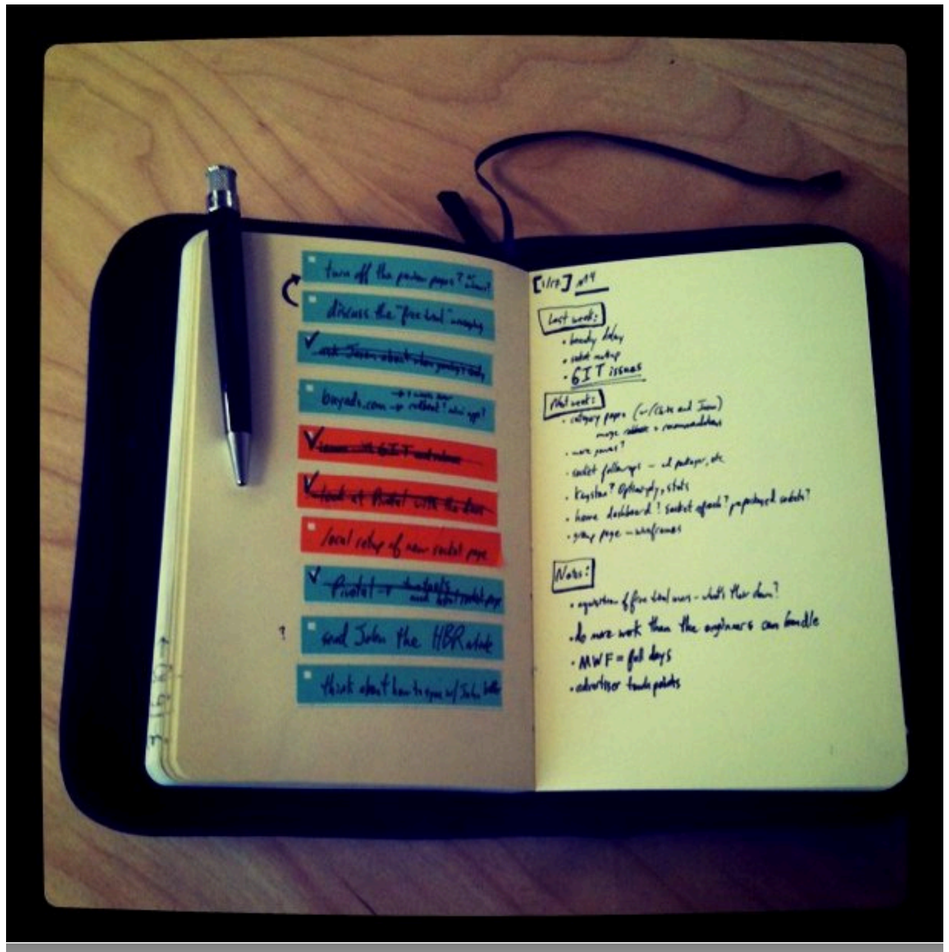
Android Development

Lecture 10
Networking

Lecture Summary

- Android & Networking
- HTTP
- Connectivity Manager
- WifiManager
- TelephonyManager
- Bluetooth



Android & Networking

- Applications written with networking components are far more dynamic and content-rich than those that are not.
- Networking capabilities can be used for a variety of reasons:
  - to delivery fresh and updated content
  - to enable social networking features
  - to offload heavy processing to high-power servers
  - to enable data storage beyond what the user can do on the device
- Networking on the Android platform is standardized, using a combination of powerful and easy to use solutions such as **java.net**.
- Android support traditional networking API like Socket and SocketServer and HTTP (Hypertext Transfer Protocol).

