Introduction to Android Programming

Instructor: Randeep Bhatia
AGENDA

1. Android Basics
2. Eclipse Demo
3. Programming Basics
4. Networking (APIs, C2DM)
Android

- Mobile OS
  - Linux Kernel
  - Open Source (OHA/Google)

- Programming Environment
  - SDK -- compiler, debugger, device emulator
    - Multiplatform dev. support – Windows, Linux, Mac
  - Java Programming: has its own JVM (Dalvik VM) and special bytecode
Architecture
Android Development Process

- Setup Dev. Environment (JDK, SDK, Eclipse...)

- Create app.
  - Android Project containing java files + resource files

- Test app.
  - Pack project into debuggable *.apk
  - Install, run and debug on device or emulator

- Publish app. in Android Market

- Get Rich!
Setup SDK within Eclipse (in Windows)

1. Download and Install
   - JDK
   - Eclipse

2. Install and configure Android SDK plugin in Eclipse
   2. It will prompt to install the Android SDK
   3. Use Android SDK Manager to install specific versions of Android
## Android SDK Manager

**SDK Path:** D:\install\android-sdk_26-windows\android-sdk-windows\nn

### Packages

<table>
<thead>
<tr>
<th>Name</th>
<th>API</th>
<th>Rev.</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tools</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Android SDK Tools</td>
<td>16</td>
<td>1</td>
<td>Installed</td>
</tr>
<tr>
<td>Android SDK Platform-tools</td>
<td>10</td>
<td>1</td>
<td>Installed</td>
</tr>
<tr>
<td><strong>Android 4.0.3 (API 15)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation for Android SDK</td>
<td>15</td>
<td>1</td>
<td>Installed</td>
</tr>
<tr>
<td>SDK Platform</td>
<td>15</td>
<td>1</td>
<td>Installed</td>
</tr>
<tr>
<td>Samples for SDK</td>
<td>15</td>
<td>1</td>
<td>Installed</td>
</tr>
<tr>
<td>ARM EABI v7a System Image</td>
<td>15</td>
<td>1</td>
<td>Installed</td>
</tr>
<tr>
<td>Google APIs by Google Inc.</td>
<td>15</td>
<td>1</td>
<td>Not installed</td>
</tr>
<tr>
<td>Sources For Android SDK</td>
<td>15</td>
<td>1</td>
<td>Not installed</td>
</tr>
<tr>
<td><strong>Android 4.0 (API 14)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDK Platform</td>
<td>14</td>
<td>3</td>
<td>Installed</td>
</tr>
<tr>
<td>Samples for SDK</td>
<td>14</td>
<td>2</td>
<td>Installed</td>
</tr>
<tr>
<td>ARM EABI v7a System Image</td>
<td>14</td>
<td>2</td>
<td>Not installed</td>
</tr>
<tr>
<td>Google APIs by Google Inc.</td>
<td>14</td>
<td>2</td>
<td>Not installed</td>
</tr>
<tr>
<td>Sources For Android SDK</td>
<td>14</td>
<td>1</td>
<td>Not installed</td>
</tr>
</tbody>
</table>

**Show:** Updates, New, Installed, Obsolete  Select New or Updates

**Sort by:** API level, Repository  Deselect All

**Done loading packages.**
Option 1: Use an Emulator

Create an AVD (Android Virtual Device)
- Lets you specify the configuration of a device to be emulated by the Android Emulator.
- Create AVD In Eclipse by selecting Window > AVD Manager.
Option 2: Use a device

- Install drivers for device
- Connect device to PC via USB cable
  - Make sure turned on USB debugging (Settings → Application → Development)
  - Also turn on install of non market Apps (Settings → Application → Unknown Sources)
- Device will be recognized within Eclipse (DDMS view)
Continue after Eclipse Demo
Android App

- Runs in its own Virtual Machine & process
  - Isolation among apps
- Typically an app cannot directly access other apps data
- Is composed of basic “components”
- App components can be activated individually
  - Android starts the app process when any of its component needs to be executed
# Android App Components

<table>
<thead>
<tr>
<th>Basic Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>Deals with UI aspects. Typically corresponds to a single screen</td>
</tr>
<tr>
<td>Service</td>
<td>Background tasks (e.g. play music in background while user is web surfing) that typically have no UI.</td>
</tr>
<tr>
<td>BroadcastReceiver</td>
<td>Can receive messages (e.g. “Low Battery”) from system/apps and act upon them.</td>
</tr>
<tr>
<td>ContentProvider</td>
<td>Provide an interface to app data. Lets apps share data with each other</td>
</tr>
</tbody>
</table>
Activities

- UI portion of an App
- One activity typically corresponds to a single screen of an app (but can also be faceless)
- Conceptually laid out as a stack
  - The Activity on top of the stack is visible/in foreground
  - Background activities are stopped but state is retained
  - Back button resumes previous Activity in the stack
- HOME button moves app and its activities in background
Activity Example

```java
import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    // savedInstanceState holds any data that may have been saved for the activity just before it got
    killed by the system (e.g. to save memory) the last time
}
```

```xml
<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>
```
Views

