Android Developers

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

This page focuses on *Classic Bluetooth*. Classic Bluetooth is the right choice for more battery-intensive operations, which include streaming and communicating between Android devices. For Bluetooth devices with low power requirements, Android 4.3 (API level 18) introduces API support for Bluetooth Low Energy. To learn more, see Bluetooth Low Energy (https://developer.android.com/guide/topics/connectivity/bluetooth-le.html).

In this document

Related samples

Android BluetoothChat

This document describes different Bluetooth profiles, including the Health Device Profile. It then explains how to use the Android Bluetooth APIs to accomplish the four major tasks necessary to communicate using Bluetooth: setting up Bluetooth, finding devices that are either paired or available in the local area, connecting devices, and transferring data between devices.

The Basics

In order for Bluetooth-enabled devices to transmit data between each other, they must first form a channel of communication using a *pairing* process. One device, a *discoverable device*, makes itself available for incoming connection requests. Another device finds the discoverable device using a *service discovery* process. After the discoverable device accepts the pairing request, the two devices complete a *bonding* process where they exchange security keys. The devices cache these keys for later use. After the pairing and bonding processes are complete, the two devices exchange information. When the session is complete, the device that initiated the pairing request releases the channel that had linked it to the discoverable device. The two devices remain bonded, however, so they can reconnect automatically during a future session as long as they're in range of each other and neither device has removed the bond.

Bluetooth Permissions

In order to use Bluetooth features in your application, you must declare two permissions. The first of these is `BLUETOOTH` (https://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH). You need this permission to perform any Bluetooth communication, such as requesting a connection, accepting a connection, and transferring data.

The other permission that you must declare is either `ACCESS_COARSE_LOCATION` (https://developer.android.com/reference/android/Manifest.permission.html#ACCESS_COARSE_LOCATION) or `ACCESS_FINE_LOCATION` (https://developer.android.com/reference/android/Manifest.permission.html#ACCESS_FINE_LOCATION). A location permission is required because Bluetooth scans can be used to gather information about the location of the user. This information may come from the user's own devices, as well as Bluetooth beacons in use at locations such as shops and transit facilities.

If you want your app to initiate device discovery or manipulate Bluetooth settings, you must declare the `BLUETOOTH_ADMIN` (https://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH_ADMIN) permission in addition to the `BLUETOOTH` (https://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH) permission. 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 modifies Bluetooth settings upon user request.

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

See the <uses-permission> (https://developer.android.com/guide/topics/manifest/uses-permission-element.html) reference for more information about declaring application permissions.

Working with Profiles

Starting in Android 3.0, the Bluetooth API includes support for working with Bluetooth profiles. A *Bluetooth profile* is a wireless interface specification for Bluetooth-based communication between devices. An example is the Hands-Free profile. For a mobile phone to connect to a wireless headset, both devices must support the Hands-Free profile.

The Android Bluetooth API provides implementations for the following Bluetooth profiles:

- **Headset**. The Headset profile provides support for Bluetooth headsets to be used with mobile phones. Android provides the `BluetoothHeadset` (https://developer.android.com/reference/android/bluetooth/BluetoothHeadset.html) class, which is a proxy for controlling the Bluetooth Headset Service. This includes both Bluetooth Headset and Hands-Free (v1.5) profiles. The `BluetoothHeadset` (https://developer.android.com/reference/android/bluetooth/BluetoothHeadset.html) class includes support for AT commands. For more discussion of this topic, see Vendor-specific AT commands (#AT-Commands)
- **A2DP**. The Advanced Audio Distribution Profile (A2DP) profile defines how high quality audio can be streamed from one device to another over a Bluetooth connection. Android provides the `BluetoothA2dp` (https://developer.android.com/reference/android/bluetooth/BluetoothA2dp.html) class, which is a proxy for controlling the Bluetooth A2DP Service.
- **Health Device**. Android 4.0 (API level 14) introduces support for the Bluetooth Health Device Profile (HDP). This lets you create applications that use Bluetooth to communicate with health devices that support Bluetooth, such as heart-rate monitors, blood meters, thermometers, scales, and so on. For a list of supported devices and their corresponding device data specialization codes, refer to Bluetooth's HDP Device Data Specializations (https://www.bluetooth.com/specifications/assigned-numbers/health-device-profile) . These values are also referenced in the ISO/IEEE 11073-20601 [7] specification as MDC_DEV_SPEC_PROFILE_* in the Nomenclature Codes Annex. For more

Here are the basic steps for working with a profile:

1. Get the default adapter, as described in Setting Up Bluetooth (https://developer.android.com/guide/topics/connectivity/bluetooth.html#SettingUp).
2. Set up a `BluetoothProfile.ServiceListener` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.ServiceListener.html). This listener notifies `BluetoothProfile` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.html) clients when they have been connected to or disconnected from the service.
3. Use `getProfileProxy()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#getProfileProxy(android.content.Context, android.bluetooth.BluetoothProfile.ServiceListener, int)) to establish a connection to the profile proxy object associated with the profile. In the example below, the profile proxy object is an instance of `BluetoothHeadset` (https://developer.android.com/reference/android/bluetooth/BluetoothHeadset.html).
4. In `onServiceConnected()` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.ServiceListener.html#onServiceConnected(int, android.bluetooth.BluetoothProfile)), get a handle to the profile proxy object.
5. Once you have the profile proxy object, you can use it to monitor the state of the connection and perform other operations that are relevant to that profile.