Accessing the Web (HTTP)

- The most common way to transfer data to and from the network is to use HTTP.
- You can use HTTP to encapsulate almost any type of data and to secure the data with Secure Sockets Layer (SSL), which can be important when you transmit data that falls under privacy requirements.
- Most common ports used by HTTP are typically open from the phone networks.
- Reading data from the Web can be really simple. If you only need to read some data from a website and you have the web address of that data, you can use the **URL** class to read a fixed amount of a file stored on a web server.
- Remember that since we are working with network resources, errors can be more common and should be properly managed informing the user about the status of the task and keeping the application active and responding.
- Typical problems are:
  - The phone might not have network coverage
  - The remote server might be down for a short or long period
  - The URL might be invalid

HTTP and Java URL

```
Url urlObj = new URL("http://....");

InputStream is = urlObj.openStream();

byte[] buffer = new byte[250];
int readSize = is.read(buffer);

Log.d(TAG,"readSize = " + readSize);
Log.d(TAG,"read content = " + new String(buffer));

is.close();
```

Create URL Object with the target HTTP url.

Open the InputStream Object to read content from the URL using a buffer.

Close the InputStream at the end of the operation.

HttpURLConnection

- URL based approach might work in some simple situations (retrieve lightweight data), but in complex scenarios you should use advanced solutions to obtain detailed information about available data before downloading it and then use additional API features to build the request.
- The HttpURLConnection object allows to retrieve some information about the resource referenced by the URL Object, including HTTP status and header information.
  - Some of available information are:
    - content length
    - content type
    - date-time information (to check if the data is changed since the last time that you accessed the URL)
- Using this API you can determine if a remote content is appropriate and then you can retrieve it using a standard InputStream object obtained through the method **http.getInputStream().**

HttpURLConnection

```
Url urlObj = new URL("http://....");

HttpURLConnection connection = (HttpURLConnection)url.openConnection();
```

Create URL Object and open an HTTP Connection.

```
Log.d(TAG,"lenght = " + connection.getContentLenght());
Log.d(TAG,"respCode = " + connection.getResponseCode());
Log.d(TAG,"contentType = " + connection.getContentType());
Log.d(TAG,"content = " + connection.getContent());
```

Read useful information about available data.

```
connection.connect();

// read the output from the server
reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
StringBuilder stringBuilder = new StringBuilder();

String line = null;

while ((line = reader.readLine()) != null) {
   stringBuilder.append(line + "\n");
}

return stringBuilder.toString();
```

Using an InputStream and a BufferReader retrieve the content from the remote server.

A StringBuilder object is used to store the obtained data.

Android & Network Status

- The Android SDK provides utilities for gathering information about the current state of the network.
- This is useful to determine if a network connection is even available before trying to use a network resource.
- The **ConnectivityManager** class provides a number of methods to do this. You can for example determine if the mobile or WiFi network are available and connected.
- An instance of ConnectivityManager object is retrieved using the Context method getSystemService(). Then using this object is possible to retrieve NetworkInfo objects for TYPE_WIFI and TYPE_MOBILE.
- For your application to read the status of the network, it needs explicit permission in the AndroidManifest.xml.

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

Connectivity Manager

- Class that answers queries about the state of network connectivity. It also notifies applications when network connectivity changes. Get an instance of this class by calling Context.getSystemService(Context.CONNECTIVITY_SERVICE).
- The primary responsibilities of this class are to:
  - Monitor network connections (Wi-Fi, GPRS, UMTS, etc.)
  - Send broadcast intents when network connectivity changes
  - Attempt to "fail over" to another network when connectivity to a network is lost
  - Provide an API that allows applications to query the coarse-grained or fine-grained state of the available networks

ConnectivityManager

Get SystemService

```
ConnectivityManager cm = (ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE);

NetworkInfo mobNi= cm.getNetworkInfo(ConnectivityManager.TYPE_WIFI);

mobNi.isConnected();
mobNi.isAvailable();

NetworkInfo wifiNi= cm.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);

wifiNi.isConnected();
wifiNi.isAvailable();
```

Retrieve NetworkInfo objects for WiFi and mobile network

NetworkInfo

- The NetworkInfo class describes the status of a network interface of a given type (currently either Mobile or Wifi).
- Main available methods are:
  - **getState()**: Reports the current coarse-grained state of the network.
  - **getType()**: Reports the type of network (currently mobile or Wi-Fi) to which the info in this object pertains.
  - **isAvailable()**: Indicates whether network connectivity is possible.
  - **isConnected()**: Indicates whether network connectivity exists and it is possible to establish connections and pass data.
  - **isConnectedOrConnecting()**: Indicates whether network connectivity exists or is in the process of being established.
  - **isFailover()**: Indicates whether the current attempt to connect to the network resulted from the ConnectivityManager trying to fail over to this network following a disconnect from another network.
  - **isRoaming()**: Indicates whether the device is currently roaming on this network.