- Views are building blocks of Activities/UI
  - TextView, EditText, ListView, ImageView, MapView, WebView...

```
main.xml
<TextView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/hello"
/></TextView>
```

```
MyActivity.java
public class MyActivity extends Activity
{
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
}
```
Views Continued

- Views can also be created using "programmatic" UI layout

```java
MyActivity.java
public class MyActivity extends Activity
{
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        // setContentView(R.layout.main);
        TextView tv = new TextView(this);
        tv.setText("Hello, Android");
        setContentView(tv);
    }
}
```
Layouts

- Controls how Views are laid out:
  - LinearLayout: single row or column
  - TableLayout: rows and columns
  - RelativeLayout: relative to other Views

```java
MyActivity.java
public class MyActivity extends Activity {
    public void onCreate(...) {
        ....
        setContentView(R.layout.main);
    }
}
```

```xml
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">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello1" />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello2" />
</LinearLayout>
```
Services

- Faceless components that typically run in the background
  - music player, network download, etc.
- Services can be started in two ways
  - A component can start the service by calling `startService()`
  - A component can call `bindService()` to create the service
- Service started using `startService()` remains running until explicitly stopped
- Service started using `bindService()` runs as long as the component that created it is still “bound” to it.
- The Android system can force-stop a service when memory is low
  - However “foreground” services are almost never killed.
  - If the system kills a service, it restarts it as soon as resources become available again
Service Example

ExampleService.java
public class ExampleService extends IntentService {

    // Called from the default worker thread. Service stopped when method returns
    @Override
    protected void onHandleIntent(Intent intent) {
        // Do some work here, like download a file.
    }
}

AndroidManifest.xml
<manifest ... >
    ...
    <application ... >
        <service android:name="ExampleService"/>
    ...
</application>
</manifest>

Caller.java
Intent msgIntent = new Intent(this, ExampleService.class);
startService(msgIntent);
Broadcast Receivers

- Components designed to respond to broadcast messages (called Intents)
- Can receive broadcast messages from the system. For example when:
  - A new phone call comes in
  - There is a change in the battery level or cellID
- Can receive messages broadcast by Applications
  - Apps can also define new broadcast messages
Broadcast Receiver Example

**PhoneCallReceiver.java**

```java
public class PhoneCallReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        if (extras != null) {
            String state = extras.getString(TelephonyManager.EXTRA_STATE);
            if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
                String phoneNumber = extras.getString(TelephonyManager.EXTRA_INCOMING_NUMBER);
                Log.w("DEBUG", phoneNumber);
            }
        }
    }
}
```

**AndroidManifest.xml**

```xml
<application android:icon="@drawable/icon" android:label="@string/app_name">

    <receiver android:name="PhoneCallReceiver">
        <intent-filter>
            <action android:name="android.intent.action.PHONE_STATE"></action>
        </intent-filter>
    </receiver>

</application>
```

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

- Enables sharing of data across applications
  - address book, photo gallery, etc.

- Provides uniform APIs for:
  - Query, delete, update, and insert rows
  - Content is represented by URI and MIME type
ContentProvider Example

**BooksContentProvider.java**

```java
public class BooksContentProvider extends ContentProvider {
    @Override
    public int delete(Uri arg0, String arg1, String[] arg2) {...................}

    @Override
    public String getType(Uri uri) {.................................}

    @Override
    public Uri insert(Uri uri, ContentValues values) {.........................}

    @Override
    public boolean onCreate() {..........................}

    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
        ........................................}

    @Override
    public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {...............}
}
```

**AndroidManifest.xml**

```xml
<provider android:name="edu.columbia.BooksContentProvider" android:authorities="books"/>
```

**CallingApp.java**

```java
Uri empsUri=Uri.parse("content://books");
Cursor cursor=getContentResolver().query(empsUri, null, null, null, null);
```
Intent

- Intent are messages used for activating components
- Intent Object:
  - Helps identify the receiving component(s)
  - May contain action to be taken and data to act on
  - Serve as notification for a system event (e.g. new call)
- Intents can be:
  - Explicit: Specify receiving component (java class)
  - Implicit: Specify action/data. Components registered for the action/data pair can receive the Intent
    - Register via IntentFilters in AndroidManifest.xml
    - BroadCastRecievers can also register programmatically
Explicit Intent Example

**ExampleService.java**
public class ExampleService extends IntentService {

// Called from the default worker thread. Service stopped when method returns
@Override
protected void onHandleIntent(Intent intent) {
    // Do some work here, like download a file.

