ANDROID

SAI TECH

History Of Android

What Is Android ?

Android is a operating system for mobile device and also open source software for developing mobile applications.

Android was released by "Android Incorporation Company" in 2003. Later in 2005 "google" company acquired this company. In 2007 Google company formed one group that group name is called as "OHA"(Open Handset Alliance).This OHA contains more than 30companies. Among of them 80%companies are mobile manufacture companies. Some of the mnc companies are Samsung, LG, Alcatel, HTC, Intel, MicroMax, Motorola, karbon, Vodafone etc..

The First Android mobile device was released by "HTC T-MobileG1" model in 2008. That model name is "T-MobileG1" in 2008. This course was introduced first in ameerpet by SAITECHNOLOGIES in 2009 on feb 3rd. Today Except reliance, aircel and airtel remaining all companies they are using Android Operating system.

Today Android is not only operating System for mobiles and also we are using it in different areas that are Tablets, Internet TV's, Mp4 players, E-ReaderDevices. More than 7lakhs apps are available in android market. Today android mobile we can get at less price and more features. Akasha Tablet also comes with Android OS.

What you need for become a ANDROID developer :

Qt : Does complete java need for learning android?

Ans : No, we need basic knowledge of core java. Generally Java categorized in three ways . They are 1. J2SE

2. J2EE(servlets,jsp,struts,spring,JSF,Hibernate,ajax)
3. J2ME(For developing mobile apps in java)

From the above we don't need j2ee and j2me concepts.we need some of the concepts from J2SE part.

They are
1. Basics of java.
2. OOP's concepts.
3. Basics of Exceptions and Threads.
4. Basics of Files(File,FileInputStream,FileOutputStream)
5. Basics of Collections(ArrayList,Vector).
6. String Manipulation (String, StringBuilder and StringBuffer, StringTokenizer).

CONTENTS

Chapter 1 What Is Android?

Android is a mobile operating system that is based on a modified version of Linux. It was originally developed by a startup of the same name, Android, Inc. In 2005, as part of its strategy to enter the mobile space, Google purchased Android and took over its development work (as well as its development team).

Google wanted Android to be open and free; hence, most of the Android code was released under the open-source Apache License, which means that anyone who wants to use Android can do so by downloading the full Android source code. Moreover, vendors (typically hardware manufacturers) can add their own proprietary extensions to Android and customize Android to differentiate their products from others. This simple development model makes Android very attractive and has thus piqued the interest of many vendors. This has been especially true for companies affected by the phenomenon of Apple's iPhone, a hugely successful product that revolutionized the smartphone industry. Such companies include Motorola and Sony Ericsson, which for many years have been developing their own mobile operating systems. When the iPhone was launched, many of these manufacturers had to scramble to find new ways of revitalizing their products. These manufacturers see Android as a solution — they will continue to design their own hardware and use Android as the operating system that powers it.

The main advantage of adopting Android is

- It offers a unified approach to application development.
- Developers need only develop for Android, and their applications should be able to run on numerous different devices, as long as the devices are powered using Android. In the world of smart phones, applications are the most important part of the success chain.
- Device manufacturers therefore see Android as their best hope to challenge the onslaught of the iPhone, which already commands a large base of applications.

Android is a combination of three components:

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

Native Android Applications

Android phones will normally come with a suite of generic preinstalled applications that are part of the **Android Open Source Project (AOSP),** including, but not necessarily limited to:

- 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
- The home screen
- An alarm clock

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

- The Android Market client for downloading third-party Android applications
- A fully-featured mobile Google Maps application including StreetView, driving directions
- and turn-by-turn navigation, satellite view, and traffic conditions
- The Gmail mail client
- The Google Talk instant-messaging client
- The YouTube video player.

Android SDK Features

The true appeal of Android as a development environment lies in the APIs it provides.

- No licensing, distribution, or development fees or release approval processes
- Wi-Fi hardware access
- GSM, EDGE, and 3G 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
- Full multimedia hardware control, including playback and recording with the camera and microphone
- APIs for using sensor hardware, including accelerometers and the compass
- Libraries for using Bluetooth for peer-to-peer data transfer

- IPC message passing
- Shared data stores
- Background applications and processes
- Home-screen Widgets, Live Folders, and LiveWallpaper
- The ability to integrate application search results into the system search
- An integrated open-source HTML5WebKit-based browser
- Full support for applications that integrate map controls as part of their user interface
- Mobile-optimized hardware-accelerated graphics, including a path-based 2D graphics library
- and support for 3D graphics using OpenGL ES 2.0
- Media libraries for playing and recording a variety of audio/video or still image formats
- Localization through a dynamic resource framework
- An application framework that encourages reuse of application components and the replacement of native applications

Android Project Structure

The Android build system is organized around a specific directory tree structure for your Android project, much like any other Java project. The specifics, though, are fairly unique to Android and what it all does to prepare the actual application that will run on the device or emulator.

When you create a new Android project we get five key items in the project's root directory:

• **AndroidManifest.xml,** which is an XML file describing the application being built and what components – activities, services, etc. – are being supplied by that application

• **build.xml** - which is an Ant script for compiling the application and installing it on the device

• **bin** - which holds the application once it is compiled

• **src** - which holds the Java source code for the application

• **res** - which holds "resources", such as icons, GUI layouts, and the like, that get packaged with the compiled Java in the application

• **assets** - which hold other static files you wish packaged with the application for deployment onto the device

We will also find that your project has a res/ directory tree. This holds **"resources"** – static files that are packaged along with your application, either in their original form or, occasionally, in a preprocessed form. Some of the subdirectories you will find or create under res/ include:

• **res/drawable/** for images (PNG, JPEG, etc.)

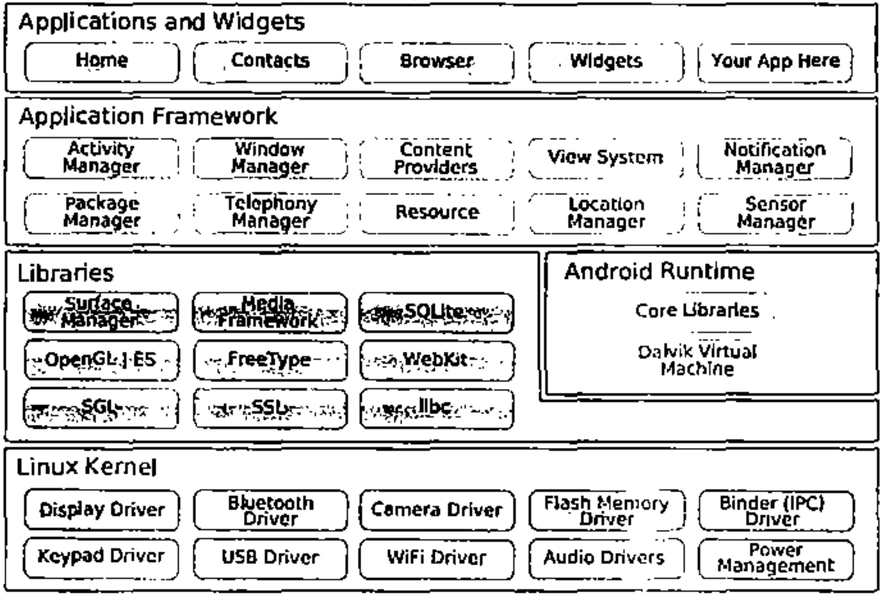
- **res/layout/** for XML-based UI layout specifications
- **res/raw/** for general-purpose files
- **res/values/** for strings, dimensions, and the like
- **res/xml/** for other general-purpose XML files you wish to ship

The Dalvik Virtual Machine (DVM)

One of the key elements of Android is the Dalvik Virtual Machine. Rather than use a traditional Java virtual machine (VM) such as Java ME (Java Mobile Edition), 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 directly on the underlying Linux OS.

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 never have to worry about a particular hardware implementation. The Dalvik VM executes Dalvik executable files, a format optimized to ensure minimal memory footprint. We create.dex executables by transforming Java language compiled classes using the tools supplied within the SDK.

Android Architecture



The Android OS is roughly divided into five sections in four main layers:

- **Linux kernel** — This is the kernel on which Android is based. This layer contains all the lowlevel device drivers for the various hardware components of an Android device.

- **Libraries** — These contain all the code that provides the main features of an Android OS. For example, the SQLite library provides database support so that an application can use it for data storage. The WebKit library provides functionalities for web browsing.
- **Android runtime** — At the same layer as the libraries, the Android runtime provides a set of core libraries that enable developers to write Android apps using the Java programming language. The Android runtime also includes the Dalvik virtual machine, which enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine (Android applications are compiled into the Dalvik executables). Dalvik is a specialized virtual machine designed specifically for Android and optimized for battery-powered mobile devices with limited memory and CPU.
- **Application framework** — Exposes the various capabilities of the Android OS to application developers so that they can make use of them in their applications.
- **Applications** — At this top layer, you will find applications that ship with the Android device (such as Phone, Contacts, Browser, etc.), as well as applications that you download and install from the Android Market.

Chapter 2 Android Installation

Because Android applications run within the Dalvik virtual machine, you can write them on any platform that supports the developer tools. This currently includes the following:

- MicrosoftWindows (XP or later)
- Mac OS X 10.4.8 or later (Intel chips only)
- Linux

To get started, you'll need to download and install the following:

The Android SDK

- Java Development Kit (JDK) 5 or 6
- Download the latest JDK from Sun at **http://java.sun.com/javase/downloads/index.jsp**

Downloading and Installing the SDK

- We can download the latest version of the SDK for your development platform from the Android development homepage at **http://developer.android.com/sdk/index.html**
- The SDK is presented as a ZIP file containing only the latest version of the Android developer tools. Install it by unzipping the SDK into a new folder.
- Before we can begin development you need to add at least one SDK Platform; do this on Windows by running the ''SDK Setup.exe'' executable, or on MacOS or Linux by running the ''android'' executable in the tools subfolder.
- In the screen that appears, select the ''Available Packages'' option on the left panel, and then select the SDK Platform versions you wish to install in the ''Sources, Packages, and Archives'' panel on the right.
- The selected platform will then be downloaded to your SDK installation folder and will contain the API libraries, documentation, and several sample applications.

Developing with Eclipse

Using Eclipse with the ADT plug-in for your Android development offers some significant advantages.

- Eclipse is an open-source IDE (integrated development environment) particularly popular for Java. development.
- It's available for download for each of the development platforms supported by Android Windows,MacOS,andLinux) from the Eclipse foundation homepage: **www.eclipse.org/downloads/**

There are many variations available; the following is the recommended configuration for Android:

- Eclipse 3.4 or 3.5 (Galileo)
- Eclipse JDT plug-in
- WST

WST and the JDT plug-in are included in most Eclipse IDE packages.

Installing Eclipse consists of uncompressing the download into a new folder. When that's done, run the eclipse executable. When it starts for the first time, create a new workspace for your Android development projects.

Installing the ADT Plug-In

Install the developer tools plug-in by following these steps:

1. Select **Help ⇨ Install New Software. . .** from within Eclipse.

2. In the resulting dialog box enter the following address into the **Work With** text entry box and press Enter: https://dl-ssl.google.com/android/eclipse/

3. Eclipse will now search for the ADT plug-in. When finished it will display the available plugin. Select it by clicking the checkbox next to the Developer Tools root node, and click **Next**.

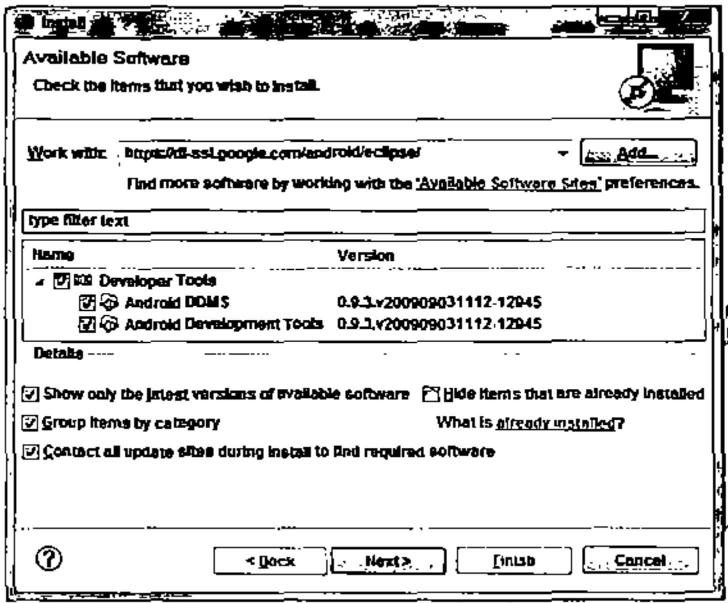
4. Eclipse will now download the plug-in. When it's finished, ensure both the Android DDMS and Android Developer Tools plug-ins are selected and click **Next.**

Read and then **Accept** the terms of the license agreement, and click **Next** and then **Finish**. As the ADT plug-in is not signed, you'll be prompted before the installation continues.

6. When installation is complete you'll have to restart Eclipse and update the ADT preferences. Restart and select **Window ⇨ Preferences. . .** (or **Eclipse ⇨ Preferences** for MacOS).

7. Then select **Android** from the left panel.

8. Click **Browse. . .** and navigate to the folder into which you unzipped the Android SDK; then click **Apply.** The list will then update to display each of the available SDK targets, as in Figure 2-3. Click **OK** to complete the SDK installation.



Available Software

Check the items that you wish to install.

Work with: https://dl-ssl.google.com/android/eclipse/

Find more software by working with the 'Available Software Sites' preferences.

type filter text

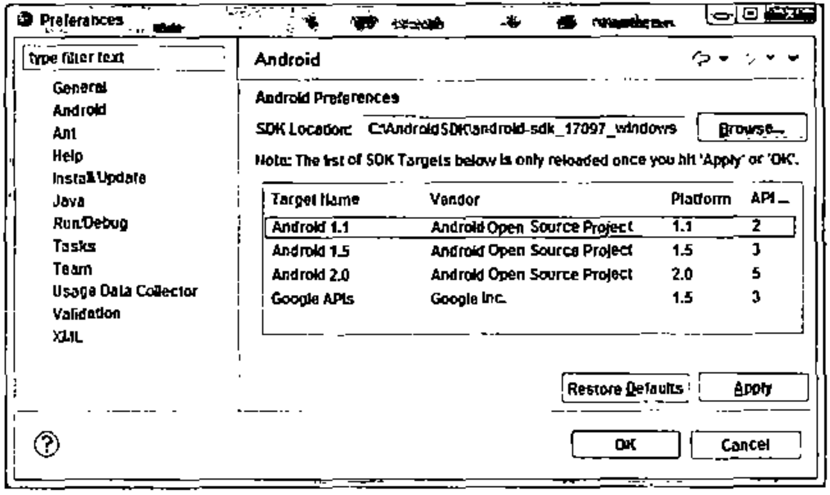
Name	Version
☑ Developer Tools	
☑ Android DDMS	0.9.3.v200909031112-12945
☑ Android Development Tools	0.9.3.v200909031112-12945

Details

☑ Show only the latest versions of available software ☐ Hide items that are already installed

☑ Group items by category What is already installed?

☑ Contact all update sites during install to find required software



Android

Android Preferences

SDK Location: C:\AndroidSDK\android-sdk_17097_windows

Note: The list of SDK Targets below is only reloaded once you hit 'Apply' or 'OK'.

Target Name	Vendor	Platform	API...
Android 1.1	Android Open Source Project	1.1	2
Android 1.5	Android Open Source Project	1.5	3
Android 2.0	Android Open Source Project	2.0	5
Google APIs	Google Inc.	1.5	3

Creating Your First Android Application

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

Starting a New Android Project

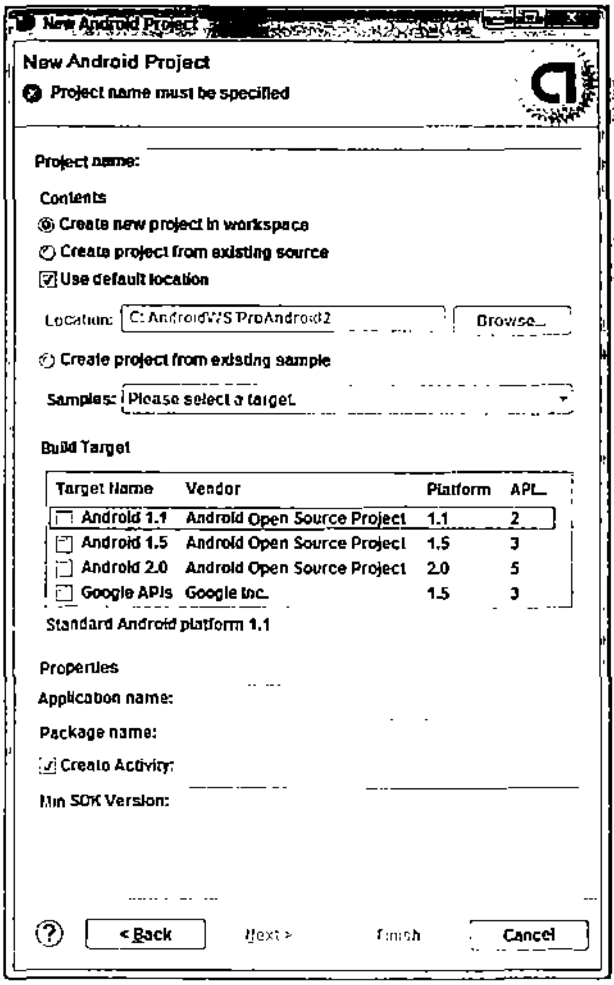
To create a new Android project using the Android New Project Wizard, do the following:

1. Select **File** ⇨ New ⇨ **Project.**

2. Select the **Android Project** application type from the Android folder and click **Next.**

3. In the dialog that appears (shown in Figure 2-4), enter the details for your new project. The ''Project name'' is the name of your project file; the ''Package name'' specifies its java package; Create Activity lets you specify the name of a class that will be your initial Activity; and the ''Application name'' is the friendly name for your application. ''Min SDK Version'' lets you specify the minimum version of the SDK that your application will run on.

When you've entered the details, click **Finish.**



New Android Project

New Android Project

Project name must be specified

Project name:

Contents

◉ Create new project in workspace

○ Create project from existing source

☑ Use default location

Location: C:\AndroidWS\ProAndroid2

○ Create project from existing sample

Samples: Please select a target.

Build Target

Target Name	Vendor	Platform	API...
□ Android 1.1	Android Open Source Project	1.1	2
□ Android 1.5	Android Open Source Project	1.5	3
□ Android 2.0	Android Open Source Project	2.0	5
□ Google APIs	Google Inc.	1.5	3

Standard Android platform 1.1

Properties

Application name:

Package name:

☑ Create Activity:

Min SDK Version:

Creating AVDManager (Emulator) :

Window
 |..........>AVDmanager
   |.................>new

|.........>Name(enter some name Ex:saitech)

|................>Target(select version with API level Ex :2.2,8)
|...............> create AVD.

Run our application on created Emulator(Ex : On saitech) :

Select Project
|....................>Right Click(Run As)
|................>Run Configuration
|............>Target
|................>saitech (Select)
|...............> Run

Creating a Launch Configuration

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 virtual device and emulator options to use
- Input/output settings (including console defaults)

We can specify different launch configurations for running and debugging applications.

The following steps show how to create a launch configuration for an Android application:

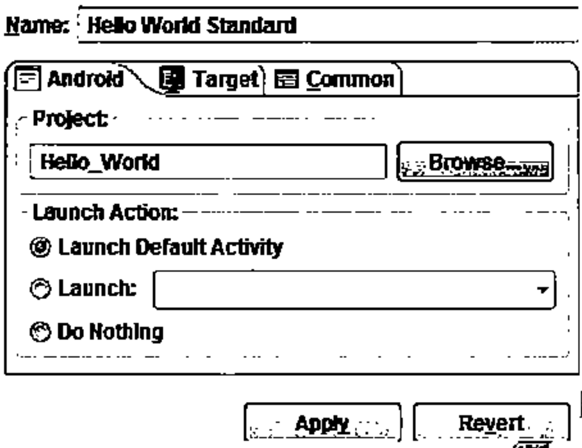
1. Select **Run Configurations. . .** or **Debug Configurations. . .** from the **Run** menu.

2. Right-click **Android Application** on the project type list, 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. Now 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.

5. Use the **Target** tab shown in Figure 2-6 to select the default virtual device to launch on, or select manual to select a device or AVD each time.



Name: Hello World Standard

Android | Target | Common

Project:

Hello_World

Launch Action:

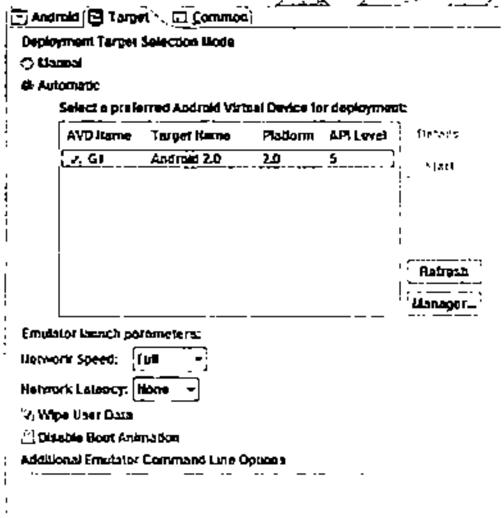
● Launch Default Activity

○ Launch:

○ Do Nothing

Finally, set any additional properties in the **Common** tab.

7. Click **Apply,** and your launch configuration will be saved. **Running and Debugging Your Android Applications**



Android | Target | Common

Deployment Target Selection Mode

○ Manual

● Automatic

Select a preferred Android Virtual Device for deployment:

AVD Name	Target Name	Platform	API Level
✓ G1	Android 2.0	2.0	5

Emulator launch parameters:

Network Speed: Full

Network Latency: None

☑ Wipe User Data

□ Disable Boot Animation

Additional Emulator Command Line Options

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 to use.

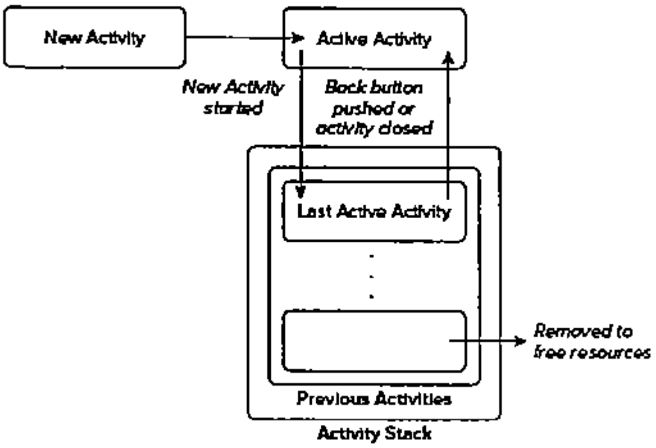
Chapter 3 Android Application & Activities

An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" queue mechanism, so, when the user is done with the current activity and presses the BACK key, it is popped from the stack (and destroyed) and the previous activity resumes. (The back stack is discussed more in the Tasks and Back Stack document.)

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

The rest of this document discusses the basics of how to build and use an activity, including a complete discussion of how the activity lifecycle works, so you can properly manage the transition between various activity states.



Creating an Activity

To create an activity, you must create a subclass of Activity (or an existing subclass of it). In your subclass, you need to implement callback methods that the system calls when the activity transitions between various states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed. The two most important callback methods are:

onCreate()

> You must implement this method. The system calls this when creating your activity. Within your implementation, you should initialize the essential components of your activity. Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface.

onPause()

> The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

There are several other lifecycle callback methods that you should use in order to provide a fluid user experience between activities and handle unexpected interuptions that cause your activity to be stopped and even destroyed. All of the lifecycle callback methods are discussed later, in the section about Managing the Activity Lifecycle.

Implementing a user interface

The user interface for an activity is provided by a hierarchy of views—objects derived from the View class. Each view controls a particular rectangular space within the activity's window and can respond to user interaction. For example, a view might be a button that initiates an action when the user touches it.

Android provides a number of ready-made views that you can use to design and organize your layout. "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image. "Layouts" are views derived from ViewGroup that provide a unique layout model for its child views, such as a linear layout, a grid layout, or relative layout. You can also subclass the View and ViewGroup classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout.

The most common way to define a layout using views is with an XML layout file saved in your application resources. This way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior. You can set the layout as the UI for your activity with setContentView(), passing the resource ID for the layout. However, you can also create new Views in your activity code and build a view hierarchy by inserting new Views into a ViewGroup, then use that layout by passing the root ViewGroup to setContentView().

For information about creating a user interface, see the User Interface documentation.