Connectivity Manager & Broadcast Receiver

- The ConnectivityManager class allows to send broadcast intents when network connectivity changes.

```
registerReceiver(mConnReceiver,new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION));
```

- Available Intent Extras are:
  - EXTRA_EXTRA_INFO: The lookup key for a string that provides optionally supplied extra information about the network state.
  - EXTRA_IS_FAILOVER: The lookup key for a boolean that indicates whether a connect event is for a network to which the connectivity manager was failing over following a disconnect on another network.
  - EXTRA_NETWORK_INFO: This constant is deprecated. Since NetworkInfo can vary based on UID, applications should always obtain network information through getActiveNetworkInfo() or getAllNetworkInfo().
  - EXTRA_NO_CONNECTIVITY: The lookup key for a boolean that indicates whether there is a complete lack of connectivity, i.e., no network is available.
  - EXTRA_OTHER_NETWORK_INFO: The lookup key for a NetworkInfo object.
  - EXTRA_REASON: The lookup key for a string that indicates why an attempt to connect to a network failed.

ConnectivityManager & Broadcast Receiver

```
private BroadcastReceiver mConnReceiver = new BroadcastReceiver() {
      @Override
      public void onReceive(Context context, Intent intent) {

         Log.d(TAG,"Received Connectivity Intent: " + intent);

         boolean noConnectivity = intent.getBooleanExtra(ConnectivityManager.EXTRA_NO_CONNECTIVITY, false);
         String reason = intent.getStringExtra(ConnectivityManager.EXTRA_REASON);
         boolean isFailover = intent.getBooleanExtra(ConnectivityManager.EXTRA_IS_FAILOVER, false);

         NetworkInfo currentNetworkInfo = (NetworkInfo) intent.getParcelableExtra(ConnectivityManager.EXTRA_NETWORK_INFO);
         NetworkInfo otherNetworkInfo = (NetworkInfo) intent.getParcelableExtra(ConnectivityManager.EXTRA_OTHER_NETWORK_INFO);

         Log.d(TAG,"No Connectivity: " + noConnectivity + " reason:" + reason + " isFailover: " + isFailover);


         if(currentNetworkInfo != null)
            Log.d(TAG,"Current Network Info: " + currentNetworkInfo.getTypeName()
                  + " ("+currentNetworkInfo.getType()+") ["+currentNetworkInfo.getState()+"]");

         if(otherNetworkInfo != null)
            Log.d(TAG,"Other Network Info: " + otherNetworkInfo.getTypeName()
               + " ("+currentNetworkInfo.getType()+") ["+currentNetworkInfo.getState()+"]");
      }
};
```

Working with WiFi

- The WiFi sensor can read network status and determine nearby wireless access points.
- The Android SDK provides a set of APIs for retrieving information about the WiFi networks available to the device and WiFi network connection details.
- This information can be used for tracking signal strength, finding access points of interest or perform actions when connected to a specific access points.
- The class used to work with WiFi is called WifiManager and can be retrieved as a System Service through the Context method getSystemService().
- Working with WiFi require two different and additional Android permissions:

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

WifiManager

- This WifiManager class provides the primary API for managing all aspects of Wi-Fi connectivity. Get an instance of this class by calling Context.getSystemService(Context.WIFI_SERVICE). It allows to obtain:
  - The list of configured networks. The list can be viewed and updated, and attributes of individual entries can be modified.
  - The currently active Wi-Fi network, if any. Connectivity can be established or torn down, and dynamic information about the state of the network can be queried.
  - Results of access point scans, containing enough information to make decisions about what access point to connect to.
  - It defines the names of various Intent actions that are broadcast upon any sort of change in Wi-Fi state.

