for alternative hardware configurations, languages, and locations. When an application starts, Android automatically selects the correct resources without you having to write a line of code. Among other things, this lets you change the layout based on the screen size and orientation, images based on screen density, and customize text prompts based on a user's language and country.

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. If you start a project using the ADT Wizard, it creates a res folder that contains subfolders for the values, drawable-ldpi, drawable-mdpi, drawable-hdpi, and layout resources that contain the default string resource definitions, application icon, and layouts respectively, as shown in figure. Note that three drawable resource folders contain three different icons: one each for low, medium, and high density displays respectively.



Each resource type is stored in a different folder: simple values, Drawables, colors, layouts, animations, styles, menus, XML files (including searchables), and raw resources. 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. This enables you to reference the resources in your code, with the advantage of design-time syntax checking. The following sections describe many of the specific resource types available within these categories and how to create them for your applications. In all cases, the resource filenames should contain only lowercase letters, numbers, and the period (.) and underscore (_) symbols.

Simple Values

Supported simple values include strings, colors, dimensions, styles, and string or integer arrays. All simple values are stored within XML files in the res/values folder. This example includes all the simple value types. By convention, resources are generally stored in separate files, one for each type; for example, res/values/strings.xml would contain only string resources. The following sections detail the options for defining simple resources.

Strings

Externalizing your strings helps maintain consistency within your application and makes it much easier to internationalize them. 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 <b>, <i>, and <u> to apply bold, italics, or underlining, respectively, to parts of your text strings, as shown in the following example:

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

You can use resource strings as input parameters for the String.format method. However, String.format does not support the text styling previously described. To apply styling to a format string, you have to escape the HTML tags when creating your resource, as shown in the following snippet:

```
<string name=”stop_message”>&lt;b>Stop&lt;/b>. %1$s</string>
```

Within your code, use the Html.fromHtml method to convert this back into a styled character sequence.

Colors

Use the color tag to define a new color resource. Specify the color value using a # symbol followed by the (optional) alpha channel, and then the red, green, and blue values using one or two hexadeci- mal numbers 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)

Although you can use any of these measurements to define a dimension, it's best practice to use either density-or scale-independent pixels. These alternatives let you define a dimension using relative scales that account for different screen resolutions and densities to simplify scaling on different hardware. Scale-independent pixels are particularly well suited when defining font sizes because they automatically scale if the user changes the system font size. 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. The most common use of themes and styles is to store the colors and fonts for an application. 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. The tag itself should then contain the value, as shown in the following skeleton code.

```
<?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>
```

Styles support inheritance using the parent attribute on the style tag, making it easy to create simple variations:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
<style name=”small_text” parent=”base_text”>
<item name=”android:textSize”>8sp</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. Note that it's good practice to store bitmap image assets in the appropriate drawable -ldpi, -mdpi, -hdpi, and -xhdpi folders, as described earlier in this chapter. The resource identifier for a Drawable resource is the lowercase file name without its extension.

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. 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.

Using layouts to construct your screens in XML is best practice in Android. The decoupling of the layout from the code enables you to create optimized layouts for different hardware configurations, such as varying screen sizes, orientation, or the presence of keyboards and touchscreens.

Each layout definition is stored in a separate file, each containing a single layout, in the res/layout folder. The filename then becomes the resource identifier.

Animations

Defining animations as external resources enables you to reuse the same sequence in multiple places and provides you with the opportunity to present different animations based on device hardware or orientation. 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.
    a. Property animators were introduced in Android 3.0 (API level 11). It is a powerful framework that can be used to animate almost anything.
    b. Each property animation is stored in a separate XML file in the project's res/animator folder. As with layouts and Drawable resources, the animation's filename is used as its resource identifier.
    c. You can use a property animator to animate almost any property on a target object. You can define animators that are tied to a specific property, or a generic value animator that can be allocated to any property and object. Property animators are extremely useful and are used extensively for animating Fragments in Android.

2. **View animations:** Tweened animations that can be applied to rotate, move, and stretch a View.
    a. Each view animation is stored in a separate XML file in the project's res/anim folder. As with layouts and Drawable resources, the animation's filename is used as its resource identifier. An animation can be defined for changes in alpha (fading), scale (scaling), translate (movement), or rotate (rotation).
    b. You can create a combination of animations using the set tag. An animation set contains one or more animation transformations and supports various additional tags and attributes to customize when and how each animation within the set is run.
    c. The following list shows some of the set tags available:
        i. duration — Duration of the full animation in milliseconds.
        ii. startOffset — Millisecond delay before the animation starts.
        iii. fillBeforetrue — Applies the animation transformation before it begins.
        iv. fillAftertrue — Applies the animation transformation after it ends.
        v. interpolator — Sets how the speed of this effect varies over time.
3. **Frame animations:** Frame-by-frame "cell" animations used to display a sequence of Drawable images.
    a. Frame-by-frame animations produce a sequence of Drawables, each of which is displayed for a specified duration.
    b. Because frame-by-frame animations represent animated Drawables, they are stored in the res/ drawable folder and use their filenames (without the .xml extension) as their resource Ids.

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. When defined in XML, 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 file, each containing a single menu, in the res/menu folder — the filename then becomes the resource identifier.

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. For example, a dimension resource might be referenced in a layout definition.

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. The R class contains static subclasses for each of the resource types for which you’ve defined at least one resource. For example, the default new project includes the R.string and R.drawable subclasses.

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.

Referencing Resources within Resources

You can also use resource references as attribute values in other XML resources. This is particularly useful for layouts and styles, letting you create specialized variations on themes and localized strings and image assets. It’s also a useful way to support different images and spacing for a layout to ensure that it’s optimized for different screen sizes and resolutions. To reference one resource from another, use the @ notation, as shown in the following snippet:

```
attribute=”@[packagename:]resourcetype/resourceidentifier”
```

Using System Resources

The Android framework makes many native resources available, providing you with various strings, images, animations, styles, and layouts to use in your applications. 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);
```

To access system resources in XML, specify android as the package name, as shown in this XML snippet:

```
<EditText
android:id=”@+id/myEditText”
android:layout_width=”match_parent”
android:layout_height=”wrap_content”
android:text=”@android:string/httpErrorBadUrl”
android:textColor=”@android:color/darker_gray”

/>
```

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"
/>
```

Creating Resources for Di"erent 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 for. The following list gives the qualifiers you can use to customize your resource values:

- **Mobile Country Code and Mobile Network Code (MCC/MNC):** The country and optionally the network, associated with the SIM currently used in the device. 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).
- **Language and Region:** Language specified by the lowercase two-letter ISO 639-1 language code, followed optionally by a region specified by a lowercase r followed by the uppercase two-letter ISO 3166-1-alpha-2 language code (for example, en, en-rUS, or en-rGB).
- **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). This is generally used when providing multiple layouts, where the value specified should be the smallest screen width that your layout requires in order to render correctly. Where you supply multiple directories with different smallest screen width qualifiers, Android selects the largest value that doesn't exceed the smallest dimension available on the device.
- **Available Screen Width:** The minimum screen width required to use the contained resources, specified in the form w<Dimension value>dp (for example, w600dp, w320dp, or w720dp). Also used to supply multiple layouts alternatives, but unlike smallest screen width, 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.
- **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). Like available screen width, the available screen height changes when the device orientation changes to reflect the current screen height. Android selects the largest value that doesn't exceed the currently available screen height.

- **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). Because each of these screen categories can include devices with significantly different screen sizes (particularly tablets), it's good practice to use the more specific smallest screen size, and available screen width and height whenever possible. Because they precede this screen size qualifier, where both are specified, the more specific qualifiers will be used in preference where supported.
- **Screen Aspect Ratio:** Specify long or notlong for resources designed specifically for wide screen. (For example, WVGA is long; QVGA is not long.)
- **Screen Orientation:** One of port (portrait), land (landscape), or square (square).
- **Dock Mode:** One of car or desk. Introduced in API level 8.
- **Night Mode:** One of night (night mode) or notnight (day mode). Introduced in API level Used in combination with the dock mode qualifier, this provides a simple way to change the theme and/or color scheme of an application to make it more suitable for use at night in a car dock.
- **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 extra high (320 dpi) pixel density assets, respectively. You can specify nodpi for bitmap resources you don't want scaled to support an exact screen density. To better support applications targeting televisions running Android, you can also use the tvdpi qualifier for assets of approximately 213dpi. This is generally unnecessary for most applications, where including medium- and high-resolution assets is sufficient for a good user experience. Unlike with other resource types, Android does not require an exact match to select a resource. When selecting the appropriate folder, it chooses the nearest match to the device's pixel density and scales the resulting Drawables accordingly.
- **Touchscreen Type:** One of notouch, stylus, or finger, allowing you to provide layouts or dimensions optimized for the style of touchscreen input available on the host device.
- **Keyboard Availability:** One of keysexposed, keyshidden, or keyssoft.
- **Keyboard Input Type:** One of nokeys, qwerty, or 12key.
- **Navigation Key Availability:** One of navexposed or navhidden.
- **UI Navigation Type:** One of nonav, dpad, trackball, or wheel.
- **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.

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. In some special cases this default behavior may be inconvenient, particularly for applications that don't want to present a different UI based on screen orientation changes. You can customize your application's response to such changes by detecting and reacting to them yourself. To have an Activity listen for runtime configuration changes, add an android:configChanges attribute to its manifest node, specifying the configuration changes you want to handle.

11. <u>The Android Application Lifecycle:</u>

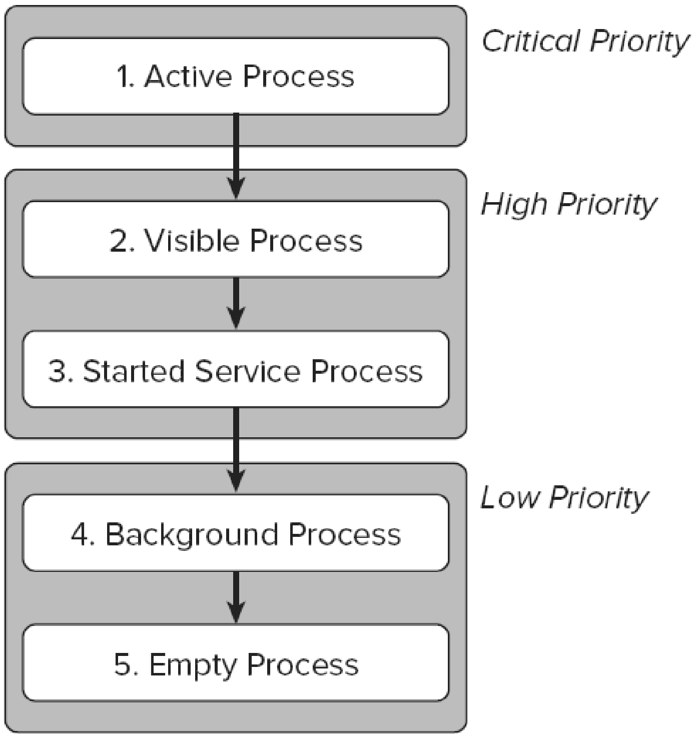
Unlike many traditional application platforms, Android applications have limited control over their own lifecycles. Instead, application components must listen for changes in the application state and react accordingly, taking particular care to be prepared for untimely termination. By default, each 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 (and their hosted applications) will be killed, in some case without warning, to free resources for higher-priority applications.

The order in which processes are killed to reclaim resources is determined by the priority of their hosted 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 a dependency on a Service or Content Provider supplied by a second application, the secondary application has at least as high a priority as the application it supports.

The below figure shows the priority tree used to determine the order of application termination. It's important to structure your application to ensure that its priority is appropriate for the work it's doing. If you don't, your application could be killed while it's in the middle of something important, or it could remain running when it is no longer needed. The following list details each of the application states shown in figure, explaining how the state is determined by the application components of which it comprises:

- **Active processes:** Active (foreground) processes have application components the user is interacting with. These are the processes Android tries to keep responsive by reclaiming resources from other applications. There are generally very few of these processes, and they will be killed only as a last resort.
- **Visible processes** — Visible but inactive processes are those hosting "visible" Activities. As the name suggests, visible Activities are visible, but they aren't in the foreground or responding to user events. This happens when an Activity is only partially obscured (by a non-full-screen or transparent Activity). There are generally very few visible processes, and they'll be killed only under extreme circumstances to allow active processes to continue.



- **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. When the system terminates a running Service it will attempt to restart them (unless you specify that it shouldn't) when resources become available.
- **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-first-killed pattern in order to obtain resources for foreground processes.
- **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.

******

UNIT - II

Building User Interface: Fundamental Android UI design - Android User Interface fundamentals - Layouts - Linear - Relative - Grid Layouts - Fragments - Creating new fragments - The Fragments Lifecycle - Introducing the Fragment Manager - Adding Fragments to Activities - Interfacing between Fragments and Activities

---

1. **<u>Fundamental Android UI design:</u>**

Android introduces some new terminology for familiar programming metaphors that will be explored 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. It is the base class for all visual components (control and widgets). All the controls present in an android app are derived from this class.
    - ❖ A View is an object that draws something on a Smartphone screen and enables a user to interact with it. It represents the basic building block for user interface components. The user interface in android app is made with a collection of View and ViewGroup objects.
    - ❖ The following are the common View subclasses which will be used in android applications. They are TextView, EditText, Button, CheckBox, RadioButton, ImageButton, ProgressBar and Spinner.
- View Groups — View Groups are extensions of the View class that can contain multiple child Views. Extend the ViewGroup class to create compound controls made up of interconnected child Views. The ViewGroup class is also extended to provide the Layout Managers that help you lay out controls within your Activities.
- Fragments — Fragments, introduced in Android 3.0 (API level 11), are used to encapsulate portions of your UI. This encapsulation makes Fragments particularly useful when optimizing your UI layouts for different screen sizes and creating reusable UI elements. Each Fragment includes its own UI layout and receives the related input events but is tightly bound to the Activity into which each must be embedded. Fragments are similar to UI View Controllers 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 traditional Windows desktop development. To display a UI, you assign a View (usually a layout or Fragment) to an Activity.

Android provides several common UI controls, widgets, and Layout Managers. For most graphical applications, it's likely that you'll need to extend and modify these standard Views or create composite or entirely new Views to provide your own user experience.

2. **<u>Android User Interface Fundamentals:</u>**

The ViewGroup class is an extension of View designed to contain multiple Views. View Groups are used most commonly to manage the layout of child Views, but they can also be used to create atomic reusable components. View Groups that perform the former function are generally referred to as layouts. In the following sections you'll learn how to put together increasingly complex UIs, before being introduced to Fragments, the Views available in the SDK, how to extend these Views, build your own compound controls, and create your own custom Views from scratch.

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 empty screens aren't particularly inspiring, you will almost always use setContentView to assign an Activity's UI when overriding its onCreate handler. The setContentView method accepts either a layout's resource ID or a single View instance. This lets 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 different layouts optimized for different hardware configurations, even changing them at run time based on 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");
}
```

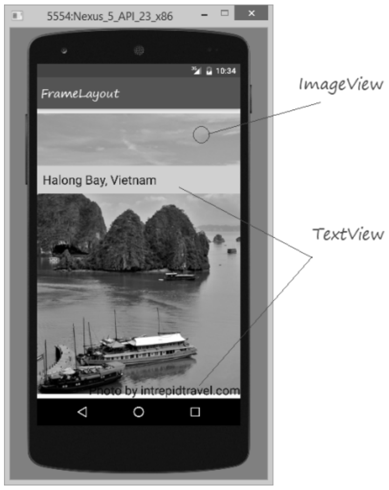
The setContentView method accepts a single View instance; as a result, you use layouts to add multiple controls to your Activity. If you're using Fragments to encapsulate portions of your Activity's UI, the View inflated within your Activity's onCreate handler will be a layout that describes the relative position of each of your Fragments (or their containers).

The UI used for each Fragment is defined in its own layout and inflated within the Fragment itself. Note that once a Fragment has been inflated into an Activity, the Views it contains become part of that Activity's View hierarchy. As a result you can find any of its child Views from within the parent Activity, using findViewById as described previously.

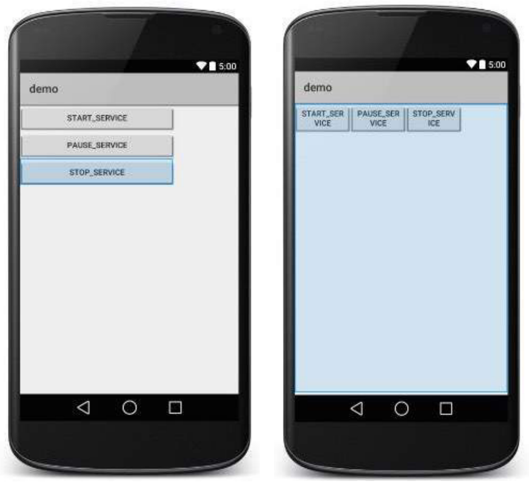
3. <u>Layouts:</u>

Layout defines a visual structure of an Activity. It may be considered as a set of rules according to which controls (buttons, text fields, input fields etc.) are placed on the View. It is used to define the user interface for an app or activity and it will hold the UI elements that will appear to the user. The Android SDK includes a number of layout classes. You can use these, modify them, or create your own to construct the UI for your Views, Fragments, and Activities. It's up to you to select and use the right combination of layouts to make your UI aesthetically pleasing, easy to use, and efficient to display. The following list includes some of the most commonly used layout classes available in the Android SDK:

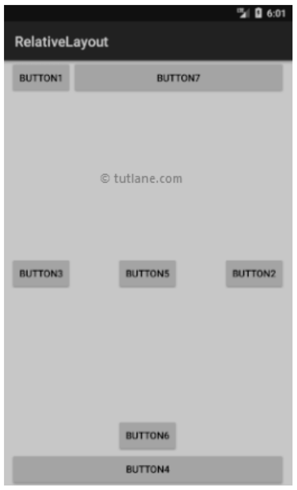
FrameLayout:
FrameLayout is a ViewGroup subclass which is used to specify the position of View instances it contains on the top of each other to display only single View inside the FrameLayout. The FrameLayout is a placeholder on screen that you can use to display a single View.



LinearLayout:
LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally. You can specify the layout direction with the android: orientation attribute. All children of a LinearLayout are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding). LinearLayout respects margins between children and the *gravity* (right, center, or left alignment) of each child.

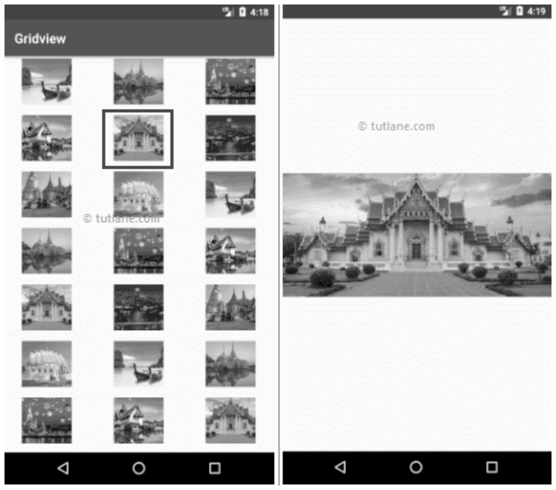


RelativeLayout:



RelativeLayout is a ViewGroup which is used to specify the position of child View instances relative to each other (Child A to the left of Child B) or relative to the parent (Aligned to the top of parent). It is very useful to design user interface because by using relative layout, eliminate the nested view groups and keep our layout hierarchy flat, which improves performance of application.

GridLayout:



GridView is a ViewGroup which is used to display items in a two dimensional, scrollable grid and grid items are automatically inserted to the GridView layout using a list adapter. Adapter will act as an intermediate between the data sources and adapter views such as ListView, Gridview to fill the data into adapter views. The adapter will hold the data and iterates through items in data set and generate the views for each item in the list.

Each of these layouts is designed to scale to suit the host device's screen size by avoiding the use of absolute positions or predetermined pixel values. This makes them particularly useful when designing applications that work well on a diverse set of Android hardware.

4. <u>Fragments:</u>

Fragments enable you to divide your Activities into fully encapsulated reusable components, each with its own lifecycle and UI. The primary advantage of Fragments is the ease with which you can create dynamic and flexible UI designs 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 to suit multipane tablet UIs and added to, removed from, and exchanged within a running Activity to help build dynamic UIs.

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

5. Creating New Fragments

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