Declaring the activity in the manifest

You must declare your activity in the manifest file in order for it to be accessible to the system. To declare your activity, open your manifest file and add an <activity> element as a child of the <application> element. For example:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
  ...
</manifest >
```

There are several other attributes that you can include in this element, to define properties such as the label for the activity, an icon for the activity, or a theme to style the activity's UI. The android:name attribute is the only required attribute—it specifies the class name of the activity. Once you publish your application, you should not change this name, because if you do, you might break some functionality, such as application shortcuts (read the blog post, Things That Cannot Change).

See the <activity> element reference for more information about declaring your activity in the manifest.

Using intent filters

An <activity> element can also specify various intent filters—using the <intent-filter> element—in order to declare how other application components may activate it.

When you create a new application using the Android SDK tools, the stub activity that's created for you automatically includes an intent filter that declares the activity responds to the "main" action and should be placed in the "launcher" category. The intent filter looks like this:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The <action> element specifies that this is the "main" entry point to the application. The <category> element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

If you intend for your application to be self-contained and not allow other applications to activate its activities, then you don't need any other intent filters. Only one activity should have the "main" action and "launcher" category, as in the previous example. Activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents (as discussed in the following section).

However, if you want your activity to respond to implicit intents that are delivered from other applications (and your own), then you must define additional intent filters for your activity. For each type of intent to which you want to respond, you must include an <intent-filter> that includes an <action> element and, optionally, a <category> element and/or a <data> element. These elements specify the type of intent to which your activity can respond.

For more information about how your activities can respond to intents, see the Intents and Intent Filters document.

Starting an Activity

You can start another activity by calling startActivity(), passing it an Intent that describes the activity you want to start. The intent specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An intent can also carry small amounts of data to be used by the activity that is started.

When working within your own application, you'll often need to simply launch a known activity. You can do so by creating an intent that explicitly defines the activity you want to start, using the class name. For example, here's how one activity starts another activity named SignInActivity:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

However, your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable—you can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the intent, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

The EXTRA_EMAIL extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

Starting an activity for a result

Sometimes, you might want to receive a result from the activity that you start. In that case, start the activity by calling startActivityForResult() (instead of startActivity()). To then receive the result from the subsequent activity, implement the onActivityResult() callback method. When the subsequent activity is done, it returns a result in an Intent to your onActivityResult() method.

For example, perhaps you want the user to pick one of their contacts, so your activity can do something with the information in that contact. Here's how you can create such an intent and handle the result:

```
private void pickContact() {
   // Create an intent to "pick" a contact, as defined by the content provider URI
   Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
   startActivityForResult(intent, PICK_CONTACT_REQUEST);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
   // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
   if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST) {
      // Perform a query to the contact's content provider for the contact's name
      Cursor cursor = getContentResolver().query(data.getData(),
      new String[] {Contacts.DISPLAY_NAME}, null, null, null);
      if (cursor.moveToFirst()) { // True if the cursor is not empty
         int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
         String name = cursor.getString(columnIndex);
         // Do something with the selected contact's name...
      }
   }
```

}

This example shows the basic logic you should use in your onActivityResult() method in order to handle an activity result. The first condition checks whether the request was successful—if it was, then the resultCode will be RESULT_OK—and whether the request to which this result is responding is known—in this case, the requestCode matches the second parameter sent with startActivityForResult(). From there, the code handles the activity result by querying the data returned in an Intent (the data parameter).

What happens is, a ContentResolver performs a query against a content provider, which returns a Cursor that allows the queried data to be read. For more information, see the Content Providers document.

For more information about using intents, see the Intents and Intent Filters document.

Shutting Down an Activity

You can shut down an activity by calling its finish() method. You can also shut down a separate activity that you previously started by calling finishActivity().

Note: In most cases, you should not explicitly finish an activity using these methods. As discussed in the following section about the activity lifecycle, the Android system manages the life of an activity for you, so you do not need to finish your own activities. Calling these methods could adversely affect the expected user experience and should only be used when you absolutely do not want the user to return to this instance of the activity.

Managing the Activity Lifecycle

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application. The lifecycle of an activity is directly affected by its association with other activities, its task and back stack.

An activity can exist in essentially three states:

Resumed

> The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running".)

Paused

Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive (the Activity object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but can be killed by the system in extremely low memory situations.

Stopped

The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive (the Activity object is retained in memory, it maintains all state and member information, but is *not* attached to the window manager). However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its finish() method), or simply killing its process. When the activity is opened again (after being finished or killed), it must be created all over.

Implementing the lifecycle callbacks

When an activity transitions into and out of the different states described above, it is notified through various callback methods. All of the callback methods are hooks that you can override to do appropriate work when the state of your activity changes. The following skeleton activity includes each of the fundamental lifecycle methods:

```
public class ExampleActivity extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }
  protected void onStart() {
    super.onStart();
    // The activity is about to become visible.
  }
rotected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be "paused").   }
protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }}
```

Taken together, these methods define the entire lifecycle of an activity. By implementing these methods, you can monitor three nested loops in the activity lifecycle:

- The **entire lifetime** of an activity happens between the call to onCreate() and the call to onDestroy(). Your activity should perform setup of "global" state (such as defining layout) in onCreate(), and release all remaining resources in onDestroy(). For example, if your activity has a thread running in the background to download data from the network, it might create that thread in onCreate() and then stop the thread in onDestroy().
- The **visible lifetime** of an activity happens between the call to onStart() and the call to onStop(). During this time, the user can see the activity on-screen and interact with it. For example, onStop() is called when a new activity starts and this one is no longer visible. Between these two methods, you can maintain resources that are needed to show the activity to the user. For example, you can register a BroadcastReceiver in onStart() to monitor changes that impact your UI, and unregister it in onStop() when the user can no longer see what you are displaying. The system might call onStart() and onStop() multiple times during the entire lifetime of the activity, as the activity alternates between being visible and hidden to the user.
- The **foreground lifetime** of an activity happens between the call to onResume() and the call to onPause(). During this time, the activity is in front of all other activities on screen and has

user input focus. An activity can frequently transition in and out of the foreground—for example, onPause() is called when the device goes to sleep or when a dialog appears. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

Figure 1 illustrates these loops and the paths an activity might take between states. The rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.

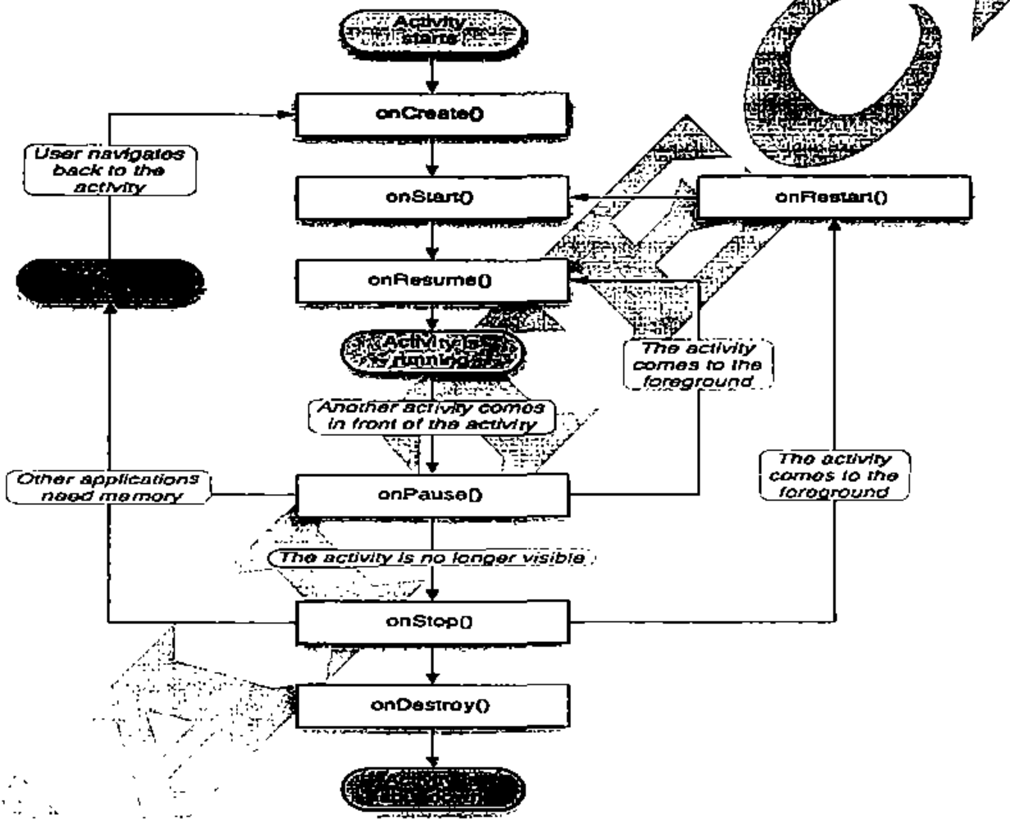

Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()

Figure 1. The activity lifecycle.

The same lifecycle callback methods are listed in table 1, which describes each of the callback methods in more detail and locates each one within the activity's overall lifecycle, including whether the system can kill the activity after the callback method completes.

Table 1. A summary of the activity lifecycle's callback methods.

Method	Description	Killable after?	Next
onCreate()	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured (see Saving Activity State, later). Always followed by onStart().	No	onStart()
onRestart()	Called after the activity has been stopped, just prior to it being started again. Always followed by onStart()	No	onStart()

onStart()	Called just before the activity becomes visible to the user.
	Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.
		No	onResume()
			or
			onStop()

onResume()	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it. Always followed by onPause().	No	onPause()

Method	Description	Killable after?	Next

onPause()	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.
	Followed either by onResume() if the activity returns back to the front, or by onStop() if it becomes invisible to the user.
		Yes	onResume()
			or
			onStop()

onStop()	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.
	Followed either by onRestart() if the activity is coming back to interact with the user, or by onDestroy() if this activity is going away.
		Yes	onRestart()
			or
			onDestroy()

onDestroy()	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called finish() on it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between	Yes	*nothing*

Method	Description	Killable after?	Next
	these two scenarios with the isFinishing() method.		

The column labeled "Killable after?" indicates whether or not the system can kill the process hosting the activity at any time *after the method returns*, without executing another line of the activity's code. Three methods are marked "yes": (onPause(), onStop(), and onDestroy()). Because onPause() is the first of the three, once the activity is created, onPause() is the last method that's guaranteed to be called before the process *can* be killed—if the system must recover memory in an emergency, then onStop() and onDestroy() might not be called. Therefore, you should use onPause() to write crucial persistent data (such as user edits) to storage. However, you should be selective about what information must be retained during onPause(), because any blocking procedures in this method block the transition to the next activity and slow the user experience.

Methods that are marked "No" in the Killable column protect the process hosting the activity from being killed from the moment they are called. Thus, an activity is killable from the time onPause() returns to the time onResume() is called. It will not again be killable until onPause() is again called and returns.

Note: An activity that's not technically "killable" by this definition in table 1 might still be killed by the system—but that would happen only in extreme circumstances when there is no other recourse. When an activity might be killed is discussed more in the Processes and Threading document.

Saving activity state

The introduction to Managing the Activity Lifecycle briefly mentions that when an activity is paused or stopped, the state of the activity is retained. This is true because the Activity object is still held in memory when it is paused or stopped—all information about its members and current state is still alive. Thus, any changes the user made within the activity are retained in memory, so that when the activity returns to the foreground (when it "resumes"), those changes are still there.

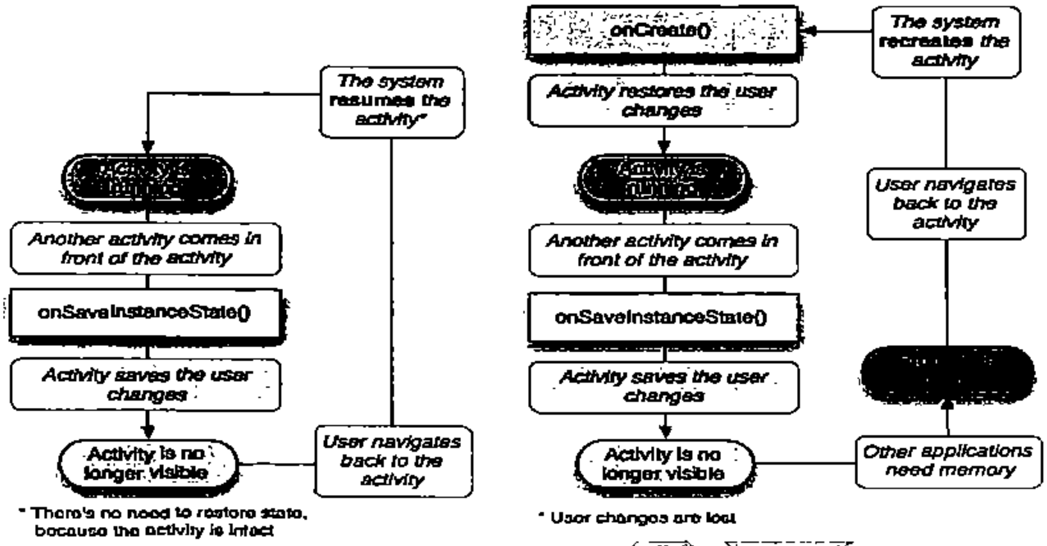

The system resumes the activity*
Another activity comes in front of the activity
onSaveInstanceState()
Activity saves the user changes
Activity is no longer visible
User navigates back to the activity
* There's no need to restore state, because the activity is intact
onCreate()
The system recreates the activity
Activity restores the user changes
Another activity comes in front of the activity
onSaveInstanceState()
Activity saves the user changes
Activity is no longer visible
User navigates back to the activity
Other applications need memory
* User changes are lost

Figure 2. The two ways in which an activity returns to user focus with its state intact: either the activity is stopped, then resumed and the activity state remains intact (left), or the activity is destroyed, then recreated and the activity must restore the previous activity state (right).

However, when the system destroys an activity in order to recover memory, the Activity object is destroyed, so the system cannot simply resume it with its state intact. Instead, the system must recreate the Activity object if the user navigates back to it. Yet, the user is unaware that the system destroyed the activity and recreated it and, thus, probably expects the activity to be exactly as it was. In this situation, you can ensure that important information about the activity state is preserved by implementing an additional callback method that allows you to save information about the state of your activity and then restore it when the the system recreates the activity.

The callback method in which you can save information about the current state of your activity is onSaveInstanceState(). The system calls this method before making the activity vulnerable to being destroyed and passes it a Bundle object. The Bundle is where you can store state information about the activity as name-value pairs, using methods such as putString(). Then, if the system kills your activity's process and the user navigates back to your activity, the system passes the Bundle to onCreate() so you can restore the activity state you saved during onSaveInstanceState(). If there is no state information to restore, then the Bundle passed to onCreate() is null.

Note: There's no guarantee that onSaveInstanceState() will be called before your activity is destroyed, because there are cases in which it won't be necessary to save the state (such as when the user leaves your activity using the BACK key, because the user is explicitly closing the activity). If the method is called, it is always called before onStop() and possibly before onPause().

However, even if you do nothing and do not implement onSaveInstanceState(), some of the activity state is restored by the Activity class's default implementation of onSaveInstanceState(). Specifically, the default implementation calls onSaveInstanceState() for every View in the layout, which allows each view to provide information about itself that should be saved. Almost every widget in the Android framework implements this method as appropriate, such that any visible changes to the UI are automatically saved and restored when your activity is recreated. For example, the EditText widget saves any text entered by the user and the CheckBox widget saves whether it's checked or not. The only work required by you is to provide a unique ID (with the android:id attribute) for each widget you want to save its state. If a widget does not have an ID, then it cannot save its state.

You can also explicitly stop a view in your layout from saving its state by setting the android:saveEnabled attribute to "false" or by calling the setSaveEnabled() method. Usually, you should not disable this, but you might if you want to restore the state of the activity UI differently.

Although the default implementation of onSaveInstanceState() saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default).

Because the default implementation of onSaveInstanceState() helps save the state of the UI, if you override the method in order to save additional state information, you should always call the superclass implementation of onSaveInstanceState() before doing any work.

Note: Because onSaveInstanceState() is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use onPause() to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

A good way to test your application's ability to restore its state is to simply rotate the device so that the screen orientation changes. When the screen orientation changes, the system destroys and recreates the activity in order to apply alternative resources that might be available for the new orientation. For this reason alone, it's very important that your activity completely restores its state when it is recreated, because users regularly rotate the screen while using applications.

Handling configuration changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android restarts the running Activity (onDestroy() is called, followed immediately by onCreate()). The restart behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources that you've provided. If you design your activity to properly handle this event, it will be more resilient to unexpected events in the activity lifecycle.

The best way to handle a configuration change, such as a change in the screen orientation, is to simply preserve the state of your application using onSaveInstanceState() and onRestoreInstanceState() (or onCreate()), as discussed in the previous section.

For a detailed discussion about configuration changes that happen at runtime and how you should handle them, read Handling Runtime Changes.

Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity pauses and stops (though, it won't stop if it's still visible in the background), while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other. Here's the order of operations that occur when Activity A starts Acivity B:

1. Activity A's onPause() method executes.
2. Activity B's onCreate(), onStart(), and onResume() methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its onStop() method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another. For example, if you must write to a database when the first activity stops so that the following activity can read it, then you should write to the database during onPause() instead of during onStop().

Chapter 4 Android Manifest File

Each Android project includes a manifest file, AndroidManifest.xml, stored in the root of the project hierarchy. The manifest lets you define the structure and metadata of your application, its components, and its requirements.

It includes nodes for each of the components (Activities, Services, Content Providers, and Broadcast Receivers) that make up your application and, using Intent Filters and Permissions, determines how they interact with each other and with other applications.

The manifest also offers attributes to specify application metadata (like its icon or theme), and additional top level nodes can be used for security settings, unit tests, and defining hardware and platform support requirements, as described below.

The manifest is made up of a root <manifest> tag with a package attribute set to the project's package. It usually includes an xmlns:android attribute that supplies several system attributes used within the file.

Use the versionCode attribute to define the current application version as an integer. This value is used internally to compare application versions. Use the versionName attribute to specify a public version number that is displayed to users.

A typical manifest node is shown in the following XML snippet:

```
<manifest xmlns:android=http://schemas.android.com/apk/res/android
package="com.my_domain.my_app"
android:versionCode="1"
android:versionName="0.9 Beta">
[ ... manifest nodes ... ]
</manifest>
```

The <manifest> tag includes nodes that define the application components, security settings, test classes, and requirements that make up your application. The following list gives a summary of the available <manifest> node tags, and an XML snippet demonstrating how each one is used:

➤ uses-sdk This node lets you define a minimum, maximum, and target SDK version that must be available on a device in order for your application to function properly. Using a combination of minSDKVersion, maxSDKVersion, and targetSDKVersion attributes you can restrict which devices your application can run on, based on the SDK version supported by the installed platform.

The minimum SDK version specifies the lowest version of the SDK that includes the APIs you have used in your application. If you fail to specify a minimum version one will be assumed and your application will crash if it attempts to access APIs that aren't available on the host device.

The maximum SDK version lets you define an upper limit you are willing to support. Your application will not be visible on the Market for devices running a higher platform release. It's good practice *not* to set the maximum SDK value unless you know your application will definitely not work on newer platform releases.

The target SDK version attribute lets you specify the platform against which you did your development and testing. Setting a target SDK version tells the system that there is no need to apply any forward- or backward- compatibility changes to support that particular version.

```
<uses-sdk android:minSdkVersion="4"
android:targetSdkVersion="5">
</uses-sdk>
```

The supported SDK version is not equivalent to the platform version and cannot be derived from it. For example, Android platform release 2.0 supports the SDK version 5. To find the correct SDK version for each platform use the table at

➤ uses-configuration Use uses-configuration nodes to specify each combination of input mechanisms supported by your application. You can specify any combination of input devices that include:

➤ reqFiveWayNav Specify true for this attribute if you require an input device capable of navigating up, down, left, and right and of clicking the current selection. This includes both trackballs and D-pads.

➤ reqHardKeyboard If your application requires a hardware keyboard specify true.

➤ reqKeyboardType Lets you specify the keyboard type as one of nokeys, qwerty, twelvekey, or undefined.

➤ reqNavigation Specify the attribute value as one of nonav, dpad, trackball, wheel, or undefined as a required navigation device.

Introducing the Application Manifest | 53

➤ reqTouchScreen Select one of notouch, stylus, finger, or undefined to specify the required touchscreen input.

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

```
<uses-configuration android:reqTouchScreen=["finger"]
```

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

When specifying required configurations be aware that your application won't be installed on any device that does not have one of the combinations specified. In the above example a device with a QWERTY keyboard and a D-pad (but no touchscreen or trackball) would not be supported. Ideally you should develop your application to ensure it works with any input configuration, in which case no uses-configuration *node is required.*

➢ uses-feature One of the advantages of Android is the wide variety of hardware platforms it runs on. Use multiple uses-feature nodes to specify each of the hardware features your application requires. This will prevent your application from being installed on a device that does not include a required hardware feature. You can require support for any hardware that is optional on a compatible device. Currently optional hardware features include

➢ android.hardware.camera For applications that require camera hardware.

➢ android.hardware.camera.autofocus If you require an autofocus camera.

As the variety of platforms on which Android is available increases, so too will the optional hardware. A full list of uses-feature hardware can be found here. You can also use the uses-feature node to specify the minimum version of OpenGL required by your application. Use the glEsVersion attribute, specifying the OpenGL ES version as an integer. The higher 16 bits represent the major number and the lower 16 bits represent the minor number.

```
<uses-feature android:glEsVersion=" 0x00010001"
android:name="android.hardware.camera" />
```

➢ supports-screens After the initial round of HVGA hardware, 2009 saw the introduction of WVGA and QVGA screens to the Android device menagerie. With future Android devices likely to feature devices with larger screens, the supports-screen node lets you specify the screen sizes your application can, and can't, support.

Exact dimensions will vary depending on hardware, but in general the supported screen sizes match resolutions as follows:

- smallScreens Screens with a resolution smaller than traditional HVGA—typically QVGA screens.
- normalScreens Used to specify typical mobile phone screens of at least HVGA, including WVGA andWQVGA.
- largeScreens Screens larger than normal. In this instance a large screen is considered to be significantly larger than a mobile phone display.
- anyDensity Set to true if your application can be scaled to accommodate any screen resolution.

As of SDK 1.6 (API level 4), the default value for each attribute is true. Use this node to specify screen sizes you do not support.

```
<supports-screens android:smallScreens=["false"]
android:normalScreens=["true"]
android:largeScreens=["true"]
android:anyDensity=["false"] />
```

Where possible you should optimize your application for different screen resolutions and densities using the resources folder, as shown later in this chapter. If you specify a supports-screen *node that excludes certain screen sizes, your application will not be available to be installed on devices with unsupported screens.*

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

The <application> node also acts as a container that includes the Activity, Service, Content Provider, and Broadcast Receiver tags used to specify the application components. You can also define your own implementation of the Application class. Later in this chapter you'll learn how to create and use your own Application class extension to manage application state.

```
<application android:icon="@drawable/icon"
android:theme="@style/my_theme"
android:name="MyApplication"
android:debuggable="true">
[ ... application nodes ... ]
</application>
```

➤ activity An <activity> tag is required for every Activity displayed by your application. Using the android:name attribute to specify the Activity class name.

You must include the main launch Activity and any other screen or dialog that can be displayed. Trying to start an Activity that's not defined in the manifest will throw a runtime exception. Each Activity node supports <intent-filter> child tags that specify which Intents launch the Activity.

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

➤ service As with the activity tag, create a new service tag for each Service class used in your application. Service tags also support <intent-filter> child tags to allow late runtime binding.

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

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

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

➤ receiver By adding a receiver tag, you can register a Broadcast Receiver without having to launch your application first. Broadcast Receivers are like global event listeners that, once registered, will execute whenever a matching Intent is broadcast by the system or an application. By registering a Broadcast Receiver in the manifest you can make this process entirely autonomous. If a matching Intent is broadcast, your application will be started automatically and the registered Broadcast Receiver will be run.