For example, this code snippet shows how to connect to a `BluetoothHeadset` (https://developer.android.com/reference/android/bluetooth/BluetoothHeadset.html) proxy object so that you can control the Headset profile:

```
BluetoothHeadset mBluetoothHeadset;

// Get the default adapter
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

private BluetoothProfile.ServiceListener mProfileListener = new BluetoothProfile
    public void onServiceConnected(int profile, BluetoothProfile proxy) {
        if (profile == BluetoothProfile.HEADSET) {
            mBluetoothHeadset = (BluetoothHeadset) proxy;
        }
    }
    public void onServiceDisconnected(int profile) {
        if (profile == BluetoothProfile.HEADSET) {
            mBluetoothHeadset = null;
        }
    }
};
```

```
// Establish connection to the proxy.
mBluetoothAdapter.getProfileProxy(context, mProfileListener, BluetoothProfile.HE

// ... call functions on mBluetoothHeadset

// Close proxy connection after use.
mBluetoothAdapter.closeProfileProxy(mBluetoothHeadset);
```

Vendor-specific AT commands

Starting in Android 3.0 (API level 11), applications can register to receive system broadcasts of predefined vendor-specific AT commands sent by headsets (such as a Plantronics +XEVENT command). For example, an application could receive broadcasts that indicate a connected device's battery level and could notify the user or take other action as needed. Create a broadcast receiver for the `ACTION_VENDOR_SPECIFIC_HEADSET_EVENT` (https://developer.android.com/reference/android/bluetooth/BluetoothHeadset.html#ACTION_VENDOR_SPECIFIC_HEADSET_EVENT) intent to handle vendor-specific AT commands for the headset.

Health Device Profile

Android 4.0 (API level 14) introduces support for the Bluetooth Health Device Profile (HDP). This lets you create applications that use Bluetooth to communicate with health devices that support Bluetooth, such as heart-rate monitors, blood meters, thermometers, and scales. The Bluetooth Health API includes the classes `BluetoothHealth` (https://developer.android.com/reference/android/bluetooth/BluetoothHealth.html), `BluetoothHealthCallback` (https://developer.android.com/reference/android/bluetooth/BluetoothHealthCallback.html), and `BluetoothHealthAppConfiguration` (https://developer.android.com/reference/android/bluetooth/BluetoothHealthAppConfiguration.html), which are described in Key Classes and Interfaces (#KeyClassesAndInterfaces).

In using the Bluetooth Health API, it's helpful to understand these key HDP concepts:

Source

A health device—such as a weight scale, glucose meter, or thermometer—that transmits medical data to a smart device, such as an Android phone or tablet.

Sink

The smart device that receives the medical data. In an Android HDP application, the sink is

/reference/android/bluetooth/BluetoothHealthAppConfiguration.html) object.

Registration

The process used to register a sink for communicating with a particular health device.

Connection

The process used to open a channel between a health device (source) and a smart device (sink).

Creating an HDP Application

Here are the basic steps involved in creating an Android HDP application:

1. Get a reference to the `BluetoothHealth` (https://developer.android.com/reference/android/bluetooth/BluetoothHealth.html) proxy object.

   Similar to regular headset and A2DP profile devices, you must call `getProfileProxy()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#getProfileProxy(android.content.Context, android.bluetooth.BluetoothProfile.ServiceListener, int)) with a `BluetoothProfile.ServiceListener` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.ServiceListener.html) and the `HEALTH` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.html#HEALTH) profile type to establish a connection with the profile proxy object.

2. Create a `BluetoothHealthCallback` (https://developer.android.com/reference/android/bluetooth/BluetoothHealthCallback.html) and register an application configuration (`BluetoothHealthAppConfiguration` (https://developer.android.com/reference/android/bluetooth/BluetoothHealthAppConfiguration.html)) that acts as a health sink.

3. Establish a connection to a health device.

   > **Note:** Some devices initiate the connection automatically. It is unnecessary to carry out this step for those devices.

4. When connected successfully to a health device, read/write to the health device using the file descriptor. The received data need to be interpreted using a health manager which implements the IEEE 11073 (http://standards.ieee.org/develop/wg/PHD.html) specifications.

5. When done, close the health channel and unregister the application. The channel also closes when there is extended inactivity

Setting Up Bluetooth

Before your application can communicate over Bluetooth, you need to verify that Bluetooth is supported on the device, and if so, ensure that it is enabled.