```
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);
}
}
```

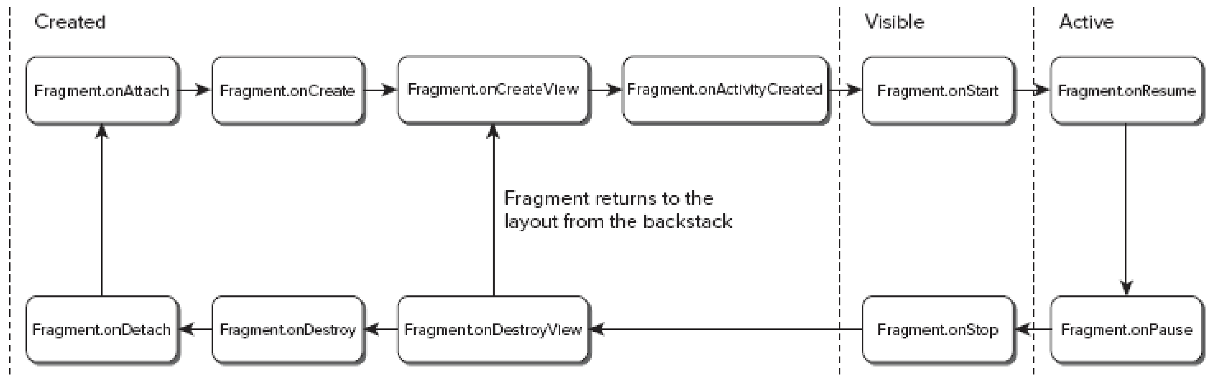
You can create a layout in code using layout View Groups; however, as with Activities, the preferred way to design Fragment UI layouts is by infl ating an XML resource. Unlike Activities, Fragments don't need to be registered in your manifest. This is because Fragments can exist only when embedded into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

6. The Fragment Lifecycle

The lifecycle events of a Fragment mirror those of its parent Activity; however, after the containing Activity is in its active — resumed — state adding or removing a Fragment will affect its lifecycle independently.

Fragments include a series of event handlers that mirror those in the Activity class. They are triggered as the Fragment is created, started, resumed, paused, stopped, and destroyed.

Fragments also include a number of additional callbacks that signal binding and unbinding the Fragment from its parent Activity, creation (and destruction) of the Fragment's View hierarchy, and the completion of the 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 in which 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's been 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 onDetach will 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 Fragment itself or its parent Activity have fi nished their initialization. Typically, the onAttach event is used to gain 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 fi rst call to onCreate and the final call to onDestroy. As it's not uncommon for an Activity's process to be terminated *without* the corresponding onDestroy method being called, so a Fragment can't rely on its onDestroy handler being triggered. As with Activities, you should use the onCreate method to initialize your Fragment. It's good practice to create any class scoped objects here to ensure they're created only once in the Fragment's lifetime.

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: Infl ate 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

The fate of a Fragment is inextricably bound to that of the Activity to which it belongs. As a result, 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 non deterministically 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 to compose your Activity's UI 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.

Whatever the trigger for a Fragment's transition through its lifecycle, managing its state transitions is critical in ensuring a seamless user experience. 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.

7. <u>Introducing the Fragment Manager:</u>

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.

8. <u>Adding Fragments to Activities:</u>

The simplest way to add a Fragment to an Activity is by including it within the Activity's layout using the fragment tag, as shown in below code:

```
<?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>
```

Once the Fragment has been inflated, it becomes a View Group, laying out and managing its UI within the Activity. 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();
```

Each of these transaction types and options will be explored in the following sections.

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 first 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);
```

If you’ve added a Fragment using a Fragment Transaction, you should specify the resource identifier of the container View to which you added the Fragment you want to fi nd. 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);
```

The find FragmentByTag method is essential for interacting with these Fragments. Because they’re not part of the Activity’s View hierarchy, they don’t have a resource identifier or a container resource identifier to pass in to the findFragmentById method.

Fragments and the Back Stack

Fragments enable you to create dynamic Activity layouts that can be modified to present significant changes in the UIs. In some cases these changes could be considered a new screen in which case a user may reasonably expect the back button to return to the previous layout. This involves reversing previously executed Fragment Transactions. Android provides a convenient technique for providing this functionality. To add the Fragment Transaction to the back stack,call addToBackStack on a Fragment Transaction before calling commit.

9. 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 fi nding the current Context, accessing other Fragments using the Fragment Manager, and fi nding 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), it's good practice to create a callback interface within the Fragment that a host Activity must implement.

Fragments without User Interfaces

In most circumstances, Fragments are used to encapsulate modular components of the UI; however, you can also create a Fragment without a UI to provide background behavior that persists across Activity restarts. 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.

Android 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.

- 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.
- 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.

****

UNIT - III

Intents and Broadcasts Receivers: Introducing Intents - Using intents to launch Activities - Introducing Linkify - Using Intents to Broadcast Events - Introducing the Local Broadcast Manager - Introducing pending intents - Using Intent filters to service implicit Intents - Using Intent Filters for Plug-ins and extensibility - Listening for Native Broadcast Intents - Monitoring Device State Changes Using Broadcast Intents.

1. Introducing Intents:

Intents are used as a message-passing mechanism that works both within your application and between applications. You can use Intents to do the following:

- Explicitly start a particular Service or Activity using its class name
- Start an Activity or Service to perform an action with (or on) a particular piece of data
- Broadcast that an event has occurred

Intents are the objects of android.content.Intent type and intents are mainly useful to perform following things.

- By passing an Intent object to startActivity() method start a new Activity or existing Activity to perform required things.
- By passing an Intent object to startService() method start a new Service or send required instructions to an existing Service.
- By passing an Intent object to sendBroadcast() method deliver our messages to other app broadcast receivers.

Generally, in android Intent object contains the information that required that which component to start and the information about the action to be performed by recipient component. It has following characteristics:

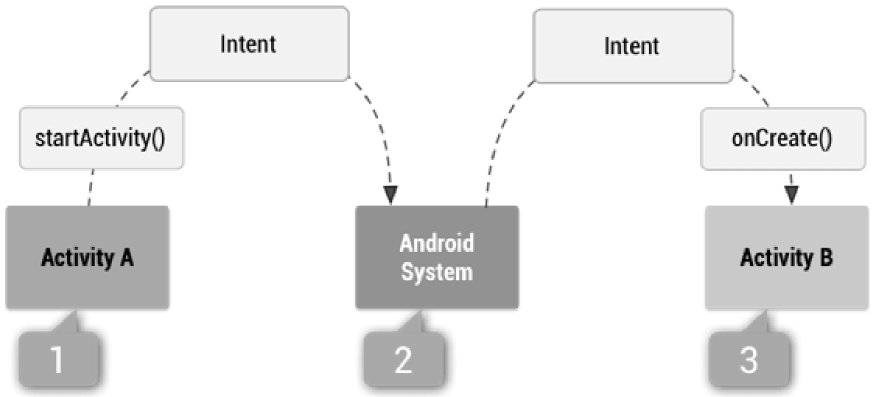
- **Component Name**: Defines a name of the component to start and by using component name android system will deliver intent to the specific app component defined by the component name. In case if we didn’t define component name then android system will decide which component should receive intent based on other intent information such as action, data, etc.
- **Action**: It defines a name of the action to be performed to start an activity.
    - ❖ ACTION_VIEW: Use this action in an intent with startActivity(), when we have information that an activity can show to the user.
    - ❖ ACTION_SEND: Use this action in an intent with startActivity(), when we have a some data that the user can share through another app such as email app, social sharing app.

❖ **Data**: It specifies a type of the data to an intent filter. When we create intent, it's important to specify the type of data (MIME type) in addition to its URI. By specifying a MIME type of data, it helps android system to decide which the best component to receive our intent is.

❖ **Category**: Generally, in android category is an optional for intents and it specifying the additional information about type of the component that should handle intent.

Implicit Intents:

- Implicit Intents won't specify any name of the component to start, instead it declare an action to perform and it allow a component from other app to handle it. For example, by using implicit intents, request another app to show the location details of user or etc.



- From the above figure Activity A creates an intent with required action and send it to an android system using startActivity() method. The android system will search for an intent filter that matches the intent in all apps. Whenever the match found the system starts matching activity (Activity B) by invoking onCreate() method.
- In android when we create implicit intents, the android system will search for matching component by comparing the contents of intent with intent filters which defined in manifest file of other apps on the device. If matching component found, the system starts that component and send it to Intent object. In case if multiple intent filters are matched then the system displays a dialog so that user can pick which app to use.

```
Intent intent=new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.google.com"));
startActivity(intent);
```

Explicit Intents:

- Explicit intents explicitly specify the name of component to be invoked by an activity and we use explicit intents to start a component in our own app. For example we can start a new activity in response to a user action using explicit intents.

- By using explicit intents we can send or share data / content from one activity to another activity based on our requirements. To create an Explicit Intent, we need to define the component name for Intent object.

```
Intent di = new Intent(this, ActivityView.class);
di.setData(Uri.parse("http://www.tutlane.com"));
startService(di);
```

2. **<u>Using Intents to Launch Activities:</u>**

- The most common use of Intents is to bind your application components and communicate between them. Intents are used to start Activities, allowing you to create a workflow of different screens.
- To create and display an Activity, call startActivity, passing in an Intent, as follows: startActivity(myIntent); The startActivity method finds and starts the single Activity that best matches your Intent.
- Construct the Intent to explicitly specify the Activity class to open, or to include an action that the target Activity must be able to perform. In the latter case, the run time will choose an Activity dynamically using a process known as intent resolution.

Explicitly Starting New Activities

- To select a specific Activity class to start, create a new Intent, specifying the current Activity's Context and the class of the Activity to launch. Pass this Intent into startActivity.
- Intent intent = new Intent(MyActivity.this, MyOtherActivity.class); startActivity(intent);
- After startActivity is called, the new Activity (in this example, MyOtherActivity) will be created, started, and resumed moving to the top of the Activity stack.
- Calling finish on the new Activity, or pressing the hardware back button, closes it and removes it from the stack. Alternatively, you can continue to navigate to other Activities by calling startActivity. Note that each time you call startActivity, a new Activity will be added to the stack; pressing back (or calling finish) will remove each of these Activities, in turn.

Implicit Intents and Late Runtime Binding

- An implicit Intent is a mechanism that lets anonymous application components service action requests. That means you can ask the system to start an Activity to perform an action without knowing which application, or Activity, will be started.
- When constructing a new implicit Intent, you specify an action to perform and, optionally, supply the URI of the data on which to perform that action. You can send additional data to the target Activity by adding extras to the Intent.

- For example, to let users make calls from your application, you could implement a new dialer, or you could use an implicit Intent that requests the action (dialing) be performed on a phone number (represented as a URI).

  ```
  if (somethingWeird && itDontLookGood)
  {
  Intent intent =
  new Intent(Intent.ACTION_DIAL, Uri.parse(“tel:555-2368”));
  startActivity(intent);
  }
  ```

- Android resolves this Intent and starts an Activity that provides the dial action on a telephone number in this case, typically the Phone Dialer.
- When constructing a new implicit Intent, you specify an action to perform and, optionally, supply the URI of the data on which to perform that action. You can send additional data to the target Activity by adding extras to the Intent.
- Extras are a mechanism used to attach primitive values to Intent. The extras are stored within the Intent as a Bundle object that can be retrieved using the getExtras method.
- When you use an implicit Intent to start an Activity, Android will at run time resolve it into the Activity class best suited to performing the required action on the type of data specified. This means you can create projects that use functionality from other applications without knowing exactly which application you’re borrowing functionality from ahead of time.
- Multiple Activities can potentially perform a given action; the user is presented with a choice. The process of intent resolution is determined through an analysis of the registered Broadcast Receivers.
- Broadcast receiver (receiver) is an Android component which allows you to register for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens.

Determining If an Intent Will Resolve

- Incorporating the Activities and Services of a third-party application into your own is incredibly powerful; however, there is no guarantee that any particular application will be installed on a device, or that any application capable of handling your request is available.
- To determine if your call will resolve to an Activity *before* calling startActivity.
- You can query the Package Manager to determine which, if any, Activity will be launched to service a specific Intent by calling resolveActivity on your Intent object, passing in the Package Manager.

- If no Activity is found, you can choose to either disable the related functionality (and associated user interface controls) or direct users to the appropriate application in the Google Play Store. Note that Google Play is not available on all devices, nor the emulator.

Returning Results from Activities

- An Activity started via startActivity is independent of its parent and will not provide any feedback when it closes. Where feedback is required, you can start an Activity as a sub-Activity that can pass results back to its parent.
- Sub-Activities are actually just Activities opened in a different way. You must register them in the application manifest in the same way as any other Activity.
- When a sub-Activity is finished, it triggers the onActivityResult event handler within the calling Activity. Sub-Activities are particularly useful in situations in which one Activity is providing data input for another, such as a user selecting an item from a list.

Launching Sub Activities,

- The startActivityForResult method works much like startActivity, but with one important difference. In addition to passing in the explicit or implicit Intent used to determine which Activity to launch, you also pass in a *request code*. This value will later be used to uniquely identify the sub Activity that has returned a result.

```
private static final int SHOW_SUBACTIVITY = 1;
private void startSubActivity()
{
Intent intent = new Intent(this, MyOtherActivity.class);
startActivityForResult(intent, SHOW_SUBACTIVITY);
}
```

Returning Results

- When your sub-Activity is ready to return, call setResult before finish to return a result to the calling Activity. The setResult method takes two parameters: the result code and the result data itself, represented as Intent.
- The result code is the “result” of running the sub-Activity — generally, either Activity. RESULT_OK or Activity.RESULT_CANCELED. In some circumstances, where OK and cancelled don’t sufficiently or accurately describe the available return results, you’ll want to use your own response codes to handle application-specific choices; setResult supports any integer value.
- If the Activity is closed by the user pressing the hardware back key, or finish is called without a prior call to setResult, the result code will be set to RESULT_CANCELED and the result Intent set to null. For handling Sub-Activity Results, The onActivityResult handler receives a number of parameters: Request code**,** Result code and Data.

- If the Activity is closed by the user pressing the hardware back key, or finish is called without a prior call to setResult, the result code will be set to RESULT_CANCELED and the result Intent set to null.

Handling Sub-Activity Results

When a sub-Activity closes, the onActivityResult event handler is fired within the calling Activity. Override this method to handle the results returned by sub-Activities. The onActivityResult handler receives a number of parameters: Request code**,** Result code and Data. The onActivityResult handler receives a number of parameters:

- **Request code:** The request code that was used to launch the returning sub-Activity.
- **Result code:** The result code set by the sub-Activity to indicate its result. It can be any integer value, but typically will be either Activity.RESULT_OK or Activity.RESULT_CANCELED.
- **Data:** An Intent used to package returned data. Depending on the purpose of the sub-Activity, it may include a URI that represents a selected piece of content. The sub Activity can also return information as an extra within the returned data Intent.

Native Android Actions

Native Android applications also use Intents to launch Activities and sub-Activities. The following (non comprehensive) list shows some of the native actions available as static string constants in the Intent class. When creating implicit Intents, you can use these actions, known as *Activity Intents*, to start Activities and sub-Activities within your own applications.

- ACTION_ALL_APPS — Opens an Activity that lists all the installed applications. Typically, this is handled by the launcher.
- ACTION_ANSWER — Opens an Activity that handles incoming calls. This is normally handled by the native in-call screen.
- ACTION_BUG_REPORT — Displays an Activity that can report a bug. This is normally handled by the native bug-reporting mechanism.
- ACTION_CALL — Brings up a phone dialer and immediately initiates a call using the num- ber supplied in the Intent's data URI. This action should be used only for Activities that replace the native dialer application. In most situations it is considered better form to use ACTION_DIAL.
- ACTION_CALL_BUTTON — Triggered when the user presses a hardware "call button." This typically initiates the dialer Activity.
- ACTION_DELETE — Starts an Activity that lets you delete the data specified at the Intent's data URI.

- ACTION_DIAL — Brings up a dialer application with the number to dial prepopulated from the Intent's data URI. By default, this is handled by the native Android phone dialer. The dialer can normalize most number schemas — for example, tel: 555-1234 and tel:(212) 555 1212 are both valid numbers.
- ACTION_EDIT — Requests an Activity that can edit the data at the Intent's data URI.
- ACTION_INSERT — Opens an Activity capable of inserting new items into the Cursor specified in the Intent's data URI. When called as a sub-Activity, it should return a URI to the newly inserted item.
- ACTION_PICK — launches a sub-Activity that lets you pick an item from the Content Provider specified by the Intent's data URI. When closed, it should return a URI to the item that was picked. The Activity launched depends on the data being picked — for example, passing content://contacts/people will invoke the native contacts list.
- ACTION_SEARCH — typically used to launch a specific search Activity. When it's fired with- out a specific Activity, the user will be prompted to select from all applications that support search. Supply the search term as a string in the Intent's extras using SearchManager.QUERY as the key.
- ACTION_SEARCH_LONG_PRESS — enables you to intercept long presses on the hardware search key. This is typically handled by the system to provide a shortcut to a voice search.
- ACTION_SENDTO — Launches an Activity to send data to the contact specified by the Intent's data URI.
- ACTION_SEND — Launches an Activity that sends the data specified in the Intent. The recipient contact needs to be selected by the resolved Activity. Use setType to set the MIME type of the transmitted data. The data itself should be stored as an extra by means of the key EXTRA_TEXT or EXTRA_STREAM, depending on the type. In the case of email, the native Android applications will also accept extras via the EXTRA_EMAIL, EXTRA_CC, EXTRA_BCC, and EXTRA_SUBJECT keys. Use the ACTION_SEND action only to send data to a remote recipi- ent (not to another application on the device).
- ACTION_VIEW — This is the most common generic action. View asks that the data supplied in the Intent's data URI be viewed in the most reasonable manner. Different applications will handle view requests depending on the URI schema of the data supplied. Natively http: addresses will open in the browser; tel: addresses will open the dialer to call the number; geo: addresses will be displayed in the Google Maps application; and contact content will be displayed in the Contact Manager.
- ACTION_WEB_SEARCH — Opens the Browser to perform a web search based on the query sup- plied using the SearchManager.QUERY key.