```
<receiver android:enabled="true"
android:label="My Intent Receiver"
android:name=".MyIntentReceiver">
</receiver>
```

➤ uses-permission As part of the security model, uses-permission tags declare the permissions you've determined your application needs to operate properly. The permissions you include will be presented to the user before installation commences. Permissions are required for many of the native Android services, particularly those with a cost or security implication (such as dialing, receiving SMS, or using the location-based services).

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

➤ permission Third-party applications can also specify permissions before providing access to shared application components. Before you can restrict access to an application component, you need to define a permission in the manifest. Use the permission tag to create a permission definition. Application components can then require permissions by adding the android:permission attribute. Other applications will then need to include a uses-permission tag in their manifests to use these protected components.

Within the permission tag, you can specify the level of access the permission will permit (normal, dangerous, signature, signatureOrSystem), a label, and an external resource containing the description that explains the risks of granting the specified permission.

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

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

```
<instrumentation android:label="My Test"
android:name=".MyTestClass"
android:targetPackage="com.paad.aPackage">
</instrumentation>
```

The ADT New Project Wizard automatically creates a new manifest file when it creates a new project.

USING THE MANIFEST EDITOR

The ADT plug-in includes a visual Manifest Editor so you don't have to manipulate the underlying XML directly.

To use the Manifest Editor in Eclipse, right-click the AndroidManifest.xml file in your project folder and select **Open With . . . ➪ Android Manifest Editor.** This presents the Android Manifest Overview screen, as shown in Figure 3-1. This screen gives you a high-level view of your application structure, enabling you to set your application version information and root level manifest nodes, including

<uses-sdk> and <uses-features>, as described previously in this chapter. It also provides shortcut links to the Application, Permissions, Instrumentation, and raw XML screens. Each of the next three tabs contains a visual interface for managing the application, security, and instrumentation (testing) settings, while the last tag (using the manifest's file name) gives access to the raw XML. Of particular interest is the Application tab, shown in Figure 3-2. Use it to manage the application node and the application component hierarchy, where you specify the application components. You can specify an application's attributes — including its icon, label, and theme — in the Application Attributes panel. The Application Nodes tree beneath it lets you manage the application components, including their attributes and any associated Intent Filter subnodes.

THE ANDROID APPLICATION LIFE CYCLE

Unlike most traditional environments, Android applications have limited control over their own life cycles. 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. *While it's uncommon, it's possible to force application components within the same application to run in different processes or to have multiple applications share the same process using the* android:process *attribute on the affected component nodes within the manifest.*

Android aggressively manages its resources, doing whatever it takes to ensure that the device remains responsive. This means that processes (and their hosted applications) will be killed, without warning in some cases, to free resources for higher-priority applications — generally those interacting directly with the user at the time. The prioritization process is discussed in the next section.

Chapter 5: Intents

Before you can begin to interact with the phone dialer, you need to understand the type of code that you will use to do the job. Android uses *Intents* to do specific jobs within applications. Once you master the use of Intents, a whole new world of application development will be open to you. This section defines what an Intent is and how it is used. An Intent is Android's method for relaying certain information from one Activity to another.

An Intent, in simpler terms, expresses to Android your intent to do something. You can think of an Intent as a message passed between Activities. For example, assume that you have an Activity that needs to open a web browser and display a page on your Android device. Your Activity would send an "intent to open *x* page in the web browser," known as a WEB_SEARCH_ACTION Intent, to the Android Intent Resolver. The Intent Resolver parses through a list of Activities and chooses the one that would best match your Intent; in this case, the Web Browser Activity. The Intent Resolver then passes your page to the web browser and starts the Web Browser Activity. Intents are broken up into two main categories:

- **Activity Action Intents** Intents used to call Activities outside of your application. Only one Activity can handle the Intent. For example, for a web browser, you need to open the Web Browser Activity to display a page.
- **Broadcast Intents** Intents that are sent out for multiple Activities to handle. An example of a Broadcast Intent would be a message sent out by Android about the current battery level. Any Activity can process this Intent and react accordingly—for example, cancel an Activity if the battery level is below a certain point. the current Activity Action Intents that are available to you. As you'll notice, in most cases, the name of the Intent does a good job of describing what that Intent does.

The Intent is only one-third of the picture. An Intent is really just that, an intent to do something; an Intent cannot actually do anything by itself. You need Intent Filters and Intent Receivers to listen for, and interpret, the Intents. An Intent Receiver is like the mailbox of an Activity. The Intent Receiver is used to allow an Activity to receive the specified Intent. Using the previous web browser example, the Web Browser Activity is set up to receive web browser Intents. A system like this allows unrelated Activities to ignore Intents that they would not be able to process. It also allows Activities that need the assistance of another Activity to utilize that Activity without needing to know how to call it.

Intents are used for navigation purpose in android that means pass the control from one activity(class) to another activity(class) . In three application components we are using this intent mechanism(Activity, Service, Receiver)but not for Content Provider. Intents are two types .

They are : Implicit Intents
Explicit Intents.

Explicit Intents :

If we want to navigate the control from one screens to and another screen at that time we are using this intents.

Passing the control from one Activity to Activity:

By using two methods we can do navigation from one activity to another activity.

They are 1 . startActivity(Intent intent)
2 . startActivityForResult(Intent intent , int requestCode).

startActivity() :

This method is from context class and it supports only for single direction communication.

```
Intent intent = new Intent(context c,class);
        startActivtity(intent);
```

For Example : If I want to pass the control from A(activity) to B(activity) at that time We write this logic in A activity.

```
Coding :    Intent intent = new Intent(A.this,B.Class);
             startActivity(intent);
```

startActivityForResult() :

This method is from Activity class and it supports only for bi-direction communication. Here three steps will be involved as follows.

Step – I : In A class we write this code .

```
Intent intent = new Intent(A.this,B.class);
        startActivityForResult(intent,101);
```

Step – II : In **B** class we need to call the **setResult()** from **Activity**. write this logic.

```
setResult(int ResultCode, Intent intent);
```

Logic :

```
Intent data = getIntent();
```

```
        String result = F + S;
        data.putExtra("Result",result);
        setResult(1001,data)
```

Step – III : Again in A class we have to call callback method **onActivityResult(int requestcode, int resultCode, intent data).** Here automatically the request code and result code will be bonded to this variables.

Logic :

```
protected void onActivityResult(int requestCode,int resultCode, Intent data) {
                if (requestCode == 100 && resultCode == 1001) {
                String total = data.getStringExtra("result");
                Result.setText(total);
        }
```

Implicit Intents :

If we want to interact with default software components and hardware components at that time we are using this type of intents.

For Opening Browser :

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

For opening calling screen :

```
        Intent intent = new Intent(Intent.ACTION_CALL,Uri.parse("tel:9493577797"));
                        startActivity(intent);
```

For opening Dialing screen :

```
        Intent intent = new Intent(Intent.ACTION_DIAL,Uri.parse("tel:9493577797"));
                        startActivity(intent);
```

For opening contacts :

```
        Intent intent = new Intent(Intent.ACTION_VIEW,
Uri.parse("content://contacts/people/1"));
                        startActivity(intent);
```

For opening Camera :

```
        Intent  intent = new Intent("android.media.action.IMAGE_CAPTURE");
                        startActivityForResult(intent, 0);
```

Chapter 6 Layouts

An Android layout is a class that handles arranging the way its children appear on the screen. Anything that is a View (or inherits from View) can be a child of a layout. All of the layouts inherit from ViewGroup (which inherits from View) so you can nest layouts. You could also create your own custom layout by making a class that inherits from ViewGroup.

The standard Layouts are:

AbsoluteLayout
FrameLayout
LinearLayout
RelativeLayout
TableLayout

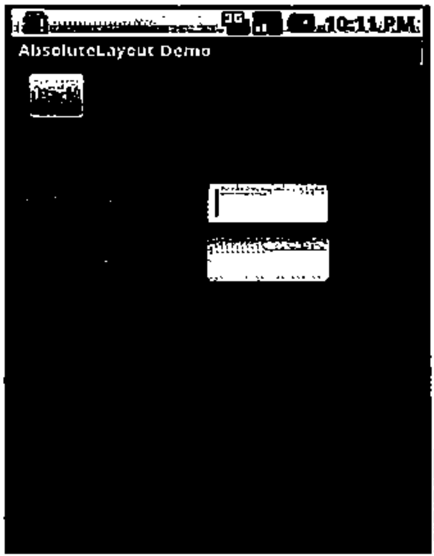
Absolute Layout

Absolute Layout is based on the simple idea of placing each control at an absolute position. You specify the exact x and y coordinates on the screen for each control. This is not recommended for most UI development (in fact Absolute Layout is currently deprecated) since absolutely positioning every element on the screen makes an inflexible UI that is much more difficult to maintain. Consider what happens if a control needs to be added to the UI. You would have to change the position of every single element that is shifted by the new control.

Here is a sample Layout XML using AbsoluteLayout.

```
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <Button
        android:id="@+id/backbutton"
        android:text="Back"
        android:layout_x="10px"
        android:layout_y="5px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
  <TextView
        android:layout_x="10px"
        android:layout_y="110px"
        android:text="First Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
  <EditText
        android:layout_x="150px"
        android:layout_y="100px"
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
  <TextView
        android:layout_x="10px"
        android:layout_y="160px"
        android:text="Last Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
        <EditText
        android:layout_x="150px"
        android:layout_y="150px"
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</AbsoluteLayout>
```

Note how each element has android:layout_x and android:layout_y specified. Android defines the top left of the screen as (0,0) so the layout_x value will move the control to the right, and the layout_y value will move the control down. Here is a screenshot of the layout produced by this XML.



FrameLayout

FrameLayout is designed to display a single item at a time. You can have multiple elements within a FrameLayout but each element will be positioned based on the top left of the screen. Elements that overlap will be displayed overlapping. I have created a simple XML layout using FrameLayout that shows how this works.

```
<FrameLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <ImageView
                android:src="@drawable/icon"
                android:scaleType="fitCenter"
                android:layout_height="fill_parent"
                android:layout_width="fill_parent"/>
        <TextView
                android:text="Learn-Android.com"
                android:textSize="24sp"
                android:textColor="#000000"
                android:layout_height="fill_parent"
                android:layout_width="fill_parent"
                android:gravity="center"/>
</FrameLayout>
```

FrameLayout can become more useful when elements are hidden and displayed programmatically. You can use the attribute android:visibility in the XML to hide specific elements. You can call setVisibility from the code to accomplish the same thing. The three available visibility values are visible, invisible (does not display, but still takes up space in the layout), and gone (does not display, and does not take space in the layout).

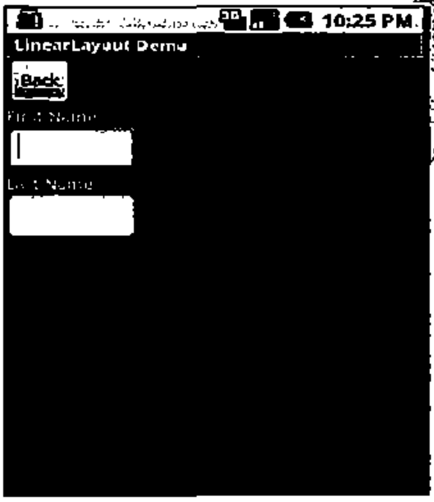
LinearLayout

LinearLayout organizes elements along a single line. You specify whether that line is verticle or horizontal using android:orientation. Here is a sample Layout XML using LinearLayout.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
  android:layout_height="fill_parent">
   <Button
        android:id="@+id/backbutton"
        android:text="Back"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
  <TextView
        android:text="First Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
  <EditText
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
  <TextView
```

```
        android:text="Last Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
  <EditText
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Here is a screenshot of the result of the above XML.



RelativeLayout

RelativeLayout lays out elements based on their relationships with one another, and with the parent container. This is arguably the most complicated layout, and we need several properties to actually get the layout we want.

Relative To Container

These properties will layout elements relative to the parent container.

- android:layout_alignParentBottom – Places the bottom of the element on the bottom of the container
- android:layout_alignParentLeft – Places the left of the element on the left side of the container
- android:layout_alignParentRight – Places the right of the element on the right side of the container
- android:layout_alignParentTop – Places the element at the top of the container
- android:layout_centerHorizontal – Centers the element horizontally within its parent container
- android:layout_centerInParent – Centers the element both horizontally and vertically within its container
- android:layout_centerVertical – Centers the element vertically within its parent container

Relative To Other Elements

These properties allow you to layout elements relative to other elements on screen. The value for each of these elements is the id of the element you are using to layout the new element. Each element that is used in this way must have an ID defined using android:id="@+id/XXXXX" where XXXXX is replaced with the desired id. You use "@id/XXXXX" to reference an element by its id. One thing to remember is that referencing an element before it has been declared will produce an error.

- android:layout_above – Places the element above the specified element
- android:layout_below – Places the element below the specified element
- android:layout_toLeftOf – Places the element to the left of the specified element
- android:layout_toRightOf – Places the element to the right of the specified element

Alignment With Other Elements

These properties allow you to specify how elements are aligned in relation to other elements.

- android:layout_alignBaseline – Aligns baseline of the new element with the baseline of the specified element
- android:layout_alignBottom – Aligns the bottom of new element in with the bottom of the specified element

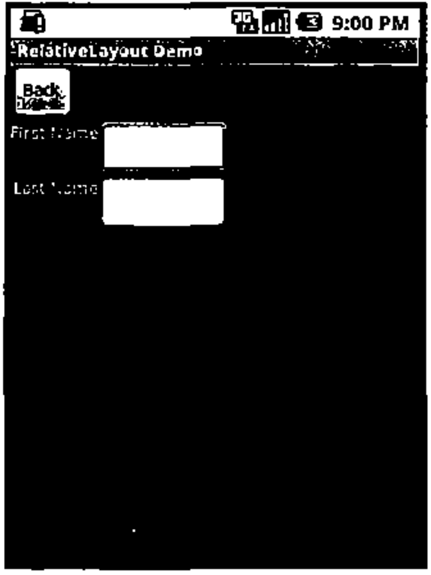
- android:layout_alignLeft – Aligns left edge of the new element with the left edge of the specified element
- android:layout_alignRight – Aligns right edge of the new element with the right edge of the specified element
- android:layout_alignTop – Places top of the new element in alignment with the top of the specified element

Here is a sample XML Layout

```
<RelativeLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <Button
                android:id="@+id/backbutton"
                android:text="Back"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        <TextView
                android:id="@+id/firstName"
                android:text="First Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_below="@id/backbutton" />
        <EditText
                android:id="@+id/editFirstName"
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_toRightOf="@id/firstName"
                android:layout_below="@id/backbutton"/>
        <EditText
                android:id="@+id/editLastName"
```

```
            android:width="100px"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/editFirstName"
            android:layout_alignLeft="@id/editFirstName"/>
    <TextView
            android:id="@+id/lastName"
            android:text="Last Name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_toLeftOf="@id/editLastName"
            android:layout_below="@id/editFirstName"/>
</RelativeLayout>
```

You probably noticed that I had to rearrange the elements in the XML since, as I already mentioned, you cannot reference an element that has not already been laid out. Here is what the updated RelativeLayout produces.



TableLayout

TableLayout organizes content into rows and columns. The rows are defined in the layout XML, and the columns are determined automatically by Android. This is done by creating at least one column for each element. So, for example, if you had a row with two elements and a row with five elements then you would have a layout with two rows and five columns.

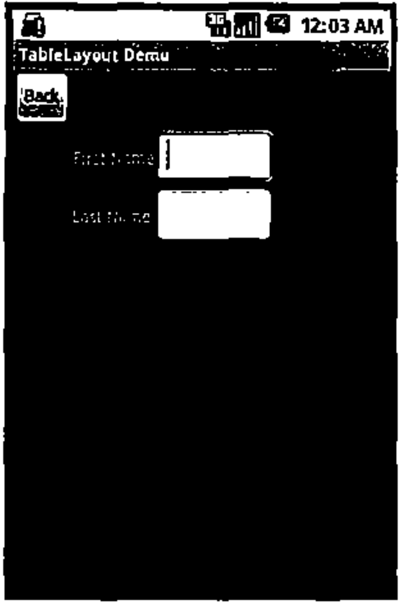
You can specify that an element should occupy more than one column using android:layout_span. This can increase the total column count as well, so if we have a row with two elements and each element has android:layout_span="3" then you will have at least six columns in your table.

By default, Android places each element in the first unused column in the row. You can, however, specify the column an element should occupy using android:layout_column.

Here is some sample XML using TableLayout.

```
<TableLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <TableRow>
                <Button
                android:id="@+id/backbutton"
                android:text="Back"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        </TableRow>
        <TableRow>
                <TextView
                android:text="First Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_column="1" />
                <EditText
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        </TableRow>
        <TableRow>
                <TextView
                android:text="Last Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_column="1" />
                <EditText
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        </TableRow>
</TableLayout>
```

This is the result of that XML.



Notifications

What are Notifications? The name itself implies their functionality. They are a way of alerting a user about an event that he needs to be informed about or even take some action on getting that information. Notification on Android can be done in any of the following ways:

They are

1. Toast Notification
2. Status Bar Notification
3. Dialog Boxes

Toast Notifications

A toast notification is a message that pops up on the surface of the window. It only fills the amount of space required for the message and the user's current activity remains visible and interactive. The notification automatically fades in and out, and does not accept interaction events.First, instantiate a Toast object with one of the makeText() methods. This method takes three parameters: the application Context, the text message, and the duration for the toast. It returns a properly initialized Toast object. You can display the toast notification with show(), as shown in the following example:

```
Context context = getApplicationContext();
CharSequence text = "Hello Sai_tech!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

This example demonstrates everything you need for most toast notifications. You should rarely need anything else. You may, however, want to position the toast differently or even use your own layout instead of a simple text message. The following sections describe how you can do these things.

You can also chain your methods and avoid holding on to the Toast object, like this:

```
Toast.makeText(context, text, duration).show();
```

Positioning your Toast

A standard toast notification appears near the bottom of the screen, centered horizontally. You can change this position with the setGravity(int, int, int) method. This accepts three parameters: a Gravity constant, an x-position offset, and a y-position offset.

For example, if you decide that the toast should appear in the top-left corner, you can set the gravity like this:

```
toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);
```

Status Bar Notification :

This Notification on Android can be done in any of the following ways:

- Status Bar Notification
- Vibrate
- Flash lights
- Play a sound

From the Notification, you can allow the user to launch a new activity as well. Now we will look at status bar notification as this can be easily tested on the emulator.

To create a status bar notification, you will need to use two classes: Notification and NotificationManager.

- Notification – defines the properties of the status bar notification like the icon to display, the test to display when the notification first appears on the status bar and the time to display.

- NotificationManager is an android system service that executes and manages all notifications. Hence you cannot create an instance of the NotificationManager but you can retrieve a reference to it by calling the getSystemService() method.

Once you procure this handle, you invoke the notify() method on it by passing the notification object created.

So far, you have all the information to display on the status bar. However, when the user clicks the notification icon on the status bar, what detailed information should you show the user? This is yet to be created. This is done by calling the method setLatestEventInfo() on the notification object.

Creating a Notification

A Notification object defines the details of the notification message that is displayed in the status bar and notifications window, and any other alert settings, such as sounds and blinking lights.

A status bar notification *requires* all of the following:

- An icon for the status bar
- A title and message, unless you define a custom notification layout
- A PendingIntent, to be fired when the notification is selected

Optional settings for the status bar notification include:

- A ticker-text message for the status bar
- An alert sound
- A vibrate setting
- A flashing LED setting

Updating the notification

You can update the information in your status bar notification as events continue to occur in your application. For example, when a new SMS text message arrives before previous messages have been read, the Messaging application updates the existing notification to display the total number of new messages received. This practice of updating an existing notification is much better than adding new notifications, because it avoids clutter in the notifications window.

Because each notification is uniquely identified by the NotificationManager with an integer ID, you can revise the notification by calling setLatestEventInfo() with new values, change some field values of the notification, and then call notify() again.

You can revise each property with the object member fields (except for the Context and the notification title and text). You should always revise the text message when you update the notification by calling setLatestEventInfo() with new values for *contentTitle* and *contentText*. Then call notify() to update the notification. (Of course, if you've created a custom notification layout, then updating these title and text values has no effect.)

Adding a sound

You can alert the user with the default notification sound (which is defined by the user) or with a sound specified by your application.

To use the user's default sound, add "DEFAULT_SOUND" to the *defaults* field:

```
notification.defaults |= Notification.DEFAULT_SOUND;
```

Adding vibration

You can alert the user with the the default vibration pattern or with a vibration pattern defined by your application.

To use the default pattern, add DEFAULT_VIBRATE to the *defaults* field:

```
notification.defaults |= Notification.DEFAULT_VIBRATE;
```

Adding flashing lights

To alert the user by flashing LED lights, you can implement the default light pattern (if available), or define your own color and pattern for the lights.

To use the default light setting, add DEFAULT_LIGHTS to the *defaults* field:

```
notification.defaults |= Notification.DEFAULT_LIGHTS;
```

Dialog Boxes :

A dialog is usually a small window that appears in front of the current Activity. The underlying Activity loses focus and the dialog accepts all user interaction. Dialogs are normally used for notifications that should interupt the user and to perform short tasks that directly relate to the application in progress (such as a progress bar or a login prompt).

The Dialog class is the base class for creating dialogs. However, you typically should not instantiate a Dialog directly. Instead, you should use one of the following subclasses:

AlertDialog

A dialog that can manage zero, one, two, or three buttons, and/or a list of selectable items that can include checkboxes or radio buttons. The AlertDialog is capable of constructing most dialog user interfaces and is the suggested dialog type. See Creating an AlertDialog below.

ProgressDialog

A dialog that displays a progress wheel or progress bar. Because it's an extension of the AlertDialog, it also supports buttons. See Creating a ProgressDialog below.

DatePickerDialog

A dialog that allows the user to select a date. See the Hello DatePicker tutorial.

TimePickerDialog

A dialog that allows the user to select a time. See the Hello TimePicker tutorial.

If you would like to customize your own dialog, you can extend the base Dialog object or any of the subclasses listed above and define a new layout. See the section on Creating a Custom Dialog below.

Showing a Dialog

A dialog is always created and displayed as a part of an Activity. You should normally create dialogs from within your Activity's onCreateDialog(int) callback method. When you use this callback, the Android system automatically manages the state of each dialog and hooks them to the Activity, effectively making it the "owner" of each dialog. As such, each dialog inherits certain properties from the Activity. For example, when a dialog is open, the Menu key reveals the options menu defined for the Activity and the volume keys modify the audio stream used by the Activity.

Note: If you decide to create a dialog outside of the onCreateDialog() method, it will not be attached to an Activity. You can, however, attach it to an Activity with setOwnerActivity(Activity).

When you want to show a dialog, call showDialog(int) and pass it an integer that uniquely identifies the dialog that you want to display.

When a dialog is requested for the first time, Android calls onCreateDialog(int) from your Activity, which is where you should instantiate the Dialog. This callback method is passed the same ID that you passed to showDialog(int). After you create the Dialog, return the object at the end of the method.

The best way to define the onCreateDialog(int) and onPrepareDialog(int, Dialog) callback methods is with a *switch* statement that checks the *id* parameter that's passed into the method. Each *case* should check for a unique dialog ID and then create and define the respective Dialog. For example, imagine a game that uses two different dialogs: one to indicate that the game has paused and another to indicate that the game is over. First, define an integer ID for each dialog:

```
static final int DIALOG_PAUSED_ID = 0;
static final int DIALOG_GAMEOVER_ID = 1;
```

Then, define the onCreateDialog(int) callback with a switch case for each ID:

```
protected Dialog onCreateDialog(int id) {
   Dialog dialog;
   switch(id) {
   case DIALOG_PAUSED_ID:
      // do the work to define the pause Dialog
      break;
   case DIALOG_GAMEOVER_ID:
      // do the work to define the game over Dialog
      break;
   default:
      dialog = null;
   }
   return dialog;
}
```

Note: In this example, there's no code inside the case statements because the procedure for defining your Dialog is outside the scope of this section. See the section below about Creating an AlertDialog, offers code suitable for this example.

When it's time to show one of the dialogs, call showDialog(int) with the ID of a dialog:

showDialog(DIALOG_PAUSED_ID);

Dismissing a Dialog

When you're ready to close your dialog, you can dismiss it by calling dismiss() on the Dialog object. If necessary, you can also call dismissDialog(int) from the Activity, which effectively calls dismiss() on the Dialog for you.

If you are using onCreateDialog(int) to manage the state of your dialogs (as discussed in the previous section), then every time your dialog is dismissed, the state of the Dialog object is retained by the Activity. If you decide that you will no longer need this object or it's important that the state is cleared, then you should call removeDialog(int). This will remove any internal references to the object and if the dialog is showing, it will dismiss it.

Creating an AlertDialog

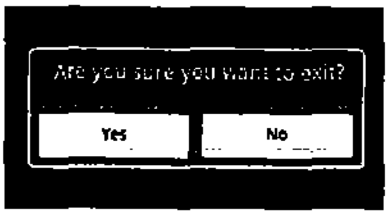
An AlertDialog is an extension of the Dialog class. It is capable of constructing most dialog user interfaces and is the suggested dialog type. You should use it for dialogs that use any of the following features:

- A title
- A text message
- One, two, or three buttons
- A list of selectable items (with optional checkboxes or radio buttons)

To create an AlertDialog, use the AlertDialog.Builder subclass. Get a Builder with AlertDialog.Builder(Context) and then use the class's public methods to define all of the AlertDialog properties. After you're done with the Builder, retrieve the AlertDialog object with create().

The following topics show how to define various properties of the AlertDialog using the AlertDialog.Builder class. If you use any of the following sample code inside your onCreateDialog() callback method, you can return the resulting Dialog object to display the dialog.

Adding buttons



Are you sure you want to exit?