WifiManager

```
WifiManager wm = (WifiManager) getSystemService(Context.WIFI_SERVICE);
```

```
if(wm.isWifiEnabled())
{
  Log.d(TAG,"Stopping WiFi");
  wm.disconnect();
  wm.setWifiEnabled(false);
}
else
{
  Log.d(TAG,"Enabling WiFi");
  wm.setWifiEnabled(true);
}
```

Check if the WiFi interface is enabled

You can disconnect and disable the interface or re-enable when it is needed

WifiManager and AccessPoint Scan

- WiFiManager allows to perform a manual scan of available Access Point (AP) near the mobile device using the method **wm.startScan()**.
- This method returns immediately and the availability of the results is made known later by means of an asynchronous event sent on completion of the scan (available thorough a BroadcastReceived).
- To obtain the list of detected APs you can use the method **wm.getScanResults()** returning a list of **ScanResult** Objects.
- ScanResult class Describes information about a detected access point. In addition to the attributes described here, the supplicant keeps track of quality, noise, and maxbitrate attributes.
- Available information are:
  - **BSSID**: The address of the access point.
  - **SSID**: The network name.
  - **capabilities**: Describes the authentication, key management, and encryption schemes supported by the access point.
  - **frequency**: The frequency in MHz of the channel over which the client is communicating with the access point.
  - **level**: The detected signal level in dBm.

WifiManager and Configured Networks

- WifiManager allows also to retrieve the list of all the networks configured in by the user.
- Through the method The set of group ciphers supported by this configuration. you can obtain a list of WifiConfiguration containing the following information:
  - **networkId**: The ID number that the supplicant uses to identify this network configuration entry.
  - **SSID**: The network's SSID. (Note: Contains additional “” that are not present in the ScanResult SSID field)
  - **BSSID**: When set, this network configuration entry should only be used when associating with the AP having the specified BSSID.
  - **priority**: Priority determines the preference given to a network by wpa_supplicant when choosing an access point with which to associate.
  - **allowedProtocols**: The set of security protocols supported by this configuration.
  - **allowedKeyManagement:The** set of key management protocols supported by this configuration.
  - **allowedAuthAlgorithms**: The set of authentication protocols supported by this configuration.
  - **allowedPairwiseCiphers**: The set of pairwise ciphers for WPA supported by this configuration.
  - **allowedGroupCiphers**: The set of group ciphers supported by this configuration.

http://developer.android.com/reference/android/net/wifi/WifiConfiguration.html

Wifi Network Configuration and Connection

- WifiManager provides also the methods to create and save a new WifiConfiguration on the user device.

```
WifiManager wifi = (WifiManager) getSystemService(Context.WIFI_SERVICE);

WifiConfiguration wc = new WifiConfiguration();

wc.SSID = "\"SSIDName\"";
wc.preSharedKey  = "\"password\"";
wc.hiddenSSID = true;
wc.status = WifiConfiguration.Status.ENABLED;
wc.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.TKIP);
wc.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.CCMP);
wc.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_PSK);
wc.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.TKIP);
wc.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.CCMP);
wc.allowedProtocols.set(WifiConfiguration.Protocol.RSN);

int res = wifi.addNetwork(wc);

Log.d("WifiPreference", "add Network returned " + res );

boolean b = wifi.enableNetwork(res, true);

Log.d("WifiPreference", "enableNetwork returned " + b );
```

Allow to enable a configured network.

WifiManager & Broadcast Receiver

- The WifiManager class allows to set a dedicated BroadcastReceiver to retrieve the status of WiFi Interface and results of access point scans, containing enough information to make decisions about what access point to connect to.
- Available Intent Actions are:
  - NETWORK_IDS_CHANGED_ACTION: The network IDs of the configured networks could have changed.
  - NETWORK_STATE_CHANGED_ACTION: Broadcast intent action indicating that the state of Wi-Fi connectivity has changed.
  - RSSI_CHANGED_ACTION: The RSSI (signal strength) has changed.
  - SCAN_RESULTS_AVAILABLE_ACTION: An access point scan has completed, and results are available from the supplicant.
  - SUPPLICANT_CONNECTION_CHANGE_ACTION: Broadcast intent action indicating that a connection to the supplicant has been established (and it is now possible to perform Wi-Fi operations) or the connection to the supplicant has been lost.
  - SUPPLICANT_STATE_CHANGED_ACTION: Broadcast intent action indicating that the state of establishing a connection to an access point has changed.One extra provides the newSupplicantState.
  - WIFI_STATE_CHANGED_ACTION: Broadcast intent action indicating that Wi-Fi has been enabled, disabled, enabling, disabling, or unknown.