3. <u>Introducing Linkify:</u>

- The Linkify class is used to create the links from the *TextView* or the Spannable. It converts the text and regular expression to the clickable links on the basis of the pattern match of text value and the regex. The Linkify class creates the links for web URL, email address, phone number, and map address by using pattern. Android clickable links can be created by two different ways:
- Using layout (.xml) file: it uses autoLink attribute with the specified type.

```
<TextView
 android:id="@+id/url"
 android:autoLink="web"/>
```

- Using Java class: it uses the addlLinks() method with specified types of Linkify class.

```
TextView webURL = new TextView(this);
webURL.setText("https://www.google.com/");
Linkify.addLinks(webURL , Linkify.WEB_URLS);
```

Native Linkify Link Types

- The Linkify class has presets that can detect and linkify web URLs, email addresses, and phone numbers. To apply a preset, use the static Linkify.addLinks method, passing in a View to Linkify and a bitmask of one or more of the following self-describing Linkify class constants: WEB_URLS, EMAIL_ADDRESSES, PHONE_NUMBERS, and ALL.

```
TextView textView = (TextView)findViewById(R.id.myTextView);
Linkify.addLinks(textView, Linkify.WEB_URLS|Linkify.EMAIL_ADDRESSES);
```

- You can also linkify Views directly within a layout using the android:autoLink attribute. It sup- ports one or more of the following values: none, web, email, phone, and all.

```
<TextView android:layout_width=”match_parent”
android:layout_height=”match_parent”
android:text=”@string/linkify_me”
android:autoLink=”phone|email”
/>
```

Creating Custom Link Strings

- To linkify your own data, you need to define your own linkify strings. Do this by creating a new RegEx pattern that matches the text you want to display as hyperlinks. As with the native types, you can linkify the target Text View by calling Linkify.addLinks; however, rather than passing in one of the preset constants, pass in your RegEx pattern. You can also pass in a prefix that will be prepended to the target URI when a link is clicked.

Using the Match Filter

- To add additional conditions to RegEx pattern matches, implement the acceptMatch method in a Match Filter. When a potential match is found, acceptMatch is triggered, with the match start and end index (along with the full text being searched) passed in as parameters. Below code shows a MatchFilter implementation that cancels any match immediately preceded by an exclamation mark.

```
class MyMatchFilter implements MatchFilter
{
public boolean acceptMatch(CharSequence s, int start, int end)
        {
                return (start == 0 || s.charAt(start-1) != '!');
        }
}
```

Using the Transform Filter

- The Transform Filter lets you modify the implicit URI generated by matching link text. Decoupling the link text from the target URI gives you more freedom in how you display data strings to your users. To use the Transform Filter, implement the transformUrl method in your Transform Filter. When Linkify finds a successful match, it calls transformUrl, passing in the RegEx pattern used and the matched text string (before the base URI is prepended). You can modify the matched string and return it such that it can be appended to the base string as the data for a View Intent.

```
class MyTransformFilter implements TransformFilter
{
public String transformUrl(Matcher match, String url)
        {
        return url.toLowerCase().replace(" ", "");
        }
}
```

4. <u>Using Intents to Broadcast Events:</u>

- Intents to broadcast messages anonymously *between* components via the sendBroadcast method. As a system-level message-passing mechanism, Intents are capable of sending structured messages across process boundaries. As a result, you can implement Broadcast Receivers to listen for, and respond to, these Broadcast Intents within your applications.
- Broadcast Intents are used to notify applications of system or application events, extending the event-driven programming model between applications.
- Broadcasting Intents helps make your application more open; by broadcasting an event using Intent, you let yourself and third-party developers react to events without having to modify

your original application. Within your applications you can listen for Broadcast Intents to to react to device state changes and third-party application events.

- Android uses Broadcast Intents extensively to broadcast system events, such as changes in network connectivity, docking state, and incoming calls.

Broadcasting Events with Intents

- Within your application, construct the Intent you want to broadcast and call sendBroadcast to send it. Set the action, data, and category of your Intent in a way that lets Broadcast Receivers accurately determine their interest. In this scenario, the Intent action string is used to identify the event being broadcast, so it should be a unique string that identifies the event. By convention, action strings are constructed using the same form as Java package names:

```
public static final String NEW_LIFEFORM_DETECTED =
“com.paad.action.NEW_LIFEFORM”;
```

- If you want to include data within the Intent, you can specify a URI using the Intent’s data property. You can also include extras to add additional primitive values. Considered in terms of an event- driven paradigm, the extras equate to optional parameters passed into an event handler.

Listening for Broadcasts with Broadcast Receivers

- Broadcast Receivers (commonly referred to simply as Receivers) are used to listen for Broadcast Intents. For a Receiver to receive broadcasts, it must be registered, either in code or within the application manifest — the latter case is referred to as a manifest Receiver. In either case, use an Intent Filter to specify which Intent actions and data your Receiver is listening for.

- In the case of applications that include manifest Receivers, the applications don’t have to be running when the Intent is broadcast for those receivers to execute; they will be started automatically when a matching Intent is broadcast. This is excellent for resource management, as it lets you create event-driven applications that will still respond to broadcast events even after they’ve been closed or killed. To create a new Broadcast Receiver, extend the BroadcastReceiver class and override the onReceive event handler:

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
public class MyBroadcastReceiver extends BroadcastReceiver
{
@Override
public void onReceive(Context context, Intent intent)
{
//TODO: React to the Intent received. }}
```

- The onReceive method will be executed on the main application thread when a Broadcast Intent is received that matches the Intent Filter used to register the Receiver. The onReceive handler must complete within five seconds; otherwise, the Force Close dialog will be displayed.
- Typically, Broadcast Receivers will update content, launch Services, update Activity UI, or notify the user using the Notification Manager. The five-second execution limit ensures that major processing cannot, and should not, be done within the Broadcast Receiver itself.

Registering Broadcast Receivers in Code

- Broadcast Receivers that affect the UI of a particular Activity are typically registered in code. A Receiver registered programmatically will respond to Broadcast Intents only when the application component it is registered within is running.
- This is useful when the Receiver is being used to update UI elements in an Activity. In this case, it's good practice to register the Receiver within the onResume handler and unregister it during onPause.

Registering Broadcast Receivers in Your Application Manifest

- To include a Broadcast Receiver in the application manifest, add a receiver tag within the application node, specifying the class name of the Broadcast Receiver to register. The receiver node needs to include an intent-filter tag that specifies the action string being listened for.

```
<receiver android:name=".LifeformDetectedReceiver">
<intent-filter>
<action android:name="com.paad.alien.action.NEW_LIFEFORM"/>
</intent-filter>
</receiver>
```

- Broadcast Receivers registered this way are always active and will receive Broadcast Intents even when the application has been killed or hasn't been started.

Broadcasting Ordered Intents

- When the order in which the Broadcast Receivers receive the Intent is important — particularly where you want to allow Receivers to affect the Broadcast Intent received by future Receivers — you can use sendOrderedBroadcast, as follows:

```
String requiredPermission = "com.paad.MY_BROADCAST_PERMISSION";
sendOrderedBroadcast(intent, requiredPermission);
```

- Using this method, your Intent will be delivered to all registered Receivers that hold the required permission (if one is specified) in the order of their specified priority. You can specify the priority of a Broadcast Receiver using the android:priority attribute within its Intent Filter manifest node, where higher values are considered higher priority.

```
<receiver android:name=".MyOrderedReceiver"
android:permission="com.paad.MY_BROADCAST_PERMISSION">
<intent-filter android:priority="100">
<action android:name="com.paad.action.ORDERED_BROADCAST" />
</intent-filter>
</receiver>
```

- It's good practice to send ordered broadcasts, and specify Receiver priorities, only for Receivers used within your application that specifically need to impose a specific order of receipt.
- One common use-case for sending ordered broadcasts is to broadcast Intents for which you want to receive result data. Using the sendOrderedBroadcast method, you can specify a Broadcast Receiver that will be placed at the end of the Receiver queue, ensuring that it will receive the Broadcast Intent after it has been handled (and modified) by the ordered set of registered Broadcast Receivers.
- In this case, it's often useful to specify default values for the Intent result, data, and extras that may be modified by any of the Receivers that receive the broadcast before it is returned to the final result Receiver.

Broadcasting Sticky Intents

- Sticky Intents are useful variations of Broadcast Intents that persist the values associated with their last broadcast, returning them as Intent when a new Receiver is registered to receive the broadcast.
- When you call registerReceiver, specifying an Intent Filter that matches a sticky Broadcast Intent, the return value will be the last Intent broadcast, such as the battery-level changed broadcast:

```
IntentFilter battery = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);

Intent currentBatteryCharge = registerReceiver(null, battery);
```

- As shown in the preceding snippet, it is not necessary to specify a Receiver to obtain the current value of a sticky Intent. As a result, many of the system device state broadcasts (such as battery and docking state) use sticky Intents to improve efficiency. To broadcast your own sticky Intents, your application must have the BROADCAST_STICKY uses-per- mission before calling sendStickyBroadcast and passing in the relevant Intent:

```
sendStickyBroadcast(intent);
```

- To remove a sticky Intent, call removeStickyBroadcast, passing in the sticky Intent to remove:

```
removeStickyBroadcast(intent);
```

5. Introducing the Local Broadcast Manager:

- The Local Broadcast Manager was introduced to the Android Support Library to simplify the process of registering for, and sending, Broadcast Intents between components within your application.
- Because of the reduced broadcast scope, using the Local Broadcast Manager is more efficient than sending a global broadcast. It also ensures that the Intent you broadcast cannot be received by any components outside your application, ensuring that there is no risk of leaking private or sensitive data, such as location information.
- Similarly, other applications can’t transmit broadcasts to your Receivers, negating the risk of these Receivers becoming vectors for security exploits.
- Use the LocalBroadcastManager.getInstance method to return an instance of the Local Broadcast Manager:

```
LocalBroadcastManager lbm = LocalBroadcastManager.getInstance(this);
```

- To register a local broadcast Receiver, use the Local Broadcast Manager’s registerReceiver method, much as you would register a global receiver, passing in a Broadcast Receiver and an Intent Filter:

```
lbm.registerReceiver(new BroadcastReceiver() { @Override
public void onReceive(Context context, Intent intent) {
// TODO Handle the received local broadcast
}
}, new IntentFilter(LOCAL_ACTION));
```

- Note that the Broadcast Receiver specified can also be used to handle global Intent broadcasts. To transmit a local Broadcast Intent, use the Local Broadcast Manager’s sendBroadcast method, passing in the Intent to broadcast:

```
lbm.sendBroadcast(new Intent(LOCAL_ACTION));
```

- The Local Broadcast Manager also includes a sendBroadcastSync method that operates synchronously, blocking until each registered Receiver has been dispatched.

6. Introducing Pending Intents:

- The PendingIntent class provides a mechanism for creating Intents that can be fired on your application’s behalf by another application at a later time. A Pending Intent is commonly used to package Intents that will be fired in response to a future event, such as a Widget or Notification being clicked.

- The PendingIntent class offers static methods to construct Pending Intents used to start an Activity, to start a Service, or to broadcast an Intent:

```
int requestCode = 0; int flags = 0;
// Start an Activity
Intent startActivityIntent = new Intent(this, MyOtherActivity.class);
PendingIntent.getActivity(this, requestCode, startActivityIntent, flags);

// Start a Service
Intent startServiceIntent = new Intent(this, MyService.class);
PendingIntent.getService(this, requestCode, startServiceIntent , flags);

// Broadcast an Intent
Intent broadcastIntent = new Intent(NEW_LIFEFORM_DETECTED);
PendingIntent.getBroadcast(this, requestCode, broadcastIntent, flags);
```

- The PendingIntent class includes static constants that can be used to specify flags to update or cancel any existing Pending Intent that matches your specified action, as well as to specify if this Intent is to be fired only once.

Creating Intent Filters and Broadcast Receivers

- Having learned to use Intents to start Activities/Services and to broadcast events, it's important to understand how to create the Broadcast Receivers and Intent Filters that listen for Broadcast Intents and allow your application to respond to them.
- In the case of Activities and Services, an Intent is a request for an action to be performed on a set of data, and an Intent Filter is a declaration that a particular application component is capable of per- forming an action on a type of data. Intent Filters are also used to specify the actions a Broadcast Receiver is interested in receiving.

7. <u>Using Intent Filters to Service Implicit Intents:</u>

- If an Intent is a request for an action to be performed on a set of data, how does Android know which application (and component) to use to service that request? Using Intent Filters, application components can declare the actions and data they support.
- To register an Activity or Service as a potential Intent handler, add an intent-filter tag to its manifest node using the following tags (and associated attributes):
    - ❖ action — Uses the android:name attribute to specify the name of the action being serviced. Each Intent Filter must have at least one action tag. Actions should be unique strings that are self-describing. Best practice is to use a naming system based on the Java package naming conventions.

- category — Uses the android:name attribute to specify under which circumstances the action should be serviced. Each Intent Filter tag can include multiple category tags. You can specify your own categories or use the following standard values provided by Android:
    - ALTERNATIVE — This category specifies that this action should be available as an alternative to the default action performed on an item of this data type. For example, where the default action for a contact is to view it, the alternative could be to edit it.
    - SELECTED_ALTERNATIVE — Similar to the ALTERNATIVE category, but whereas that category will always resolve to a single action using the intent resolution described next, SELECTED_ALTERNATIVE is used when a list of possibilities is required. As you'll see later in this chapter, one of the uses of Intent Filters is to help populate context menus dynamically using actions.
    - BROWSABLE — Specifies an action available from within the browser. When Intent is fired from within the browser, it will always include the browsable category.
    - If you want your application to respond to actions triggered within the browser (e.g., intercepting links to a particular website), you must include the browsable category.
    - DEFAULT — Set this to make a component the default action for the data type specified in the Intent Filter. This is also necessary for Activities that are launched using an explicit Intent.
    - HOME — By setting an Intent Filter category as home without specifying an action, you are presenting it as an alternative to the native home screen.
    - LAUNCHER — Using this category makes an Activity appear in the application launcher.
- data — The data tag enables you to specify which data types your component can act on; you can include several data tags as appropriate. You can use any combination of the following attributes to specify the data your component supports:
    - android:host — Specifies a valid hostname (e.g., google.com).
    - android:mimetype — Specifies the type of data your component is capable of han- dling. For example, <type android:value="vnd.android.cursor.dir/*"/> would match any Android cursor.
    - android:path — Specifies valid path values for the URI (e.g., /transport/boats/).

    - android:port — Specifies valid ports for the specified host.
    - android:scheme — Requires a particular scheme (e.g., content or http).

- The following snippet shows an Intent Filter for an Activity that can perform the SHOW_DAMAGE action as either a primary or an alternative action based on its mime type.

```
<intent-filter>
<action android:name=”com.paad.earthquake.intent.action.SHOW_DAMAGE”
/>
<category android:name=”android.intent.category.DEFAULT”/>
<category android:name=”android.intent.category.SELECTED_ALTERNATIVE”/>
<data android:mimeType=”vnd.earthquake.cursor.item/*”/>
</intent-filter>
```

- You may have noticed that clicking a link to a YouTube video or Google Maps location on an Android device prompts you to use YouTube or Google Maps, respectively, rather than the browser. This is achieved by specifying the scheme, host, and path attributes within the data tag of an Intent Filter.

8. **Using Intent Filters for Plug-Ins and Extensibility:**

- Having used Intent Filters to declare the actions your Activities can perform on different types of data, it stands to reason that applications can also query to find which actions are available to be performed on a particular piece of data.
- Android provides a plug-in model that lets your applications take advantage of functionality, pro- vided anonymously from your own or third-party application components you haven’t yet conceived of, without your having to modify or recompile your projects.

Supplying Anonymous Actions to Applications

- To use this mechanism to make your Activity’s actions available anonymously for existing applications, publish them using intent-filter tags within their manifest nodes, as described earlier.
- The Intent Filter describes the action it performs and the data upon which it can be performed. The latter will be used during the intent-resolution process to determine when this action should be available. The category tag must be either ALTERNATIVE or SELECTED_ALTERNATIVE, or both. The android:label attribute should be a human-readable label that describes the action.