To create an AlertDialog with side-by-side buttons like the one shown in the screenshot to the right, use the set...Button() methods:

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Are you sure you want to exit?")
    .setCancelable(false)
    .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int id) {
         MyActivity.this.finish();
      }
    })
    .setNegativeButton("No", new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int id) {
         dialog.cancel();
      }
    });
AlertDialog alert = builder.create();
```

First, add a message for the dialog with setMessage(CharSequence). Then, begin method-chaining and set the dialog to be *not cancelable* (so the user cannot close the dialog with the back button) with setCancelable(boolean). For each button, use one of the set...Button() methods, such as setPositiveButton(), that accepts the name for the button and a DialogInterface.OnClickListener that defines the action to take when the user selects the button.

Note: You can only add one of each button type to the AlertDialog. That is, you cannot have more than one "positive" button. This limits the number of possible buttons to three: positive, neutral, and negative. These names are technically irrelevant to the actual functionality of your buttons, but should help you keep track of which one does what.

Adding a list

To create an AlertDialog with a list of selectable items like the one shown to the right, use the setItems() method:

```
final CharSequence[] items = {"Red", "Green", "Blue"};

AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Pick a color");
builder.setItems(items, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int item) {
        Toast.makeText(getApplicationContext(), items[item], Toast.LENGTH_SHORT).show();
    }
});
AlertDialog alert = builder.create();
```

First, add a title to the dialog with setTitle(CharSequence). Then, add a list of selectable items with setItems(), which accepts the array of items to display and a DialogInterface.OnClickListener that defines the action to take when the user selects an item.

Adding checkboxes and radio buttons

To create a list of multiple-choice items (checkboxes) or single-choice items (radio buttons) inside the dialog, use the setMultiChoiceItems() and setSingleChoiceItems() methods, respectively. If you create one of these selectable lists in the onCreateDialog() callback method, Android manages the state of the list for you. As long as the Activity is active, the dialog remembers the items that were previously selected, but when the user exits the Activity, the selection is lost.

Note: To save the selection when the user leaves or pauses the Activity, you must properly save and restore the setting throughout the activity lifecycle. To permanently save the selections, even when the Activity process is completely shutdown, you need to save the settings with one of the Data Storage techniques.

To create an AlertDialog with a list of single-choice items like the one shown to the right, use the same code from the previous example, but replace the setItems() method with setSingleChoiceItems():

```
final CharSequence[] items = {"Red", "Green", "Blue"};

AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Pick a color");
builder.setSingleChoiceItems(items, -1, new DialogInterface.OnClickListener() {
  public void onClick(DialogInterface dialog, int item) {
     Toast.makeText(getApplicationContext(), items[item], Toast.LENGTH_SHORT).show();
  }
});
AlertDialog alert = builder.create();
```

The second parameter in the setSingleChoiceItems() method is an integer value for the *checkedItem*, which indicates the zero-based list position of the default selected item. Use "-1" to indicate that no item should be selected by default.

ProgressDialog :

A ProgressDialog is an extension of the AlertDialog class that can display a progress animation in the form of a spinning wheel, for a task with progress that's undefined, or a progress bar, for a task that has a defined progression. The dialog can also provide buttons, such as one to cancel a download.

Opening a progress dialog can be as simple as calling ProgressDialog.show(). For example, the progress dialog shown to the right can be easily achieved without managing the dialog through the onCreateDialog(int) callback, as shown here:

```
ProgressDialog dialog = ProgressDialog.show(MyActivity.this, "",
            "Loading. Please wait...", true);
```

The first parameter is the application Context, the second is a title for the dialog (left empty), the third is the message, and the last parameter is whether the progress is indeterminate (this is only relevant when creating a progress bar, which is discussed in the next section).

The default style of a progress dialog is the spinning wheel. If you want to create a progress bar that shows the loading progress with granularity, some more code is required, as discussed in the next section.

Showing a progress bar :

To show the progression with an animated progress bar:

1. Initialize the ProgressDialog with the class constructor, ProgressDialog(Context).
2. Set the progress style to "STYLE_HORIZONTAL" with setProgressStyle(int) and set any other properties, such as the message.
3. When you're ready to show the dialog, call show() or return the ProgressDialog from the onCreateDialog(int) callback.
4. You can increment the amount of progress displayed in the bar by calling either setProgress(int) with a value for the total percentage completed so far or incrementProgressBy(int) with an incremental value to add to the total percentage completed so far.

For example, your setup might look like this:

```
ProgressDialog progressDialog;
progressDialog = new ProgressDialog(mContext);
progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
progressDialog.setMessage("Loading...");
progressDialog.setCancelable(false);
```

The setup is simple. Most of the code needed to create a progress dialog is actually involved in the process that updates it. You might find that it's necessary to create a second thread in your application for this work and then report the progress back to the Activity's UI thread with a Handler object. If you're not familiar with using additional threads with a Handler, see the example Activity below that uses a second thread to increment a progress dialog managed by the Activity.

DatePickerDialog:

To provide a widget for selecting a date, use the DatePicker widget, which allows the user to select the month, day, and year, in a familiar interface.

Main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <TextView android:id="@+id/dateDisplay"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=""/>
  <Button android:id="@+id/pickDate"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Change the date"/>
</LinearLayout>
```

HelloDatePicker.java

```
Public class HelloDatePicker extends Activity

{

  private TextView mDateDisplay;
    private Button mPickDate;
    private int mYear;
    private int mMonth;
    private int mDay;

    static final int DATE_DIALOG_ID = 0;
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.main);

        // capture our View elements
        mDateDisplay = (TextView) findViewById(R.id.dateDisplay);
        mPickDate = (Button) findViewById(R.id.pickDate);

        // add a click listener to the button
        mPickDate.setOnClickListener(new View.OnClickListener() {
          public void onClick(View v) {
            showDialog(DATE_DIALOG_ID);
          }
        });

        // get the current date
        final Calendar c = Calendar.getInstance();
        mYear = c.get(Calendar.YEAR);
        mMonth = c.get(Calendar.MONTH);
        mDay = c.get(Calendar.DAY_OF_MONTH);

        // display the current date (this method is below)
        updateDisplay();
      }
  // updates the date in the TextView
      private void updateDisplay() {
        mDateDisplay.setText(
          new StringBuilder()
               // Month is 0 based so add 1
               .append(mMonth + 1).append("-")
               .append(mDay).append("-")
               .append(mYear).append(" "));
      }
```

```
// the callback received when the user "sets" the date in the dialog
   private DatePickerDialog.OnDateSetListener mDateSetListener =
         new DatePickerDialog.OnDateSetListener() {

            public void onDateSet(DatePicker view, int year,
                              int monthOfYear, int dayOfMonth) {
               mYear = year;
               mMonth = monthOfYear;
               mDay = dayOfMonth;
               updateDisplay();
            }
         };
@Override
protected Dialog onCreateDialog(int id) {
   switch (id) {
   case DATE_DIALOG_ID:
      return new DatePickerDialog(this,
               mDateSetListener,
               mYear, mMonth, mDay);
   }
   return null;
}
```

Time Picker Dialog:

To provide a widget for selecting a time, use the TimePicker widget, which allows the user to select the hour and minute in a familiar interface.

main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:orientation="vertical">
   <TextView android:id="@+id/timeDisplay"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text=""/>
   <Button android:id="@+id/pickTime"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Change the time"/>
</LinearLayout>
```

HelloTimePicker.java:

```
Public class HelloTimePicker extends Activity

{

  private TextView mTimeDisplay;
    private Button mPickTime;


    private int mHour;
    private int mMinute;


    static final int TIME_DIALOG_ID = 0;
  @Override
    protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
    // capture our View elements
    mTimeDisplay = (TextView) findViewById(R.id.timeDisplay);
    mPickTime = (Button) findViewById(R.id.pickTime);

    // add a click listener to the button
    mPickTime.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        showDialog(TIME_DIALOG_ID);
      }
    });

    // get the current time
    final Calendar c = Calendar.getInstance();
    mHour = c.get(Calendar.HOUR_OF_DAY);
    mMinute = c.get(Calendar.MINUTE);

    // display the current date
    updateDisplay();
  }
// updates the time we display in the TextView
private void updateDisplay() {
  mTimeDisplay.setText(
    new StringBuilder()
        .append(pad(mHour)).append(":")
        .append(pad(mMinute)));
}

private static String pad(int c) {
  if (c >= 10)
    return String.valueOf(c);
  else
```

```
      return "0" + String.valueOf(c);
  }
  // the callback received when the user "sets" the time in the dialog
  private TimePickerDialog.OnTimeSetListener mTimeSetListener =
    new TimePickerDialog.OnTimeSetListener() {
      public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
        mHour = hourOfDay;
        mMinute = minute;
        updateDisplay();
      }
    };
  @Override
  protected Dialog onCreateDialog(int id) {
    switch (id) {
    case TIME_DIALOG_ID:
      return new TimePickerDialog(this,
          mTimeSetListener, mHour, mMinute, false);
    }
    return null;
  }
```

AutoCompleteTextView:

An editable text view that shows completion suggestions automatically while the user is typing. The list of suggestions is displayed in a drop down menu from which the user can choose an item to replace the content of the edit box with.

The drop down can be dismissed at any time by pressing the back key or, if no item is selected in the drop down, by pressing the enter/dpad center key.The list of suggestions is obtained from a data adapter and appears only after a given number of characters defined by the threshold.

Example:

Main.java:

```
public class main extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

          AutoCompleteTextView textView = (AutoCompleteTextView)
                    findViewById(R.id.autocomplete_country);
      ArrayAdapter<String> adapter  = new ArrayAdapter<String>(this,
                    R.layout.list_item, DISTRICTS);
      textView.setAdapter(adapter);
   }

      static final String[] DISTRICTS = new String[] {

"SRIKAKULAM","VIJAYAGANAGARAM","VISHAKAPATNAM","EASTGODAVARI","WES
TGODAVARI","KRISHNA"
      };
}
```

Main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:padding="5dp">
   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
```

```
        android:text="Country" />

    <AutoCompleteTextView android:id="@+id/autocomplete_country"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="5dp"/>
</LinearLayout>
```

ListItem.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="10dp"
    android:textSize="16sp"
    android:textColor="#000"
    >
</TextView>
```

SPINNER:

Any good web designed knows how important the <select> element can be. In a touch screen environment, like Android, where users often don't have a full keyboard, like many Android phones, the ability to simply select an item from a pre-determined list is absolutely vital. It's easier for the user to select what they want - rather than have to type it, it protects your databases from junk data and typos by limiting choices to pre-defined, pre-screened responses. The Android OS has a very robust, user friendly and - relatively - programmer friendly way to handle this: the Android Spinner.

Spinners are simple to use, easy to program and just downright good-looking. However, there are several different ways to populate the list of items in an Android Spinner, and - as of the writing of this article - the Android Developer's Website (developer.android.com) only focuses on one of those methods. This article will break down both of the most popular ways to do populate an Android Spinner: with an array contained in your strings.xml file, and from a database query. Then it will also demonstrate how to manually insert items into a spinner array using code. For ease of navigation, I'm placing each of those on a separate page and including links to them so you can go to the page that best fits your needs as a developer.

- Populate Android Spinner from string array resource
- Populate Android Spinner from database query (using a cursor)
- Populate Android Spinner manually with Java code

Example:

main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:padding="10dip"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dip"
    android:text="@string/planet_prompt"
  />
  <Spinner
    android:id="@+id/spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:prompt="@string/planet_prompt"
  />
</LinearLayout>
```

strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="planet_prompt">Choose a planet</string>
   <string-array name="planets_array">
      <item>Mercury</item>
      <item>Venus</item>
      <item>Earth</item>
      <item>Mars</item>
      <item>Jupiter</item>
      <item>Saturn</item>
      <item>Uranus</item>
      <item>Neptune</item>
   </string-array>
</resources>
```

SpinnerDemo.java:

```
Public class SpinnerDemo extends Activity

{

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

   Spinner spinner = (Spinner) findViewById(R.id.spinner);
   ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(
        this, R.array.planets_array,android.R.layout.simple_spinner_item);

     adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
   spinner.setAdapter(adapter);
}
```

```
        public class MyOnItemSelectedListener implements OnItemSelectedListener {

          public void onItemSelected(AdapterView<?> parent,View view, int pos, long id) {
            Toast.makeText(parent.getContext(), "The planet is " +
        parent.getItemAtPosition(pos).toString(),  Toast.LENGTH_LONG).show();
          }

          public void onNothingSelected(AdapterView parent) {
           // Do nothing.
          }
        }
spinner.setOnItemSelectedListener(new MyOnItemSelectedListener());

}
```

ListView:

Android provides the view "ListView" which is capable of displaying a scrollable list of items. "ListView" gets the data to display via an adapter. An adapter which must extend "BaseAdapter" and is responsible for providing the data model for the list and for converting the data into the fields of the list.

Android has two standard adapters, ArrayAdapter and CursorAdapter . "ArrayAdapter" can handle data based on Arrays or Lists while "SimpleCursorAdapter" handle database related data. You can develop your own Adapter by extending these classes or the BaseAdapter class.

The following description will focus on ArrayAdapter and how to develop your own adapter. It will also give a small example for "CursorAdapter".

ListActivity

You can directly use the "ListView" in your layout as any other UI component. In case your Activity is primary showing a list you can extend the activity "ListActivity" which simplifies the handling of a "ListView". "ListActivity" extends "Activity" and provides simplified handling of lists. For example you have a predefine method if someone clicks on a list element.

"ListActivity" contains a "ListAdapter" which is responsible for managing the data. This adapter must be set in the onCreate() method of your Activity via the method setListAdapter().

If the user select in the list a list entry the method onListItemClick() will be called. This method allows to access the selected element.

Android provides already some default layouts which you can use in your Adapter, e.g. "android.R.layout.simple_list_item1". In case you don't want to use one of the pre-defined layouts your own layout must have an element with the id "@android:id/list" which is the ListView. You can also use a view with the id "@android:id/empty". This view is displayed if the list is empty. For example you could display here an error message.

ListViews and performance

Displaying a large dataset must be efficiently implemented on a mobile device. Therefore the ListView only creates views (widget) if needed and attach them to the view hierarchy. The default Adapter implementation for a ListView will recycle views, e.g. if a row is not displayed anymore it will be recycled and only its content will change. If you implement your own adapter for a view you also should do this to avoid performance problems.

This technique will be demonstrated in this tutorial.

ListActivity with ArrayAdapter and Android standard layout

Create a new Android project "de.vogella.android.listactivity" with the activity "MyList". You do not need to change the default layout "main.xml". Create the following activity.

```
package sk.saitech.android.receiver;


import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.Toast;
```

```
public class MyList extends ListActivity {



/** Called when the activity is first created. */

        public void onCreate(Bundle icicle) {
                super.onCreate(icicle);
                // Create an array of Strings, that will be put to our ListActivity          static final
String[] DISTRICTS = new String[] {

"SRIKAKULAM","VIJAYAGANAGARAM","VISHAKAPATNAM","EASTGODAVARI","WES
TGODAVARI","KRISHNA"
    };
                // Create an ArrayAdapter, that will actually make the Strings above
                // appear in the ListView
                this.setListAdapter(new ArrayAdapter<String>(this,
                                android.R.layout.simple_list_item_1, DISTRICTS));
        }

        @Override
        protected void onListItemClick(ListView l, View v, int position, long id) {
                super.onListItemClick(l, v, position, id);
                // Get the item that was clicked
                Object o = this.getListAdapter().getItem(position);
                String keyword = o.toString();
                Toast.makeText(this, "You selected: " + keyword, Toast.LENGTH_LONG)
                                .show();
        }
}
```

ListActivity with own layout

You can also define your own layout for the rows and assign this layout to your row adapter. We will add a graphic to each list entry.

Create the following layout file "rowlayout.xml" in the res/layout folder of your project

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="wrap_content" android:layout_height="wrap_content">
        <ImageView android:id="@+id/icon"
                android:layout_height="wrap_content"
                android:src="@drawable/icon" android:layout_width="22px"
                android:layout_marginTop="4px" android:layout_marginRight="4px"
                android:layout_marginLeft="4px">
        </ImageView>
        <TextView android:text="@+id/TextView01" android:layout_width="wrap_content"
                android:layout_height="wrap_content" android:id="@+id/label"
                android:textSize="30px"></TextView>
</LinearLayout>
```

Change your activity "MyList" to the following. This is almost the same coding as in the previous example, the only difference is that we are using our own layout in the ArrayAdapter and telling the adapter which UI element should contains the text.

```
package sk.saitech.android.receiver;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.Toast;
```

```
public class MyList extends ListActivity {

/** Called when the activity is first created. */

        public void onCreate(Bundle icicle) {
                super.onCreate(icicle);
                // Create an array of Strings, that will be put to our ListActivity
                static final String[] DISTRICTS = new String[] {

"SRIKAKULAM","VIJAYAGANAGARAM","VISHAKAPATNAM","EASTGODAVARI","WES
TGODAVARI","KRISHNA"
   };
                // Use your own layout and point the adapter to the UI elements which
                // contains the label
                this.setListAdapter(new ArrayAdapter<String>(this, R.layout.rowlayout,
                                R.id.label, DISTRICTS));
        }

        @Override
        protected void onListItemClick(ListView l, View v, int position, long id) {
                super.onListItemClick(l, v, position, id);
                // Get the item that was clicked
                Object o = this.getListAdapter().getItem(position);
                String keyword = o.toString();
                Toast.makeText(this, "You selected: " + keyword, Toast.LENGTH_LONG)
                                .show();

        }
}
```

GridView:

GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid. The grid items are automatically inserted to the layout using a ListAdapter.

Main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/gridview"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:columnWidth="90dp"
  android:numColumns="auto_fit"
  android:verticalSpacing="10dp"
  android:horizontalSpacing="10dp"
  android:stretchMode="columnWidth"
  android:gravity="center"
/>
```

Hellogrid.java:

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

  GridView gridview = (GridView) findViewById(R.id.gridview);
  gridview.setAdapter(new ImageAdapter(this));

  gridview.setOnItemClickListener(new OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
      Toast.makeText(HelloGridView.this, "" + position, Toast.LENGTH_SHORT).show();
    }
  });
}
```

```
public class ImageAdapter extends BaseAdapter {
  private Context mContext;

  public ImageAdapter(Context c) {
    mContext = c;
  }

  public int getCount() {
    return mThumbIds.length;
  }

  public Object getItem(int position) {
    return null;
  }

  public long getItemId(int position) {
    return 0;
  }

  // create a new ImageView for each item referenced by the Adapter
  public View getView(int position, View convertView, ViewGroup parent) {
    ImageView imageView;
    if (convertView == null) {  // if it's not recycled, initialize some attributes
      imageView = new ImageView(mContext);
      imageView.setLayoutParams(new GridView.LayoutParams(85, 85));
      imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
      imageView.setPadding(8, 8, 8, 8);
    } else {
      imageView = (ImageView) convertView;
    }

    imageView.setImageResource(mThumbIds[position]);
```

```
        return imageView;
    }

    // references to our images
    private Integer[] mThumbIds = {
            R.drawable.sample_2, R.drawable.sample_3,
            R.drawable.sample_4, R.drawable.sample_5,
            R.drawable.sample_6, R.drawable.sample_7,
            R.drawable.sample_0, R.drawable.sample_1,
            R.drawable.sample_2, R.drawable.sample_3,
            R.drawable.sample_4, R.drawable.sample_5,
            R.drawable.sample_6, R.drawable.sample_7,
            R.drawable.sample_0, R.drawable.sample_1,
            R.drawable.sample_2, R.drawable.sample_3,
            R.drawable.sample_4, R.drawable.sample_5,
            R.drawable.sample_6, R.drawable.sample_7
    };
  }
```

Gallery:

Gallery is a layout widget used to display items in a horizontally scrolling list and positions the current selection at the center of the view.

Example:

Main.xml:

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

<LinearLayout android:id="@+id/LinearLayout01"

        android:layout_width="fill_parent" android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android"

        android:orientation="vertical">

        <Gallery xmlns:android="http://schemas.android.com/apk/res/android"

                android:id="@+id/examplegallery" android:layout_width="fill_parent"

                android:layout_height="wrap_content" />

        <ImageView android:id="@+id/ImageView01"

                android:layout_width="wrap_content" android:layout_height="wrap_content"/>

</LinearLayout>
```

GalleryExample.java

```
public class GalleryExample extends Activity {

  private Gallery gallery;

  private ImageView imgView;

  private Integer[] Imgid = {

      R.drawable.a_1, R.drawable.a_2, R.drawable.a_3, R.drawable.a_4, R.drawable.a_5,
R.drawable.a_6, R.drawable.a_7

  };  @Override

  public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);
    imgView = (ImageView)findViewById(R.id.ImageView01);

    imgView.setImageResource(Imgid[0]);

    gallery = (Gallery) findViewById(R.id.examplegallery);

    gallery.setAdapter(new AddImgAdp(this));

    gallery.setOnItemClickListener(new OnItemClickListener() {

      public void onItemClick(AdapterView parent, View v, int position, long id) {

        imgView.setImageResource(Imgid[position]);

      }     }); }

  public class AddImgAdp extends BaseAdapter {

    int GalItemBg;

    private Context cont;

    public AddImgAdp(Context c) {

      cont = c;

      TypedArray typArray = obtainStyledAttributes(R.styleable.GalleryTheme);

      GalItemBg =
typArray.getResourceId(R.styleable.GalleryTheme_android_galleryItemBackground, 0);

      typArray.recycle();     }

    public int getCount() {

      return Imgid.length;     }

    public Object getItem(int position) {

      return position;     }
```

```
public long getItemId(int position) {

  return position;

}     public View getView(int position, View convertView, ViewGroup parent) {

  ImageView imgView = new ImageView(cont);

  imgView.setImageResource(Imgid[position]);

  imgView.setLayoutParams(new Gallery.LayoutParams(80, 70));

  imgView.setScaleType(ImageView.ScaleType.FIT_XY);

  imgView.setBackgroundResource(GalItemBg);

  return imgView;
}  }}
```

Other Important Views In ANDROID :

Edit Text

In this section, you will create a text field for user input, using the EditText widget. Once text has been entered into the field, the "Enter" key will display the text in a toast message.

1. Open the res/layout/main.xml file and add the EditText element (inside the LinearLayout):

```
<EditText
  android:id="@+id/edittext"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"/>
```

2. To do something with the text that the user types, add the following code to the end of the onCreate() method:

```
final EditText edittext = (EditText) findViewById(R.id.edittext);
edittext.setOnKeyListener(new OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent event) {
        // If the event is a key-down event on the "enter" button
        if ((event.getAction() == KeyEvent.ACTION_DOWN) &&
            (keyCode == KeyEvent.KEYCODE_ENTER)) {
          // Perform action on key press
          Toast.makeText(HelloFormStuff.this, edittext.getText(),
Toast.LENGTH_SHORT).show();
          return true;
        }
        return false;
    }
});
```

This captures the EditText element from the layout and adds an View.OnKeyListener. The View.OnKeyListener must implement the onKey(View, int, KeyEvent) method, which defines the action to be made when a key is pressed while the widget has focus. In this case, the method is defined to listen for the Enter key (when pressed down), then pop up a Toast message with the text that has been entered. The onKey(View, int, KeyEvent) method should always return true if the event has been handled, so that the event doesn't bubble-up (which would result in a carriage return in the text field).

3. Run the application.

Checkbox

In this section, you will create a checkbox for selecting items, using the CheckBox widget. When the checkbox is pressed, a toast message will indicate the current state of the checkbox.

1. Open the res/layout/main.xml file and add the CheckBox element (inside the LinearLayout):

```
<CheckBox android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="check it out" />
```

2. To do something when the state is changed, add the following code to the end of the onCreate() method:

```
final CheckBox checkbox = (CheckBox) findViewById(R.id.checkbox);
checkbox.setOnClickListener(new OnClickListener() {
  public void onClick(View v) {
    // Perform action on clicks, depending on whether it's now checked
    if (((CheckBox) v).isChecked()) {
      Toast.makeText(HelloFormStuff.this, "Selected", Toast.LENGTH_SHORT).show();
    } else {
      Toast.makeText(HelloFormStuff.this, "Not selected",
Toast.LENGTH_SHORT).show();
    }
  }
});
```

This captures the CheckBox element from the layout, then adds an View.OnClickListener. The View.OnClickListener must implement the onClick(View) callback method, which defines the action to be made when the checkbox is clicked. When clicked, isChecked() is called to check the new state of the check box. If it has been checked, then a Toast displays the message "Selected", otherwise it displays "Not selected". Note that the View object that is passed in the onClick(View) callback must be cast to a CheckBox because the isChecked() method is not defined by the parent View class. The CheckBox handles its own state changes, so you only need to query the current state.

3. Run it.

Tip: If you need to change the state yourself (such as when loading a saved CheckBoxPreference), use the setChecked(boolean) or toggle() method.

Radio Buttons

In this section, you will create two mutually-exclusive radio buttons (enabling one disables the other), using the RadioGroup and RadioButton widgets. When either radio button is pressed, a toast message will be displayed.

1. Open the res/layout/main.xml file and add two RadioButtons, nested in a RadioGroup (inside the LinearLayout):

```
<RadioGroup
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <RadioButton android:id="@+id/radio_red"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Red" />
  <RadioButton android:id="@+id/radio_blue"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Blue" />
</RadioGroup>
```

It's important that the RadioButtons are grouped together by the RadioGroup element so that no more than one can be selected at a time. This logic is automatically handled by the Android system. When one RadioButton within a group is selected, all others are automatically deselected.

2. To do something when each RadioButton is selected, you need an View.OnClickListener. In this case, you want the listener to be re-usable, so add the following code to create a new member in the HelloFormStuff Activity:

```
private OnClickListener radio_listener = new OnClickListener() {
  public void onClick(View v) {
    // Perform action on clicks
    RadioButton rb = (RadioButton) v;
    Toast.makeText(HelloFormStuff.this, rb.getText(), Toast.LENGTH_SHORT).show();
  }
};
```

First, the View that is passed to the onClick(View) method is cast into a RadioButton. Then a Toast message displays the selected radio button's text.

3. Now, at the bottom of the onCreate() method, add the following:

```
final RadioButton radio_red = (RadioButton) findViewById(R.id.radio_red);
final RadioButton radio_blue = (RadioButton) findViewById(R.id.radio_blue);
radio_red.setOnClickListener(radio_listener);
radio_blue.setOnClickListener(radio_listener);
```

   This captures each of the RadioButtons from the layout and adds the newly-created View.OnClickListener to each.

4. Run the application.

Tip: If you need to change the state yourself (such as when loading a saved CheckBoxPreference), use the setChecked(boolean) or toggle() method.

Toggle Button

In this section, you'll create a button used specifically for toggling between two states, using the ToggleButton widget. This widget is an excellent alternative to radio buttons if you have two simple states that are mutually exclusive ("on" and "off", for example).

1. Open the res/layout/main.xml file and add the ToggleButton element (inside the LinearLayout):

```
<ToggleButton android:id="@+id/togglebutton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Vibrate on"
    android:textOff="Vibrate off"/>
```

   The attributes android:textOn and android:textOff specify the text for the button when the button has been toggled on or off. The default values are "ON" and "OFF".

2. To do something when the state is changed, add the following code to the end of the onCreate() method:

```
final ToggleButton togglebutton = (ToggleButton) findViewById(R.id.togglebutton);
togglebutton.setOnClickListener(new OnClickListener() {
   public void onClick(View v) {
      // Perform action on clicks
      if (togglebutton.isChecked()) {
         Toast.makeText(HelloFormStuff.this, "Checked", Toast.LENGTH_SHORT).show();
      } else {
         Toast.makeText(HelloFormStuff.this, "Not checked",
Toast.LENGTH_SHORT).show();
      }
   }
});
```

This captures the ToggleButton element from the layout, then adds an View.OnClickListener. The View.OnClickListener must implement the onClick(View) callback method, which defines the action to perform when the button is clicked. In this example, the callback method checks the new state of the button, then shows a Toast message that indicates the current state.

Notice that the ToggleButton handles its own state change between checked and unchecked, so you just ask which it is.

3. Run the application.

Tip: If you need to change the state yourself (such as when loading a saved CheckBoxPreference), use the setChecked(boolean) or toggle() method.

Rating Bar

In this section, you'll create a widget that allows the user to provide a rating, with the RatingBar widget.

1. Open the res/layout/main.xml file and add the RatingBar element (inside the LinearLayout):

```
<RatingBar android:id="@+id/ratingbar"
   android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:numStars="5"
    android:stepSize="1.0"/>
```

The android:numStars attribute defines how many stars to display for the rating bar. The android:stepSize attribute defines the granularity for each star (for example, a value of 0.5 would allow half-star ratings).

2. To do something when a new rating has been set, add the following code to the end of the onCreate() method:

```
final RatingBar ratingbar = (RatingBar) findViewById(R.id.ratingbar);
ratingbar.setOnRatingBarChangeListener(new OnRatingBarChangeListener() {
  public void onRatingChanged(RatingBar ratingBar, float rating, boolean fromUser) {
    Toast.makeText(HelloFormStuff.this, "New Rating: " + rating,
Toast.LENGTH_SHORT).show();
  }
});
```

This captures the RatingBar widget from the layout with findViewById(int) and then sets an RatingBar.OnRatingBarChangeListener. The onRatingChanged() callback method then defines the action to perform when the user sets a rating. In this case, a simple Toast message displays the new rating.

3. Run the application.

Services

What is Service?

A service is a component which runs in the background, without interacting with the user. The Android platform provides a lot of pre-defined services, usually exposed via a Manager class. In your activity you access services via the method getSystemService().

Own Services must be declared via the "AndroidManifest.xml". They run the main thread of their hosting process. Therefore you should run performance intensive tasks in the background .

Own Services:

You can declare your own service to perform long running operations without user interaction or to supply functionality to other applications. A activity can start a service via the startService() method and stop the service via the stopService() method. If the activity want to interact with the service it can use the bindService() method of the service. This requires an "ServiceConnection" object which allows to connect to the service and which return a IBinder object. This IBinder object can be used by the activity to communicate with the service.

Once a service is started its onCreate() method is called. Afterwards the onStartCommand() is called with the Intent data provided by the activity. startService also allows to provide a flag with determines the lifecycle behavior of the services. START_STICKY is used for services which are explicit started or stopped. Services started with START_NOT_STICKY will end automatically after the onStartCommand() method is done. A service is started within the main thread of the application therefore all long running tasks should be performed in the background .

A Services needs to be declared in the "AndroidManifest.xml" via a <service android:name="yourclasss"> </service> and the implementing class must extend the class "Service" or one of its subclasses.

One common case is that the service and the activity are very closely related. In this case class just case the IBinder to the class the service provides. See Local Service Example for an example.

Another example would be the usage of Remote Messenger Service. In this case activities can send messages to the service and receive messages as well if they register for them.

Broadcast Receiver:

A broadcast receiver is a class which extends "BroadcastReceiver" and which is registered as a receiver in an Android Application via the AndroidManifest.xml (or via code). This class will be able to receive intents via the sendBroadcast() method. "BroadCastReceiver" defines the method "onReceive()". Only during this method your broadcast receiver object will be valid, afterwards the Android system will consider your object as no longer active. Therefore you cannot perform any asynchronous operation.

Pending Intent:

This tutorial will also use a PendingIntent. A PendingIntent is a token that you give to another application (e.g. Notification Manager, Alarm Manager or other 3rd party applications), which allows this other application to use the pennissions of your application to execute a predefined piece of code. To perform a broadcast via a pending intent so get a PendingIntent via PendingIntent.getBroadcast(). To perform an activity via an pending intent you receive the activity via PendingIntent.getActivity().

Broadcast Receiver Example:

We will define a broadcast receiver which listens to telephone state changes. If the phone receives a phone call then our receiver will be notified and log a message.

Create a new project "saitech.android.receiver.phone". We do not need an activity. Create the following "AndroidManifest.xml".

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="sk.saitech.android.receiver" android:versionCode="1"
          android:versionName="1.0">
          <application android:icon="@drawable/icon" android:label="@string/app_name">
                    <receiver android:name="MyPhoneReceiver">
```

```
                        <intent-filter>
                <action android:name="android.intent.action.PHONE_STATE"></action>
                        </intent-filter>
                </receiver>
        </application>
        <uses-sdk android:minSdkVersion="9" />
        <uses-permission android:name="android.permission.READ_PHONE_STATE"></uses-
permission>
</manifest>
```

Create the following class "MyPhoneReceiver".

```
package sk.saitech.android.receiver;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.TelephonyManager;
import android.util.Log;
public class MyPhoneReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
                Bundle extras = intent.getExtras();
                if (extras != null) {
                        String state = extras.getString(TelephonyManager.EXTRA_STATE);
                        Log.w("DEBUG", state);
                        if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
                                String phoneNumber = extras

        .getString(TelephonyManager.EXTRA_INCOMING_NUMBER);
                                Log.w("DEBUG", phoneNumber);
                        }
                }       }}
```

If you install your application and receive a call, e.g simulated by the DDMS perspective in Eclipse, then your receiver will be called and lot a message to the console.

Menus

Menus are an important part of an activity's user interface, which provide users a familiar way to perform actions. Android offers a simple framework for you to add standard menus to your application.There are three types of application menus:

Options Menu

The primary collection of menu items for an activity, which appears when the user touches the MENU button. When your application is running on Android 3.0 or later, you can provide quick access to select menu items by placing them directly in the Action Bar, as "action items."

Context Menu

A floating list of menu items that appears when the user touches and holds a view that's registered to provide a context menu.

Submenu

A floating list of menu items that appears when the user touches a menu item that contains a nested menu.

This document shows you how to create each type of menu, using XML to define the content of the menu and callback methods in your activity to respond when the user selects an item.

Creating a Menu Resource

Instead of instantiating a Menu in your application code, you should define a menu and all its items in an XML menu resource, then inflate the menu resource (load it as a programmable object) in your application code. Using a menu resource to define your menu is a good practice because it separates the content for the menu from your application code. It's also easier to visualize the structure and content of a menu in XML.To create a menu resource, create an XML file inside your project's res/menu/ directory and build the menu with the following elements:

<menu>

Defines a Menu, which is a container for menu items. A <menu> element must be the root node for the file and can hold one or more <item> and <group> elements.

<item>

Creates a MenuItem, which represents a single item in a menu. This element may contain a nested <menu> element in order to create a submenu.

<group>

An optional, invisible container for <item> elements. It allows you to categorize menu items so they share properties such as active state and visibility. See the section about Menu groups.

Here's an example menu named game_menu.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/new_game"
      android:icon="@drawable/ic_new_game"
      android:title="@string/new_game" />
  <item android:id="@+id/help"
      android:icon="@drawable/ic_help"
      android:title="@string/help" />
</menu>
```

This example defines a menu with two items. Each item includes the attributes:

android:id

A resource ID that's unique to the item, which allows the application can recognize the item when the user selects it.

android:icon

A reference to a drawable to use as the item's icon.

android:title

A reference to a string to use as the item's title.

There are many more attributes you can include in an <item>, including some that specify how the item may appear in the Action Bar. For more information about the XML syntax and attributes for a menu resource, see the Menu Resource reference.

Inflating a Menu Resource

From your application code, you can inflate a menu resource (convert the XML resource into a programmable object) using MenuInflater.inflate(). For example, the following code inflates the game_menu.xml file defined above, during the onCreateOptionsMenu() callback method, to use the menu as the activity's Options Menu:

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

The getMenuInflater() method returns a MenuInflater for the activity. With this object, you can call inflate(), which inflates a menu resource into a Menu object. In this example, the menu resource defined by game_menu.xml is inflated into the Menu that was passed into onCreateOptionsMenu(). (This callback method for the Options Menu is discussed more in the next section.)

Creating an Options Menu

The Options Menu is where you should include basic activity actions and necessary navigation items (for example, a button to open the application settings). Items in the Options Menu are accessible in two distinct ways: the MENU button or in the Action Bar (on devices running Android 3.0 or higher).

When running on a device with Android 2.3 and lower, the Options Menu appears at the bottom of the screen, as shown in figure 1. When opened, the first visible portion of the Options Menu is the icon menu. It holds the first six menu items. If you add more than six items to the Options Menu, Android places the sixth item and those after it into the overflow menu, which the user can open by touching the "More" menu item.

On Android 3.0 and higher, items from the Options Menu is placed in the Action Bar, which appears at the top of the activity in place of the traditional title bar. By default all items from the Options Menu are placed in the overflow menu, which the user can open by touching the menu icon on the right side of the Action Bar. However, you can place select menu items directly in the Action Bar as "action items," for instant access, as shown in figure 2.

When the Android system creates the Options Menu for the first time, it calls your activity's onCreateOptionsMenu() method. Override this method in your activity and populate the Menu that is passed into the method, Menu by inflating a menu resource as described above in Inflating a Menu Resource. For example:

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

You can also populate the menu in code, using add() to add items to the Menu.

Note: On Android 2.3 and lower, the system calls onCreateOptionsMenu() to create the Options Menu when the user opens it for the first time, but on Android 3.0 and greater, the system creates it as soon as the activity is created, in order to populate the Action Bar.

Responding to user action

When the user selects a menu item from the Options Menu (including action items in the Action Bar), the system calls your activity's onOptionsItemSelected() method. This method passes the MenuItem that the user selected. You can identify the menu item by calling getItemId(), which returns the unique ID for the menu item (defined by the android:id attribute in the menu resource or with an integer given to the add() method). You can match this ID against known menu items and perform the appropriate action. For example:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   // Handle item selection
   switch (item.getItemId()) {
   case R.id.new_game:
      newGame();
      return true;
   case R.id.help:
      showHelp();
      return true;
   default:
      return super.onOptionsItemSelected(item);
   }
}
```

In this example, getItemId() queries the ID for the selected menu item and the switch statement compares the ID against the resource IDs that were assigned to menu items in the XML resource. When a switch case successfully handles the menu item, it returns true to indicate that the item selection was handled. Otherwise, the default statement passes the menu item to the super class, in case it can handle the item selected. (If you've directly extended the Activity class, then the super class returns false, but it's a good practice to pass unhandled menu items to the super class instead of directly returning false.)

Additionally, Android 3.0 adds the ability for you to define the on-click behavior for a menu item in the menu resource XML, using the android:onClick attribute. So you don't need to implement onOptionsItemSelected(). Using the android:onClick attribute, you can specify a method to call when the user selects the menu item. Your activity must then implement the method specified in the android:onClick attribute so that it accepts a single MenuItem parameter—when the system calls this method, it passes the menu item selected.

Tip: If your application contains multiple activities and some of them provide the same Options Menu, consider creating an activity that implements nothing except the onCreateOptionsMenu() and onOptionsItemSelected() methods. Then extend this class for each activity that should share the same Options Menu. This way, you have to manage only one set of code for handling menu actions and each descendant class inherits the menu behaviors.

If you want to add menu items to one of your descendant activities, override onCreateOptionsMenu() in that activity. Call super.onCreateOptionsMenu(menu) so the original menu items are created, then add new menu items with menu.add(). You can also override the super class's behavior for individual menu items.

Changing menu items at runtime

Once the activity is created, the onCreateOptionsMenu() method is called only once, as described above. The system keeps and re-uses the Menu you define in this method until your activity is destroyed. If you want to change the Options Menu any time after it's first created, you must override the onPrepareOptionsMenu() method. This passes you the Menu object as it currently exists. This is useful if you'd like to remove, add, disable, or enable menu items depending on the current state of your application.

On Android 2.3 and lower, the system calls onPrepareOptionsMenu() each time the user opens the Options Menu.

On Android 3.0 and higher, you must call invalidateOptionsMenu() when you want to update the menu, because the menu is always open. The system will then call onPrepareOptionsMenu() so you can update the menu items.

Note: You should never change items in the Options Menu based on the View currently in focus. When in touch mode (when the user is not using a trackball or d-pad), views cannot take focus, so you should never use focus as the basis for modifying items in the Options Menu. If you want to provide menu items that are context-sensitive to a View, use a Context Menu.

If you're developing for Android 3.0 or higher, be sure to also read the Action Bar developer guide.

Creating a Context Menu

A context menu is conceptually similar to the menu displayed when the user performs a "right-click" on a PC. You should use a context menu to provide the user access to actions that pertain to a specific item in the user interface. On Android, a context menu is displayed when the user performs a "long-press" (press and hold) on an item.

You can create a context menu for any View, though context menus are most often used for items in a ListView. When the user performs a long-press on an item in a ListView and the list is registered to provide a context menu, the list item signals to the user that a context menu is available by animating its background color—it transitions from orange to white before opening the context menu. (The Contacts application demonstrates this feature.)

Register a ListView

If your activity uses a ListView and you want all list items to provide a context menu, register all items for a context menu by passing the ListView to registerForContextMenu(). For example, if you're using a ListActivity, register all list items like this:

```
registerForContextMenu(getListView());
```

In order for a View to provide a context menu, you must "register" the view for a context menu. Call registerForContextMenu() and pass it the View you want to give a context menu. When this View then receives a long-press, it displays a context menu.

To define the context menu's appearance and behavior, override your activity's context menu callback methods, onCreateContextMenu() and onContextItemSelected().

For example, here's an onCreateContextMenu() that uses the context_menu.xml menu resource:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                ContextMenuInfo menuInfo) {
 super.onCreateContextMenu(menu, v, menuInfo);
 MenuInflater inflater = getMenuInflater();
 inflater.inflate(R.menu.context_menu, menu);
}
```

MenuInflater is used to inflate the context menu from a menu resource. (You can also use add() to add menu items.) The callback method parameters include the View that the user selected and a ContextMenu.ContextMenuInfo object that provides additional information about the item selected. You might use these parameters to determine which context menu should be created, but in this example, all context menus for the activity are the same.

Then when the user selects an item from the context menu, the system calls onContextItemSelected(). Here is an example of how you can handle selected items:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
 AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
 switch (item.getItemId()) {
 case R.id.edit:
  editNote(info.id);
  return true;
 case R.id.delete:
  deleteNote(info.id);
  return true;
 default:
  return super.onContextItemSelected(item);
 }
}
```

The structure of this code is similar to the example for Creating an Options Menu, in which getItemId() queries the ID for the selected menu item and a switch statement matches the item to the IDs that are defined in the menu resource. And like the options menu example, the default statement calls the super class in case it can handle menu items not handled here, if necessary.

In this example, the selected item is an item from a ListView. To perform an action on the selected item, the application needs to know the list ID for the selected item (it's position in the ListView). To get the ID, the application calls getMenuInfo(), which returns a AdapterView.AdapterContextMenuInfo object that includes the list ID for the selected item in the id field. The local methods editNote() and deleteNote() methods accept this list ID to perform an action on the data specified by the list ID.

Note: Items in a context menu do not support icons or shortcut keys.

Creating Submenus

A submenu is a menu that the user can open by selecting an item in another menu. You can add a submenu to any menu (except a submenu). Submenus are useful when your application has a lot of functions that can be organized into topics, like items in a PC application's menu bar (File, Edit, View, etc.).

When creating your menu resource, you can create a submenu by adding a <menu> element as the child of an <item>. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/file"
        android:icon="@drawable/file"
        android:title="@string/file" >
      <!-- "file" submenu -->
      <menu>
         <item android:id="@+id/create_new"
              android:title="@string/create_new" />
         <item android:id="@+id/open"
              android:title="@string/open" />
      </menu>
   </item>
</menu>
```

When the user selects an item from a submenu, the parent menu's respective on-item-selected callback method receives the event. For instance, if the above menu is applied as an Options Menu, then the onOptionsItemSelected() method is called when a submenu item is selected.

You can also use addSubMenu() to dynamically add a SubMenu to an existing Menu. This returns the new SubMenu object, to which you can add submenu items, using add().

Chapter 8 Background Services

A Service is an application component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use. Each service class must have a corresponding <service> declaration in its package's AndroidManifest.xml. Services can be started with Context.startService() and Context.bindService().

Note that services, like other application objects, run in the main thread of their hosting process. This means that, if your service is going to do any CPU intensive (such as MP3 playback) or blocking (such as networking) operations, it should spawn its own thread in which to do that work. More information on this can be found in Processes and Threads. The IntentService class is available as a standard implementation of Service that has its own thread where it schedules its work to be done.

1. What is a Service?
2. Service Lifecycle
3. Permissions
4. Process Lifecycle
5. Local Service Sample
6. Remote Messenger Service Sample

What is a Service?

Most confusion about the Service class actually revolves around what it is *not*:

- A Service is **not** a separate process. The Service object itself does not imply it is running in its own process; unless otherwise specified, it runs in the same process as the application it is part of.
- A Service is **not** a thread. It is not a means itself to do work off of the main thread (to avoid Application Not Responding errors).

Thus a Service itself is actually very simple, providing two main features:

- A facility for the application to tell the system *about* something it wants to be doing in the background (even when the user is not directly interacting with the application). This corresponds to calls to Context.startService(), which ask the system to schedule work for the service, to be run until the service or someone else explicitly stop it.

- A facility for an application to expose some of its functionality to other applications. This corresponds to calls to Context.bindService(), which allows a long-standing connection to be made to the service in order to interact with it.

When a Service component is actually created, for either of these reasons, all that the system actually does is instantiate the component and call its onCreate() and any other appropriate callbacks on the main thread. It is up to the Service to implement these with the appropriate behavior, such as creating a secondary thread in which it does its work.

Note that because Service itself is so simple, you can make your interaction with it as simple or complicated as you want: from treating it as a local Java object that you make direct method calls on (as illustrated by Local Service Sample), to providing a full remoteable interface using AIDL.

Service Lifecycle

There are two reasons that a service can be run by the system. If someone calls Context.startService() then the system will retrieve the service (creating it and calling its onCreate() method if needed) and then call its onStartCommand(Intent, int, int) method with the arguments supplied by the client. The service will at this point continue running until Context.stopService() or stopSelf() is called. Note that multiple calls to Context.startService() do not nest (though they do result in multiple corresponding calls to onStartCommand()), so no matter how many times it is started a service will be stopped once Context.stopService() or stopSelf() is called; however, services can use their stopSelf(int) method to ensure the service is not stopped until started intents have been processed.

For started services, there are two additional major modes of operation they can decide to run in, depending on the value they return from onStartCommand(): START_STICKY is used for services that are explicitly started and stopped as needed, while START_NOT_STICKY or START_REDELIVER_INTENT are used for services that should only remain running while processing any commands sent to them. See the linked documentation for more detail on the semantics.

Clients can also use Context.bindService() to obtain a persistent connection to a service. This likewise creates the service if it is not already running (calling onCreate() while doing so), but does not call onStartCommand(). The client will receive the IBinder object that the service returns from its onBind(Intent) method, allowing the client to then make calls back to the service. The service will remain running as long as the connection is established (whether or not the client retains a reference on the service's IBinder). Usually the IBinder returned is for a complex interface that has been written in aidl.

A service can be both started and have connections bound to it. In such a case, the system will keep the service running as long as either it is started *or* there are one or more connections to it with the Context.BIND_AUTO_CREATE flag. Once neither of these situations hold, the service's onDestroy() method is called and the service is effectively terminated. All cleanup (stopping threads, unregistering receivers) should be complete upon returning from onDestroy().

Permissions

Global access to a service can be enforced when it is declared in its manifest's <service> tag. By doing so, other applications will need to declare a corresponding <uses-permission> element in their own manifest to be able to start, stop, or bind to the service.

In addition, a service can protect individual IPC calls into it with permissions, by calling the checkCallingPermission(String) method before executing the implementation of that call.

Process Lifecycle

The Android system will attempt to keep the process hosting a service around as long as the service has been started or has clients bound to it. When running low on memory and needing to kill existing processes, the priority of a process hosting the service will be the higher of the following possibilities:

- If the service is currently executing code in its onCreate(), onStartCommand(), or onDestroy() methods, then the hosting process will be a foreground process to ensure this code can execute without being killed.
- If the service has been started, then its hosting process is considered to be less important than any processes that are currently visible to the user on-screen, but more important than any process not visible. Because only a few processes are generally visible to the user, this means that the service should not be killed except in extreme low memory conditions.

- If there are clients bound to the service, then the service's hosting process is never less important than the most important client. That is, if one of its clients is visible to the user, then the service itself is considered to be visible.
- A started service can use the startForeground(int, Notification) API to put the service in a foreground state, where the system considers it to be something the user is actively aware of and thus not a candidate for killing when low on memory. (It is still theoretically possible for the service to be killed under extreme memory pressure from the current foreground application, but in practice this should not be a concern.)

Note this means that most of the time your service is running, it may be killed by the system if it is under heavy memory pressure. If this happens, the system will later try to restart the service. An important consequence of this is that if you implement onStartCommand() to schedule work to be done asynchronously or in another thread, then you may want to use START_FLAG_REDELIVERY to have the system re-deliver an Intent for you so that it does not get lost if your service is killed while processing it.

Other application components running in the same process as the service (such as an Activity) can, of course, increase the importance of the overall process beyond just the importance of the service itself.

Local Service Sample

One of the most common uses of a Service is as a secondary component running alongside other parts of an application, in the same process as the rest of the components. All components of an .apk run in the same process unless explicitly stated otherwise, so this is a typical situation.

When used in this way, by assuming the components are in the same process, you can greatly simplify the interaction between them: clients of the service can simply cast the IBinder they receive from it to a concrete class published by the service.

An example of this use of a Service is shown here. First is the Service itself, publishing a custom class when bound:

```
public class LocalService extends Service {
   private NotificationManager mNM;

   // Unique Identification Number for the Notification.
// We use it on Notification start, and to cancel it.
private int NOTIFICATION = R.string.local_service_started;


public class LocalBinder extends Binder {
   LocalService getService() {
      return LocalService.this;
   }
}   public void onCreate() {
   mNM = (NotificationManager)getSystemService(NOTIFICATION_SERVICE);


   // Display a notification about us starting.  We put an icon in the status bar.
   showNotification();
}
public int onStartCommand(Intent intent, int flags, int startId) {
   Log.i("LocalService", "Received start id " + startId + ": " + intent);
   // We want this service to continue running until it is explicitly
   // stopped, so return sticky.
   return START_STICKY;
}
public void onDestroy() {
   // Cancel the persistent notification.
   mNM.cancel(NOTIFICATION);


   // Tell the user we stopped.
   Toast.makeText(this, R.string.local_service_stopped, Toast.LENGTH_SHORT).show();
}
public IBinder onBind(Intent intent) {
   return mBinder;
}
// This is the object that receives interactions from clients.  See
// RemoteService for a more complete example.
private final IBinder mBinder = new LocalBinder();
```