http://developer.android.com/reference/android/net/wifi/WifiManager.html

WifiManager & Broadcast Receiver

- For example you can create an in-line IntentFilter to receive the following notifications:

```
IntentFilter intfil = new IntentFilter();
intfil.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
intfil.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
intfil.addAction(WifiManager.NETWORK_IDS_CHANGED_ACTION);
intfil.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
intfil.addAction(WifiManager.RSSI_CHANGED_ACTION);
```

```
wifiNetworkEventsReceiver = new WiFiNetworkChangesReceiver();
registerReceiver(wifiNetworkEventsReceiver, intfil);
```

- Available IntentExtras are:
  - EXTRA_BSSID, EXTRA_NETWORK_INFO, EXTRA_NEW_RSSI, EXTRA_NEW_STATE, EXTRA_PREVIOUS_WIFI_STATE, EXTRA_SUPPLICANT_CONNECTED, EXTRA_SUPPLICANT_ERROR, EXTRA_WIFI_INFO and EXTRA_WIFI_STATE.

WifiManager & Broadcast Receiver

```
public class WiFiNetworkChangesReceiver extends BroadcastReceiver{

   @Override
   public void onReceive(Context context, Intent intent) {
      if(intent.getAction().equals(WifiManager.NETWORK_STATE_CHANGED_ACTION)){
         NetworkInfo ni = (NetworkInfo)intent.getParcelableExtra(WifiManager.EXTRA_NETWORK_INFO);
         ...
      }
      if(intent.getAction().equals(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION)){
         SupplicantState ss = (SupplicantState)intent.getParcelableExtra(WifiManager.EXTRA_NEW_STATE);
         ...
      }
      if(intent.getAction().equals(WifiManager.NETWORK_IDS_CHANGED_ACTION)){
         Log.d(TAG, "WiFiNetworkChangesReceiver: action " + intent.getAction());
      }
      if( intent.getAction().equals(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION))
      {
         Log.d(TAG, "WiFiNetworkChangesReceiver: action " + intent.getAction());
         ...
      }
      if(intent.getAction().equals(WifiManager.RSSI_CHANGED_ACTION)){

         int newRssi = intent.getIntExtra(WifiManager.EXTRA_NEW_RSSI, 0);
         ...
      }
   }
}
```

Android & Telephony Utilities

- The Android SDK provides a number of useful utilities for applications to integrate phone features available on the device.
- This information are really important to add features to your application. For example your app should not interrupt a phone conversation or should stop media playing when an incoming call arrives.
- To avoid this kind of behavior, your application must know something about what the user is doing in order to properly react.
- Another example is to adapt application features according to the type of user provider for example extracting SIM information like IMSI, Operator Name and Cell Locations.

Telephony Manager

- The TelephonyManager object is the platform interface to access Phone State Information.
- It provides access to information about the telephony services on the device. Applications can use the methods in this class to determine telephony services and states, as well as to access some types of subscriber information. Applications can also register a listener to receive notification of telephony state changes.
- You do not instantiate this class directly; instead, you retrieve a reference to an instance through Context.getSystemService(Context.TELEPHONY_SERVICE).
- Note that access to some telephony information is permission-protected. Your application cannot access the protected information unless it has the appropriate permissions declared in its manifest file. Where permissions apply, they are noted in the the methods through which you access the protected information.