If Bluetooth isn't supported, then you should gracefully disable any Bluetooth features. If Bluetooth is supported, but disabled, then you can request that the user enable Bluetooth without leaving your application. This setup is accomplished in two steps, using the `BluetoothAdapter` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html):

1. Get the `BluetoothAdapter` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html).

   The `BluetoothAdapter` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html) is required for any and all Bluetooth activity. To get the `BluetoothAdapter` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html), call the static `getDefaultAdapter()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#getDefaultAdapter()) method. This returns a `BluetoothAdapter` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html) 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()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#getDefaultAdapter()) returns `null`, then the device doesn't support Bluetooth. For example:

   ```
   BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
   if (mBluetoothAdapter == null) {
       // Device doesn't support Bluetooth
   }
   ```

2. Enable Bluetooth.

   Next, you need to ensure that Bluetooth is enabled. Call `isEnabled()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#isEnabled()) to check whether Bluetooth is currently enabled. If this method returns false, then Bluetooth is disabled. To request that Bluetooth be enabled, call `startActivityForResult()` (https://developer.android.com/reference/android

ACTION_REQUEST_ENABLE (https://developer.android.com/reference/android/bluetooth /BluetoothAdapter.html#ACTION_REQUEST_ENABLE) intent action. This call issues a request to enable Bluetooth through the system settings (without stopping your application). For example:

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

A dialog appears requesting user permission to enable Bluetooth, as shown in Figure 1. If the user responds "Yes", the system begins to enable Bluetooth, and focus returns to your application once the process completes (or fails).

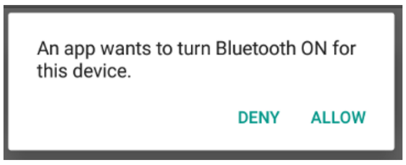


Figure 1: The enabling Bluetooth dialog.

The REQUEST_ENABLE_BT constant passed to startActivityForResult() (https://developer.android.com /reference/android /app/Activity.html#startActivityForResult(android.content.Intent, int)) is a locally defined integer that must be greater than 0. The system passes this constant back to you in your onActivityResult() (https://developer.android.com/reference/android /app/Activity.html#onActivityResult(int, int, android.content.Intent)) implementation as the requestCode parameter.

If enabling Bluetooth succeeds, your activity receives the RESULT_OK (https://developer.android.com/reference/android/app/Activity.html#RESULT_OK) result code in the onActivityResult() (https://developer.android.com/reference/android /app/Activity.html#onActivityResult(int, int, android.content.Intent)) callback. If Bluetooth was not enabled due to an error (or the user responded "No") then the result code is RESULT_CANCELED (https://developer.android.com/reference/android /app/Activity.html#RESULT_CANCELED).

Optionally, your application can also listen for the ACTION_STATE_CHANGED (https://developer.android.com/reference/android/bluetooth /BluetoothAdapter.html#ACTION_STATE_CHANGED) broadcast intent, which the system broadcasts whenever the Bluetooth state changes. This broadcast contains the extra fields EXTRA_STATE (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_STATE) and EXTRA_PREVIOUS_STATE (https://developer.android.com/reference/android/bluetooth /BluetoothAdapter.html#EXTRA_PREVIOUS_STATE), containing the new and old Bluetooth states, respectively. Possible values for these extra fields are STATE_TURNING_ON

`STATE_ON` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#STATE_ON), `STATE_TURNING_OFF` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#STATE_TURNING_OFF), and `STATE_OFF` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#STATE_OFF). Listening for this broadcast can be useful if your app needs to detect runtime changes made to the Bluetooth state.

> **Tip:** Enabling discoverability automatically enables Bluetooth. If you plan to consistently enable device discoverability before performing Bluetooth activity, you can skip step 2 above. For more information, read the enabling discoverability (#EnablingDiscoverability), section on this page.

Finding Devices

Using the `BluetoothAdapter` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html), you can find remote Bluetooth devices either through device discovery or by querying the list of paired devices.

Device discovery is a scanning procedure that searches the local area for Bluetooth-enabled devices and requests some information about each one. This process is sometimes referred to as *discovering*, *inquiring*, or *scanning*. However, a nearby Bluetooth device responds to a discovery request only if it is currently accepting information requests by being *discoverable*. If a device is discoverable, it responds to the discovery request by sharing some information, such as the device's name, its class, and its unique MAC address. Using this information, the device that is performing the discovery process can then choose to initiate a connection to the discovered device.

Once a connection is made with a remote device for the first time, a pairing request is automatically presented to the user. When a device is paired, the basic information about that device—such as the device's name, class, and MAC address—is saved and can be read using the Bluetooth APIs. Using the known MAC address for a remote device, a connection can be initiated with it at any time without performing discovery, assuming the device is still within range.