```
  /**
   * Show a notification while this service is running.
   */
  private void showNotification() {
    // In this sample, we'll use the same text for the ticker and the expanded notification
    CharSequence text = getText(R.string.local_service_started);

    // Set the icon, scrolling text and timestamp
    Notification notification = new Notification(R.drawable.stat_sample, text,
        System.currentTimeMillis());

    // The PendingIntent to launch our activity if the user selects this notification
    PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
        new Intent(this, LocalServiceActivities.Controller.class), 0);

    // Set the info for the views that show in the notification panel.
    notification.setLatestEventInfo(this, getText(R.string.local_service_label),
           text, contentIntent);

    // Send the notification.
    mNM.notify(NOTIFICATION, notification);
  }
}
```

With that done, one can now write client code that directly accesses the running service, such as:

```
private LocalService mBoundService;

private ServiceConnection mConnection = new ServiceConnection() {
  public void onServiceConnected(ComponentName className, IBinder service) {
    // This is called when the connection with the service has been
    // established, giving us the service object we can use to
    // interact with the service.  Because we have bound to a explicit
    // service that we know is running in our own process, we can
```

```
        // cast its IBinder to a concrete class and directly access it.
        mBoundService = ((LocalService.LocalBinder)service).getService();

        // Tell the user about this for our demo.
        Toast.makeText(Binding.this, R.string.local_service_connected,
                Toast.LENGTH_SHORT).show();
    }
    public void onServiceDisconnected(ComponentName className) {
        // This is called when the connection with the service has been
        // unexpectedly disconnected -- that is, its process crashed.
        // Because it is running in our same process, we should never
        // see this happen.
        mBoundService = null;
        Toast.makeText(Binding.this, R.string.local_service_disconnected,
                Toast.LENGTH_SHORT).show();
    }};

void doBindService() {
    // Establish a connection with the service.  We use an explicit
    // class name because we want a specific service implementation that
    // we know will be running in our own process (and thus won't be
    // supporting component replacement by other applications).
    bindService(new Intent(Binding.this,
            LocalService.class), mConnection, Context.BIND_AUTO_CREATE);
    mIsBound = true;
}

void doUnbindService() {
    if (mIsBound) {
        // Detach our existing connection.
        unbindService(mConnection);
        mIsBound = false;
    }
```

```
}

@Override
protected void onDestroy() {
   super.onDestroy();
   doUnbindService();
}
```

Remote Messenger Service Sample

If you need to be able to write a Service that can perform complicated communication with clients in remote processes (beyond simply the use of Context.startService to send commands to it), then you can use the Messenger class instead of writing full AIDL files.

An example of a Service that uses Messenger as its client interface is shown here. First is the Service itself, publishing a Messenger to an internal Handler when bound:

```
          public class MessengerService extends Service {
             /** For showing and hiding our notification. */
             NotificationManager mNM;
             /** Keeps track of all current registered clients. */
             ArrayList<Messenger> mClients = new ArrayList<Messenger>();
             /** Holds last value set by a client. */
             int mValue = 0;
             static final int MSG_REGISTER_CLIENT = 1;
            static final int MSG_UNREGISTER_CLIENT = 2;
             static final int MSG_SET_VALUE = 3;
   class IncomingHandler extends Handler {
      @Override
      public void handleMessage(Message msg) {
         switch (msg.what) {
            case MSG_REGISTER_CLIENT:
               mClients.add(msg.replyTo);
               break;
            case MSG_UNREGISTER_CLIENT:
```

```
          mClients.remove(msg.replyTo);
          break;
        case MSG_SET_VALUE:
          mValue = msg.arg1;
          for (int i=mClients.size()-1; i>=0; i–) {
            try {
              mClients.get(i).send(Message.obtain(null,
                    MSG_SET_VALUE, mValue, 0));
            } catch (RemoteException e) {
              // The client is dead.  Remove it from the list;
              // we are going through the list from back to front
              // so this is safe to do inside the loop.
              mClients.remove(i);
            }
          }
          break;
        default:
          super.handleMessage(msg);
    }     }   }

  /**
   * Target we publish for clients to send messages to IncomingHandler.
   */
  final Messenger mMessenger = new Messenger(new IncomingHandler());

  @Override
  public void onCreate() {
    mNM = (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    // Display a notification about us starting.
    showNotification();
  }
```

```
@Override
public void onDestroy() {
   // Cancel the persistent notification.
   mNM.cancel(R.string.remote_service_started);

   // Tell the user we stopped.
   Toast.makeText(this, R.string.remote_service_stopped, Toast.LENGTH_SHORT).show();
}

/**
 * When binding to the service, we return an interface to our messenger
 * for sending messages to the service.
 */
@Override
public IBinder onBind(Intent intent) {
   return mMessenger.getBinder();
}

/**
 * Show a notification while this service is running.
 */
private void showNotification() {
   // In this sample, we'll use the same text for the ticker and the expanded notification
   CharSequence text = getText(R.string.remote_service_started);

   // Set the icon, scrolling text and timestamp
   Notification notification = new Notification(R.drawable.stat_sample, text,
         System.currentTimeMillis());

   // The PendingIntent to launch our activity if the user selects this notification
   PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
         new Intent(this, Controller.class), 0);
```

```
        // Set the info for the views that show in the notification panel.
        notification.setLatestEventInfo(this, getText(R.string.remote_service_label),
                       text, contentIntent);

        // Send the notification.
        // We use a string id because it is a unique number.  We use it later to cancel.
        mNM.notify(R.string.remote_service_started, notification);
    }
}
```

If we want to make this service run in a remote process (instead of the standard one for its .apk), we can use android:process in its manifest tag to specify one:

```
<service android:name=".app.MessengerService" android:process=":remote" />
```

Note that the name "remote" chosen here is arbitrary, and you can use other names if you want additional processes. The ':' prefix appends the name to your package's standard process name.

With that done, clients can now bind to the service and send messages to it. Note that this allows clients to register with it to receive messages back as well:

```
/** Messenger for communicating with service. */
Messenger mService = null;
/** Flag indicating whether we have called bind on the service. */
boolean mIsBound;
/** Some text view we are using to show state information. */
TextView mCallbackText;

/**
 * Handler of incoming messages from service.
 */
class IncomingHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MessengerService.MSG_SET_VALUE:
```

```
            mCallbackText.setText("Received from service: " + msg.arg1);
            break;
          default:
            super.handleMessage(msg);
      }  }}

/**
 * Target we publish for clients to send messages to IncomingHandler.
 */
final Messenger mMessenger = new Messenger(new IncomingHandler());

/**
 * Class for interacting with the main interface of the service.
 */
private ServiceConnection mConnection = new ServiceConnection() {
   public void onServiceConnected(ComponentName className,
         IBinder service) {
      // This is called when the connection with the service has been
      // established, giving us the service object we can use to
      // interact with the service.  We are communicating with our
      // service through an IDL interface, so get a client-side
      // representation of that from the raw service object.
      mService = new Messenger(service);
      mCallbackText.setText("Attached.");

      // We want to monitor the service for as long as we are
      // connected to it.
      try {
        Message msg = Message.obtain(null,
              MessengerService.MSG_REGISTER_CLIENT);
        msg.replyTo = mMessenger;
        mService.send(msg);
```

```
            // Give it some value as an example.
            msg = Message.obtain(null,
                    MessengerService.MSG_SET_VALUE, this.hashCode(), 0);
            mService.send(msg);
        } catch (RemoteException e) {
            // In this case the service has crashed before we could even
            // do anything with it; we can count on soon being
            // disconnected (and then reconnected if it can be restarted)
            // so there is no need to do anything here.
        }

        // As part of the sample, tell the user what happened.
        Toast.makeText(Binding.this, R.string.remote_service_connected,
                Toast.LENGTH_SHORT).show();
    }

    public void onServiceDisconnected(ComponentName className) {
        // This is called when the connection with the service has been
        // unexpectedly disconnected -- that is, its process crashed.
        mService = null;
        mCallbackText.setText("Disconnected.");

        // As part of the sample, tell the user what happened.
        Toast.makeText(Binding.this, R.string.remote_service_disconnected,
                Toast.LENGTH_SHORT).show();
    }
};

void doBindService() {
    // Establish a connection with the service.  We use an explicit
    // class name because there is no reason to be able to let other
    // applications replace our component.
    bindService(new Intent(Binding.this,
        MessengerService.class), mConnection, Context.BIND_AUTO_CREATE);
    mIsBound = true;
    mCallbackText.setText("Binding.");
}
void doUnbindService() {
    if (mIsBound) {
        // If we have received the service, and hence registered with
        // it, then now is the time to unregister.
        if (mService != null) {
            try {
                Message msg = Message.obtain(null,
                        MessengerService.MSG_UNREGISTER_CLIENT);
                msg.replyTo = mMessenger;
                mService.send(msg);
            } catch (RemoteException e) {
                // There is nothing special we need to do if the service
                // has crashed.
            }     }

        // Detach our existing connection.
        unbindService(mConnection);
        mIsBound = false;
        mCallbackText.setText("Unbinding.");
    }
}
```

Summary

Constants

int START_CONTINUATION_MASK	Bits returned by onStartCommand(Intent, int, int) describing how to continue the service if it is killed.
int START_FLAG_REDELIVERY	This flag is set in onStartCommand(Intent, int, int) if the Intent is a re-delivery of a previously delivered

	intent, because the service had previously returned START_REDELIVER_INTENT but had been killed before calling stopSelf(int) for that Intent.
int START_FLAG_RETRY	This flag is set in onStartCommand(Intent, int, int) if the Intent is a a retry because the original attempt never got to or returned from onStartCommand(Intent, int, int).
int START_NOT_STICKY	Constant to return from onStartCommand(Intent, int, int): if this service's process is killed while it is started (after returning from onStartCommand(Intent, int, int)), and there are no new start intents to deliver to it, then take the service out of the started state and don't recreate until a future explicit call to Context.startService(Intent).
int START_REDELIVER_INTENT	Constant to return from onStartCommand(Intent, int, int): if this service's process is killed while it is started (after returning from onStartCommand(Intent, int, int)), then it will be scheduled for a restart and the last delivered Intent re-delivered to it again via onStartCommand(Intent, int, int).
int START_STICKY	Constant to return from onStartCommand(Intent, int, int): if this service's process is killed while it is started (after returning from onStartCommand(Intent, int, int)), then leave it in the started state but don't retain this delivered intent.

Many a times the application needs to do some tasks in the background for which user interventions is not required (or very less intervention is required). These background processes keeps working even if user is using some other application on the phone.

To define such background processes android has a concept of Services. Service in android is long lived application component. Service doesn't implement any User Interface. Common example of service is Media Player application that keeps playing song in the background, file download application that can download the file in the background.Let's see how to create a service.

Creating a service

Android has defined a base class for all services as 'Service'. All the services have to extend from this Service class. Service class defines service lifecycle methods like onCreate(), onStart(), onDestroy(). Here is the example a service class

```
package com.sivaji.testApp.service;
public class MyService extends Service {
        public IBinder onBind(Intent intent) {
                return null;
        }
        public void onCreate() {
                super.onCreate();
                Toast.makeText(this, "Service created...", Toast.LENGTH_LONG).show();
        }
        public void onDestroy() {
                super.onDestroy();
                Toast.makeText(this, "Service destroyed...", Toast.LENGTH_LONG).show();
        }
}
```

The above service is notifying the user when the service is created and service is destroyed.

Like every thing else in android, Service in android are also associated with the intents. This intent is required while using the service.The service entry has to be done in the AndroidManifest.xml file along with the service intent as shown below:

```
<service class=".service.MyService">
<intent-filter>
<action android:value="com.sivaji.testApp.service.MY_SERVICE" />
</intent-filter>
</service>
```

Now our service is created and can be used by the application code.

Using the service:

The application can start the service with the help of Context.startService method. The method will call the onCreate method of the service if service is not already created; else onStart method will be called. Here is the code to start the MyService

```
Intent serviceIntent = new Intent();
serviceIntent.setAction("com.sivaji.testApp.service.MY_SERVICE");
startService(serviceIntent);
```

The service started with startService method will keep on running until stopService() is called or stopSelf() method is called.

Another way to use service is to bind to the service. The service contented this way will be considered required by the system only for as long as the calling context exists. To bind to the service a service connection object need to be created. The service connection object tell the application when the service is connected or disconnected. Here is how you can bind to the service.

```
ServiceConnection conn = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
```

```
        Log.i("INFO", "Service bound ");
        @Override
        public void onServiceDisconnected(ComponentName arg0) {
        Log.i("INFO", "Service Unbound ");
        }        }
        bindService(new Intent("com.sivaji.testApp.service.MY_SERVICE"), conn,
Context.BIND_AUTO_CREATE);
}
```

The application can communicate with the service when application is connected with the service. Generally the service communicate is done with the help of Service Interface. Service interface defines methods for which service can provider implementation. For example here is some interface:

```
        package com.sivaji.testApp;
        public interface IMyService {
        public int getStatusCode();
                }
```

Using this interface the application can ask the Service about its status. Lets see how the service can support this interface. Previously we saw a method called onBind which return IBinder object, the method gets called when some client of the service binds to the service. This is the same object that is passed to the onServiceConnected method. The application can communicate with the service using this IBinder object. Here is how this can be done:

```
public class MyService extends Service {
        private int statusCode;
        private MyServiceBinder myServiceBinder = new MyServiceBinder();
        @Override
        public IBinder onBind(Intent intent) {
                return myServiceBinder; // object of the class that implements Service
        }
        public class MyServiceBinder extends Binder implements IMyService {
                public int getStatusCode() {
                        return statusCode;          }          }}
```

And here is how application can call getStatusCode method:

```
        ServiceConnection conn = new ServiceConnection() {

                public void onServiceConnected(ComponentName name, IBinder service) {
                        IMyService myService = (IMyService) service;
                        statusCode = myService.getStatusCode();
                        Log.i("INFO", "Service bound ");
                }       };
```

You can also define ServiceListener interface which the client of the service has to be implemented to gets update from the service. In that case the service interface will have to define methods to register and unregister the ServiceListener objects.

Communication with Remote Service:

The services that we defined until now run in the application processes, you can define service that can run in their own process. For two processes to communicate with each other they need to marshal the object to sent to other process.Android provide an AIDL tool (Android Interface definition Language) to handle all marshalling and communication part.

The service has to provide the Service interface as an aidl file. The AIDL tool will create a java interface corresponding for the aidl Service Interface. The AIDL tool also defines a stub class in the generated service interface, which implements the Service Interface (as abstract methods) and also provides some other required functionality. The service interface implementation class has to extend this stub class and define the service interface methods. The service onBind method will return object of this implementation class so that the client application can use the service methods. Here is the how the communication can be done:

Create a file as IMyRemoteService.aidl as follows:

```
 package com.sivaji.testApp;

interface IMyRemoteService {
int getStatusCode();

}
```

The eclipse android plug-in will create a Java interface for the aidl file created above as the part of build process.The interface generated above will have a Stub inner class. Define a class that extends this stub class. Here is the code for the RemoteService class:

```
package com.wissen.testApp;
class RemoteService implements Service {
        int statusCode;

        @Override
        public IBinder onBind(Intent arg0) {
                return myRemoteServiceStub;
        }
        private IMyRemoteService.Stub myRemoteServiceStub = new IMyRemoteService.Stub() {
                public int getStatusCode() throws RemoteException {
                        return 0;
                }
        };}
```

When the client application connect to the service the onServiceConnected method will be called and client will get the IBinder object of the service. The Stub class also provides a method to obtain the Service Interface object from the IBinder object. Here is the client onServiceConnected code

```
        ServiceConnection conn = new ServiceConnection() {
                @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
                        IMyRemoteService myRemoteService = IMyRemoteService.Stub        .asInterface(se
                        try {
                        statusCode = myRemoteService.getStatusCode();
                        } catch (RemoteException e) {
                                // handle exception
                        }
                        Log.i("INFO", "Service bound ");
```

```
                    }
        };
```

Permissions:

Service may specify required user permissions in the AndroidManifest.xml in <service> tag like this,

```
<service class=".service.MyService"
android:permission="com.wissen.permission.MY_SERVICE_PERMISSION">
<intent-filter>
<action android:value="com.wissen.testApp.service.MY_SERVICE" />
</intent-filter>
</service>
```

Then to use above service the application has to ask for permission with the help of <user-permission> tag as follows:

```
<uses-permission android:name="com.wissen.permission.MY_SERVICE_PERMISSION"></uses-permission>
```

So in today's post we saw how to create service and use it. In the next post we will see how to use ContentProviders.

Interview Questions :

1. **What is the present version API level?**
2. **Write a four methods from every Android class?**
3. **What is the difference between JVM and DVM?**
4. **Write a process of generating "apk" file from java code?**
5. **What is the use of adb tool?**
6. **Who is the vendor of android?**
7. **What is the API level for 2.3.1?**
8. **What is the use of setContentView(int id)? And from which class it is?**
9. **What is the purpose of R.java file?**
10. **Write a three points for AndroidManifest.xml?**
11. **What is the use of import statement in program?**
12. **What is the difference between margin and padding attribute?**
13. **How to perform event handling with out using onClickListener Interface?**
14. **What is application component and what are they?**

15. **Write a two third party libraries in android?**
16. **What is the difference between this() and super()?**
17. **How to create a object for Context class?**
18. **What are the parameters we are passing in makeText() of Toast class?**
19. **From which package we are getting View Class? And what is a super class for View?**
20. **All the application component classes are coming from which package?**
21. **What is the dex file name ll be generated for all the android projects?**
22. **onclickListener contains How many methods? What are they?**
23. **Write a seven tools from android?**
24. **What is a Layout? What are they?**
25. **What is the difference between ImageButtton and ToggleButton?**
26. **What is the difference between RadioGroup and RadioButton?**
27. **What is the super class for LinearLayout and TableLayout?**
28. **$How many ways we can store the data in android? What are they?**
29. **Write a code for retrieving the data from EditText?**
30. **Difference between android.view.View.onClickListener and ndroid.content.DialogInterfaces.onClickListener?**
31. **Write a syntax for normal method definition in java and android?**
32. **aapt stands for what? What is the use of it?**
33. **What is a variable and how many types of variables we have in android?**
34. **All the views are coming from which package?**
35. **Difference between wrap_content and match_parent?**
36. **What is the difference between layout_gravity and gravity?**

SQLITE DATABASE

(with example of Sqlite)

SQLite is not a Database. It is file but it acts as if database. What are the operations (Commands/queries) we can perform on normal database concepts like oracle, SQLServer those we can perform on this file. In Other words It is file. Without using javaI/O Streams we can do operations on this file by using SQLite Commands. In this we can store the data Approximately up to 200kb data.

Note : This File doesn't support to JOIN and FORIENKEY reference mechanism.

No databases are automatically supplied to you by Android. If you want to use SQLite, you have to create your own database, then populate it with your own tables, indexes, and data. To create and open a database, your best option is to create a subclass of SQLiteOpenHelper. This class wraps up the logic to create and upgrade a database, per your specifications, as needed by your application. Your subclass of SQLiteOpenHelper will need three methods:

- The constructor, chaining upward to the SQLiteOpenHelper constructor. This takes the Context (e.g., an Activity), the name of the database, an optional cursor factory(typically, just pass null),and an integer representing the version of the database schema you are using.
- onCreate(), which passes you a SQLiteDatabase object that you need to populate with tables and initial data, as appropriate.
- onUpgrade(), which passes you a SQLiteDatabase object and the old and new version numbers, so you can figure out how best to convert the database from the old schema to the new one. The simplest, albeit least friendly, approach is to simply drop the old tables and create new ones.

You can always use execSQL() method from SQLiteDatabase, just like you did for creating the tables. The execSQL() method works for any SQL that does not return results, so it can handle INSERT, UPDATE, DELETE, etc. just fine. So,
For example:

```
        db.execSQL("CREATE TABLE Employee (_id INTEGER PRIMARY KEY AUTOINCREMENT,
Name TEXT, Salary  REAL);");
```

After Creating the table we need to insert the data and also interact with that data it means update it and delete it. For this operations in Sqlite Concept we have alternative methods those are called as insert(), update(), and delete() methods on the SQLiteDatabase object. These are "builder" sorts of methods, in that the break down the SQL statements into discrete chunks, then take those chunks as parameters.

These methods make use of ContentValues objects, which implement a Map-esque interface, albeit one that has additional methods for working with SQLite types. For example, in addition to get() to retrieve a value by its key, you have getAsInteger(), getAsString(), and so forth.

DML (Data Manipulation Language) OPERATIONS:

INSERT OPERATION BY USING insert() METHOD :

The insert() method takes the name of the table, the name of one column as the "null column hack", and a ContentValues with the initial values you want put into this row. The "null column hack" is for the case where the ContentValues instance is empty – the column named as the "null columnhack" will be explicitly assigned the value NULL in the SQL INSERT statement generated by insert().

For example:

```
        ContentValues cv=newContentValues();
        cv.put("Name","shivaji_sai");
        cv.put("Salary",35000);
        db.insert("constants",getNullColumnHack(), cv);
```

Note : Here db is object reference of SQLiteDatabase.

UPDATE OPERATION BY USING update() METHOD :

The update() method takes the name of the table, a ContentValues representing the columns and replacement values to use, an optional WHERE clause, and an optional list of parameters to fill into the WHERE clause, to replace any embedded question marks (?). Since update() only replaces columns with fixed values, versus ones computed based on other information, you may need to use execSQL() to accomplish some ends.

The WHERE clause and parameter list works akin to the positional SQL parameters you may be used to from other SQL APIs.

For example:

db.update(tablename , contentvalues , whereClause , whereArgs);

DELETE OPERATION BY USING delete() METHOD :

The delete() method works akin to update(), taking the name of the table ,the optional WHERE clause, and the corresponding parameters to fill into the WHERE clause.

DRL(Data Retrieval Language):

RETRIVENING DATA FROM SQLite DATABASE :

As with INSERT, UPDATE, and DELETE, you have two main options for retrieving data from a SQLite database using SELECT:

1. You can use rawQuery() to invoke a SELECT statement directly, or
2. You can use query() to build up a query from its component parts

rawQuery() method Using for Fetch the Data from Table:

The simplest solution, at least in terms of the API, is rawQuery(). Simply callit with your SQL SELECT statement. The SELECT statement can includepositional parameters; the array of these forms your second parameter to rawQuery(). So, we wind up with:

For Example :

Cursor c=db.rawQuery("SELECT salary FROM employee WHERE name= "sai_shivaji",null);

query() method Using for Fetch the Data from Table:

The query() method takes the discrete pieces of a SELECT statement andbuilds the query from them. The pieces, in order that they appear asparameters to query(), are:

- The name of the table to query against
- The list of columns to retrieve
- The WHERE clause, optionally including positional parameters
- The list of values to substitute in for those positional parameters
- The GROUP BY clause, if any
- The ORDER BY clause, if any
- The HAVING clause, if any

These can be null when they are not needed (except the table name, ofcourse). So, our previous snippet converts into:

For Example :

```
Cursor result=db.query("Employee", null, null, null, null, null, null);
```

Note : Above two methods return Cursor object.

Usage of Cursor

No matter how you execute the query, you get a Cursor back. This is the Android/ SQLite edition of the database cursor, a concept used in many database systems. With the cursor, you can:

- Find out how many rows are in the result set via getCount()
- Iterate over the rows via moveToFirst(), moveToNext(), and isAfterLast()
- Find out the names of the columns via getColumnNames(), convert those into column numbers via getColumnIndex(), and get values for the current row for a given column via methods like getString(), getInt(), etc.

- Re-execute the query that created the cursor via requery()
- Release the cursor's resources via close()

WHERE THIS FILE IS LOCATED AFTER RUN THITS POGRAM:

After started the Emulator it means running our project at that time this file will be appeared/generated.

Click Window

|......................> Open Perspective

|........................>Other

|...............> DDMS

|..............>File Explorer

Under this folders we can see.

/data/data/<packagename>/databases/dbfile.db.

Example :

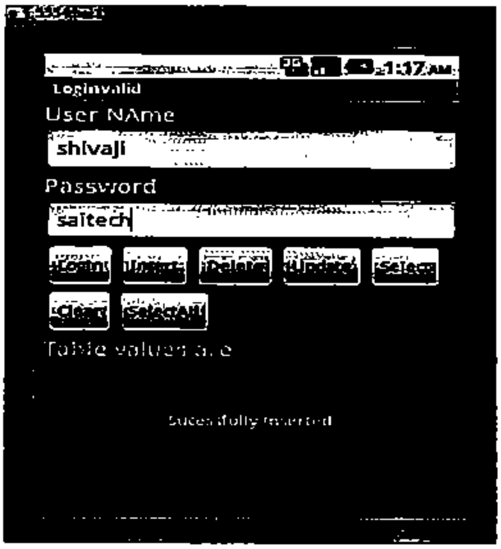
res/layout/main.xml :

Note : 1. By using this diagram you have to prepare GUI configuration in main.xml file.

2. The actual Business Logic only provided with this handout in activity class.

AndroidManifest.xml :

Note : you don't need to modify anything in this manifest.xml file. Leave this file with default configuration.



User NAme

shivaji

Password

saitech

Table values are

Sucessfully inserted

Src/in.co.sai /MainActivity :