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

TelephonyManager & BroadcastReceiver

- You can register to receive updates using the Intent Action ACTION_PHONE_STATE_CHANGED.
- Available Intent Extras are: EXTRA_STATE, EXTRA_INCOMING_NUMBER, EXTRA_STATE_IDLE, EXTRA_STATE_OFFHOOK, EXTRA_STATE_RINGING.

```
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);
         }
      }
   }
}
```

TelephonyManager

```
TelephonyManager tm  = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
if(tm != null)
{
    int simState = tm.getSimState();
    //Checks the status of the SIM CARD
    if(simState != TelephonyManager.SIM_STATE_ABSENT)
    {
        //Returns the serial number of the SIM, if applicable. Return null if it is unavailable.
        //Requires Permission: READ_PHONE_STATE
        String simSerialNumber = tm.getSimSerialNumber();

        //Returns the MCC+MNC (mobile country code + mobile network code) of the provider of the SIM.
        String simOperatorCode = tm.getSimOperator();

        //Returns the Service Provider Name (SPN).
        String simOperatorName = tm.getSimOperatorName();

        //Returns the alphabetic name of current registered operator.
        String operatorName = tm.getNetworkOperatorName();

        //Returns the numeric name (MCC+MNC) of current registered operator.
        String operatorCode = tm.getNetworkOperator();

        //Returns the ISO country code equivalent of the current registered operator's MCC (Mobile Country Code).
        String operatorISO = tm.getNetworkCountryIso();

        //Returns the unique subscriber ID, for example, the IMSI for a GSM phone.
        String subscriberId = tm.getSubscriberId();
    }
}
```

Bluetooth

- The Android platform includes support for the Bluetooth network stack, which allows a device to wirelessly exchange data with other Bluetooth devices. The application framework provides access to the Bluetooth functionality through the Android Bluetooth APIs. These APIs let applications wirelessly connect to other Bluetooth devices, enabling point-to-point and multipoint wireless features.
  - Using the Bluetooth APIs, an Android application can perform the following:
    - Scan for other Bluetooth devices
    - Query the local Bluetooth adapter for paired Bluetooth devices
    - Establish RFCOMM channels
    - Connect to other devices through service discovery
    - Transfer data to and from other devices
    - Manage multiple connections

Bluetooth

- The **android.bluetooth** package contains all available bluetooth API. Main classes are:
  - **BluetoothAdapter**: Represents the local Bluetooth adapter (Bluetooth radio). The BluetoothAdapter is the entry-point for all Bluetooth interaction. Using this, you can discover other Bluetooth devices, query a list of bonded (paired) devices, instantiate a BluetoothDevice using a known MAC address, and create aBluetoothServerSocket to listen for communications from other devices.
  - **BluetoothDevice**: Represents a remote Bluetooth device. Use this to request a connection with a remote device through a BluetoothSocket or query information about the device such as its name, address, class, and bonding state.
  - **BluetoothSocket**: Represents the interface for a Bluetooth socket (similar to a TCP Socket). This is the connection point that allows an application to exchange data with another Bluetooth device via InputStream and OutputStream.
  - **BluetoothServerSocket**: Represents an open server socket that listens for incoming requests (similar to a TCP ServerSocket). In order to connect two Android devices, one device must open a server socket with this class. When a remote Bluetooth device makes a connection request to the this device, the BluetoothServerSocket will return a connected BluetoothSocket when the connection is accepted.
  - **BluetoothClass**: Describes the general characteristics and capabilities of a Bluetooth device. This is a read-only set of properties that define the device's major and minor device classes and its services. However, this does not reliably describe all Bluetooth profiles and services supported by the device, but is useful as a hint to the device type.
  - **BluetoothProfile**: An interface that represents a Bluetooth profile. A Bluetooth profile is a wireless interface specification for Bluetooth-based communication between devices. An example is the Hands-Free profile. For more discussion of profiles, see Working with Profiles

Bluetooth Permission

- In order to use Bluetooth features in your application, you need to declare at least one of two Bluetooth permissions: BLUETOOTH and BLUETOOTH_ADMIN.
- You must request the BLUETOOTH permission in order to perform any Bluetooth communication, such as requesting a connection, accepting a connection, and transferring data.
- You must request the BLUETOOTH_ADMIN permission in order to initiate device discovery or manipulate Bluetooth settings. Most applications need this permission solely for the ability to discover local Bluetooth devices. The other abilities granted by this permission should not be used, unless the application is a "power manager" that will modify Bluetooth settings upon user request. Note: If you use BLUETOOTH_ADMIN permission, then must also have the BLUETOOTH permission.