Note that there is a difference between being paired and being connected:

- To be *paired* means that two devices are aware of each other's existence, have a shared link-key that can be used for authentication, and are capable of establishing an encrypted connection with each other.
- To be *connected* means that the devices currently share an RFCOMM channel and are able to transmit data with each other. The current Android Bluetooth API's require devices to be paired before an RFCOMM connection can be established. Pairing is automatically performed when you

The following sections describe how to find devices that have been paired, or discover new devices using device discovery.

> **Note:** Android-powered devices are not discoverable by default. A user can make the device discoverable for a limited time through the system settings, or an application can request that the user enable discoverability without leaving the application. For more information, see the enable discoverability (#EnablingDiscoverability) section on this page.

Querying paired devices

Before performing device discovery, it's worth querying the set of paired devices to see if the desired device is already known. To do so, call `getBondedDevices()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#getBondedDevices()). This returns a set of `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) objects representing paired devices. For example, you can query all paired devices and get the name and MAC address of each device, as the following code snippet demonstrates:

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();

if (pairedDevices.size() > 0) {
    // There are paired devices. Get the name and address of each paired device.
    for (BluetoothDevice device : pairedDevices) {
        String deviceName = device.getName();
        String deviceHardwareAddress = device.getAddress(); // MAC address
    }
}
```

To initiate a connection with a Bluetooth device, all that's needed from the associated `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) object is the MAC address, which you retrieve by calling `getAddress()` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#getAddress()). You can learn more about creating a connection in the section about Connecting Devices (#ConnectingDevices).

> **Caution:** Performing device discovery consumes a lot of the Bluetooth adapter's resources. After you have found a device to connect to, be certain that you stop discovery with `cancelDiscovery()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#cancelDiscovery()) before attempting a connection. Also, you shouldn't perform discovery while connected to a device because the discovery process significantly reduces the bandwidth available for any existing connections.

Discovering devices

To start discovering devices, simply call `startDiscovery()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#startDiscovery()). The process is asynchronous and returns a boolean value 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 device found to retrieve its Bluetooth name.

In order to receive information about each device discovered, your application must register a BroadcastReceiver for the `ACTION_FOUND` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#ACTION_FOUND) intent. The system broadcasts this intent for each device. The intent contains the extra fields `EXTRA_DEVICE` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#EXTRA_DEVICE) and `EXTRA_CLASS` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#EXTRA_CLASS), which in turn contain a `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) and a `BluetoothClass` (https://developer.android.com/reference/android/bluetooth/BluetoothClass.html), respectively. The following code snippet shows how you can register to handle the broadcast when devices are discovered:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...

    // Register for broadcasts when a device is discovered.
    IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
    registerReceiver(mReceiver, filter);
}

// Create a BroadcastReceiver for ACTION_FOUND.
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            // Discovery has found a device. Get the BluetoothDevice
            // object and its info from the Intent.
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.E
            String deviceName = device.getName();
            String deviceHardwareAddress = device.getAddress(); // MAC address
        }
    }
};
```

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

    // Don't forget to unregister the ACTION_FOUND receiver.
    unregisterReceiver(mReceiver);
}
```

To initiate a connection with a Bluetooth device, all that's needed from the associated `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) object is the MAC address, which you retrieve by calling `getAddress()` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#getAddress()). You can learn more about creating a connection in the section about Connecting Devices (#ConnectingDevices).

> **Caution:** Performing device discovery consumes a lot of the Bluetooth adapter's resources. After you have found a device to connect to, be certain that you stop discovery with `cancelDiscovery()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#cancelDiscovery()) before attempting a connection. Also, you shouldn't perform discovery while connected to a device because the discovery process significantly reduces the bandwidth available for any existing connections.

Enabling discoverability

If you would like to make the local device discoverable to other devices, call `startActivityForResult(Intent, int)` (https://developer.android.com/reference/android/app/Activity.html#startActivityForResult(android.content.Intent, int)) with the `ACTION_REQUEST_DISCOVERABLE` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#ACTION_REQUEST_DISCOVERABLE) intent. This issues a request to enable the system's discoverable mode without having to navigate to the Settings app, which would stop your own app. By default, the device becomes discoverable for 120 seconds, or 2 minutes. You can define a different duration, up to 3600 seconds (1 hour), by adding the `EXTRA_DISCOVERABLE_DURATION` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_DISCOVERABLE_DURATION) extra.

> **Caution:** If you set the `EXTRA_DISCOVERABLE_DURATION` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_DISCOVERABLE_DURATION) extra's value to 0, the device is always discoverable. This configuration is insecure and therefore highly discouraged.

The following code snippet sets the device to be discoverable for 5 minutes (300 seconds):