```
<activity android:name=”.NostromoController”>
<intent-filter android:label=”@string/Nuke_From_Orbit”>
<action android:name=”com.pad.nostromo.NUKE_FROM_ORBIT”/>
<data android:mimeType=”vnd.moonbase.cursor.item/*”/>
<category android:name=”android.intent.category.ALTERNATIVE”/>
```

```
<category android:name=”android.intent.category.SELECTED_ALTERNATIVE”
/>
</intent-filter>
</activity>
```

Discovering New Actions from Third-Party Intent Receivers

- Using the Package Manager, you can create an Intent that specifies a type of data and a category of action, and have the system return a list of Activities capable of performing an action on that data.
- The elegance of this concept is best explained by an example. If the data your Activity displays is a list of places, you might include functionality to View them on a map or “Show directions
- to” each. Jump a few years ahead and you’ve created an application that interfaces with your car, allowing your phone to handle driving. Thanks to the runtime menu generation, when a new Intent Filter — with a DRIVE_CAR action — is included within the new Activity’s node, Android will resolve this new action and make it available to your earlier application.
- This provides you with the ability to retrofit functionality to your application when you create new components capable of performing actions on a given type of data. Many of Android’s native appli- cations use this functionality, enabling you to provide additional actions to native Activities.
- The Intent you create will be used to resolve components with Intent Filters that supply actions for the data you specify. The Intent is being used to find actions, so don’t assign it one; it should specify only the data to perform actions on. You should also specify the category of the action, either CATEGORY_ALTERNATIVE or CATEGORY_SELECTED_ALTERNATIVE. The skeleton code for creating an Intent for menu-action resolution is shown here:

```
Intent intent = new Intent();
intent.setData(MyProvider.CONTENT_URI);
intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
```

- Pass this Intent into the Package Manager method queryIntentActivityOptions, specifying any options flags.

Incorporating Anonymous Actions as Menu Items

- The most common way to incorporate actions from third-party applications is to include them within your Menu Items or Action Bar Actions. The addIntentOptions method, available from the Menu class, lets you specify an Intent that describes the data acted upon within your Activity, as described previously; however, rather than simply returning a list of possible Receivers, a new Menu Item will be created for each, with the text populated from the matching Intent Filters' labels.

- To add Menu Items to your Menus dynamically at run time, use the addIntentOptions method on the Menu object in question: Pass in an Intent that specifies the data for which you want to provide actions. Generally, this will be handled within your Activities' onCreateOptionsMenu or onCreate- ContextMenu handlers.
- As in the previous section, the Intent you create will be used to resolve components with Intent Filters that supply actions for the data you specify. The Intent is being used to find actions, so don't assign it one; it should specify only the data to perform actions on. You should also specify the cat- egory of the action, either CATEGORY_ALTERNATIVE or CATEGORY_SELECTED_ALTERNATIVE. The skeleton code for creating Intent for menu-action resolution is shown here:

```
Intent intent = new Intent();
intent.setData(MyProvider.CONTENT_URI);
intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
```

- Pass this Intent in to addIntentOptions on the Menu you want to populate, as well as any options flags, the name of the calling class, the Menu group to use, and the Menu ID values. You can also specify an array of Intents you'd like to use to create additional Menu Items.

9. <u>Listening for Native Broadcast Intents:</u>

- Many of the system Services broadcast Intents to signal changes. You can use these messages to add functionality to your own projects based on system events, such as time-zone changes, data-connection status, incoming SMS messages, or phone calls.
- The following list introduces some of the native actions exposed as constants in the Intent class; these actions are used primarily to track device status changes:
    - ❖ ACTION_BOOT_COMPLETED — Fired once when the device has completed its startup sequence. An application requires the RECEIVE_BOOT_COMPLETED permission to receive this broadcast.
    - ❖ ACTION_CAMERA_BUTTON — Fired when the camera button is clicked.
    - ❖ ACTION_DATE_CHANGED and ACTION_TIME_CHANGED — These actions are broadcast if the date or time on the device is manually changed (as opposed to changing through the inexo- rable progression of time).
    - ❖ ACTION_MEDIA_EJECT — If the user chooses to eject the external storage media, this event is fired first. If your application is reading or writing to the external media storage, you should listen for this event to save and close any open file handles.

- ❖ ACTION_MEDIA_MOUNTED and ACTION_MEDIA_UNMOUNTED — These two events are broadcast whenever new external storage media are successfully added to or removed from the device, respectively.
- ❖ ACTION_NEW_OUTGOING_CALL — Broadcast when a new outgoing call is about to be placed. Listen for this broadcast to intercept outgoing calls. The number being dialed is stored in the EXTRA_PHONE_NUMBER extra, whereas the resultData in the returned Intent will be the num- ber actually dialed. To register a Broadcast Receiver for this action, your application must declare the PROCESS_OUTGOING_CALLS uses-permission.
- ❖ ACTION_SCREEN_OFF and ACTION_SCREEN_ON — Broadcast when the screen turns off or on, respectively.
- ❖ ACTION_TIMEZONE_CHANGED — This action is broadcast whenever the phone's current time zone changes. The Intent includes a time-zone extra that returns the ID of the new java. util.TimeZone.

- Android also uses Broadcast Intents to announce application-specific events, such as incoming SMS messages, changes in dock state, and battery level. The actions and Intents associated with these events will be discussed in more detail in later chapters when you learn more about the associated Services.

10. <u>Monitoring Device State Changes Using Broadcast Intents:</u>

- Monitoring the device state is an important part of creating efficient and dynamic applications whose behavior can change based on connectivity, battery charge state, and docking status.
- Android broadcasts Intents for changes in each of these device states. The following sections examine how to create Intent Filters to register Broadcast Receivers that can react to such changes, and how to extract the device state information accordingly.

Listening for Battery Changes

- To monitor changes in the battery level or charging status within an Activity, you can register a Receiver using an Intent Filter that listens for the Intent.ACTION_BATTERY_CHANGED broadcast by the Battery Manager.
- The Broadcast Intent containing the current battery charge and charging status is a sticky Intent, so you can retrieve the current battery status at any time without needing to implement a Broadcast Receiver.

```
IntentFilter batIntentFilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent battery = context.registerReceiver(null, batIntentFilter);
int status = battery.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging =
status == BatteryManager.BATTERY_STATUS_CHARGING ||
status == BatteryManager.BATTERY_STATUS_FULL;
```

- Note that you can’t register the battery changed action within a manifest Receiver; however, you can monitor connection and disconnection from a power source and a low battery level using the following action strings, each prefixed with android.intent.action:
    - ACTION_BATTERY_LOW
    - ACTION_BATTERY_OKAY
    - ACTION_POWER_CONNECTED
    - ACTION_POWER_DISCONNECTED

Listening for Connectivity Changes

- Changes in connectivity, including the bandwidth, latency, and availability of an Internet connec- tion, can be significant signals for your application. In particular, you might choose to suspend recurring updates when you lose connectivity or to delay downloads of significant size until you have a Wi-Fi connection.
- To monitor changes in connectivity, register a Broadcast Receiver (either within your appli-cation or within the manifest) to listen for the android.net.conn.CONNECTIVITY_CHANGE (ConnectivityManager.CONNECTIVITY_ACTION) action.
- The connectivity change broadcast isn’t sticky and doesn’t contain any additional information regarding the change. To extract details on the current connectivity status, you need to use the Connectivity Manager.

```
String svcName = Context.CONNECTIVITY_SERVICE;
ConnectivityManager cm = (ConnectivityManager)context.getSystemService(svcName);
NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
boolean isConnected = activeNetwork.isConnectedOrConnecting();
boolean isMobile = activeNetwork.getType() ==
ConnectivityManager.TYPE_MOBILE;
```

Listening for Docking Changes

- Android devices can be docked in either a car dock or desk dock. These, in term, can be either analog or digital docks. By registering a Receiver to listen for the Intent.ACTION_DOCK_EVENT (android. intent.action.ACTION_DOCK_EVENT), you can determine the docking status and type of dock.
- Like the battery status, the dock event Broadcast Intent is sticky. Below code shows how to extract the current docking status from the Intent returned when registering a Receiver for docking events.

```
IntentFilter dockIntentFilter =
new IntentFilter(Intent.ACTION_DOCK_EVENT);
```

```
Intent dock = registerReceiver(null, dockIntentFilter);
int dockState = dock.getIntExtra(Intent.EXTRA_DOCK_STATE,
Intent.EXTRA_DOCK_STATE_UNDOCKED);
boolean isDocked = dockState != Intent.EXTRA_DOCK_STATE_UNDOCKED;
```

Managing Manifest Receivers at Run Time

- Using the Package Manager, you can enable and disable any of your application's manifest Receivers at run time using the setComponentEnabledSetting method. You can use this technique to enable or disable any application component (including Activities and Services), but it is particularly useful for manifest Receivers.
- To minimize the footprint of your application, it's good practice to disable manifest Receivers that listen for common system events (such as connectivity changes) when your application doesn't need to respond to those events. This technique also enables you to schedule an action based on a system event — such as downloading a large file when the device is connected to Wi-Fi — without gaining the overhead of having the application launch every time a connectivity change is broadcast. The below code shows how to enable and disable a manifest Receiver at run time.

```
ComponentName myReceiverName = new ComponentName(this, MyReceiver.class);
PackageManager pm = getPackageManager();

// Enable a manifest receiverpm.setComponentEnabledSetting(myReceiverName,
PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
PackageManager.DONT_KILL_APP);

// Disable a manifest receiver pm.setComponentEnabledSetting(myReceiverName,

PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
PackageManager.DONT_KILL_APP);
```

****

UNIT - IV

Files , Saving State And Preferences: Saving Simple Application Data - creating and Saving Shared Preferences - Retrieving Shared Preferences – Introducing the Preference Framework and the Preference Activity – Working with the File System – Introducing Android Databases - Introducing SQLite – Content Values and Cursors – Working with SQLite Databases - Creating Content Providers, Using Content Providers

----

1. INTRODUCTION:

Saving and loading data is essential for most applications. At a minimum, an Activity should save its user interface (UI) state before it becomes inactive to ensure the same UI is presented when it restarts. It's also likely that you'll need to save user preferences and UI selections.

Android's nondeterministic Activity and application lifetimes make persisting UI state and application data between sessions particularly important, as your application process may have been killed and restarted before it returns to the foreground. Android offers several alternatives for saving application data, each optimized to fulfill a particular need.

Shared Preferences are a simple, lightweight name/value pair (NVP) mechanism for saving primitive application data, most commonly a user's application preferences. Android also offers a mechanism for recording application state within the Activity lifecycle handlers, as well as for providing access to the local file system, through both specialized methods and the java.io classes.

Android also offers a rich framework for user preferences, allowing you to create settings screens consistent with the system settings.

2. SAVING SIMPLE APPLICATION DATA:

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

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

There are two lightweight techniques for saving simple application data for Android applications:

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

3. <u>CREATING AND SAVING SHARED PREFERENCES:</u>

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

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

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

Shared Preferences are stored within the application's sandbox, so they can be shared between an application's components but aren't available to other applications.

To modify a Shared Preference, use the SharedPreferences.Editor class. Get the Editor object by calling edit on the Shared Preferences object you want to change.

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

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

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

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

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

4. <u>RETRIEVING SHARED PREFERENCES:</u>

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

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

```
// Retrieve the saved values.
boolean isTrue = mySharedPreferences.getBoolean("isTrue", false);
float lastFloat = mySharedPreferences.getFloat("lastFloat", 0f);
int wholeNumber = mySharedPreferences.getInt("wholeNumber", 1);
```

```
long aNumber = mySharedPreferences.getLong("aNumber", 0);
String stringPreference =
mySharedPreferences.getString("textEntryValue", "");
```

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

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

5. INTRODUCING THE PREFERENCE FRAMEWORK AND THE PREFERENCE ACTIVITY:

Android offers an XML-driven framework to create system-style Preference Screens for your applications. By using this framework you can create Preference Activities that are consistent with those used in both native and other third-party applications.

This has two distinct advantages:

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

The preference framework consists of four parts:

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

5.1 Defining a Preference Screen Layout in XML:

Unlike in the standard UI layout, preference definitions are stored in the res/xml resources folder.

Each preference layout is defi ned as a hierarchy, beginning with a single PreferenceScreen element:

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

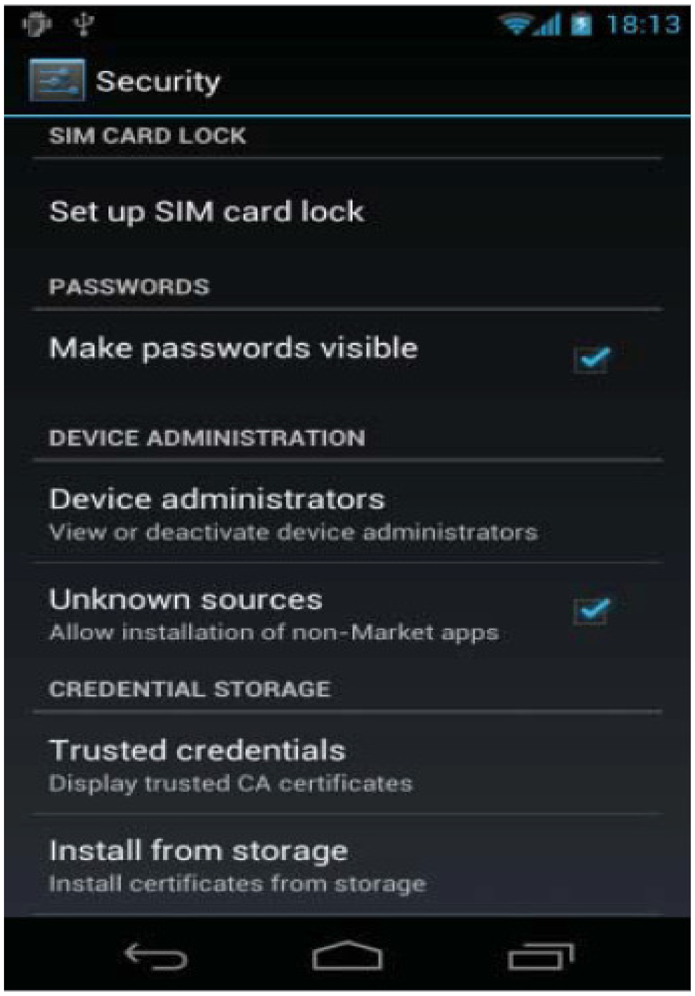
```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">
</PreferenceScreen>
```

You can include additional Preference Screen elements, each of which will be represented as a selectable element that will display a new screen when clicked.

Within each Preference Screen you can include any combination of PreferenceCategory and Preference<control> elements. Preference Category elements, as shown in the following snippet, are used to break each Preference Screen into subcategories using a title bar separator:

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

Figure shows the SIM card lock, device administration, and credential storage Preference Categories used on the Security Preference Screen.



SIM CARD LOCK

Set up SIM card lock

PASSWORDS

Make passwords visible ✓

DEVICE ADMINISTRATION

Device administrators
View or deactivate device administrators

Unknown sources ✓
Allow installation of non-Market apps

CREDENTIAL STORAGE

Trusted credentials
Display trusted CA certificates

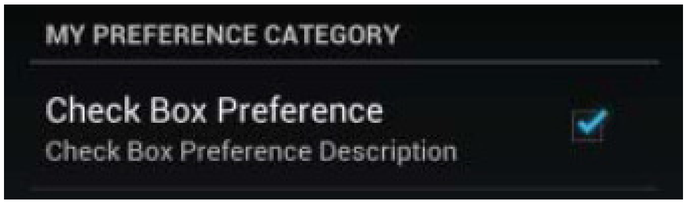
Install from storage
Install certificates from storage

All that remains is to add the preference controls that will be used to set the preferences. Although the specific attributes available for each preference control vary, each of them includes at least the following four:

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

<u>A simple Shared Preferences screen:</u>

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



When displayed, this Preference Screen will appear as shown above.

5.2 <u>Native Preference Controls:</u>

Android includes several preference controls to build your Preference Screens:

- ✓ *CheckBoxPreference* - A standard preference check box control used to set preferences to true or false.
- ✓ *EditTextPreference* - Allows users to enter a string value as a preference. Selecting the preference text at run time will display a text-entry dialog.
- ✓ *ListPreference* - The preference equivalent of a spinner. Selecting this preference will display a dialog box containing a list of values from which to select. You can specify different arrays to contain the display text and selection values.

- ✓ *MultiSelectListPreference* - Introduced in Android 3.0 (API level 11), this is the preference equivalent of a check box list.
- ✓ *RingtonePreference* - A specialized List Preference that presents the list of available ringtones for user selection. This is particularly useful when you're constructing a screen to configure notification settings.

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

5.3 <u>Using Intents to Import System Preferences into Preference Screens:</u>

In addition to including your own Preference Screens, preference hierarchies can include Preference Screens from other applications - including system preferences.

You can invoke any Activity within your Preference Screen using an Intent. If you add an Intent node within a Preference Screen element, the system will interpret this as a request to call startActivity using the specified action. The following XML snippet adds a link to the system display settings:

```
<?xml version=”1.0” encoding=”utf-8”?>
<PreferenceScreen xmlns:android=”http://schemas.android.com/apk/res/android”
android:title=”Intent preference”
android:summary=”System preference imported using an intent”>
<intent android:action=”android.settings.DISPLAY_SETTINGS “/>
</PreferenceScreen>
```

The android.provider.Settings class includes a number of android.settings.* constants that can be used to invoke the system settings screens. To make your own Preference Screens available for invocation using this technique, simply add an Intent Filter to the manifest entry for the host Preference Activity.

```
<activity android:name=”.UserPreferences” android:label=”My User Preferences”>
<intent-filter>
<action android:name=”com.paad.myapp.ACTION_USER_PREFERENCE” />
</intent-filter>
</activity>
```

5.4 <u>Introducing the Preference Fragment:</u>

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

```
public class MyPreferenceFragment extends PreferenceFragment
```

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

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

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

5.5 Defining the Preference Fragment Hierarchy Using Preference Headers:

Preference headers are XML resources that describe how your Preference Fragments should be grouped and displayed within a Preference Activity. Each header identifies and allows you to select a particular Preference Fragment.

The layout used to display the headers and their associated Fragments can vary depending on the screen size and OS version. Figure 7-4 shows examples of how the same Preference Header definition is displayed on a phone and tablet.

Preference Headers are XML resources stored in the res/xml folder of your project hierarchy. The resource ID for each header is the filename (without extension).

Defining a Preference Headers resource

```
<preference-headers xmlns:android=”http://schemas.android.com/apk/res/android”>
<header android:fragment=”com.paad.preferences.MyPreferenceFragment”
android:icon=”@drawable/preference_icon”
android:title=”My Preferences”
android:summary=”Description of these preferences” />
</preference-headers>
```

Like Preference Screens, you can invoke any Activity within your Preference Headers using an Intent. If you add an Intent node within a header element, as shown in the following snippet, the system will interpret this as a request to call startActivity using the specified action:

```
<header android:icon=”@drawable/ic_settings_display”
android:title=”Intent”
android:summary=”Launches an Intent.”>
<intent android:action=”android.settings.DISPLAY_SETTINGS “/>
</header>
```

5.6 Introducing the Preference Activity:

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

To create a new Preference Activity, extend the PreferenceActivity class as follows:

```
public class MyFragmentPreferenceActivity extends PreferenceActivity
```

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

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

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

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

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

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

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

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

5.7 <u>Backward Compatibility and Preference Screens:</u>

The Preference Fragment and associated Preference Headers are not supported on Android platforms prior to Android 3.0 (API level 11). As a result, if you want to create applications that support devices running on both pre- and post-Honeycomb devices, you need to implement separate Preference Activities to support both, and launch the appropriate Activity at run time, as shown below.

Runtime selection of pre- or post-Honeycomb Preference Activities

```
Class c = Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB ?
MyPreferenceActivity.class : MyFragmentPreferenceActivity.class;
Intent i = new Intent(this, c);
startActivityForResult(i, SHOW_PREFERENCES);
```

5.8 <u>Finding and Using the Shared Preferences Set by Preference Screens:</u>

The Shared Preference values recorded for the options presented in a Preference Activity are stored within the application’s sandbox. This lets any application component, including

Activities, Services, and Broadcast Receivers, access the values, as shown in the following snippet:

```
Context context = getApplicationContext();
SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(context);
// TODO Retrieve values using get<type> methods.
```

5.9 <u>Introducing On Shared Preference Change Listeners:</u>

The onSharedPreferenceChangeListener can be implemented to invoke a callback whenever a particular Shared Preference value is added, removed, or modified.

This is particularly useful for Activities and Services that use the Shared Preference framework to set application preferences. Using this handler, your application components can listen for changes to user preferences and update their UIs or behavior, as required.

Register your On Shared Preference Change Listeners using the Shared Preference you want to monitor:

```
public class MyActivity extends Activity implements
OnSharedPreferenceChangeListener {
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
// Register this OnSharedPreferenceChangeListener
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(this);
prefs.registerOnSharedPreferenceChangeListener(this);
}
public void onSharedPreferenceChanged(SharedPreferences prefs,
String key) {
// TODO Check the shared preference and key parameters
// and change UI or behavior as appropriate.
}
}
```

6. <u>WORKING WITH THE FILE SYSTEM:</u>

It’s good practice to use Shared Preferences or a database to store your application data, but there may still be times when you’ll want to use fi les directly rather than rely on Android’s managed mechanisms - particularly when working with multimedia files.

6.1 <u>File-Management Tools:</u>

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

- *deleteFile* - Enables you to remove fi les created by the current application

- *fileList* - Returns a string array that includes all the fi les created by the current application

These methods are particularly useful for cleaning up temporary fi les left behind if your application crashes or is killed unexpectedly.

6.2 Using Application-Specific Folders to Store Files:

Many applications will create or download fi les that are specifi c to the application. There are two options for storing these application-specifi c fi les: internally or externally.

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

The most important thing to remember when storing files on external storage is that no security is enforced on files stored here. Any application can access, overwrite, or delete files stored on the external storage.

Android offers two corresponding methods via the application Context, getDir and getExternalFilesDir, both of which return a File object that contains the path to the internal and external application fi le storage directory, respectively.

All files stored in these directories or the subfolders will be erased when your application is uninstalled.

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

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

6.3 Creating Private Application Files:

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

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

These methods support only those files in the current application folder; specifying path separators will cause an exception to be thrown.

If the filename you specify when creating a FileOutputStream does not exist, Android will create it for you. The default behavior for existing files is to overwrite them; to append an existing file, specify the mode as Context.MODE_APPEND.

By default, files created using the openFileOutput method are private to the calling application - a different application will be denied access. The standard way to share a file between applications is to use a Content Provider. Alternatively, you can specify either Context.MODE_WORLD_READABLE or Context.MODE_WORLD_WRITEABLE when creating the output file, to make it available in other applications, as shown in the following snippet:

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

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

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

6.4 Using the Application File Cache:

Should your application need to cache temporary files, Android offers both a managed internal cache, and (since Android API level 8) an unmanaged external cache. You can access them by calling the getCacheDir and getExternalCacheDir methods, respectively, from the current Context.

Files stored in either cache location will be erased when the application is uninstalled. Files stored in the internal cache will potentially be erased by the system when it is running low on available storage; files stored on the external cache will not be erased, as the system does not track available storage on external media.

In either case it’s good form to monitor and manage the size and age of your cache, deleting files when a reasonable maximum cache size is exceeded.

6.5 Storing Publicly Readable Files:

Android 2.2 (API level 8) also includes a convenience method, Environment.getExternal StoragePublicDirectory, that can be used to find a path in which to store your application files. The returned location is where users will typically place and manage their own files of each type.

This is particularly useful for applications that provide functionality that replaces or augments system applications, such as the camera, that store files in standard locations. The getExternalStoragePublicDirectory method accepts a String parameter that determines which subdirectory you want to access using a series of Environment static constants:

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

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

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

7. <u>INTRODUCING ANDROID DATABASES:</u>

Android provides structured data persistence through a combination of SQLite databases and Content Providers.

SQLite databases can be used to store application data using a managed, structured approach. Android offers a full SQLite relational database library. Every application can create its own databases over which it has complete control.

Content Providers offer a generic, well-defined interface for using and sharing data that provides a consistent abstraction from the underlying data source.

7.1 SQLite Databases:

Using SQLite you can create fully encapsulated relational databases for your applications. Use them to store and manage complex, structured application data.

Android databases are stored in the /data/data/<package_name>/databases folder on your device (or emulator). All databases are private, accessible only by the application that created them.

In particular, when you're creating databases for resource-constrained devices (such as mobile phones), it's important to normalize your data to minimize redundancy.

7.2 Content Providers:

Content Providers provide an interface for publishing and consuming data, based around a simple URI addressing model using the content:// schema. They enable you to decouple your application layers from the underlying data layers, making your applications data-source agnostic by abstracting the underlying data source.

Content Providers can be shared between applications, queried for results, have their existing records updated or deleted, and have new records added. Any application - with the appropriate permissions - can add, remove, or update data from any other application, including the native Android Content Providers.

Several native Content Providers have been made accessible for access by third-party applications, including the contact manager, media store, and calendar.

8. INTRODUCING SQLITE:

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.

Each SQLite database is an integrated part of the application that created it. This reduces external dependencies, minimizes latency, and simplifies transaction locking and synchronization.

SQLite has a reputation for being extremely reliable and is the database system of choice for many consumer electronic devices, including many MP3 players and smartphones.

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; instead, 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.

9. 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. Cursors provide a managed way of controlling your position (row) in the result set of a database query.

The Cursor class includes a number of navigation functions, including, but not limited to, the following:

- ✓ *moveToFirst* - Moves the cursor to the fi rst 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
- ✓ *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

Android provides a convenient mechanism to ensure queries are performed asynchronously. The CursorLoader class were introduced in Android 3.0 (API level 11) and are now also available as part of the support library.

10. WORKING WITH SQLITE DATABASES:

When working with databases, it's good form to encapsulate the underlying database and expose only the public methods and constants required to interact with that database, generally using what's often referred to as a contract or helper class. This class should expose database constants, particularly column names, which will be required for populating and querying the database.

10.1 SQLiteOpenHelper:

SQLiteOpenHelper is an abstract class used to implement the best practice pattern for creating, opening, and upgrading databases.

The SQLite Open Helper caches database instances after they have been successfully opened, so make requests to open the database immediately prior to performing a query or transaction.

For the same reason, there is no need to close the database manually unless you no longer need to use it again.

Database operations, especially opening or creating databases, can be time consuming. To ensure this doesn’t impact the user experience, make all database transactions asynchronous.

Implementing an SQLite Open Helper

```
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;
// SQL Statement to create a new database.
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);
}
// Called when no database exists in disk and the helper class needs
// to create a new one.
@Override
public void onCreate(SQLiteDatabase db) {
db.execSQL(DATABASE_CREATE);
}
// Called when there is a database version mismatch meaning that
// the version of the database on disk needs to be upgraded to
// the current version.
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
// Log the version upgrade.
Log.w(“TaskDBAdapter”, “Upgrading from version “ +
oldVersion + “ to “ +
newVersion + “, which will destroy all old data”);
// Upgrade the existing database to conform to the new
// version. Multiple previous versions can be handled by
// comparing oldVersion and newVersion values.
// The simplest case is to drop the old table and create a new one.
db.execSQL(“DROP TABLE IF IT EXISTS “ + DATABASE_TABLE);
// Create a new one.
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.

10.2 <u>Opening and Creating Databases Without the SQLite Open Helper:</u>

Use the application Context's openOrCreateDatabase method to create the database itself:

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

After creating the database, must handle the creation and upgrade logic handled within the onCreate and onUpgrade handlers of the SQLite Open Helper.

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.

At a minimum, any such operations must be handled asynchronously to avoid impacting the main application thread.

10.3 <u>Querying a Database:</u>

Each database query is returned as a Cursor. 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.
- ❖ An array of selection argument strings
- ❖ 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.

```
// Specify the result column projection. Return the minimum set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
KEY_ID, KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
KEY_GOLD_HOARDED_COLUMN };
```

```
// Specify the where clause that will limit our results.
String where = KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + “=” + 1;
// Replace these with valid SQL statements as necessary.
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);
```

10.4 <u>Extracting Values from a Cursor:</u>

To extract values from a Cursor, first use the moveTo<location> methods described earlier to position the cursor at the correct row of the result Cursor, and then use the type-safe get<type> methods (passing in a column index) 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;
// Find the index to the column(s) being used.
int GOLD_HOARDED_COLUMN_INDEX =
cursor.getColumnIndexOrThrow(KEY_GOLD_HOARDED_COLUMN);
// Iterate over the cursors rows.
// The Cursor is initialized at before first, so we can
// check only if there is a “next” row available. If the
// result Cursor is empty this will return false.
while (cursor.moveToNext()) {
float hoard = cursor.getFloat(GOLD_HOARDED_COLUMN_INDEX);
totalHoard += hoard;
}
// Calculate an average -- checking for divide by zero errors.
float cursorCount = cursor.getCount();
averageHoard = cursorCount > 0 ?
(totalHoard / cursorCount) : Float.NaN;
// Close the Cursor when you’ve finished with it.
cursor.close();
```

Because SQLite database columns are loosely typed, you can cast individual values into valid types, as required. For example, values stored as fl oats can be read back as strings. When you have finished using your result Cursor, it’s important to close it to avoid memory leaks and reduce your application’s resource load:

cursor.close();

10.5 <u>Adding, Updating, and Removing Rows:</u>

The SQLiteDatabase class exposes insert, delete, and update methods that encapsulate the SQL statements required to perform these actions. Additionally, the execSQL method executes any valid SQL statement on database tables.

Any time of modification the underlying database values, should update Cursors by running a new query.

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 want to update.

Call the update method on the database, passing in the table name, the updated Content Values object, and a where clause that specifies the rows 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, where,
whereArgs);
```

Deleting Rows

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

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

11. CREATING CONTENT PROVIDERS:

Content Providers provide an interface for publishing data that will be consumed using a Content Resolver. They allow you to decouple the application components that consume data from their underlying data sources, providing a generic mechanism through which applications can share their data or consume data provided by others.

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

```
public class MyContentProvider extends ContentProvider
```

It's good practice to include static database constants - particularly column names and the Content Provider authority - that will be required for transacting with, and querying, the database.

You will also need to override the onCreate handler to initialize the underlying data source, as well as the query, update, delete, insert, and getType methods to implement the interface used by the Content Resolver to interact with the data.

11.1 Registering Content Providers:

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.

Use the authorities tag to define the base URI of the Provider's authority. A Content Provider's authority is used by the Content Resolver as an address and used to find the database you want to interact with.

Each Content Provider authority must be unique, so it's good practice to base the URI path on your package name. 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 show in the following XML snippet:

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

11.2 Creating the Content Provider's Database:

To initialize the data source you plan to access through the Content Provider, override the onCreate method. This is typically handled using an SQLite Open Helper implementation, allowing you to effectively defer creating and opening the database until it's required.

```
private MySQLiteOpenHelper myOpenHelper;
@Override
public boolean onCreate() {
// Construct the underlying database.
// Defer opening the database until you need to perform
// a query or transaction.
myOpenHelper = new MySQLiteOpenHelper(getContext(),
MySQLiteOpenHelper.DATABASE_NAME, null,
MySQLiteOpenHelper.DATABASE_VERSION);
return true;
}
```

11.3 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. The most common scenario is to use a Content Provider to provide access to an SQLite database, but within these methods you can access any source of data.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
// Open the database.
SQLiteDatabase db;
try {
db = myOpenHelper.getWritableDatabase();
} catch (SQLiteException ex) {
db = myOpenHelper.getReadableDatabase();
}
// Replace these with valid SQL statements if necessary.
String groupBy = null;
String having = null;
// Use an SQLite Query Builder to simplify constructing the
// database query.
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
// If this is a row query, limit the result set to the passed in row.
switch (uriMatcher.match(uri)) {
case SINGLE_ROW :
String rowID = uri.getPathSegments().get(1);
queryBuilder.appendWhere(KEY_ID + "=" + rowID);
default: break;
}
// Specify the table on which to perform the query. This can
// be a specific table or a join as required.
queryBuilder.setTables(MySQLiteOpenHelper.DATABASE_TABLE);
// Execute the query.
Cursor cursor = queryBuilder.query(db, projection, selection,
selectionArgs, groupBy, having, sortOrder);
// Return the result Cursor.
return cursor;
}
```

Having implemented queries, you must also specify a MIME type to identify the data returned. Override the getType method to return a string that uniquely describes your data type.

The type returned should include two forms, one for a single entry and another for all the entries, following these forms:

- ❖ Single item:

  vnd.android.cursor.item/vnd.<companyname>.<contenttype>

- ❖ All items:

  vnd.android.cursor.dir/vnd.<companyname>.<contenttype>

11.4 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.

When performing transactions that modify the dataset, it's good practice to call the Content Resolver's notifyChange method. This will notify any Content Observers, registered for a given Cursor using the Cursor.registerContentObserver method, that the underlying table (or a particular row) has been removed, added, or updated.

As with Content Provider queries, the most common use case is performing transactions on an SQLite database, though this is not a requirement.

Typical Content Provider transaction implementations

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
// Open a read / write database to support the transaction.
SQLiteDatabase db = myOpenHelper.getWritableDatabase();
// If this is a row URI, limit the deletion to the specified row.
switch (uriMatcher.match(uri)) {
case SINGLE_ROW :
String rowID = uri.getPathSegments().get(1);
selection = KEY_ID + "=" + rowID
+ (!TextUtils.isEmpty(selection) ?
" AND (" + selection + ')' : "");
default: break;
}
// To return the number of deleted items you must specify a where
// clause. To delete all rows and return a value pass in "1".
if (selection == null)
```

```
selection = "1";
// Perform the deletion.
int deleteCount = db.delete(MySQLiteOpenHelper.DATABASE_TABLE,
selection, selectionArgs);
// Notify any observers of the change in the data set.
getContext().getContentResolver().notifyChange(uri, null);
// Return the number of deleted items.
return deleteCount;
}

@Override
public Uri insert(Uri uri, ContentValues values) {
// Open a read / write database to support the transaction.
SQLiteDatabase db = myOpenHelper.getWritableDatabase();
// To add empty rows to your database by passing in an empty
// Content Values object you must use the null column hack
// parameter to specify the name of the column that can be
// set to null.
String nullColumnHack = null;
// Insert the values into the table
long id = db.insert(MySQLiteOpenHelper.DATABASE_TABLE,
nullColumnHack, values);
// Construct and return the URI of the newly inserted row.
if (id > -1) {
// Construct and return the URI of the newly inserted row.
Uri insertedId = ContentUris.withAppendedId(CONTENT_URI, id);
// Notify any observers of the change in the data set.
getContext().getContentResolver().notifyChange(insertedId, null);
return insertedId;
}
else
return null;
}

@Override
public int update(Uri uri, ContentValues values, String selection,
String[] selectionArgs) {
// Open a read / write database to support the transaction.
SQLiteDatabase db = myOpenHelper.getWritableDatabase();
// If this is a row URI, limit the deletion to the specified row.
switch (uriMatcher.match(uri)) {
case SINGLE_ROW :
String rowID = uri.getPathSegments().get(1);
selection = KEY_ID + "=" + rowID
+ (!TextUtils.isEmpty(selection) ?
" AND (" + selection + ')' : "");
default: break;
}
// Perform the update.
int updateCount = db.update(MySQLiteOpenHelper.DATABASE_TABLE,
values, selection, selectionArgs);
// Notify any observers of the change in the data set.
getContext().getContentResolver().notifyChange(uri, null);
return updateCount;
}
```

11.5 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.

It's typical for a Content Provider to include two tables, one that is used only to store the external files, and another that includes a user-facing column containing a URI reference to the rows in the file table.

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
throws FileNotFoundException {
// Find the row ID and use it as a filename.
String rowID = uri.getPathSegments().get(1);
// Create a file object in the application's external
// files directory.
String picsDir = Environment.DIRECTORY_PICTURES;
File file =
new File(getContext().getExternalFilesDir(picsDir), rowID);
// If the file doesn't exist, create it now.
if (!file.exists()) {
try {
file.createNewFile();
} catch (IOException e) {
Log.d(TAG, "File creation failed: " + e.getMessage());
}
}
// Translate the mode parameter to the corresponding Parcel File
// Descriptor open mode.
int fileMode = 0;
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;
// Return a Parcel File Descriptor that represents the file.
return ParcelFileDescriptor.open(file, fileMode);
}
```

12. <u>USING CONTENT PROVIDERS:</u>

12.1 <u>Content Resolver:</u>

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

ContentResolver cr = getContentResolver();

When Content Providers are used to expose data, Content Resolvers are the corresponding class used to query and perform transactions on those Content Providers.

A Content Provider's URI is its authority as defined by its manifest node and typically published as a static constant on the Content Provider implementation.

Content Providers usually accept two forms of URI, one for requests against all data and another that specifies only a single row. The form for the latter appends the row identifier (in the form /<rowID>) to the base URI.

12.2 <u>Querying Content Providers:</u>

Content Provider queries take a form very similar to that of database queries.

Using the query method on the ContentResolver object, pass in the following:

- A URI to the Content Provider you want to query.
- A projection that lists the columns you want to include in the result set.
- A where clause that defines the rows to be returned. You can include ? wildcards that will be replaced by the values passed into the selection argument parameter.
- An array of selection argument strings that will replace the ? wildcards in the where clause.
- A string that describes the order of the returned rows.

Querying a Content Provider with a Content Resolver

```
// Get the Content Resolver.
ContentResolver cr = getContentResolver();
// Specify the result column projection. Return the minimum set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
```

```
MyHoardContentProvider.KEY_ID,
MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN };
// Specify the where clause that will limit your results.
String where =
MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN
+ "=" + 1;
// Replace these with valid SQL statements as necessary.
String whereArgs[] = null;
String order = null;
// Return the specified rows.
Cursor resultCursor = cr.query(MyHoardContentProvider.CONTENT_URI,
result_columns, where, whereArgs, order);
```

12.3 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 (API level 11) introduced the Loader class. Loaders are now also available within the Android Support Library, making them available for use with every Android platform back to Android 1.6.

12.4 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.

12.5 Using the Cursor Loader:

The Cursor Loader handles all the management tasks required to use a Cursor within an Activity or Fragment, effectively deprecating the managedQuery and startManagingCursor Activity methods.

This includes managing the Cursor lifecycle to ensure Cursors are closed when the Activity is terminated. Cursor Loaders also observe changes in the underlying query, so you no longer need to implement your own Content Observers.

12.5.1 *Implementing Cursor Loader Callbacks:*

To use a Cursor Loader, create a new LoaderManager.LoaderCallbacks implementation. Loader Callbacks are implemented using generics, so you should specify the explicit type being loaded, in this case Cursors, when implementing your own.

```
LoaderManager.LoaderCallbacks<Cursor> loaderCallback
= new LoaderManager.LoaderCallbacks<Cursor>() {
```

The Loader Callbacks consist of three handlers:

- *onCreateLoader* - Called when the loader is initialized, this handler should create and return new Cursor Loader object. The Cursor Loader constructor arguments mirror those required for executing a query using the Content Resolver. Accordingly, when this handler is executed, the query parameters you specify will be used to perform a query using the Content Resolver.
- *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.
- *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.

12.5.2 *Initializing and Restarting the Cursor Loader:*

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 identifi er.

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

This is generally done within the onCreate method of the host Activity (or the onActivityCreated handler in the case of Fragments).

After a Loader has been created, repeated calls to initLoader will simply return the existing Loader. Should you want to discard the previous Loader and re-create it, use the restartLoader method.

```
loaderManager.restartLoader(LOADER_ID, args, myLoaderCallbacks);
```

12.6 <u>Adding, Deleting, and Updating Content:</u>

<u>Inserting Content:</u>

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.

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();
// Assign values for each row.
newValues.put(MyHoardContentProvider.KEY_GOLD_HOARD_NAME_COLUMN,
hoardName);
newValues.put(MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN,
hoardValue);
newValues.put(MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLU
MN,
hoardAccessible);
// [ ... Repeat for each column / value pair ... ]
// Get the Content Resolver
ContentResolver cr = getContentResolver();
// Insert the row into your table
Uri myRowUri = cr.insert(MyHoardContentProvider.CONTENT_URI,
newValues);
```

<u>Deleting Content:</u>

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.

```
// Specify a where clause that determines which row(s) to delete.
// Specify where arguments as necessary.
String where = MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN +
"=" + 0;
String whereArgs[] = null;
// Get the Content Resolver.
ContentResolver cr = getContentResolver();
// Delete the matching rows
int deletedRowCount =
cr.delete(MyHoardContentProvider.CONTENT_URI, where, whereArgs);
```

<u>Updating Content:</u>

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.

```
// Create the updated row content, assigning values for each row.
ContentValues updatedValues = new ContentValues();
updatedValues.put(MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN,
newHoardValue);
// [ ... Repeat for each column to update ... ]
// Create a URI addressing a specific row.
Uri rowURI =
ContentUris.withAppendedId(MyHoardContentProvider.CONTENT_URI,
hoardId);
// Specify a specific row so no selection clause is required.
String where = null;
String whereArgs[] = null;
```

12.7 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.