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

Setting Up Bluetooth

The BluetoothAdapter is required for any and all Bluetooth activity. To get the BluetoothAdapter, call the static getDefaultAdapter() method. This returns aBluetoothAdapter that represents the device's own Bluetooth adapter (the Bluetooth radio). There's one Bluetooth adapter for the entire system, and your application can interact with it using this object. If getDefaultAdapter() returns null, then the device does not support Bluetooth and your story ends here.

```
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
if (mBluetoothAdapter == null) {
    // Device does not support Bluetooth
}
```

```
if (!mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

Finding Devices / Querying paired devices

Before performing device discovery, its worth querying the set of paired devices to see if the desired device is already known. To do so, call getBondedDevices(). This will return a Set of BluetoothDevices representing paired devices.

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
// If there are paired devices
if (pairedDevices.size() > 0) {
    // Loop through paired devices
    for (BluetoothDevice device : pairedDevices) {
        // Add the name and address to an array adapter to show in a ListView
        mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
    }
}
```

Discovering Devices

- To start discovering devices, simply call startDiscovery().
- The process is asynchronous and the method will immediately return with a boolean indicating whether discovery has successfully started. The discovery process usually involves an inquiry scan of about 12 seconds, followed by a page scan of each found device to retrieve its Bluetooth name.
- Your application must register a BroadcastReceiver for the ACTION_FOUND Intent in order to receive information about each device discovered.
- For each device, the system will broadcast the ACTION_FOUND Intent. This Intent carries the extra fields EXTRA_DEVICE and EXTRA_CLASS, containing a BluetoothDevice and a BluetoothClass, respectively. For example, here's how you can register to handle the broadcast when devices are discovered:

Discovering Devices

```
// Create a BroadcastReceiver for ACTION_FOUND
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);
            // Add the name and address to an array adapter to show in a ListView
            mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
        }
    }
};

// Register the BroadcastReceiver
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
registerReceiver(mReceiver, filter); // Don't forget to unregister during onDestroy
```

Bluetooth Connecting Devices

- When you want to connect two devices, one must act as a server by holding an open BluetoothServerSocket.
- The purpose of the server socket is to listen for incoming connection requests and when one is accepted, provide a connected BluetoothSocket.
- When the BluetoothSocket is acquired from the BluetoothServerSocket, theBluetoothServerSocket can (and should) be discarded, unless you want to accept more connections.
- In order to initiate a connection with a remote device (a device holding an open server socket), you must first obtain a BluetoothDevice object that represents the remote device. You must then use the BluetoothDevice to acquire a BluetoothSocket and initiate the connection.

Bluetooth Server

```
private class AcceptThread extends Thread {
    private final BluetoothServerSocket mmServerSocket;

    public AcceptThread() {

        BluetoothServerSocket tmp = null;
        try {
            // MY_UUID is the app's UUID string, also used by the client code
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
        } catch (IOException e) { }
        mmServerSocket = tmp;
    }
```

```
    public void cancel() {
        try {
            mmServerSocket.close();
        } catch (IOException e) { }
    }
}
```

```
public void run() {
    BluetoothSocket socket = null;

    while (true) {
        try {
            socket = mmServerSocket.accept();
        } catch (IOException e) {
            break;
        }
        if (socket != null) {
            manageConnectedSocket(socket);
            mmServerSocket.close();
            break;
        }
    }
}
```

Bluetooth Client

```
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        BluetoothSocket tmp = null;
        mmDevice = device;
        try {
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) { }
        mmSocket = tmp;
    }

    public void run() {
        // Cancel discovery because it will slow down the connection
        mBluetoothAdapter.cancelDiscovery();

        try {
            // Connect the device through the socket. This will block
            // until it succeeds or throws an exception
            mmSocket.connect();
        } catch (IOException connectException) {
            // Unable to connect; close the socket and get out
            try {
                mmSocket.close();
            } catch (IOException closeException) { }
            return;
        }

        // Do work to manage the connection (in a separate thread)
        manageConnectedSocket(mmSocket);
    }

    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) { }
    }
}
```

Android Development

Lecture 10
Networking