```
        new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
startActivity(discoverableIntent);
```

A dialog is displayed, requesting the user's permission to make the device discoverable, as shown in Figure 2. If the user responds "Yes," then the device becomes discoverable for the specified amount of time. Your activity then receives a call to the `onActivityResult()` (https://developer.android.com/reference/android/app/Activity.html#onActivityResult(int, int, android.content.Intent)) callback, with the result code equal to the duration that the device is discoverable. If the user responded "No", or if an error occurred, the result code is `RESULT_CANCELED` (https://developer.android.com/reference/android/app/Activity.html#RESULT_CANCELED).

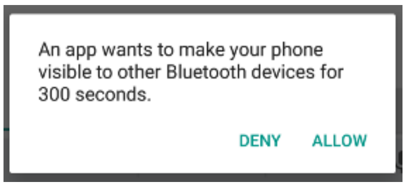

An app wants to make your phone visible to other Bluetooth devices for 300 seconds.

Figure 2: The enabling discoverability dialog.

> **Note:** If Bluetooth has not been enabled on the device, then making the device discoverable automatically enables Bluetooth.

The device silently remains in discoverable mode for the allotted time. If you would like to be notified when the discoverable mode has changed, you can register a BroadcastReceiver for the `ACTION_SCAN_MODE_CHANGED` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#ACTION_SCAN_MODE_CHANGED) intent. This intent contains the extra fields `EXTRA_SCAN_MODE` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_SCAN_MODE) and `EXTRA_PREVIOUS_SCAN_MODE` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_PREVIOUS_SCAN_MODE), which provide the new and old scan mode, respectively. Possible values for each extra are as follows:

`SCAN_MODE_CONNECTABLE_DISCOVERABLE` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#SCAN_MODE_CONNECTABLE_DISCOVERABLE)

The device is in discoverable mode.

`SCAN_MODE_CONNECTABLE` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#SCAN_MODE_CONNECTABLE)

The device isn't in discoverable mode but can still receive connections.

`SCAN_MODE_NONE` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#SCAN_MODE_NONE)

If you are initiating the connection to a remote device, you don't need to enable device discoverability. Enabling discoverability is only necessary when you want your application to host a server socket that accepts incoming connections, as remote devices must be able to discover other devices before initiating connections to those other devices.

Connecting Devices

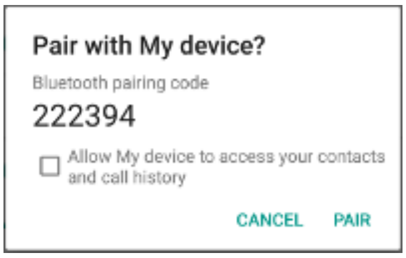
In order to create a connection between two devices, you must implement both the server-side and client-side mechanisms because one device must open a server socket, and the other one must initiate the connection using the server device's MAC address. The server device and the client device each obtain the required `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) in different ways. The server receives socket information when an incoming connection is accepted. The client provides socket information when it opens an RFCOMM channel to the server.

The server and client are considered connected to each other when they each have a connected `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) on the same RFCOMM channel. At this point, each device can obtain input and output streams, and data transfer can begin, which is discussed in the section about Managing a Connection (#ManagingAConnection). This section describes how to initiate the connection between two devices.

Connection techniques

One implementation technique is to automatically prepare each device as a server so that each device has a server socket open and listening for connections. In this case, either device can initiate a connection with the other and become the client. Alternatively, one device can explicitly host the connection and open a server socket on demand, and the other device initiates the connection.

> **Note:** If the two devices have not been previously paired, then the Android framework automatically shows a pairing request notification or dialog to the user during the connection procedure, as shown in Figure 3. Therefore, when your application attempts to connect devices, it doesn't need to be concerned about whether or not the devices are paired. Your RFCOMM connection attempt gets blocked until the user has successfully paired the two devices, and the attempt fails if the user rejects pairing, or if the pairing process fails or times out.


Pair with My device?

Bluetooth pairing code

222394

☐ Allow My device to access your contacts and call history

Figure 3: The Bluetooth pairing dialog.

Connecting as a server

When you want to connect two devices, one must act as a server by holding an open `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html). The purpose of the server socket is to listen for incoming connection requests and provide a connected `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) after a request is accepted. When the `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) is acquired from the `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html), the `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) can—and should—be discarded, unless you want the device to accept more connections.

To set up a server socket and accept a connection, complete the following sequence of steps:

About UUID

A Universally Unique Identifier (UUID) is a standardized 128-bit format for a string ID used to uniquely identify information. The point of a UUID is that it's big enough that you can select any random ID and it won't clash with any other ID. In this case, it's used to uniquely identify your application's Bluetooth service. To get a UUID to use with your application, you can use one of the many random UUID generators on the web, then initialize a `UUID` (https://developer.android.com/reference/java/util/UUID.html) with `fromString(String)` (https://developer.android.com/reference/java/util/UUID.html#fromString(java.lang.String)).

1. Get a `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) by calling `listenUsingRfcommWithServiceRecord()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#listenUsingRfcommWithServiceRecord(java.lang.String, java.util.UUID)).

   The string is an identifiable name of your service, which the system automatically writes to a new Service Discovery Protocol (SDP) database entry on the device. The name is arbitrary and can for the connection agreement with the client device. That is, when the client attempts to connect with this device, it carries a UUID that uniquely identifies the service with which it wants to connect. These UUIDs must match in order for the connection to be accepted.

2. Start listening for connection requests by calling `accept()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()).

   This is a blocking call. It returns when either a connection has been accepted or an exception has occurred. A connection is accepted only when a remote device has sent a connection request containing a UUID that matches the one registered with this listening server socket. When successful, `accept()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()) returns a connected `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html).

3. Unless you want to accept additional connections, call `close()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()).

   This method call releases the server socket and all its resources, but doesn't close the connected `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) that's been returned by `accept()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()). Unlike TCP/IP, RFCOMM allows only one connected client per channel at a time, so in most cases, it makes sense to call `close()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()) on the `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) immediately after accepting a connected socket.

Because the `accept()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()) call is a blocking call, it should not be executed in the main activity UI thread so that your application can still respond to other user interactions. It usually makes sense to do all work that involves a `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) or `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) in a new thread managed by your application. To abort a blocked call such as `accept()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()), call `close()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()) on the `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) or `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) from another thread. Note that all methods on a `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) or `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) are thread-safe.

Example

Here's a simplified thread for the server component that accepts incoming connections:

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

    public AcceptThread() {
        // Use a temporary object that is later assigned to mmServerSocket
        // because mmServerSocket is final.
        BluetoothServerSocket tmp = null;
        try {
            // MY_UUID is the app's UUID string, also used by the client code.
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_
        } catch (IOException e) {
            Log.e(TAG, "Socket's listen() method failed", e);
        }
        mmServerSocket = tmp;
    }

    public void run() {
        BluetoothSocket socket = null;
        // Keep listening until exception occurs or a socket is returned.
        while (true) {
            try {
                socket = mmServerSocket.accept();
            } catch (IOException e) {
                Log.e(TAG, "Socket's accept() method failed", e);
                break;
            }

            if (socket != null) {
                // A connection was accepted. Perform work associated with
                // the connection in a separate thread.
                manageMyConnectedSocket(socket);
                mmServerSocket.close();
                break;
            }
        }
    }

    // Closes the connect socket and causes the thread to finish.
    public void cancel() {
        try {
            mmServerSocket.close();
        } catch (IOException e) {
            Log.e(TAG, "Could not close the connect socket", e);
        }
    }
}
```