Reading and writing files from and to a Content Provider

```
public void addNewHoardWithImage(String hoardName, float hoardValue,
boolean hoardAccessible, Bitmap bitmap) {
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();
// Assign values for each row.
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);
// Get the Content Resolver
ContentResolver cr = getContentResolver();
// Insert the row into your table
Uri myRowUri =
cr.insert(MyHoardContentProvider.CONTENT_URI, newValues);
try {
// Open an output stream using the new row's URI.
OutputStream outStream = cr.openOutputStream(myRowUri);
// Compress your bitmap and save it into your provider.
bitmap.compress(Bitmap.CompressFormat.JPEG, 80, outStream);
}
catch (FileNotFoundException e) {
Log.d(TAG, “No file found for this record.”);
}
}
public Bitmap getHoardImage(long rowId) {
Uri myRowUri =
ContentUris.withAppendedId(MyHoardContentProvider.CONTENT_URI,
rowId);
try {
// Open an input stream using the new row’s URI.
InputStream inStream =
getContentResolver().openInputStream(myRowUri);
// Make a copy of the Bitmap.
Bitmap bitmap = BitmapFactory.decodeStream(inStream);
return bitmap;
}
catch (FileNotFoundException e) {
Log.d(TAG, “No file found for this record.”);
}
return null;
}
```

12.8 Creating a To-Do List Database and Content Provider:

1. Start by creating a new ToDoContentProvider class. It will be used to host the database using an SQLiteOpenHelper and manage your database interactions by extending the ContentProvider class. Include stub methods for the onCreate, query, update, insert, delete, and getType methods, and a private skeleton implementation of an SQLiteOpenHelper.

```
package com.paad.todolist;
import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;
import android.net.Uri;
import android.text.TextUtils;
import android.util.Log;
public class ToDoContentProvider extends ContentProvider {
@Override
public boolean onCreate() {
return false;
}
@Override
public String getType(Uri url) {
return null;
}
@Override
public Cursor query(Uri url, String[] projection, String selection,
String[] selectionArgs, String sort) {
return null;
}
@Override
public Uri insert(Uri url, ContentValues initialValues) {
return null;
}
@Override
public int delete(Uri url, String where, String[] whereArgs) {
return 0;
}
@Override
public int update(Uri url, ContentValues values,
String where, String[]wArgs) {
return 0;
}
private static class MySQLiteOpenHelper extends SQLiteOpenHelper {
public MySQLiteOpenHelper(Context context, String name,
CursorFactory factory, int version) {
super(context, name, factory, version);
}
// Called when no database exists in disk and the helper class needs
// to create a new one.
@Override
public void onCreate(SQLiteDatabase db) {
// TODO Create database tables.
}
// Called when there is a database version mismatch meaning that the version
// of the database on disk needs to be upgraded to the current version.
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
// TODO Upgrade database.
}
}
}
```

2. Publish the URI for this provider. This URI will be used to access this Content Provider from within other application components via the ContentResolver.

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

3. Create public static variables that defi ne the column names. They will be used within the SQLite Open Helper to create the database, and from other application components to extract values from your queries.

```
public static final String KEY_ID = "_id";
public static final String KEY_TASK = "task";
public static final String KEY_CREATION_DATE = "creation_date";
```

4. Within the MySQLiteOpenHelper, create variables to store the database name and version, along with the table name of the to-do list item table.

```
private static final String DATABASE_NAME = "todoDatabase.db";
private static final int DATABASE_VERSION = 1;
private static final String DATABASE_TABLE = "todoItemTable";
```

5. Still in the MySQLiteOpenHelper, overwrite the onCreate and onUpgrade methods to handle the database creation using the columns from step 3 and standard upgrade logic.

```
// SQL statement to create a new database.
private static final String DATABASE_CREATE = "create table " +
DATABASE_TABLE + " (" + KEY_ID +
" integer primary key autoincrement, " +
KEY_TASK + " text not null, " +
KEY_CREATION_DATE + "long);";
// Called when no database exists in disk and the helper class needs
// to create a new one.
@Override
public void onCreate(SQLiteDatabase db) {
db.execSQL(DATABASE_CREATE);
}
// Called when there is a database version mismatch, meaning that the version
// of the database on disk needs to be upgraded to the current version.
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
// Log the version upgrade.
Log.w("TaskDBAdapter", "Upgrading from version " +
oldVersion + " to " +
newVersion + ", which will destroy all old data");
// Upgrade the existing database to conform to the new version. Multiple
// previous versions can be handled by comparing oldVersion and newVersion
// values.
// The simplest case is to drop the old table and create a new one.
db.execSQL("DROP TABLE IF IT EXISTS " + DATABASE_TABLE);
// Create a new one.
onCreate(db);
}
```

6. Returning to the ToDoContentProvider, add a private variable to store an instance of the MySQLiteOpenHelper class, and create it within the onCreate handler.

```
private MySQLiteOpenHelper myOpenHelper;
@Override
public boolean onCreate() {
// Construct the underlying database.
// Defer opening the database until you need to perform
// a query or transaction.
myOpenHelper = new MySQLiteOpenHelper(getContext(),
MySQLiteOpenHelper.DATABASE_NAME, null,
MySQLiteOpenHelper.DATABASE_VERSION);
return true;
}
```

7. Still in the Content Provider, create a new UriMatcher to allow your Content Provider to differentiate between a query against the entire table and one that addresses a particular row. Use it within the getType handler to return the correct MIME type, depending on the query type.

```
private static final int ALLROWS = 1;
private static final int SINGLE_ROW = 2;
private static final UriMatcher uriMatcher;
//Populate the UriMatcher object, where a URI ending in ‘todoitems’ will
//correspond to a request for all items, and ‘todoitems/[rowID]’
//represents a single row.
static {
uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
uriMatcher.addURI(“com.paad.todoprovider”, “todoitems”, ALLROWS);
uriMatcher.addURI(“com.paad.todoprovider”, “todoitems/#”, SINGLE_ROW);
}
@Override
public String getType(Uri uri) {
// Return a string that identifies the MIME type
// for a Content Provider URI
switch (uriMatcher.match(uri)) {
case ALLROWS: return “vnd.android.cursor.dir/vnd.paad.todos”;
case SINGLE_ROW: return “vnd.android.cursor.item/vnd.paad.todos”;
default: throw new IllegalArgumentException(“Unsupported URI: “ + uri);
}
}
```

8. Implement the query method stub. Start by requesting an instance of the database, before constructing a query based on the parameters passed in. In this simple instance, you need to apply the same query parameters only to the underlying database - modifying the query only to account for the possibility of a URI that addresses a single row.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
// Open a read-only database.
SQLiteDatabase db = myOpenHelper.getWritableDatabase();
// Replace these with valid SQL statements if necessary.
String groupBy = null;
String having = null;
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
queryBuilder.setTables(MySQLiteOpenHelper.DATABASE_TABLE);
// If this is a row query, limit the result set to the passed in row.
switch (uriMatcher.match(uri)) {
case SINGLE_ROW :
String rowID = uri.getPathSegments().get(1);
queryBuilder.appendWhere(KEY_ID + "=" + rowID);
default: break;
}
Cursor cursor = queryBuilder.query(db, projection, selection,
selectionArgs, groupBy, having, sortOrder);
return cursor;
}
```

9. Implement the delete, insert, and update methods using the same approach - pass through the received parameters while handling the special case of single-row URIs.

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
// Open a read / write database to support the transaction.
SQLiteDatabase db = myOpenHelper.getWritableDatabase();
// If this is a row URI, limit the deletion to the specified row.
switch (uriMatcher.match(uri)) {
case SINGLE_ROW :
String rowID = uri.getPathSegments().get(1);
selection = KEY_ID + "=" + rowID
+ (!TextUtils.isEmpty(selection) ?
" AND (" + selection + ')' : "");
default: break;
}
// To return the number of deleted items, you must specify a where
// clause. To delete all rows and return a value, pass in "1".
if (selection == null)
selection = "1";
// Execute the deletion.
int deleteCount = db.delete(MySQLiteOpenHelper.DATABASE_TABLE, selection,
selectionArgs);
// Notify any observers of the change in the data set.
getContext().getContentResolver().notifyChange(uri, null);
return deleteCount;
}

@Override
public Uri insert(Uri uri, ContentValues values) {
// Open a read / write database to support the transaction.
SQLiteDatabase db = myOpenHelper.getWritableDatabase();
// To add empty rows to your database by passing in an empty Content Values
// object, you must use the null column hack parameter to specify the name of
// the column that can be set to null.
String nullColumnHack = null;
// Insert the values into the table
long id = db.insert(MySQLiteOpenHelper.DATABASE_TABLE,
nullColumnHack, values);
if (id > -1) {
// Construct and return the URI of the newly inserted row.
Uri insertedId = ContentUris.withAppendedId(CONTENT_URI, id);
// Notify any observers of the change in the data set.
getContext().getContentResolver().notifyChange(insertedId, null);
return insertedId;
}
else
return null;
}

@Override
public int update(Uri uri, ContentValues values, String selection,
String[] selectionArgs) {
// Open a read / write database to support the transaction.
SQLiteDatabase db = myOpenHelper.getWritableDatabase();
// If this is a row URI, limit the deletion to the specified row.
switch (uriMatcher.match(uri)) {
case SINGLE_ROW :
String rowID = uri.getPathSegments().get(1);
selection = KEY_ID + “=” + rowID
+ (!TextUtils.isEmpty(selection) ?
“ AND (“ + selection + ‘)’ : “”);
default: break;
}
// Perform the update.
int updateCount = db.update(MySQLiteOpenHelper.DATABASE_TABLE,
values, selection, selectionArgs);
// Notify any observers of the change in the data set.
getContext().getContentResolver().notifyChange(uri, null);
return updateCount;
}
```

10. That completes the Content Provider class. Add it to your application Manifest, specifying the base URI to use as its authority.

```
<provider android:name=".ToDoContentProvider"
android:authorities="com.paad.todoprovider"/>
```

11. Return to the ToDoList Activity and update it to persist the to-do list array. Start by modifying the Activity to implement LoaderManager.LoaderCallbacks<Cursor>, and then add the associated stub methods.

```
public class ToDoList extends Activity implements
NewItemFragment.OnNewItemAddedListener,
LoaderManager.LoaderCallbacks<Cursor> {
// [... Existing ToDoList Activity code ...]
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
return null;
}
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
}
public void onLoaderReset(Loader<Cursor> loader) {
}
}
```

12. Complete the onCreateLoader handler by building and returning a Loader that queries the ToDoListContentProvider for all of its elements.

```
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
CursorLoader loader = new CursorLoader(this,
ToDoContentProvider.CONTENT_URI, null, null, null, null);
return loader;
}
```

13. When the Loader's query completes, the result Cursor will be returned to the onLoadFinished handler. Update it to iterate over the result Cursor and repopulate the to-do list Array Adapter accordingly.

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
int keyTaskIndex =
cursor.getColumnIndexOrThrow(ToDoContentProvider.KEY_TASK);
todoItems.clear();
while (cursor.moveToNext()) {
ToDoItem newItem = new ToDoItem(cursor.getString(keyTaskIndex));
todoItems.add(newItem);
}
aa.notifyDataSetChanged();
}
```

14. Update the onCreate handler to initiate the Loader when the Activity is created, and the onResume handler to restart the Loader when the Activity is restarted.

```
public void onCreate(Bundle savedInstanceState) {
// [... Existing onCreate code …]
getLoaderManager().initLoader(0, null, this);
}
@Override
protected void onResume() {
super.onResume();
getLoaderManager().restartLoader(0, null, this);
}
```

15. The final step is to modify the behavior of the onNewItemAdded handler. Rather than adding the item to the to-do Array List directly, use the ContentResolver to add it to the Content Provider.

```
public void onNewItemAdded(String newItem) {
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
values.put(ToDoContentProvider.KEY_TASK, newItem);
cr.insert(ToDoContentProvider.CONTENT_URI, values);
getLoaderManager().restartLoader(0, null, this);
}
```

****

UNIT - V

Advanced Topics: Alarms - Creating and using alarms - Using Location Based Services – Using the Emulator with Location-Based Services - Finding the Current Location – Using the Geocoder - Creating Map-Based Activities

----

1. <u>USING ALARMS:</u>

- Alarms are a means of firing Intents at predetermined times or intervals.
- Unlike Timers, Alarms can also use 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 needing to be open or running.
- They are 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.
- Use alarms:
  - ✓ To schedule regular updates based on network lookups,
  - ✓ To schedule time-consuming or cost-bound operations at “off-peak” times
  - ✓ To schedule retries for failed operations.
- Alarm provides a mechanism to reduce the life time of your applications by moving scheduled events out of their control.
- Alarms in Android remain active while the device is in sleep mode and can optionally be set to wake the device. However, all Alarms are canceled whenever the device is rebooted.
- Alarm operations are handled through the AlarmManager, a system Service accessed getSystemService, as follows:

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

1.1 <u>CREATING, SETTING, AND CANCELING ALARMS:</u>

- 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:

- ✓ *RTC_WAKEUP* - Wakes the device from sleep to fire the Pending Intent at the clock time specified.
- ✓ *RTC* - Fires the Pending Intent at the time specified but does not wake the device.
- ✓ *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.
- ✓ *ELAPSED_REALTIME_WAKEUP* - Wakes the device from sleep and fires the Pending Intent after a specified length of time has passed since device boot.

Eg. Creating a waking Alarm that triggers in 10 seconds

```
//get the reference alarm(AlarmManager)getSystemService(Context.ALARM_SERVICE);
// Set the alarm to wake the device if sleeping. int alarmType =
AlarmManager.ELAPSED_REALTIME_WAKEUP;
long timeOrLengthofWait = 10000;
// Create a Pending Intent
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)
```

- When the Alarm goes off, the Pending Intent you specifi ed will be broadcast. Setting a second Alarm using the same Pending Intent replaces the preexisting Alarm.

To cancel alarm

- Call cancel on the Alarm Manager, passing in the Pending Intent you no longer want to trigger.

```
alarmManager.cancel(alarmIntent);
```

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 App, 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 AlarmManager.
    - ✓ setRepeating when you need fine-grained control over the exact interval of repeating alarm.

- ✓ setInexactRepeating 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

Eg. Setting an inexact repeating alarm

```
// 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);
```

- Repeating Alarms are canceled in the same way as one-shot Alarms, by calling cancel on the Alarm Manager and passing in the Pending Intent you no longer want to trigger.

1.2 USING REPEATING ALARMS TO SCHEDULE NETWORK REFRESHES:

- One of the most significant advantages of this approach is that it allows the Service to stop itself when it has completed a refresh, freeing significant system resources.

Start by creating a new EarthquakeAlarmReceiver class that extends BroadcastReceiver:

```
package com.paad.earthquake;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
public class EarthquakeAlarmReceiver extends BroadcastReceiver {
@Override
public void onReceive(Context context, Intent intent) {
}
}
```

Override the onReceive method to explicitly start the EarthquakeUpdateService:

```
@Override
public void onReceive(Context context, Intent intent) {
  Intent startIntent = new Intent(context, EarthquakeUpdateService.class);
  context.startService(startIntent);
}
```

Create a new public static String to define the action that will be used to trigger this Broadcast Receiver:

```
public static final String ACTION_REFRESH_EARTHQUAKE_ALARM =
“com.paad.earthquake.ACTION_REFRESH_EARTHQUAKE_ALARM”;
```

Add the new EarthquakeAlarmReceiver to the manifest, including an intent-filter tag that listens for the action defined in step 3:

```
<receiver android:name=”.EarthquakeAlarmReceiver”>
<intent-filter>
<action
android:name=”com.paad.earthquake.ACTION_REFRESH_EARTHQUAKE_ALARM”
/>
</intent-filter>
</receiver>
```

Within the Earthquake Update Service, override the onCreate method to get a reference to the AlarmManager, and create a new PendingIntent that will be fired when the Alarm is triggered. You can also remove the timerTask initialization.

```
private AlarmManager alarmManager;
private PendingIntent alarmIntent;

@Override
public void onCreate() {
  super.onCreate();
  alarmManager = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
  String ALARM_ACTION =
    EarthquakeAlarmReceiver.ACTION_REFRESH_EARTHQUAKE_ALARM;
Intent intentToFire = new Intent(ALARM_ACTION);
  alarmIntent =
    PendingIntent.getBroadcast(this, 0, intentToFire, 0);
}
```

Modify the onStartCommand handler to set an inexact repeating Alarm rather than use a Timer to schedule the refreshes (if automated updates are enabled). Setting a new Intent with the same action automatically cancels any previous Alarms. Take this opportunity to *modify the return result. Rather than setting the Service to sticky, return Service. START_NOT_STICKY. In step 7 you will stop the Service when the background refresh is complete; the use of alarms guarantees that another refresh will occur at the specified update frequency, so there's no need for the system to restart the Service if it is killed mid-refresh.*

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
// Retrieve the shared preferences
Context context = getApplicationContext();
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(context);
int updateFreq =
Integer.parseInt(prefs.getString(PreferencesActivity.PREF_UPDATE_FREQ, "60"));
boolean autoUpdateChecked =
prefs.getBoolean(PreferencesActivity.PREF_AUTO_UPDATE, false);
  if (autoUpdateChecked) {
    int alarmType = AlarmManager.ELAPSED_REALTIME_WAKEUP;
    long timeToRefresh = SystemClock.elapsedRealtime() +
                    updateFreq*60*1000;
    alarmManager.setInexactRepeating(alarmType, timeToRefresh,
                            updateFreq*60*1000, alarmIntent);
  }
  else
    alarmManager.cancel(alarmIntent);
Thread t = new Thread(new Runnable() {
public void run() {
refreshEarthquakes();
}
});
t.start();
  return Service.START_NOT_STICKY;
};
```

Within the refreshEarthquakes method, update the last try-fi nally case to call stopSelf when the background refresh has completed:

```
private void refreshEarthquakes() {
[... existing refreshEarthquakes method ...]
finally {
    stopSelf();
}
}
```

Remove the updateTimer instance variable and the Timer Task instance doRefresh.