```
public class Main extends Activity implements OnClickListener{
  /** Called when the activity is first created. */
  Button btnLogin, btninsert, btdelete, btclear, btupdate, btselect, Selall;
  EditText txtUserName, txtPassword;
  String username, password;
  TextView tv,tv1,tv2;
     public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);
..............................................................................
..............................................................................
```

You have to write a common code for getting the objects of all buttons by using findViewById(-.-.-);

And also apply listeners by using setOnClickListener(this);

```
..............................................................................
..............................................................................
}
        public void onClick(View v) {
               username = txtUserName.getText().toString();
                password = txtPassword.getText().toString();
               DBUserAdapter dbUser = new DBUserAdapter(Main.this);
         switch (v.getId()) {
        case R.id.clearbuton:
                       txtUserName.setText("");
                        txtPassword.setText("");
                       txtUserName.setFocusable(true);
               break;
        case R.id.insertbuton:
```

```
                                dbUser.open();
                               if(username.length()>0 || password.length()>0)
                               {
                                       if(dbUser.AddUser(username,password))
                                        {
        Toast.makeText(getApplicationContext(), "Sucessfully inserted",
       Toast.LENGTH_LONG).show();
                                  }else{
Toast.makeText(getApplicationContext(), "Record already
inserted",Toast.LENGTH_LONG).show();       }
                    }else{
        Toast.makeText(getApplicationContext(), "plz enter values",
Toast.LENGTH_LONG).show();     }
    break;
        case R.id.delbuton:
                try {
                                dbUser.open();
                               if(dbUser.delUser(username))
                               {
Toast.makeText(getApplicationContext(), "one record is
deleted",Toast.LENGTH_SHORT).show();
                               }else{
Toast.makeText(getApplicationContext(), "No elements found to
delete",Toast.LENGTH_SHORT).show();        }} catch (Exception e)
{Toast.makeText(getApplicationContext(),e.getMessage(),Toast.LENGTH_SHORT).show();
                }
        break;
       case R.id.updatebuton:
                try {
                               dbUser.open();
      if(dbUser.updateuser(username,password))
      {
         Toast.makeText(getApplicationContext(),"Sucessfully
Updated",Toast.LENGTH_SHORT).show();
      }          }catch (Exception e)
{Toast.makeText(getApplicationContext(),e.getMessage(),Toast.LENGTH_SHORT).show(); }

                                       break;
       case R.id.selbuton:
                try {
                        dbUser.open();
                        Cursor c = dbUser.selectuser(username);
                        StringBuilder sb=new StringBuilder();
                        txtPassword.setText(sb.append(c.getString(0)).toString());

                } catch (Exception e) {
Toast.makeText(getApplicationContext(),e.getMessage(),Toast.LENGTH_SHORT).show();
                }
                break;
        case R.id.logbuton:
                try{
        if(username.length() > 0 && password.length() >0)
        {
          dbUser.open();
          if(dbUser.Login(username, password))
          {
 Toast.makeText(Main.this,"Successfully Logged In", Toast.LENGTH_SHORT).show();
          }else{
Toast.makeText(Main.this,"Invalid Username/Password", Toast.LENGTH_SHORT).show();
          }          dbUser.close();
        } else{
Toast.makeText(Main.this,"Plz enter values", Toast.LENGTH_SHORT).show();
        }   }catch(Exception e)
     { Toast.makeText(Main.this,e.getMessage(), Toast.LENGTH_SHORT).show();      }
               break;
        case R.id.Allval:
               try {
                        dbUser.open();
                        StringBuilder sb=new StringBuilder();
                        StringBuilder sb1=new StringBuilder();
                        //List<String> list=new ArrayList<String>();
                       Cursor c=  dbUser.selectall();
                       tv1.setText("Num : Name");
                       if(c.moveToFirst())
                       {
                               do {

                               sb.append(c.getString(0).toString()).append("  :  ");
                               sb1.append(c.getString(1).toString()).append("\n");
                               tv.setText(sb);
                               tv2.setText(sb1);
                               } while (c.moveToNext());
                       }       } catch (Exception e) {
                      // TODO: handle exception
 Toast.makeText(getApplicationContext(), e.getMessage(), Toast.LENGTH_LONG).show();
                }
               break;
         }   }}
```

Src/in.co.sai /DBUserAdapter :

```
public class DBUserAdapter
{
  public static final String KEY_ROWID = "_id";
  public static final String KEY_USERNAME= "username";
  public static final String KEY_PASSWORD = "password";
  private static final String TAG = "DBAdapter";

  private static final String DATABASE_NAME = "usersdb":
   private static final String DATABASE_TABLE = "users";
   private static final int DATABASE_VERSION = 1;

   private static final String DATABASE_CREATE =
         "create table users (_id integer primary key autoincrement, " + "username text not null,
"+ "password text not null);";
   private Context context = null;
   private DatabaseHelper DBHelper;
   private SQLiteDatabase db;
   public DBUserAdapter(Context ctx)
   {
      this.context = ctx;
      DBHelper = new DatabaseHelper(context);
   }
   private static class DatabaseHelper extends SQLiteOpenHelper
   {
      DatabaseHelper(Context context)
      {
         super(context, DATABASE_NAME, null, DATABASE_VERSION);      }
      public void onCreate(SQLiteDatabase db)
      {
         db.execSQL(DATABASE_CREATE);     }
      public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
      {
         Log.w(TAG, "Upgrading database from version " + oldVersion
              + " to "
              + newVersion + ", which will destroy all old data");
         db.execSQL("DROP TABLE IF EXISTS users");
         onCreate(db);
      }   }
   public void open() throws SQLException
   {
      db = DBHelper.getWritableDatabase();     }
   public void close()
   {
      DBHelper.close();    }
   public boolean AddUser(String username, String password)
   {
         Cursor mCursor = db.rawQuery("SELECT * FROM " + DATABASE_TABLE + "
WHERE username=? ", new String[]{username});
      if (mCursor != null) {
         if(mCursor.getCount() > 0)
         {
            return false;           }       }
          ContentValues initialValues = new ContentValues();
       initialValues.put(KEY_USERNAME, username);
       initialValues.put(KEY_PASSWORD, password);
        db.insert(DATABASE_TABLE, null, initialValues);
        return true;       }
   public boolean Login(String username, String password) throws SQLException
   {
     Cursor mCursor = db.rawQuery("SELECT * FROM " + DATABASE_TABLE + "
WHERE username=? AND password=?", new String[]{username,password});
     if (mCursor != null) {
        if(mCursor.getCount() > 0)
        {
           return true;          }       }
    return false;
   }
   public boolean delUser(String username)
   {
        Cursor mCursor = db.rawQuery("SELECT * FROM " + DATABASE_TABLE + "
WHERE username=?", new String[]{username});
     if (mCursor != null) {
        if(mCursor.getCount() > 0)
        {
         db.delete(DATABASE_TABLE,KEY_USERNAME+"="+"'"+username+"'",null);
         return true;          }       }
      return false;  }
   public boolean updateuser(String username, String password)
   {
      ContentValues initialValues = new ContentValues();
       initialValues.put(KEY_PASSWORD, password);

db.update(DATABASE_TABLE,initialValues,KEY_USERNAME+"="+"'"+username+"'",null);
       return true;    }
   public Cursor selectuser(String username) throws SQLException
   {
        Cursor mCursor =
            db.query(true, DATABASE_TABLE, new String[] {
                     KEY_PASSWORD},
                     KEY_USERNAME + "=" + "'"+username+"'",   null.null,null, null,
                     null, null);
        if (mCursor != null) {
          mCursor.moveToFirst();
        }
       return mCursor;
     }
   public Cursor selectall()throws SQLException
   {
        Cursor mcur=db.rawQuery("select * from users", null);
     if(mcur!=null){
        mcur.moveToFirst();      } return mcur;    }}
```

Using SQLite3 Tool :

Open : C:\Users\Shivaji\Desktop\android-sdk\tools\SQLite3

Or

```
adb –s emulator-5554 shell
Sqlite3 data/data/in.co.shivaji/databases/usersdb
For Help :
>.help
```

Using SQLiteBrowser :

Note : in class room I ll explain that how to take generated database and open that database(table with data) in Sqlite browser and viceversa.

Content Provider

To query a content provider, you specify the query string in the form of a URI, with an optional specifier for a particular row. The format of the query URI is as follows:

<standard_prefix>://<authority>/<data_path>/<id>

The various parts of the URI are as follows:

- The *standard prefix* for content providers is always content://.
- The *authority* specifies the name of the content provider. An example would be contacts for the built-in Contacts content provider. For third-party content providers, this could be the fully qualified name, such as in.co.shivaji.
- The *data path* specifies the kind of data requested. For example, if you are getting all the contacts from the Contacts content provider, then the data path would be people, and the URI would look like this: content://contacts/people.
- The *id* specifies the specific record requested. For example, if you are looking for contact number 2 in the Contacts content provider, the URI would look like this: content:// contacts/people/2.

Query Strin	Description
content://media/internal/images	Returns a list of all the internal images on the device
content://media/external/images	Returns a list of all the images stored on the external storage (e.g., SD card) on the device
content://call_log/calls	Returns a list of all calls registered in the Call Log
content://browser/bookmarks	Returns a list of bookmarks stored in the browser

The managedQuery() method of the Activity class retrieves a managed cursor. A *managed cursor* handles all the work of unloading itself when the application pauses and requerying itself when the application restarts.

The statement

```
        Cursor c = managedQuery(allContacts, null, null, null, null);
```

is equivalent to

```
        Cursor c = getContentResolver().query(allContacts, null, null, null, null);
      startManagingCursor(c);
//---allows the activity to manage the Cursor's
// lifecyle based on the activity's lifecycle---
```

The getContentResolver() method returns a ContentResolver object, which helps to resolve a content

predefined Query String constants

Besides using the query URI, you can use a list of predefined query string constants in Android to specify the URI for the different data types.

For example, besides using the query content://contacts/people, you can rewrite the following statement

```
Uri allContacts = Uri.parse("content://contacts/people");
```

using one of the predefined constants in Android, as

```
Uri allContacts = ContactsContract.Contacts.CONTENT_URI;
```

Some examples of predefined query string constants are as follows:

- Browser.BOOKMARKS_URI
- Browser.SEARCHES_URI
- CallLog.CONTENT_URI
- MediaStore.Images.Media.INTERNAL_CONTENT_URI
- MediaStore.Images.Media.EXTERNAL_CONTENT_URI
- Settings.CONTENT_URI

If you want to retrieve the first contact, specify the ID of that contact, like this:

```
Uri oneContact = Uri.parse("content://contacts/people/1");
```

Alternatively, use the predefined constant together with the withAppendedId() method of the ContentUris class:

```
import android.content.ContentUris;

Uri oneContact = ContentUris. withAppendedId(ContactsContract.Contacts.CONTENT_URI, 1);
```

Example :

Interact with predefined ContentProvider(Contacts) :

```
public class MainActivity extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
  public void  listener(View v){
       switch (v.getId()) {
              case R.id.display:

                     displayContacts();
                           break;
              case R.id.insert:
                     insertContacts("saaitech","9493577797");
                     Toast.makeText(this, "inserted", Toast.LENGTH_LONG).show();
                           break;
              case R.id.delete:
                     deleteContacts();
                     Toast.makeText(this, "deleted", Toast.LENGTH_LONG).show();
                           break;
              default:
                     break;
              }  }
  private void displayContacts() {
    String columns[] = new String[] { ~~People~~.NAME, ~~People~~.NUMBER };
    Uri mContacts = ~~People~~.CONTENT_URI;
     Cursor cur = managedQuery(mContacts, columns, // Which columns to return
        null, // WHERE clause; which rows to return(all rows)
        null, // WHERE clause selection arguments (none)
        null // Order-by clause (ascending by name)
  );
    if (cur.moveToFirst()) {
      String name = null;
      String phoneNo = null;
      do {
        name = cur.getString(cur.getColumnIndex(~~People~~.NAME));
        phoneNo = cur.getString(cur.getColumnIndex(~~People~~.NUMBER));
        Toast.makeText(this, name + " " + phoneNo, Toast.LENGTH_LONG).show();
      } while (cur.moveToNext());
    } }

  private void insertContacts(String name, String phoneNo) {
       ContentValues values = new ContentValues();
    values.put(~~People~~.NAME, name);
    values.put(~~People~~.NUMBER, phoneNo);
    ContentResolver cr = getContentResolver();
    Uri uri = cr.insert(~~People~~.CONTENT_URI, values);
  }
  private void deleteContacts() {
    Uri uri = ~~People~~.CONTENT_URI;
    getContentResolver().delete(uri, "NAME=" + "'saitech'", null);
  }}
```

For Customised ContentProvider :

Step 1 :

In AndroidManifestFile we have to configure like this :

```
              <provider     android:name="DiaryContentProvider"
                            android:authorities="in.co.saitech" />
```

step 2:

on our class we need to extend the contentprovider class. And Let assume that we are creating table name is Employee.

Note : In Which class we are using this provider there we need to write this logic.

```
        String myUri = "content://in.co.saitech/Employee";
        Uri CONTENT_URI = Uri.parse(myUri);
        //get ContentResolver instance
                ContentResolver crInstance = getContentResolver();
                Cursor c = crInstance.query(CONTENT_URI, null, null, null, null);
                startManagingCursor(c);
        StringBuilder sb = new StringBuilder();
                if(c.moveToFirst()){
                        do{
                                sb.append(c.getString(1)).append("\n");
                        }while(c.moveToNext());
}tv.setText(sb.toString());
}}
```

Interview Questions on this topic :

37. SQLiteDatabase is database or not ? To create this database what is the software (server) we are using?
38. At most how much data we can store in SQLitedatabase ?
39. From which package we are getting all the classes for implementing SQLite database?
40. To create a database what is the constructor we are using and from which class it is?
41. Write a methods from SQLiteOpenHelper class?
42. How to create a table in SqliteDatabase?
43. What are the non abstract methods of SQLiteOpenHelper Class?
44. To perform the DML and DRL commands What is the class we are using?
45. Can we implement "joins concept" in SQLiteDatabase?
46. What are the constraints supports by SQLiteDatabase?
47. To store the image what is the datatype we are using?
48. What is the purpose of SQLiteBrowser ?
49. What is the use of ContentValues class and it is from which package?
50. What are the parameters we are passing in upadate()?
51. What is the use of Cursor class?
52. Can we insert the data without using ContentValues class?(with out using Browser)
53. What is the return types of query() and rawQuery() ?
54. What is the difference between query() and rawQuery()?
55. Once we run the application where we can see the database file ?
56. Can we create the database without using SQLiteOpenHelper class?
57. What is the use of execSQL() And from which class it is?
58. Write a methods from Cursor class?
59. What is the use of SQLite3 tool?
60. What is the purpose of onUpgrade() and onDowngrade()?

Location Based Services & Google Maps

Android provides a location framework that your application can use to determine the device's location and bearing and register for updates. A Google Maps external library is available that lets you display and manage Maps data. You can build these capabilities into your applications using the classes of the **"android.location"** package and the Google Maps external library.

Location Based Services :

In Android, location-based services are provided by the LocationManager class, located in the "android.location" package. Using the LocationManager class, your application can obtain periodic updates of the device's geographical locations, as well as fire an intent when it enters the proximity of a certain location.

- LocationManager : Class providing access to Android system location services.
- LocationListener : Interface for receiving notifications from the LocationManager when the location has changed.
- Location : Class representing a geographic location determined at a particular time.

The following steps are used for getting the location:

Step 1 :

The LocationManager needs to be initialized with the Android system service called LOCATION_SERVICE.This provides the application with the device's current location, movement and can also alert when the device enters or leaves a defined area.
An example of initialization is

```
LocationManager lm =LocationManager)getSystemService(Context.LOCATION_SERVICE);
```

Step 2 :

After the LocationManager instance is initiated, a location provider needs to be selected. Different location technologies might be available on the device (such as Assisted Global Positioning System (AGPS),Wi-Fi, and so on), and a general way to find a proper location provider is to define the accuracy and power requirement.

Note : This can be done using the Criteria class defined in android.location. Criteria . This enables the Android system to find the best available location technology for the specified requirements.

Process of selecting a location provider based on criteria is :

```
            Criteria criteria = new Criteria();
            criteria.setAccuracy(Criteria.ACCURACY_FINE);
            criteria.setPowerRequirement(Criteria.POWER_LOW);
      String locationprovider = lm.getBestProvider(criteria, true);
```

Note : It is also possible to specify the location estimation technology using the location manager's getProvider() method.

Note : The two most common providers are

1) The Satellite based Global Positioning System(GPS). That can be specified by LocationManager.GPS_PROVIDER.
2) Cell - tower identification . That can be specified by LocationManager. NETWORK_PROVIDER.

Step 3 :

permission to utilize location information needs to be granted in the AndroidManifest.xml file as follows.

I . For a more accurate location, such as GPS, add the ACCESS_FINE_LOCATION permission.

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

II . Otherwise (for Network), add the ACCESS_COARSE_LOCATION permission.

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

Step 4 : Retrieving Last Location :

Because it might take time to produce a location estimation, getLastKnownLocation() can be called to retrieve the location last saved for a given provider.The location contains a latitude, longitude, and Coordinated Universal Time (UTC) timestamp. Depending on the provider, information on altitude, speed, and bearing might also be included (usegetAltitude(), getSpeed(), and getBearing() on the Location object to retrieve these and getExtras() to retrieve satellite information).

```
Location mLocation = lm.getLastKnownLocation(locationprovider);
            double lat = mLocation.getLatitude()
            double longt = mLocation.getLongitude();
```

Providing Mock Location Data

As you develop your application, you'll certainly need to test how well your model for obtaining user location works. This is most easily done using a real Android-powered device. If, however, you don't have a device, you can still test your location-based features by mocking location data in the Android emulator. There are three different ways to send your application mock location data: using Eclipse, DDMS, or the "geo" command in the emulator console.

Note: Providing mock location data is injected as GPS location data, so you must request location updates from GPS_PROVIDER in order for mock location data to work.

Using Eclipse

Select Window > Show View > Other > Emulator Control.

In the Emulator Control panel, enter GPS coordinates under Location Controls as individual lat/long coordinates, with a GPX file for route playback, or a KML file for multiple place marks. (Be sure that you have a device selected in the Devices panel—available from Window > Show View > Other > Devices.)

Using DDMS

With the DDMS tool, you can simulate location data a few different ways:

- Manually send individual longitude/latitude coordinates to the device.
- Use a GPX file describing a route for playback to the device.
- Use a KML file describing individual place marks for sequenced playback to the device.

Using the "geo" command in the emulator console

To send mock location data from the command line:

1. Launch your application in the Android emulator and open a terminal/ console in your SDK's /tools directory.
2. Connect to the emulator console:

```
telnet localhost <console-port>
```

Send the location data:

geo fix to send a fixed geo-location.

This command accepts a longitude and latitude in decimal degrees, and an optional altitude in meters.

For example:

```
Geo fix 9493 5777 97
```

Updating Location When we change :

The **LocationListener** *interface* is used to receive notifications when the location has changed.The location manager's **requestLocationUpdates()** method needs to be called after a location provider is initialized to specify when the current activity is to be notified of changes.

In the MainActivity.java file, you first obtain a reference to the LocationManager class using the getSystemService() method. To be notified whenever there is a change in location, you need to register a request for location changes so that your program can be notified periodically. This is done via the requestLocationUpdates() method:

lm.requestLocationUpdates(LocationManager.GPS_PROVIDER,0,0,locationListener);

This method takes four parameters:

- provider — The name of the provider with which you register. In this case, you are using GPS to obtain your geographical location data.
- minTime — The minimum time interval for notifications, in milliseconds
- minDistance — The minimum distance interval for notifications, in meters
- listener — An object whose onLocationChanged() method will be called for each location update

Note : The MyLocationListener class implements the LocationListener abstract class. You need to override. Four methods in this implementation:

- onLocationChanged(Location location) : Called when the location has changed.
- onProviderDisabled(String provider) : Called when the provider is disabled by the user.
- onProviderEnabled(String provider) : Called when the provider is enabled by the user.
- onStatusChanged(String provider, int status, Bundle extras) : Called when Provider status changes

Example :

```
    public class MyLocationListener extends Activity implements LocationListener {
            LocationManager lm;
    onCreate(){
            setContentView(R.layout.main);
            lm = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
lm.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,0,0,locationListener);
}
public void onLocationChanged(Location location) {
                double lat = location.getLatitude();
                double longt = location. getLongitude()
Toast.makeText(getApplicationContext(),String.ValueOf(lat)+String.ValueOf(longt),Toast.LENGTH_SHORT).show();
}
// these methods are required
public void onProviderDisabled(String arg0) {}
public void onProviderEnabled(String provider) {}
public void onStatusChanged(String a, int b, Bundle c) {}
}
```

Manifest.xml :

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

Usage of GeoCoder :

The Geocoder class provides a method to translate from an address into a latitude longitude coordinate (geocoding) and from a latitude-longitude coordinate into an address (reverse geocoding). Reverse geocoding might produce only a partial address, such as city and postal code, depending on the level of detail available to the location provider.

Translating a Location to Address (Reverse Geocoding) :

if you know the latitude and longitude of a location, you can find out its address using a process known as **reverse geocoding**. Google Maps in Android supports this via the Geocoder class. the getFromLocation() method provides a list of addresses associated with the area around the provided location. Here the maximum number of returned results is set to one (for instance, the most likely address). The Geocoder returns a List of android.location.Address objects. This translation to an address depends on a backend service that is not included in the core Android framework.

The following code shows ReverseGeocoding:

```
        List<Address> addresses;
try {
                Geocoder GC = new Geocoder(this, Locale.ENGLISH);
                Addresses = GC.getFromLocation(mLocation.getLatitude(),
                mLocation.getLongitude(), 1);
        if(addresses != null) {
                Address currentAddr = addresses.get(0);
                StringBuilder  mSB = new StringBuilder("Address:\n");
        for(int i=0; i<currentAddr.getMaxAddressLineIndex(); i++) {
                mSB.append(currentAddr.getAddressLine(i)).append("\n");
        Toast.makeText(getApplicationContext(),mSB,Toast.LENGTH_SHORT).show();
} catch(IOException e) {}
```

Translating an Address to Location (Geocoding) :

If you know the address of a location but want to know its latitude and longitude, you can do so via geocoding. Again, you can use the Geocoder class for this purpose. The following code shows how you can find the exact location of the Empire State Building by using the getFromLocationName() method:

```
        List<Address> addresses;
                String myAddress="shivaji, Vishakapatnam,andhrapradesh";
        Geocoder gc = new Geocoder(this);
        try {
        addresses = gc.getFromLocationName(myAddress, 1);
        if(addresses != null) {
                Address x = addresses.get(0);
        StringBuilder mSB = new StringBuilder("Address:\n");
                mSB.append("latitude: ").append(x.getLatitude());
                mSB.append("\nlongitude: ").append(x.getLongitude());
        Toast.makeText(getApplicationContext(),mSB,Toast.LENGTH_SHORT).show();
}
} catch(IOException e) {}
```

Google map Display in our Application :

Google maps can be used on the Android system in two ways: user access through a browser and application access through the Google Maps Application Programming Interface (API).The MapView class is a wrapper around the Google Maps API. To use MapView, the following setup is needed:

Step 1 :

Download and install the Google API's Software Development Kit (SDK):

- Use the Android SDK and Android Virtual Device (AVD) manager in Eclipse to download the Google API.

Step 2 :

- To integrate with google maps to our application then we need to apply for free google Map API Key.

- To get the google Map API Key we need to sign up for the android Maps API.
- To sign up , we need to provide the Certificate's Fingerprint .

Step 3 : Get Certficate FingerPrint(MD5)

- We need to get the keystore file for getting the MD5 as follows :

  " C : \ Documents and Settings \ <User Name> \ .android \ debug.keystore"

- Just copy that file into some other location (Friendly to use,not necessary).

  For Example I placed this file in to " D : \ sai\ debug.keystore "

Step 4 :

- Open command prompt then go to java insatalled directory .

  C : \ Program Files \ java \ <jdk-version-name> \ bin\

- Then Type the below command .