In this example, only one incoming connection is desired, so as soon as a connection is accepted and the `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) is acquired, the app passes the acquired `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) to a separate thread, closes the `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html), and breaks out of the loop.

Note that when `accept()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()) returns the `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html), the socket is already connected. Therefore, you shouldn't call `connect()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()), as you do from the client side.

The app-specific `manageMyConnectedSocket()` method is designed to initiate the thread for transferring data, which is discussed in the section about Managing a Connection (#ManagingAConnection).

Usually, you should close your `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) as soon as you are done listening for incoming connections. In this example, `close()` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()) is called as soon as the `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) is acquired. You may also want to provide a public method in your thread that can close the private `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) in the event that you need to stop listening on that server socket.

Connecting as a client

In order to initiate a connection with a remote device that is accepting connections on an open server socket, you must first obtain a `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) object that represents the remote device. To learn how to create a `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html), see Finding Devices (#FindingDevices). You must then use the `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) to acquire a `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) and initiate the connection.

The basic procedure is as follows:

1. Using the `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html), get a `BluetoothSocket` (https://developer.android.com/reference/android

(https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#createRfcommSocketToServiceRecord(java.util.UUID)).

This method initializes a `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) object that allows the client to connect to a `BluetoothDevice` (https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html). The UUID passed here must match the UUID used by the server device when it called `listenUsingRfcommWithServiceRecord(String, UUID)` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#listenUsingRfcommWithServiceRecord(java.lang.String, java.util.UUID)) to open its `BluetoothServerSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html). To use a matching UUID, hard-code the UUID string into your application, and then reference it from both the server and client code.

2. Initiate the connection by calling `connect()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()). Note that this method is a blocking call.

   After a client calls this method, the system performs an SDP lookup to find the remote device with the matching UUID. If the lookup is successful and the remote device accepts the connection, it shares the RFCOMM channel to use during the connection, and the `connect()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()) method returns. If the connection fails, or if the `connect()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()) method times out (after about 12 seconds), then the method throws an `IOException` (https://developer.android.com/reference/java/io/IOException.html).

   Because `connect()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()) is a blocking call, you should always perform this connection procedure in a thread that is separate from the main activity (UI) thread.

   > **Note:** You should always call `cancelDiscovery()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#cancelDiscovery()) to ensure that the device isn't performing device discovery before you call `connect()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()). If discovery is in progress, then the connection attempt is significantly slowed, and it's more likely to fail.