}

**AndroidManifest.xml**
<manifest ... >
    ...
    <application ... >
        <service android:name=".ExampleService" />
    ...
</application>
</manifest>

**Caller.java**
Intent msgIntent = new Intent(this, ExampleService.class);
startService(msgIntent);
Implicit Intent Example

**AndroidManifest.xml**

```xml
<activity android:name="MyBrowserActivity" android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="http"/>
    </intent-filter>
</activity>
```

**Caller.java**

```java
intent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.google.com"));
startActivity(intent);
```
Networking
Net APIs

- Standard java networking APIs
- Two HTTP clients: HttpURLConnection and Apache HTTP Client.

```java
import java.net.Socket;

Socket socket;
try {
    socket = new Socket(hostName, port);
} catch (IOException e) {
    System.out.println(e);
}
```

```java

HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet(url);
try{
    HttpResponse response = client.execute(request);
}catch(Exception ex){
    System.out.println(ex);
}
```
Telephony APIs (android.telephony)

- Send and receive SMS
- Get mobile network info (network type, operator..)
- Get current value of network parameters (cellID, signal strength, SNR, roaming state ..)
- Monitor state changes (cellID, signal strength, SNR, call state, connectivity..)
- Get current device state (connected, idle, active)
- Get device parameters (IMSI, IMEI, device type)
Android Telephony Deep Dive

Ref: http://www.netmite.com/android/mydroid/development/pdk/docs/telephony.html
WiFi APIs (android.net.wifi)

- Get WiFi state (on or off). Turn WiFi on or off.
- Get list of configured networks. Modify attributes of individual entries
- Currently active network. Disconnect from WiFi
- Initiate scan for WiFi APs
- Receive list of WiFi APs (e.g. SSIDs) from a scan
- Connect to a particular WiFi AP
- Get current state (e.g. RSSI, connection state)
- Intents broadcast upon any sort of change in WiFi state
Cloud to Device Messaging

- Various mechanisms to keep an app in synch with changes in the server (cloud)
  - Polling: App periodically polls the servers for changes
  - Push: Servers push changes to app
- Polling can be inefficient if server data changes infrequently
  - Unnecessary Battery drain and network (signaling and data) overhead
- Several apps polling independently without coordination can also be inefficient
  - High battery drain and radio signaling every time the device moves from “idle” to “radio connected” state
Push Notifications

- Network firewalls prevent servers from directly sending messages to mobile devices
- Alternative is to have the device initiate the connection
  - Maintain a connection between device and cloud
  - “Push” cloud updates to apps on the device via this connection
  - Optimize this connection to minimize bandwidth and battery consumption
    - E.g. by adjusting the frequency of keep-alive messages
- This is the principal behind Android’s Cloud to Device Messaging (C2DM)
  - Available since Android 2.2
C2DM

- Device maintains a connection to Android Marketplace
- App Server sends message to C2DM servers (e.g. via http post)
  - Message size limited to 1024 bytes
- C2DM servers forward the message to app on the device
  - If device is not online then will wait until device comes online
  - Message sent to app via a Broadcast Intent (app has to register for it)
- Message notifies that there is an update for the app. It may trigger the App to contact the server
Using C2DM

2. Setup AndroidManifest
   - BroadcastReceiver that will receive C2DM messages
   - Permissions to register and receive C2DM messages
3. Register with C2DM in the app
4. Handle registration and other messages from C2DM in the app
   - Registration response contains a registration id which the App Server needs to be able to send C2DM messages to the device
Manifest file for using C2DM

```
<manifest>
  <permission
      android:name="edu.columbia.permission.C2D_MESSAGE"
      android:protectionLevel="signature" />
  <uses-permission android:name="edu.columbia.permission.C2D_MESSAGE" />
  <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
  <uses-permission android:name="android.permissionINTERNET" />

  <receiver android:name=".MyC2DMReceiver"
    android:permission="com.google.android.c2dm.permission.SEND">
    <!-- Receive the actual message -->
    <intent-filter>
      <action android:name="com.google.android.c2dm.intent.RECEIVE" />
      <category android:name="edu.columbia" />
    </intent-filter>
  </receiver>

  <!-- Receive the registration id -->
  <receiver>
    <intent-filter>
      <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
      <category android:name="edu.columbia" />
    </intent-filter>
  </receiver>
</manifest>
```
Registering with C2DM (device side)

Register.java

```java
Intent intent = new Intent("com.google.android.c2dm.intent.REGISTER");
intent.putExtra("app", PendingIntent.getBroadcast(this, 0, new Intent(), 0));
intent.putExtra("sender", EmailUsedToRegisterWithC2DM);
startService(intent);
```

- In main activity send the register call
- Include the email used to register with C2DM.
- PendingIntent gives C2DM info about the app (via the **this** pointer)
- The service asynchronously registers with C2DM
- Will receive "com.google.android.c2dm.intent.REGISTRATION" intent upon successful registration
Handle messages from C2DM

```java
MyC2DMReceiver.java

public class MyC2DMReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("com.google.android.c2dm.intent.REGISTRATION")) {
            String registrationId = intent.getStringExtra("registration_id");
            handleRegistration(...........);
        } else if (intent.getAction().equals("com.google.android.c2dm.intent.RECEIVE")) {
            handleMessage(.............);
        }
    }
}
```

- From the Registration response a registration id is obtained and is sent to the App Server
- App Server needs the registration ID to send C2DM messages to the app
References

- **Basics**
  - [http://www.vogella.de/android.html](http://www.vogella.de/android.html)