2. USING LOCATION-BASED SERVICES

- "Location-based services" is a term that describes the different technologies can use to find a device's current location.
- The two main Location-based services (LBS) elements are:
  - ❖ Location Manager - Provides hooks to the location-based services.
  - ❖ Location Providers - Each of these represents a different location-finding technology used to determine the device's current location.
- Using the Location Manager:
  - ➢ Obtain your current location
  - ➢ Follow movement
  - ➢ Set proximity alerts for detecting movement into and out of a specifi ed area
  - ➢ Find available Location Providers
  - ➢ Monitor the status of the GPS receiver

  Accessing the Location Manager

```
String serviceString = Context.LOCATION_SERVICE;
LocationManager locationManager;
locationManager = (LocationManager)getSystemService(serviceString);
```

- Before you can use the location-based services, you need to add one or more uses-permission tags to your manifest.
- The following snippet shows how to request the *fine* and *coarse* permissions in your application manifest:

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

2.1 USING THE EMULATOR WITH LOCATION-BASED SERVICES:

- Location-based services are dependent on device hardware used to find the current location.
- When you develop and test with the Emulator, your hardware is virtualized, and you're likely to stay in pretty much the same location.
- Android includes hooks that enable you to emulate Location Providers for testing location-based applications.

2.1.1 <u>Updating Locations in Emulator Location Providers:</u>

- Use the Location Controls available from the DDMS perspective in Eclipse to push location changes directly into the Emulator's GPS Location Provider.
- Using the manual tab you can specify particular latitude/longitude pairs. The KML and GPX tabs enable you to load Keyhole Markup Language (KML) and GPS Exchange Format (GPX) files, respectively.
- After these load you can jump to particular waypoints (locations) or play back each sequence of locations.
- All location changes applied using the DDMS location controls will be applied to the GPS receiver, which must be enabled and active.

2.1.2 <u>Configuring the Emulator to Test Location-Based Services:</u>

- The GPS values returned by getLastKnownLocation do not change unless at least one application requests location updates.
- As a result, when the Emulator is first started, the result returned from a call to getLastKnownLocation is likely to be null, as no application has made a request to receive location updates.
- Further, the techniques used to update the mock location described in the previous section are effective only when at least one application has requested location updates from the GPS.

  Eg. Enabling the GPS provider on the Emulator

```
locationManager.requestLocationUpdates(
LocationManager.GPS_PROVIDER, 0, 0,
new LocationListener() {
public void onLocationChanged(Location location) {}
public void onProviderDisabled(String provider) {}
public void onProviderEnabled(String provider) {}
public void onStatusChanged(String provider, int status, Bundle extras) {}
}
);
```

2.2 <u>SELECTING A LOCATION PROVIDER:</u>

- Depending on the device, you can use several technologies to determine the current location.
- Each technology, available as a Location Provider, offers different capabilities including differences in power consumption, accuracy, and the ability to determine altitude, speed, or heading information.

2.2.1 <u>Finding location provider:</u>

- The LocationManager class includes static string constants that return the provider name for three Location Providers:
  - ❖ LocationManager.GPS_PROVIDER
  - ❖ LocationManager.NETWORK_PROVIDER
  - ❖ LocationManager.PASSIVE_PROVIDER

- To get a list of the names of all the providers available (based on hardware available on the device, and the permissions granted the application), call getProviders, using a Boolean to indicate if you want all, or only the enabled, providers to be returned:

```
boolean enabledOnly = true; List<String> providers =
locationManager.getProviders(enabledOnly);
```

2.2.2 <u>Finding Location Providers by Specifying Criteria:</u>

- Use the Criteria class to dictate the requirements of a provider in terms of accuracy, power use (low, medium, high), financial cost, and the ability to return values for altitude, speed, and heading.

Eg. Specifing location provider

```
Criteria criteria = new Criteria();
criteria.setAccuracy(Criteria.ACCURACY_COARSE);
 criteria.setPowerRequirement(Criteria.POWER_LOW);
criteria.setAltitudeRequired(false);
criteria.setBearingRequired(false);
criteria.setSpeedRequired(false);
criteria.setCostAllowed(true);
```

- If more than one Location Provider matches your Criteria, the one with the greatest accuracy is returned. If no Location Providers meet your requirements, the Criteria are loosened, in the following order, until a provider is found:
  - ✓ Power use
  - ✓ Accuracy of returned location
  - ✓ Accuracy of bearing, speed, and altitude
  - ✓ Availability of bearing, speed, and altitude

- To get a list of names for all the providers matching your Criteria, use getProviders. It accepts a Criteria object and returns a String list of all Location Providers that match it.

As with the getBestProvider call, if no matching providers are found, this method returns null or an empty List.

List<String> matchingProviders = locationManager.getProviders(criteria, false);

2.2.3 <u>Determining Location Provider Capabilities:</u>

- To get an instance of a specific provider, call getProvider, passing in the name:

  String providerName = LocationManager.GPS_PROVIDER;
  LocationProvider gpsProvider
  = locationManager.getProvider(providerName);

- This is useful only for obtaining the capabilities of a particular provider - specifically the accuracy and power requirements through the getAccuracy and getPowerRequirement methods.

2.3 <u>FINDING YOUR CURRENT LOCATION:</u>

- One of the most powerful uses of location-based services is to find the physical location of the device.
- The accuracy of the returned location is dependent on the hardware available and the permissions requested by your application.

2.3.1 <u>Location Privacy:</u>

- Privacy is an important consideration when your application uses the user’s location particularly when it is regularly updating their current position.
- The application uses the device location data in a way that respects the user’s privacy by:
  - ❖ Only using and updating location when necessary for your application
  - ❖ Notifying users of when you track their locations, and if and how that location information is used, transmitted, and store
  - ❖ Allowing users to disable location updates, and respecting the system settings for LBS preferences

2.3.2 <u>Finding the Last Known Location:</u>

- To find the last location fix obtained by a particular Location Provider using the getLastKnownLocation method, passing in the name of the Location Provider.

  String provider = LocationManager.GPS_PROVIDER;
  Location location = locationManager.getLastKnownLocation(provider);

- The Location object returned includes all the position information available from the provider that supplied it.
- This can include,

- Time
- Accuracy of the location found
- Latitude, Longitude
- Bearing
- Altitude
- Speed.

- All these properties are available via get methods on the Location object.

Where Am I Example

- The following example features a new Activity that finds the device's last known location using the GPS Location Provider.

1. Create a new Where Am I project with a WhereAmI. It uses the GPS provider, so you need to include the uses-permission tag for ACCESS_FINE_LOCATION in your application manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.paad.whereami"
  android:versionCode="1"
  android:versionName="1.0" >
  <uses-sdk android:minSdkVersion="4" />
  <uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION"
    />
  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name" >
    <activity
      android:name=".WhereAmI"
      android:label="@string/app_name" >
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

2. Modify the main.xml layout resource to include an android:ID attribute for the TextView control so that you can access it from within the Activity.

```
<?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:id=”@+id/myLocationText”
    android:layout_width=”match_parent”
    android:layout_height=”wrap_content”
    android:text=”@string/hello”
    />
</LinearLayout>
```

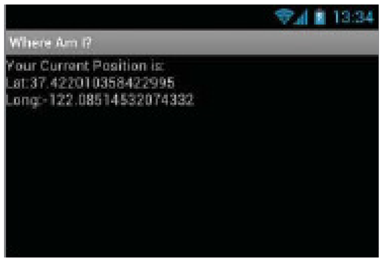
3. Override the onCreate method of the WhereAmI Activity to get a reference to the Location Manager. Call getLastKnownLocation to get the last known location, and pass it in to an updateWithNewLocation method stub.

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
LocationManager locationManager;
String svcName = Context.LOCATION_SERVICE;
locationManager = (LocationManager)getSystemService(svcName);
String provider = LocationManager.GPS_PROVIDER;
Location l = locationManager.getLastKnownLocation(provider);
updateWithNewLocation(l);
}
private void updateWithNewLocation(Location location) {}
```

4. Complete the updateWithNewLocation method to show the passed-in Location in the Text View by extracting the latitude and longitude values.

```
private void updateWithNewLocation(Location location) {
  TextView myLocationText;
  myLocationText = (TextView)findViewById(R.id.myLocationText);
  String latLongString = “No location found”;
  if (location != null) {
    double lat = location.getLatitude();
    double lng = location.getLongitude();
    latLongString = “Lat:” + lat + “\nLong:” + lng;
  }
  myLocationText.setText(“Your Current Position is:\n” +latLongString);
}
```

5. When running, your Activity should look like



```
Your Current Position is:
Lat:37.422010358422995
Long:-122.08514532074332
```

2.3.3 <u>Refreshing the Current Location:</u>

- Location Listeners also contain hooks for changes in a provider's status and availability.
- The requestLocationUpdates method accepts either a specific Location Provider name or a set of Criteria to determine the provider to use.
- To optimize efficiency and minimize cost and power use, you can also specify the minimum time and the minimum distance between location change updates.

  Requesting Location Updates using a Location Listener:

  ```
  String provider = LocationManager.GPS_PROVIDER;
  int t = 5000; // milliseconds
  int distance = 5; // meters
  LocationListener myLocationListener = new LocationListener() {
  public void onLocationChanged(Location location) {
  // Update application based on new location.
  }
  public void onProviderDisabled(String provider){
  // Update application if provider disabled.
  }
  public void onProviderEnabled(String provider){
  public void onStatusChanged(String provider, int status,
  Bundle extras){
  // Update application if provider hardware status changed.
  }
  };
  locationManager.requestLocationUpdates(provider, t, distance,
  myLocationListener);
  ```

- When the minimum time and distance values are exceeded, the attached Location Listener executes its onLocationChanged event.
- Android 3.0 (API level 11) introduced an alternative technique for receiving location changes. Rather than creating a Location Listener, you can specify a Pending Intent that will be broadcast whenever the location changes or the location provider status or

availability changes. The new location is stored as an extra with the key KEY_LOCATION_CHANGED.

- This is a particularly useful alternative if you have multiple Activities or Services that require location updates as they can listen for the same broadcast Intents.

Requesting location updates using a Pending Intent

```
String provider = LocationManager.GPS_PROVIDER;
int t = 5000; // milliseconds
int distance = 5; // meters
final int locationUpdateRC = 0;
int flags = PendingIntent.FLAG_UPDATE_CURRENT;
Intent intent = new Intent(this, MyLocationUpdateReceiver.class);
PendingIntent pendingIntent = PendingIntent.getBroadcast(this,
locationUpdateRC, intent, flags);
locationManager.requestLocationUpdates(provider, t,
distance, pendingIntent);
```

Receiving location updates using a Broadcast Receiver

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.location.Location;
import android.location.LocationManager;
public class MyLocationUpdateReceiver extends BroadcastReceiver {
@Override
public void onReceive(Context context, Intent intent) {
String key = LocationManager.KEY_LOCATION_CHANGED;
Location location = (Location)intent.getExtras().get(key);
// TODO [... Do something with the new location ...]
}
}
```

- Remember that you must add your Broadcast Receiver to the application manifest before it can begin receiving the Pending Intents.
- To stop location updates, call removeUpdates, as shown in the following code. Pass in either the Location Listener instance or Pending Intent that you no longer want to have triggered.

```
locationManager.removeUpdates(myLocationListener);
locationManager.removeUpdates(pendingIntent);
```

- To minimize the cost to battery life, you should disable updates whenever possible in your application, especially in cases where your application isn't visible and location changes are used only to update an Activity's UI.

- Where timeliness is not a significant factor, you might consider using the Passive Location Provider (introduced in Android 2.2, API level 8), as shown in the following snippet.

  String passiveProvider = LocationManager.PASSIVE_PROVIDER;
  locationManager.requestLocationUpdates(passiveProvider, 0, 0,
  myLocationListener);

- The Passive Location Provider receives location updates if, and only if, another application requests them, letting your application passively receive location updates without activating any Location Provider.
- Its passive nature makes this an excellent alternative for keeping location data fresh within your application while it is in the background, without draining the battery.

2.3.4 <u>Requesting a Single Location Update:</u>

- Not every app requires regular location updates to remain useful.
- In many cases only a single location fix is required to provide adequate context for the functionality they provide or information they display.
  - ✓ getLastKnownLocation can be used to return the last known position, there's no guarantee that this location exists, or that it is still relevant.
  - ✓ requestSingleUpdate method enables to specify a Provider or Criteria to use when requesting at least one update.

  Looper looper = null;
  locationManager.requestSingleUpdate(criteria, myLocationListener, looper);

- When using a Location Listener, you can specify a Looper parameter. This allows to schedule the callbacks on a particular thread setting the parameter to null will force it to return on the calling thread.
- Use to receive the single location update using either a Location Listener as previously shown, or through a Pending Intent as shown here.

  locationManager.requestSingleUpdate(criteria, pendingIntent);

2.4 <u>BEST PRACTICE FOR LOCATION UPDATES:</u>

- When using Location within your application, consider the following factors:
  1. **Battery life versus accuracy** - The more accurate the Location Provider, the greater its drain on the battery.
  2. **Startup time** - In a mobile environment the time taken to get an initial location can have a dramatic effect on the user experience.

3. **Update rate -** The more frequent the update rate, the more dramatic the effect on battery life. Slower updates can reduce battery drain at the price of less timely updates

3. <u>USING THE GEOCODER:</u>

- Geocoding enables to translate between the street address and longitude/latitude map coordinates.
- This can give a recognizable context for the location and coordinates used in location based services and map based activities.
- This classes are included as part of the Google Maps library, so to access it need to import by adding a uses-library node as shown here:

  <uses_libraryandroid:name="com.google.android:name="com.google.android.maps"/ >
- After that, application also require the interest uses permission in your manifest:

  <uses_permission android:name="android.permission INTERNET">
- The Geocoder class provides access to two geocoding functions:
  - ✓ Forword geocoding: Finds the latitude and longitude of an address.
  - ✓ Reverse geocoding: Finds the street address for a given latitude and longitude.
- To set the locale when creating your geocoder:

  Geocoder geocoder= new Geocoder(get.ApplicationContext().Locale.getDefault());
- If you don't specify a locale it assumes the device default. Both geocoding functions return a list of address objects.
- Geocoder looks up are performed synchronously, so they block calling thread.
- Android 2.3(API level9) introduced the ispresent() method to determine if a Geocoder implementation exists on a given device.

  Bool geocoderExists=Geocoder.isPresent();
- If there is no geocoder implementation, the forward and reverse geocoding queries will return an empty list.

3.1 <u>Forward geocoding:</u>

- It determine map coordinates for given location.
- A valid location varies depending on locale within which our search:
  - ➢ It include regular street address of varying granularity, postcodes, train stations, landmarks and hospitals.

- Valid search terms are similar terms are similar to the address and locations can enter into Google Maps search bar.

- To geocode an address call getFromLocationName() on a geocoder object:

  List<Address>result=gc.getFromLocationName(streetAddress.maxResults);

- To perform forward lookups, the locale specified when instantially the geocoder is partially important.

- The locale provides the geographical context for interpreting your search requests because the same location names can exists in multiple areas.

Example: Geocoding an address

```
Geocoder fwdGeocoder=new Geocoder(this,Locale.US);
String streetAddress = "160 Riverside Drive,New York,New York";
List<Address>locations = null;
 try{
   locations = fwdGeocoder.getFromLocationName(streetAddress,5);
}catch(IOExecption e){
  Log.e(TAG,"IO Exception",e);
}
```

- For even more specific results, you can restrict your search to within a geographical area by specifying the lower-left and upper-right latitude and longitude as shown here:

```
List<Address> locations = null;
try {
locations = fwdGeocoder.getFromLocationName(streetAddress, 10,
llLat, llLong, urLat, urLong);
} catch (IOException e) {
Log.e(TAG, "IO Exception", e);
}
```

- This overload is particularly useful with a Map View, letting you restrict the search to the visible map area.

3.2 <u>Reverse geocoding:</u>

- It returns street address for physical locations specified by latitude/longitude pairs.

- It is useful way to get recognizable context for the locations returned by location-based services.

- To perform a reverse lookups, pass the target latitude and longitude to a geocoder objects getFromLocation() method.

- If geocoder could not resolve any address for the specified coordinates, it returns null.

Example: Reverse-geocoding a given location

```
private void reverseGeocode(Location location) {
double latitude = location.getLatitude();
double longitude = location.getLongitude();
List<Address> addresses = null;
Geocoder gc = new Geocoder(this, Locale.getDefault());
try {
addresses = gc.getFromLocation(latitude, longitude, 10);
} catch (IOException e) {
Log.e(TAG, "IO Exception", e);
}
}
```

- The accuracy and granularity of reverse lookups are entirely dependent on the quality of data in the geocoding database; as a result, the quality of the results may vary widely between different countries and locales.

Example: Geocoding Where Am I

- Extend the Where Am I project to include and update the current street address whenever the device moves.

1. Start by modifying the manifest to include the Internet uses-permission:

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

2. Then open the WhereAmI Activity. Modify the updateWithNewLocation method to instantiate a new Geocoder object and call the getFromLocation method, passing in the newly received location and limiting the results to a single address.

3. Extract each line in the street address and the locality, postcode, and country, and append this information to an existing Text View string.

```
private void updateWithNewLocation(Location location) {
TextView myLocationText;
myLocationText = (TextView)findViewById(R.id.myLocationText);
String latLongString = "No location found";
String addressString = "No address found";
if (location != null) {
double lat = location.getLatitude();
double lng = location.getLongitude();
latLongString = "Lat:" + lat + "\nLong:" + lng;
double latitude = location.getLatitude();
double longitude = location.getLongitude();
Geocoder gc = new Geocoder(this, Locale.getDefault());
try {
List<Address> addresses = gc.getFromLocation(latitude, longitude, 1);
StringBuilder sb = new StringBuilder();
if (addresses.size() > 0) {
Address address = addresses.get(0);
for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
sb.append(address.getAddressLine(i)).append("\n");
sb.append(address.getLocality()).append("\n");
sb.append(address.getPostalCode()).append("\n");
sb.append(address.getCountryName());
}
addressString = sb.toString();
} catch (IOException e) {}
}
myLocationText.setText("Your Current Position is:\n" +
latLongString + "\n\n" + addressString);
}
```

Output:

```
Where Am I?
Your current postion is:
Lat:37.420418598621354
Long:122.081525398759
15  northcarst
Chidambaram
Cuddalore
India
```

4. CREATING MAP-BASED ACTIVITIES:

- One of the most intuitive ways to provide context for a physical location or address is to use a map.
- MapView, can create Activities that include an interactive map.
- Map Views support annotation using Overlays and by pinning Views to geographical locations.
- It gives full programmatic control of the map display, letting you control the zoom, location, and display mode including the option to display a satellite view.

4.1 Map View and Map Activity:

❖ MapView is the user interface element that displays the map.

❖ MapActivity is the base class you extend to create an Activity that can include a Map View.

❖ The MapActivity class handles the application life cycle and background service management required for displaying maps. You can only use Map Views within MapActivity-derived Activities.

- ❖ Overlay is the class used to annotate your maps. Using Overlays, you can use a Canvas to draw onto any number of layers displayed on top of a Map View.
- ❖ MapController is used to control the map, enabling you to set the center location and zoom levels.
- ❖ MyLocationOverlay is a special Overlay that can be used to display the current position and orientation of the device.
- ❖ ItemizedOverlays and OverlayItems are used together to let you create a layer of map markers, displayed using Drawables and associated text.