```
keytool.exe -list -alias androiddebugkey -keystore
        "D:\sai\ debug.keystore" -storepass android -keypass android
```

- After this we ll be getting the MD5 Key

For Example : 74:20:41:4A:EB:35:34:6B:B3:BB:DF:9A:9B:D8:FC:DA

Step 5 :

- In Browser type this URL for generating Google Map Api Key.

  http://code.google.com/android/add-ons/google-apis/mapkey.html

- Copy the MD5 key what we generated in 4th step.

My Certificate MD5 Fingerprint : 74:20:41:4A:EB:35:34:6B:B3:BB:DF:9A:9B:D8:FC:DA

- Click on the button " generate API Key "

Step 6:

We got this code with apiKey:

```
<com.google.android.maps.MapView
                android:id="@+id/map1"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent"
                android:clickable="true"
                android:apiKey="0ZDUMMY13442HjX491CODE44MSsJzDVlIQ"
        />
```

Sample Program For Google Maps :

res/layout/main.xml

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

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest       xmlns:android="http://schemas.android.com/apk/res/android"
                package="in.co.sai"
                android:versionCode="1"
                android:versionName="1.0">
<application android:icon="@drawable/icon" android:label="@string/app_name">
<uses-library android:name="com.google.android.maps" />

<activity       android:name=".MainActivity"
                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>
<uses-sdk android:minSdkVersion="8" />
<uses-permission android:name="android.permission.INTERNET"></uses-permission>

</manifest>
```

Src /MainActivity

Note : Add the following statements in bold to the MainActivity.java file. Note that MainActivity is now extending the MapActivity class.

```
package  in.co.sai;
import android.app.Activity;
import android.os.Bundle;
import com.google.android.maps.MapActivity;
public class MainActivity extends MapActivity {
        public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        MapView mapView = (MapView) findViewById(R.id.shivaji_mapView);
        mapView.setBuiltInZoomControls(true);
        mapView.setSatellite(true);
        //mapView.setStreetView(true);
}
protected boolean isRouteDisplayed() {
return false;}}
```

InterVIew Questions on this topic :

Location Based Service and Google maps :

1. **What is the use of LocationManager class? And how to create object for this class?**
2. **All are the classes(Location Based Classes) from which package?**
3. **What is the difference between getProvider () and getBestProvider()?**
4. **What is the use of getLastKnownLocation() method?**
5. **What is the use of requestLocationUpdates() from which class it is?**
6. **What are the parameters we are passing requestLocationUpdates()?**
7. **Write a methods of LocationListener?**
8. **What is the difference between GPS provider and N/W provider?**
9. **What is the permission configure for GPS and N/w Provider?**

10. **What is the use of criteria class?**
11. **What is the purpose of Geocoder class?**
12. **What is the difference between Geocoding and ReverseGeocoding?**
13. **Write a four methods of Address class?**
14. **What is the difference between getFromLocation() and getFromLocationName()?**
15. **How to retrieve the latitude and longitude values?**
16. **What is the third party library for googlemaps integration?**
17. **What is the use of MD5 certificate key?**
18. **What is the use of apiKey ?**
19. **How to add the default zooms to google maps?**
20. **How many ways we can pass the mock data for latitude and longitude values?**
21. **How many ways we can generate the apiKey?**
22. **How to pass the latitude and longitude values to google map?**
23. **What is the use of MapController class and it is from which package?**
24. **How to add markers on google map? And what are the classes we are using for that?**
25. **What is the use of onTap() and from which class it is?**
26. **Write a lifecycle methods of MapActivity class?**

MediaPlayer

MediaPlayer/MediaRecorder : This is the standard method to manipulate audio, but must be file- or stream-based data. Creates its own thread for processing. SoundPool utilizes this framework.

The MediaRecorder and MediaPlayer classes are used to record and play back either audio or video.This recipe focuses on audio, and the usage is straightforward. For playback,the steps are

step 1 : Create an instance of the MediaPlayer.

```
MediaPlayer mp = new MediaPlayer();
```

step 2 : Specify the source of media. It can be created from a raw resource:

```
mp = MediaPlayer.create(this, R.raw.my_music);
```

Another option is to set as a file from the filesystem (which then also needs a prepare statement).

```
mp.setDataSource(path);
mp.prepare();
```

In any case, these statements need to be surrounded by a try-catch block because the specified resource might not exist.

3. Start playback of the audio.

```
mp.start();
```

4. When the playback is done, stop the MediaPlayer and release the instance to free up resources.

```
mp.stop();
mp.release();
```

Example :

```
Context appContext = getApplicationContext();
```

From RAW Folder :

```
MediaPlayer resourcePlayer = 
MediaPlayer.create(appContext,R.raw.my_shivaji_audio);
```

From SD card :

```
MediaPlayer filePlayer = MediaPlayer.create(appContext,
                                  Uri.parse("file:///sdcard/localfile.mp3"));
```

From Internet :

```
MediaPlayer urlPlayer = MediaPlayer.create(appContext,
                                  Uri.parse("http://site.com/audio/audio.mp3"));
```

Video Player Playing :

```
VideoView videoView = (VideoView)findViewById(R.id.surface);
videoView.setKeepScreenOn(true);
videoView.setVideoPath("/sdcard/test2.3gp");
if (videoView.canSeekForward())
        videoView.seekTo(videoView.getDuration()/2);
        videoView.start();
videoView.stopPlayback();
```

Camera :

There are two ways to access the camera from an application. The implicit intent launches the default camera interface.

```
Intent intent = new Intent("android.media.action.IMAGE_CAPTURE");
        startActivity(intent);
```

The second way leverages the Camera class, which provides more flexibility in the settings.This creates a custom camera interface, which is the focus of the examples that follow. Camera hardware access requires explicit permission in the AndroidManifest XML file:

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

Camera class : Accesses the camera hardware.

Camera.Parameters class : Specifies the camera parameters such as picture size, picture quality, flash modes, and method to assign Global Positioning System (GPS) location.

Camera Preview methods : Sets the camera output display and toggles streaming video preview to the display.

SurfaceView class : Dedicates a drawing surface at the lowest level of the view hierarchy as a placeholder to display the camera preview Before describing how these are tied together, the layout structure is introduced.

res/layout/main.xml

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

```
</LinearLayout>
res/layout/cameraoverlay.xml  :
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:orientation="vertical"
            android:gravity="bottom"
            android:layout_gravity="bottom">
<LinearLayout
            xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal"
            android:gravity="center_horizontal">
<Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="take picture"/>
            </LinearLayout>
</LinearLayout>
```

The main activity involves multiple functionalities. First, the layout is set up as follows:

1. The window settings are changed to be translucent and full screen. (In this instance,they hide the title and notification bar.)

2. The SurfaceView defined in the previous layout (R.id.surface) is then filled by the camera preview. Each SurfaceView contains a SurfaceHolder for access and control over the surface.The activity is added as the SurfaceHolder's callback and the type is set to SURFACE_TYPE_PUSH_BUFFERS, which means it creates a "push" surface and the object does not own the buffer.This makes video streaming more efficient.

3. A LayoutInflater is declared to inflate another layout (cameraoverlay.xml) over the original (main.xml) layout.

Next, the activity sets a trigger for taking a picture:

1. An OnClickListener is added on the button from the cameraoverlay layout, so when clicked, it takes a picture (mCamera.takePicture()).

2. The takePicture() method needs three methods to be defined:.

ShutterCallback() to define any effects needed after the picture is taken, such as a sound to let the user know that picture has been captured.

A PictureCallback() for raw picture data if hardware has enough memory to support this feature. (Otherwise. the data might return as null.)

A second PictureCallback() for the compressed picture data.This calls the local method done() to save the picture.

Then, the activity saves any pictures that were taken:

1. The compressed picture byte array is saved to a local variable tempdata for manipulation.The BitmapFactory is used to decode the ByteArray into a Bitmap Object.
2. The media content provider is used to save the bitmap and return a URL. If this main activity were called by another activity, this URL would be the return information to the caller activity to retrieve the image.
3. After this process, finish() is called to kill the activity.

Finally, the activity sets up a response to a change in the surface view:

1. A SurfaceHolder.CallBack interface is implemented. This requires three methods to be overridden:

surfaceCreated() : Called when the surface is first created. Initialize objects here.

surfaceChanged() : Called after surface creation and when the surface changes (for example, format or size).

surfaceDestroyed() : Called between removing the surface from the view of the user and destroying the surface.This is used for memory cleanup.

2. The parameters for the camera are changed when the surface is changed (such as the PreviewSize based on the surface size).

src/in/co/shivaji/hardware/CameraApplication.java

```
package in.co.shivaji;
import android.hardware.Camera;
import android.hardware.Camera.PictureCallback;
import android.hardware.Camera.ShutterCallback;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
public class CameraApplication extends Activity implements SurfaceHolder.Callback {
        private static final String TAG = "cookbook.hardware";
        private LayoutInflater mInflater = null;
        Camera mCamera;
        byte[] tempdata;
        boolean mPreviewRunning = false;
        private SurfaceHolder mSurfaceHolder;
        private SurfaceView mSurfaceView;
        Button takepicture;
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                            WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setContentView(R.layout.main);
mSurfaceView = (SurfaceView)findViewById(R.id.surface);
mSurfaceHolder = mSurfaceView.getHolder();
mSurfaceHolder.addCallback(this);
mSurfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
        mInflater = LayoutInflater.from(this);
        View overView = mInflater.inflate(R.layout.cameraoverlay, null);
        this.addContentView(overView,
        new LayoutParams(LayoutParams.FILL_PARENT,LayoutParams.FILL_PARENT));
        takepicture = (Button) findViewById(R.id.button);
        takepicture.setOnClickListener(new OnClickListener(){
public void onClick(View view){
        mCamera.takePicture(mShutterCallback,mPictureCallback, mjpeg);
}
});
}
ShutterCallback mShutterCallback = new ShutterCallback(){
@Override
public void onShutter() {}
};
PictureCallback mPictureCallback = new PictureCallback() {
public void onPictureTaken(byte[] data, Camera c) {}
};
PictureCallback mjpeg = new PictureCallback() {
public void onPictureTaken(byte[] data, Camera c) {
if(data !=null) {
tempdata=data;
done();
}}};
void done() {
        Bitmap bm = BitmapFactory.decodeByteArray(tempdata,0, tempdata.length);
        String url = Images.Media.insertImage(getContentResolver(),bm, null, null);
        bm.recycle();
        Bundle bundle = new Bundle();
                if(url!=null) {
                        bundle.putString("url", url);
                        Intent mIntent = new Intent();
                        mIntent.putExtras(bundle);
                setResult(RESULT_OK, mIntent);
        } else {
        Toast.makeText(this, "Picture can not be saved",Toast.LENGTH_SHORT).show();
}
finish();
}
@Override
public void surfaceChanged(SurfaceHolder holder, int format,int w, int h) {
Log.e(TAG, "surfaceChanged");
        try {
        if (mPreviewRunning) {
                mCamera.stopPreview();
                mPreviewRunning = false;
        }
Camera.Parameters p = mCamera.getParameters();
        p.setPreviewSize(w, h);
        mCamera.setParameters(p);
        mCamera.setPreviewDisplay(holder);
        mCamera.startPreview();
        mPreviewRunning = true;
} catch(Exception e) {
Log.d("",e.toString());
}}
@Override
public void surfaceCreated(SurfaceHolder holder) {
Log.e(TAG, "surfaceCreated");
mCamera = Camera.open();
}
@Override
public void surfaceDestroyed(SurfaceHolder holder) {
Log.e(TAG, "surfaceDestroyed");
        mCamera.stopPreview();
        mPreviewRunning = false;
        mCamera.release();
        mCamera=null;
}}
```

Bluetooth

Bluetooth from the IEEE standard 802.15.1 is an open, wireless protocol for exchanging data between devices over short distances.A common example is from a phone to a headset, but other applications can include proximity tracking.To communicate between devices using Bluetooth, four steps need to be performed:

1. Turn on Bluetooth for the device.
2. Find paired or available devices in a valid range.
3. Connect to devices.
4. Transfer data between devices.

To use the Bluetooth Service, the application needs to have BLUETOOTH permission to receive and transmit and BLUETOOTH_ADMIN permission to manipulate Bluetooth settings or initiate device discovery.These require the following lines in the AndroidManifest XML file:

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

All the Bluetooth API functionality resides in the android.bluetooth package.There are five main classes that provide the features:

BluetoothAdapter : Represents the Bluetooth radio interface that is used to discover devices and instantiate Bluetooth connections.
BluetoothClass : Describes the general characteristics of the Bluetooth device
BluetoothDevice : Represents a remote Bluetooth device
BluetoothSocket : Represents the socket or connection point for data exchange with another Bluetooth device
BluetoothServerSocket : Represents an open socket listening for incoming requests

Turning on Bluetooth

Bluetooth is initialized using the BluetoothAdapter class.The getDefaultAdapter() method retrieves information about the Bluetooth radio interface. If null is returned, it means the device does not support Bluetooth:

```
BluetoothAdapter myBluetooth = BluetoothAdapter.getDefaultAdapter();
```

Activate Bluetooth using this BluetoothAdapter instance to query the status. If not enabled, the Android built-in activity ACTION_REQUEST_ENABLE can be used to ask the user to start Bluetooth:

```
if(!myBluetooth.isEnabled()) {
            Intent enableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
                    startActivity(enableIntent);
}
```

Discovering Bluetooth Devices :

After Bluetooth is activated, to discover paired or available Bluetooth devices, use the BluetoothAdapter instance's startdiscovery() method as an asynchronous call.This requires registering a BroadcastReceiver to listen for ACTION_FOUND events that tell the application whenever a new remote Bluetooth device is discovered.

```
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent) {
                String action = intent.getAction();
          // When discovery finds a device
  if (BluetoothDevice.ACTION_FOUND.equals(action)) {
       // Get the BluetoothDevice object from the Intent
BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
Log.v("BlueTooth Testing",device.getName() + "\n"+ device.getAddress());
}}
};
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
registerReceiver(mReceiver, filter);
myBluetooth.startDiscovery();
```

The broadcast receiver can also listen for ACTION_DISCOVERY_STARTED events and ACTION_DISCOVERY_FINISHED events that tell the application when the discovery starts and ends. For other Bluetooth devices to discover the current device, the application can enable discoverability using the ACTION_REQUEST_DISCOVERABLE intent.This activity displays another dialog on top of the application to ask users whether or not they want to make the current device discoverable:

```
Intent discoverableIntent
= new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
         startActivity(discoverableIntent);
```

Pairing with Bonded Bluetooth Devices

Bonded Bluetooth devices are those that have already paired with the current devices sometime in the past.When pairing two Bluetooth devices, one connects as a server and the other as the client using the BluetoothSocket and BluetoothServerSocket classes.

To get the bonded Bluetooth devices, the BluetoothAdapter instance's method getBondedDevices() can be used:

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
```

Opening a Bluetooth Socket :

To establish a Bluetooth connection with another device, the application needs to implement either the client-side or server-side socket. After the server and client are bonded, there is a connected Bluetooth socket for each device on the same RFCOMM (Bluetooth transport protocol). However, the client device and service device obtain the Bluetooth socket in different ways.The server receives the Bluetooth socket instance when an incoming connection is accepted.The client receives the instance when it opens an RFCOMM channel to the server.

Server-side initialization uses the generic client-server programming model with applications requiring an open socket for accepting incoming requests (similar to TCP).The interface BluetoothServerSocket should be used to create a server listening port. After the connection is accepted, a BluetoothSocket is returned and can be used to manage the connection.

The BluetoothServerSocket can be obtained from the BluetoothAdapter instance's method listenUsingRfcommWithServiceRecord().After obtaining the socket, the accept() method starts listening for a request and returns only when either a connection has been accepted or an exception has occurred.The BluetoothSocket then returns when accept() returns a valid connection. Finally, the close() method should be called to release the server socket and its resources because RFCOMM allows only one connected client per channel at a time.This does not close the connected BluetoothSocket.

The following excerpt shows how these steps are done:

```
BluetoothServerSocket myServerSocket = myBluetoothAdapter.listenUsingRfcommWithServiceRecord(name, uuid);
                myServerSocket.accept();
                myServerSocket.close();
```

Animation

In Android we have two types of animation . They are

- frame-by-frame
- Tween animation

Frame-by-frame animation :

shows a sequence of pictures in order. It enables developers to define the pictures to display, and then show them like a slideshow. Frame-by-frame animation first needs an animation-list element in the layout file containing a list of item elements specifying an ordered list of the different pictures to display.The oneshot attribute specifies whether the animation is played only once or repeatedly.

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android=http://schemas.android.com/apk/res/android
android:oneshot="false">
            <item android:drawable="@drawable/sai1" android:duration="200" />
            <item android:drawable="@drawable/sai2" android:duration="200" />
            <item android:drawable="@drawable/sai3" android:duration="200" />
</animation-list>
```

To display the frame-by-frame animation, set the animation to a view's background:
In Activity class :

```
    ImageView im = (ImageView) this.findViewById(R.id.image);
    im.setBackgroundResource(R.anim.animated);
    AnimationDrawable ad = (AnimationDrawable)im.getBackground();
    ad.start();
```

After the view background is set, a drawable can be retrieved by calling getBackground() and casting it to AnimationDrawable.Then, calling the start() method starts the animation.

Tween animation :

Tween animation uses a different approach that creates an animation by performing a series of transformations on a single image. In Android, it provides access to the following classes that are the basis for all the animations:

✓ AlphaAnimation	:	Controls transparency changes.
✓ RotateAnimation	:	Controls rotations.

- ✓ ScaleAnimation : Controls growing or shrinking.
- ✓ TranslateAnimation : Controls position changes.

These four Animation classes can be used for transitions between activities, layouts, views and so on.All these can be defined in the layout XML file as <alpha>, <rotate>, <scale>,and <translate>.They all have to be contained within an AnimationSet <set>:

<alpha> attributes:

android:fromAlpha, android:toAlpha

Note : The alpha value translates the opacity from 0.0 (transparent) to 1.0 (opaque).

<rotate> attributes:

android:fromDegrees, android:toDegrees,
android:pivotX, android:pivotY

Note : The rotate specifies the angle to rotate an animation around a center of rotation defined as the pivot.

<scale> attributes:

android:fromXScale, android:toXScale,
android:fromYScale, android:toYScale,
android:pivotX, android:pivotY

Note : The scale specifies how to change the size of a view in the x-axis or y-axis.The pivot location that stays fixed under the scaling can also be specified.

<translate> attributes:

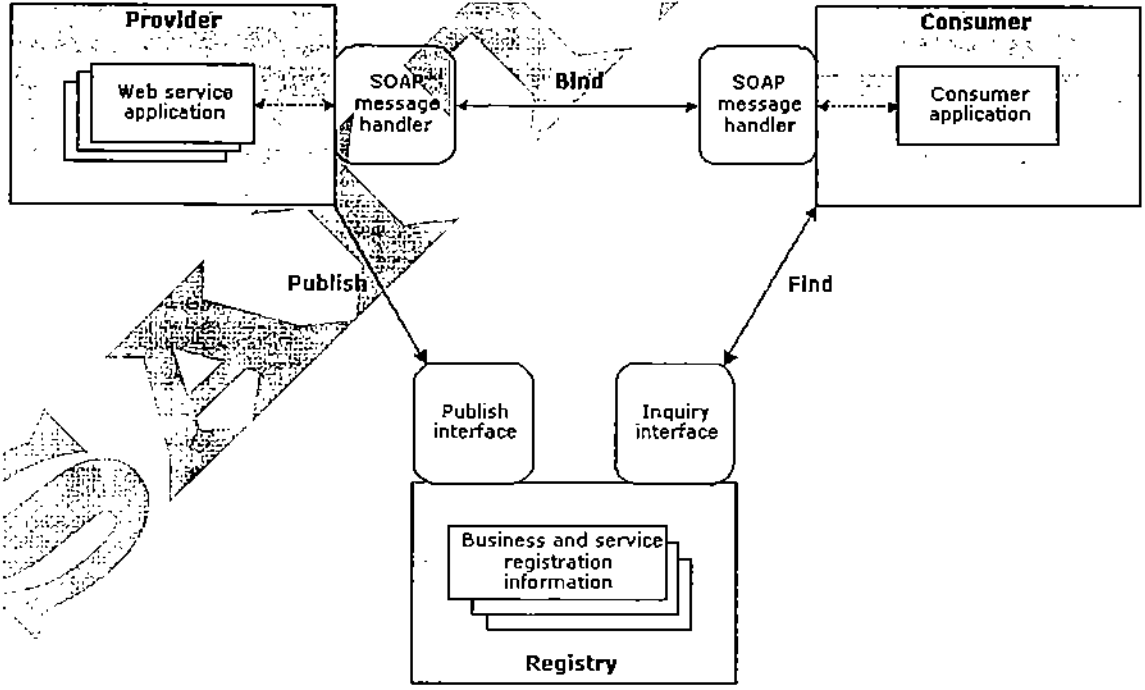
android:fromXDelta, android:toXDelta,
android:fromYDelta, android:toYDelta

Note : The translate specifies the amount of translation to perform on a View.

Web Services

Web Services are modular software components whose application functionality is accessible over the Web using Simple Object Access Protocol (SOAP), a standardized XML-based messaging protocol.Applications invoke Web Services like remote procedure calls, except that the procedure call and response are handled using SOAP messages embedded in HTTP requests and responses. An application calls a Web Service by sending a SOAP message embedded in an HTTP request to a Web location associated with that service. The Web Service performs the application logic for that message and then returns any application output in the form of another SOAP message embedded in an HTTP response.

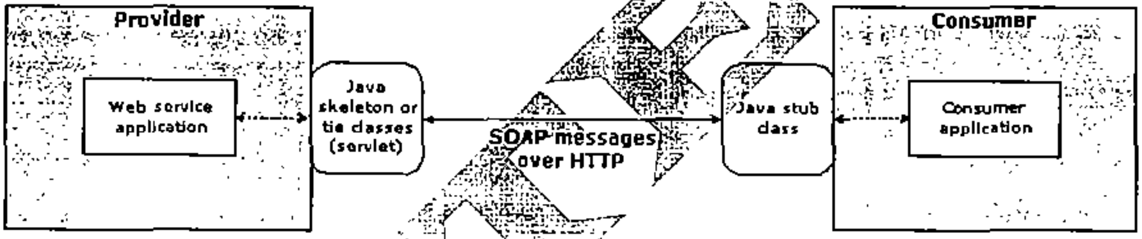
Arcitecture of Web Service :



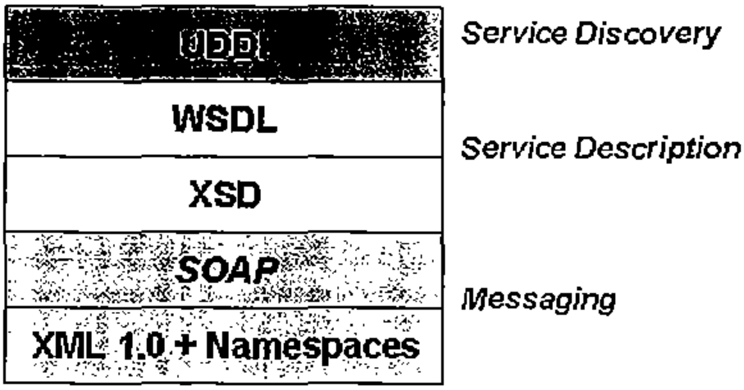
Note : As android developer we have to write consumer application.

Consumers must discover all of the details described above before they can interact with a Web service. The **Web Services Description Language (WSDL)** provides an XML grammar for describing these details. WSDL picks up where XML Schema left off by providing a way to group messages into operations and operations into interfaces. It also provides a way to define bindings for each interface and protocol combination along with the endpoint address for each one. A complete WSDL definition contains all of the information necessary to invoke a Web service. Developers that want to make it easy for others to access their services should make WSDL definitions available.

WSDL plays an important role in the overall Web services architecture since it describes the complete contract for application communication (similar to the role of IDL in the DCOM architecture). Although other techniques exist for describing Web services, the **WS-I Basic Profile Version 1.0** mandates the use of WSDL and XML Schema (see Figure 4) for describing Web services. This helps ensure interoperability at the service description layer.



Simple WebService architecture



WSDL file Structure :

```
<!-- WSDL definition structure -->
```

```
<definitions
    name="MathService"
targetNamespace="http://example.org/math/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
>
   <!-- abstract definitions -->
    <types> ...

    <message> ...
   <portType> ...

   <!-- concrete definitions -->
   <binding> ...
   <service> ...
</definition>
```

Notice that you must specify a target namespace for your WSDL definition, just like you would for an XML Schema definition. Anything that you name in the WSDL definition (like a message, portType, binding, etc.) automatically becomes part of the WSDL definition's target namespace defined by the **targetNamespace** attribute. Hence, when you reference something by name in your WSDL file, you must remember to use a qualified name. The first three elements (**types, message, and portType**) are all abstract definitions of the Web service interface.

These elements constitute the programmatic interface that you typically interface with in your code. The last two elements (**binding** and **service**) describe the concrete details of how the abstract interface maps to messages on the wire. These details are typically handled by the underlying infrastructure, not by your application code. Table 1 provides brief definitions for each of these core WSDL elements and the remaining sections discuss them in more detail.

WSDL Elements

Element Name	**Description**
sTypes	A container for abstract type definitions defined using XML Schema
Message	A definition of an abstract message that may consist of multiple parts, each part may be of a different type
portType	An abstract set of operations supported by one or more endpoints (commonly known as an interface); operations are defined by an exchange of messages
Binding	A concrete protocol and data format specification for a particular portType

Service A collection of related endpoints, where an endpoint is defined as a combination of a binding and an address (URI)