Example

Here is a basic example of a client thread that initiates a Bluetooth connection:

```
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    public ConnectThread(BluetoothDevice device) {
        // Use a temporary object that is later assigned to mmSocket
        // because mmSocket is final.
        BluetoothSocket tmp = null;
        mmDevice = device;

        try {
            // Get a BluetoothSocket to connect with the given BluetoothDevice.
            // MY_UUID is the app's UUID string, also used in the server code.
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) {
            Log.e(TAG, "Socket's create() method failed", e);
        }
        mmSocket = tmp;
    }

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

        try {
            // Connect to the remote device through the socket. This call blocks
            // until it succeeds or throws an exception.
            mmSocket.connect();
        } catch (IOException connectException) {
            // Unable to connect; close the socket and return.
            try {
                mmSocket.close();
            } catch (IOException closeException) {
                Log.e(TAG, "Could not close the client socket", closeException);
            }
            return;
        }

        // The connection attempt succeeded. Perform work associated with
        // the connection in a separate thread.
        manageMyConnectedSocket(mmSocket);
    }

    // Closes the client socket and causes the thread to finish.
    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) {
            Log.e(TAG, "Could not close the client socket", e);
        }
    }
}
```

/bluetooth/BluetoothAdapter.html#cancelDiscovery()) is called before the connection attempt occurs. You should always call `cancelDiscovery()` before `connect()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()), especially because `cancelDiscovery()` succeeds regardless of whether device discovery is currently in progress. If your app needs to determine whether device discovery is in progress, however, you can check using `isDiscovering()` (https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#isDiscovering()).

The app-specific `manageMyConnectedSocket()` method is designed to initiate the thread for transferring data, which is discussed in the section about Managing a Connection (#ManagingAConnection).

When you're done with your `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html), always call `close()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#close()). Doing so immediately closes the connected socket and release all related internal resources.

Managing a Connection

After you have successfully connected multiple devices, each one has a connected `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html). This is where the fun begins because you can share information between devices. Using the `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html), the general procedure to transfer data is as follows:

1. Get the `InputStream` (https://developer.android.com/reference/java/io/InputStream.html) and `OutputStream` (https://developer.android.com/reference/java/io/OutputStream.html) that handle transmissions through the socket using `getInputStream()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#getInputStream()) and `getOutputStream()` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#getOutputStream()), respectively.

2. Read and write data to the streams using `read(byte[])` (https://developer.android.com/reference/java/io/InputStream.html#read(byte[])) and `write(byte[])` (https://developer.android.com/reference/java/io/OutputStream.html#write(byte[])).

There are, of course, implementation details to consider. In particular, you should use a dedicated thread for reading from the stream and writing to it. This is important because both the `read(byte[])` (https://developer.android.com/reference/java/io/InputStream.html#read(byte[])) and methods are blocking calls. The `read(byte[])` (https://developer.android.com/reference/java/io/InputStream.html#read(byte[])) method blocks until there is something to read from the stream. The `write(byte[])` (https://developer.android.com/reference/java/io/OutputStream.html#write(byte[])) method doesn't usually block, but it can block for flow control if the remote device isn't calling `read(byte[])` (https://developer.android.com/reference/java/io/InputStream.html#read(byte[])) quickly enough and the intermediate buffers become full as a result. So, your main loop in the thread should be dedicated to reading from the `InputStream` (https://developer.android.com/reference/java/io/InputStream.html). A separate public method in the thread can be used to initiate writes to the `OutputStream` (https://developer.android.com/reference/java/io/OutputStream.html).