4.2 <u>Getting Maps API Key:</u>

- To use a Map View in your application, you must fi rst obtain an API key from the Android developer website at http://code.google.com/android/maps-api-signup.html.
    - ✓ Without an API key the Map View cannot download the tiles used to display the map.
    - ✓ To obtain a key, you need to specify the MD5 fingerprint of the certificate used to sign your application. Generally, sign application using two certificates: a default debug certificate and a production certificate.

4.3 <u>Creating a Map-Based Activity:</u>

- To use maps in your applications, extend the MapActivity.
- The layout for the new class must then include a MapView to display a Google Maps interface element.
- The Android maps library is not a standard Android package; as an optional API, it must be explicitly included in the application manifest before it can be used.
- Add the library to manifest using a uses-library tag within the application node:

  <uses-library android:name="com.google.android.maps"/>
- The Map View downloads its map tiles on demand.
- Add a uses-permission tag to the application manifest for INTERNET,

  <uses-permission android:name="android.permission.INTERNET"/>
- MapView controls can be used only within an Activity that extends MapActivity.
- Override the onCreate method to lay out the screen that includes a MapView, and override isRouteDisplayed to return true if the Activity will be displaying routing information (such as traffic directions).

  ➢ **A skeleton map Activity**

```
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;
import android.os.Bundle;
public class MyMapActivity extends MapActivity {
  private MapView mapView;
  private MapController mapController;
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.map_layout);
    mapView = (MapView)findViewById(R.id.map_view);
  }
  @Override
  protected boolean isRouteDisplayed() {
    // IMPORTANT: This method must return true if your Activity
    // is displaying driving directions. Otherwise return false.
    return false;
  }
}
```

- **A map Activity layout resource**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <com.google.android.maps.MapView
    android:id="@+id/map_view"
    android:layout_width="fill_parent"
android:layout_height="fill_parent"
    android:enabled="true"
    android:clickable="true"
    android:apiKey="mymapapikey"
  />
</LinearLayout>
```

4.4 Maps and Fragments:

- Map Views can be included within Fragments, provided that the Fragment is attached to a Map Activity.
- Extend FragmentActivity in order to provide Fragment support, or MapActivity in order to include a Map View element.
- At the time of writing, the support library did not include a MapFragment or MapFragmentActivity class to enable the use of Map Views within support library Fragments.

- There are several third-party support libraries that attempt to circumvent this restriction.
- One approach is to create alternative Activity classes for pre- and post-Honeycomb devices, such that Maps within Fragments are used only where supported.

4.5 <u>Configuring and Using Map Views:</u>

- By default the Map View shows the standard street map.
- In addition choose to display a satellite view ,

  mapView.setSatellite(true);
  mapView.setTraffic(true);

- Query the Map View to find the current and maximum available zoom levels:

  int maxZoom = mapView.getMaxZoomLevel();
  int currentZoom = mapView.getZoomLevel();

- Obtain the center point and currently visible longitude and latitude span (in decimal degrees). This is particularly useful for performing geographically limited Geocoder lookups:

  GeoPoint center = mapView.getMapCenter();
  int latSpan = mapView.getLatitudeSpan();
  int longSpan = mapView.getLongitudeSpan();

- Choose to display the standard map zoom controls using the setBuiltInZoomControls method.

  mapView.setBuiltInZoomControls(true);

- To customize the zoom controls use the getZoomButtonsController method to obtain an instance of the Zoom Buttons Controller.
- Use the controller to customize the zoom speed, enable or disable the zoom in or out controls, and add additional buttons to the zoom controls layout.

  ZoomButtonsController zoomButtons = mapView.getZoomButtonsController();

4.6 <u>Using the Map Controller:</u>

- Use the Map Controller to pan and zoom a MapView. get a reference to a MapView’s controller using getController.

  MapController mapController = mapView.getController();

- Map locations in the Android mapping classes are represented by GeoPoint objects, which contain a latitude and longitude measured in microdegrees. To convert degrees to microdegrees, multiply by 1E6 (1,000,000).

- Before you can use the latitude and longitude values stored in the Location objects returned by location-based services, you need to convert them to microdegrees and store them as GeoPoints.

```
Double lat = 37.422006*1E6;
Double lng = -122.084095*1E6;
GeoPoint point = new GeoPoint(lat.intValue(), lng.intValue());
```

- Recenter and zoom the Map View using the setCenter and setZoom methods available on the Map View's MapController.

```
mapController.setCenter(point);
mapController.setZoom(1);
```

- When you use setZoom, 1 represents the widest (or most distant) zoom and 21 the tightest (nearest) view.
- The actual zoom level available for a specifi c location depends on the resolution of Google's maps and imagery for that area and can be found by calling getMaxZoomLevel on the associated Map View. You can also use zoomIn and zoomOut to change the zoom level by one step or zoomToSpan to specify a latitude or longitude span to zoom to.
- The setCenter method "jumps" to a new location; to show a smooth transition, use animateTo.

```
mapController.animateTo(point);
```

➢ **Example: Mapping Where Am I**

- The following code example extends the Where Am I project again. This time you add mapping functionality by transforming it into a Map Activity. As the device location changes, the map automatically re-centers on the new position.

1. Start by checking properties
2. Modify the application manifest to add the maps library:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.paad.whereami"
 android:versionCode="1"
 android:versionName="1.0" >
 <uses-sdk android:minSdkVersion="4" />
 <uses-permission android:name="android.permission.INTERNET"/>
 <uses-permission
  android:name="android.permission.ACCESS_FINE_LOCATION"
 />
 <application
  android:icon="@drawable/ic_launcher"
```

```
    android:label="@string/app_name">
  <uses-library android:name="com.google.android.maps"/>
    <activity
      android:name=".WhereAmI"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

3. Change the inheritance of the WhereAmI Activity to extend MapActivity instead of Activity. You also need to include an override for the isRouteDisplayed method. Because this Activity won’ t show routing directions, you can return false.

```
public class WhereAmI extends MapActivity {
  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }
  [ ... existing Activity code ... ]
}
```

4. Modify the main.xml layout resource to include a MapView using the fully qualifi ed class name. You need to obtain a maps API key to include within the android:apikey attribute of the com.android.MapView node.

```
<?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:id="@+id/myLocationText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
  />
  <com.google.android.maps.MapView
    android:id="@+id/myMapView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:enabled="true"
    android:clickable="true"
    android:apiKey="myMapKey"
```

```
/>
</LinearLayout>
```

5. Running the application now should display the original address text with a MapView beneath it.
6. Returning to the WhereAmI Activity, confi gure the Map View and store a reference to its MapController as an instance variable. Set up the Map View display options to show the satellite and zoom in for a closer look.

```
private MapController mapController;
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  // Get a reference to the MapView
  MapView myMapView = (MapView)findViewById(R.id.myMapView);
  // Get the Map View's controller
  mapController = myMapView.getController();
  // Configure the map display options
  myMapView.setSatellite(true);
  myMapView.setBuiltInZoomControls(true);
  // Zoom in
  mapController.setZoom(17);
  LocationManager locationManager;
  String svcName= Context.LOCATION_SERVICE;
  locationManager = (LocationManager)getSystemService(svcName);
  Criteria criteria = new Criteria();
  criteria.setAccuracy(Criteria.ACCURACY_FINE);
  criteria.setPowerRequirement(Criteria.POWER_LOW);
  criteria.setAltitudeRequired(false);
  criteria.setBearingRequired(false);
  criteria.setSpeedRequired(false);
  criteria.setCostAllowed(true);
  String provider = locationManager.getBestProvider(criteria, true);
  Location l = locationManager.getLastKnownLocation(provider);
  updateWithNewLocation(l);
  locationManager.requestLocationUpdates(provider, 2000, 10,
                                   locationListener);
}
```

7. The final step is to modify the updateWithNewLocation method to re-center the map on the current location using the Map Controller:

```
private void updateWithNewLocation(Location location) {
  TextView myLocationText;
  myLocationText = (TextView)findViewById(R.id.myLocationText);
  String latLongString = “No location found”;
  String addressString = “No address found”;

  if (location != null) {
    // Update the map location.
    Double geoLat = location.getLatitude()*1E6;
    Double geoLng = location.getLongitude()*1E6;
    GeoPoint point = new GeoPoint(geoLat.intValue(),
                              geoLng.intValue());
    mapController.animateTo(point);
    double lat = location.getLatitude();
    double lng = location.getLongitude();
    latLongString = “Lat:” + lat + “\nLong:” + lng;

    double latitude = location.getLatitude();
    double longitude = location.getLongitude();
    Geocoder gc = new Geocoder(this, Locale.getDefault());
    if (!Geocoder.isPresent())
      addressString = “No geocoder available”;
    else {
      try {
        List<Address> addresses = gc.getFromLocation(latitude, longitude, 1);
        StringBuilder sb = new StringBuilder();
        if (addresses.size() > 0) {
          Address address = addresses.get(0);
          for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
            sb.append(address.getAddressLine(i)).append(“\n”);
          sb.append(address.getLocality()).append(“\n”);
sb.append(address.getPostalCode()).append(“\n”);
          sb.append(address.getCountryName());
        }
        addressString = sb.toString();
      } catch (IOException e) {
        Log.d(“WHEREAMI”, “IO Exception”, e);
      }
    }
  }

  myLocationText.setText(“Your Current Position is:\n” +
    latLongString + “\n\n” + addressString);
}
```

4.7 <u>Creating and Using Overlays:</u>

- Overlays enable you to add annotations and click handling to MapViews.
- Each Overlay enables you to draw 2D primitives, including text, lines, images, and shapes, directly onto a canvas, which is then overlaid onto a Map View.

- You can add several Overlays onto a single map. All the Overlays assigned to a Map View are added as layers, with newer layers potentially obscuring older ones.
- User clicks are passed through the stack until they are either handled by an Overlay or registered as clicks on the Map View itself.

4.7.1 <u>Creating New Overlays:</u>

- To add a new Overlay, create a class that extends Overlay. Override the draw method to draw the annotations you want to add, and override onTap to react to user clicks (generally made when the user taps an annotation added by this Overlay).

```
import android.graphics.Canvas;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapView;
import com.google.android.maps.Overlay;
public class MyOverlay extends Overlay {
  @Override
  public void draw(Canvas canvas, MapView mapView, boolean shadow) {
    if (shadow == false) {
      // TODO [ ... Draw annotations on main map layer ... ]
    }
else {
      // TODO [ ... Draw annotations on the shadow layer ... ]
    }
  }
  @Override
  public boolean onTap(GeoPoint point, MapView mapView) {
   // Return true if screen tap is handled by this overlay
   return false;
  }
}
```

4.7.2 <u>Introducing Projections:</u>

- The canvas used to draw Overlay annotations is a standard Canvas that represents the visible display surface.
- To add annotations based on physical locations, you need to convert between geographical points and screen coordinates.
- The Projection class enables you to translate between latitude/longitude coordinates (stored as GeoPoints) and x/y screen pixel coordinates (stored as Points).
- A map's Projection may change between subsequent calls to draw, so it's good practice to get a new instance each time. Get a Map View's Projection by calling getProjection.

```
Projection projection = mapView.getProjection();
```

- Use the fromPixel and toPixel methods to translate from GeoPoints to Points and vice versa.
- For performance reasons, you can best use the toPixel Projection method by passing a Point object to be populated (rather than relying on the return value).

```
Point myPoint = new Point();
// To screen coordinates
projection.toPixels(geoPoint, myPoint);
// To GeoPoint location coordinates
GeoPoint gPoint = projection.fromPixels(myPoint.x, myPoint.y);
```

4.7.3 <u>Drawing on the Overlay Canvas:</u>

- You handle Canvas drawing for Overlays by overriding the Overlay's draw handler.
- The Canvas object includes the methods for drawing 2D primitives on your map (including lines, text, shapes, ellipses, images, and so on). Use Paint objects to define the style and color.

Example: A simple map Overlay

```
@Override
public void draw(Canvas canvas, MapView mapView, boolean shadow) {
  Projection projection = mapView.getProjection();
  Double lat = -31.960906*1E6;
  Double lng = 115.844822*1E6;
  GeoPoint geoPoint = new GeoPoint(lat.intValue(), lng.intValue());
  if (shadow == false) {
    Point myPoint = new Point();
    projection.toPixels(geoPoint, myPoint);
    // Create and setup your paint brush
    Paint paint = new Paint();
    paint.setARGB(250, 255, 0, 0);
    paint.setAntiAlias(true);
    paint.setFakeBoldText(true);
    // Create the circle
    int rad = 5;
    RectF oval = new RectF(myPoint.x-rad, myPoint.y-rad,
                    myPoint.x+rad, myPoint.y+rad);
    // Draw on the canvas
    canvas.drawOval(oval, paint);
    canvas.drawText("Red Circle", myPoint.x+rad, myPoint.y, paint);
  }
}
```

4.7.4 Handling Map Tap Events:

- To handle map taps (user clicks), override the onTap event handler within the Overlay extension class. The onTap handler receives two parameters:
  - ❖ A GeoPoint that contains the latitude/longitude of the map location tapped
  - ❖ The MapView that was tapped to trigger this event
- When you override onTap, the method should return true if it has handled a particular tap and false to let another Overlay handle it.

```
@Override
public boolean onTap(GeoPoint point, MapView mapView) {
  // Perform hit test to see if this overlay is handling the click
  if ([ ... perform hit test ... ]) {
    // TODO [ ... execute on tap functionality ... ]
    return true;
  }
  // If not handled return false
  return false;
}
```

4.7.5 Adding and Removing Overlays:

- Each MapView contains a list of Overlays currently displayed. You can get a reference to this list by calling getOverlays, as shown in the following snippet:

```
List<Overlay> overlays = mapView.getOverlays();
```

- Adding and removing items from the list is thread-safe and synchronized, so you can modify and query the list safely. You should still iterate over the list within a synchronization block synchronized on the List.
- To add an Overlay onto a Map View, create a new instance of the Overlay and add it to the list, as shown in the following snippet.

```
MyOverlay myOverlay = new MyOverlay();
overlays.add(myOverlay);
mapView.postInvalidate();
```

- The added Overlay displays the next time the Map View is redrawn, so it' s usually a good practice to call postInvalidate after you modify the list to update the changes on the map display.

4.7.6 Introducing My Location Overlay:

- The MyLocationOverlay class is a native Overlay designed to show your current location and orientation on a MapView.

- To use My Location Overlay you need to create a new instance, passing in the application Context and target Map View, and add it to the MapView's Overlay list, as shown here:

```
List<Overlay> overlays = mapView.getOverlays();
MyLocationOverlay myLocationOverlay = new MyLocationOverlay(this, mapView);
overlays.add(myLocationOverlay);
```

- The following snippet shows how to enable both the compass and marker.

```
myLocationOverlay.enableCompass();
myLocationOverlay.enableMyLocation();
```

4.7.7 <u>Itemized Overlays and Overlay Items:</u>

- OverlayItems are used to supply simple marker functionality to your Map Views via the ItemizedOverlay class.
- ItemizedOverlay is a generic class that enables you to create extensions based on any class that implements OverlayItem.
- ItemizedOverlays provide a convenient shortcut for adding markers to a map, letting you assign a marker image and associated text to a particular geographical position.
- The ItemizedOverlay instance handles the drawing, placement, click handling, focus control, and layout optimization of each OverlayItem marker for you.
- To add an ItemizedOverlay marker layer to your map, create a new class that extends ItemizedOverlay<OverlayItem>.

  - **Creating a new Itemized Overlay**

```
import android.graphics.drawable.Drawable;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.ItemizedOverlay;
import com.google.android.maps.OverlayItem;
public class MyItemizedOverlay extends ItemizedOverlay<OverlayItem> {
  public MyItemizedOverlay(Drawable defaultMarker) {
    super(boundCenterBottom(defaultMarker));
    populate();
  }
  @Override
  protected OverlayItem createItem(int index) {
    switch (index) {
      case 0:
        Double lat = 37.422006*1E6;
Double lng = -122.084095*1E6;
        GeoPoint point = new GeoPoint(lat.intValue(), lng.intValue());
        OverlayItem oi;
        oi = new OverlayItem(point, "Marker", "Marker Text");
        return oi;
    }
    return null;
  }
  @Override
  public int size() {
    // Return the number of markers in the collection
    return 1;
  }
}
```

- To add an ItemizedOverlay implementation to your map, create a new instance (passing in the Drawable marker image to use for each marker) and add it to the map's Overlay list.
- The map markers placed by the Itemized Overlay use state to indicate if they are selected.

```
List<Overlay> overlays = mapView.getOverlays();
Drawable drawable = getResources().getDrawable(R.drawable.marker);
MyItemizedOverlay markers = new MyItemizedOverlay(drawable);
overlays.add(markers);
```

4.7.8 <u>Pinning Views to the Map and Map Positions:</u>

- You can pin any View-derived object to a Map View (including layouts and other View Groups), attaching it to either a screen position or a geographical map location.
- The View moves to follow its pinned position on the map, effectively acting as an interactive map marker. As a more resource-intensive solution, this is usually reserved for supplying the detail "balloons" often displayed on mashups to provide further detail when a marker is clicked.
- You implement both pinning mechanisms by calling addView on the MapView, usually from the onCreate or onRestore methods within the MapActivity containing it. Pass in the View you want to pin and the layout parameters to use.
- The MapView.LayoutParams parameters you pass in to addView determine how, and where, the View is added to the map.
- To add a new View to the map relative to the screen, specify a new MapView.LayoutParams, including arguments that set the height and width of the View, the x/y screen coordinates to pin to, and the alignment to use for positioning.

```
int y = 10;
int x = 10;
EditText editText1 = new EditText(getApplicationContext());
editText1.setText("Screen Pinned");
MapView.LayoutParams screenLP;
screenLP = new MapView.LayoutParams(MapView.LayoutParams.WRAP_CONTENT,
                    MapView.LayoutParams.WRAP_CONTENT,
                    x, y,
                    MapView.LayoutParams.TOP_LEFT);
mapView.addView(editText1, screenLP);
```

- To pin a View relative to a physical map location, pass four parameters when constructing the new Map View LayoutParams, representing the height, width, GeoPoint to pin to, and layout alignment.

  ```
  Double lat = 37.422134*1E6;
  Double lng = -122.084069*1E6;
  GeoPoint geoPoint = new GeoPoint(lat.intValue(), lng.intValue());
  MapView.LayoutParams geoLP;
  geoLP = new MapView.LayoutParams(MapView.LayoutParams.WRAP_CONTENT,
                          MapView.LayoutParams.WRAP_CONTENT,
                          geoPoint,
                          MapView.LayoutParams.TOP_LEFT);
  EditText editText2 = new EditText(getApplicationContext());
  editText2.setText("Location Pinned");
  mapView.addView(editText2, geoLP);
  ```

- Panning the map can leave the first TextView stationary in the upper-left corner, whereas the second TextView moves to remain pinned to a particular position on the map.
- To remove a View from a Map View, call removeView, passing in the View instance you want to remove, as shown here.

  ```
  mapView.removeView(editText2);
  ```

****