Example

Here's an example of how you can transfer data between two devices connected over Bluetooth:

```
public class MyBluetoothService {
    private static final String TAG = "MY_APP_DEBUG_TAG";
    private Handler mHandler; // handler that gets info from Bluetooth service

    // Defines several constants used when transmitting messages between the
    // service and the UI.
    private interface MessageConstants {
        public static final int MESSAGE_READ = 0;
        public static final int MESSAGE_WRITE = 1;
        public static final int MESSAGE_TOAST = 2;

        // ... (Add other message types here as needed.)
    }

    private class ConnectedThread extends Thread {
        private final BluetoothSocket mmSocket;
        private final InputStream mmInStream;
        private final OutputStream mmOutStream;
        private byte[] mmBuffer; // mmBuffer store for the stream

        public ConnectedThread(BluetoothSocket socket) {
            mmSocket = socket;
            InputStream tmpIn = null;
            OutputStream tmpOut = null;

            // Get the input and output streams; using temp objects because
            // member streams are final.
            try {
                tmpIn = socket.getInputStream();
            } catch (IOException e) {
                Log.e(TAG, "Error occurred when creating input stream", e);
            }
            try {
            } catch (IOException e) {
                Log.e(TAG, "Error occurred when creating output stream", e);
            }

            mmInStream = tmpIn;
            mmOutStream = tmpOut;
        }

        public void run() {
            mmBuffer = new byte[1024];
            int numBytes; // bytes returned from read()

            // Keep listening to the InputStream until an exception occurs.
            while (true) {
                try {
                    // Read from the InputStream.
                    numBytes = mmInStream.read(mmBuffer);
                    // Send the obtained bytes to the UI activity.
                    Message readMsg = mHandler.obtainMessage(
                            MessageConstants.MESSAGE_READ, numBytes, -1,
                            mmBuffer);
                    readMsg.sendToTarget();
                } catch (IOException e) {
                    Log.d(TAG, "Input stream was disconnected", e);
                    break;
                }
            }
        }

        // Call this from the main activity to send data to the remote device.
        public void write(byte[] bytes) {
            try {
                mmOutStream.write(bytes);

                // Share the sent message with the UI activity.
                Message writtenMsg = mHandler.obtainMessage(
                        MessageConstants.MESSAGE_WRITE, -1, -1, mmBuffer);
                writtenMsg.sendToTarget();
            } catch (IOException e) {
                Log.e(TAG, "Error occurred when sending data", e);

                // Send a failure message back to the activity.
                Message writeErrorMsg =
                        mHandler.obtainMessage(MessageConstants.MESSAGE_TOAST);
                Bundle bundle = new Bundle();
                bundle.putString("toast",
                        "Couldn't send data to the other device");
                writeErrorMsg.setData(bundle);
                mHandler.sendMessage(writeErrorMsg);
            }
        }
        // Call this method from the main activity to shut down the connection.
        public void cancel() {
            try {
                mmSocket.close();
            } catch (IOException e) {
                Log.e(TAG, "Could not close the connect socket", e);
            }
        }
    }
}
```

After the constructor acquires the necessary streams, the thread waits for data to come through the `InputStream` (https://developer.android.com/reference/java/io/InputStream.html). When `read(byte[])` (https://developer.android.com/reference/java/io/InputStream.html#read(byte[])) returns with data from the stream, the data is sent to the main activity using a member `Handler` (https://developer.android.com/reference/android/os/Handler.html) from the parent class. The thread then waits for more bytes to be read from the `InputStream` (https://developer.android.com/reference/java/io/InputStream.html).

Sending outgoing data is as simple as calling the thread's `write()` method from the main activity and passing in the bytes to be sent. This method calls `write(byte[])` (https://developer.android.com/reference/java/io/OutputStream.html#write(byte[])) to send the data to the remote device. If an `IOException` (https://developer.android.com/reference/java/io/IOException.html) is thrown when calling `write(byte[])` (https://developer.android.com/reference/java/io/OutputStream.html#write(byte[])), the thread sends a toast to the main activity, explaining to the user that the device couldn't send the given bytes to the other (connected) device.

The thread's `cancel()` method allows the connection to be terminated at any time by closing the `BluetoothSocket` (https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html). This method should always be called when you're done using the Bluetooth connection.

For a demonstration of using the Bluetooth APIs, see the Bluetooth Chat sample app (https://developer.android.com/samples/BluetoothChat/index.html).

Key Classes and Interfaces

All of the Bluetooth APIs are available in the `android.bluetooth` (https://developer.android.com/reference/android/bluetooth/package-summary.html) package. Here's a summary of the classes and interfaces you need to create Bluetooth connections:

[illegible]

Represents the local Bluetooth adapter (Bluetooth radio). The `BluetoothAdapter` `(https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html)` 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` `(https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html)` using a known MAC address, and create a `BluetoothServerSocket` `(https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html)` to listen for communications from other devices.

`BluetoothDevice` `(https://developer.android.com/reference/android/bluetooth/BluetoothDevice.html)`

Represents a remote Bluetooth device. Use this to request a connection with a remote device through a `BluetoothSocket` `(https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html)` or query information about the device such as its name, address, class, and bonding state.

`BluetoothSocket` `(https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html)`

Represents the interface for a Bluetooth socket (similar to a TCP `Socket` `(https://developer.android.com/reference/java/net/Socket.html)`). This is the connection point that allows an application to exchange data with another Bluetooth device using `InputStream` `(https://developer.android.com/reference/java/io/InputStream.html)` and `OutputStream` `(https://developer.android.com/reference/java/io/OutputStream.html)`.

`BluetoothServerSocket` `(https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html)`

Represents an open server socket that listens for incoming requests (similar to a TCP `ServerSocket` `(https://developer.android.com/reference/java/net/ServerSocket.html)`). 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 this device, the device accepts the connection, then returns a connected `BluetoothSocket` `(https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html)`.

`BluetoothClass` `(https://developer.android.com/reference/android/bluetooth/BluetoothClass.html)`

Describes the general characteristics and capabilities of a Bluetooth device. This is a read-only set of properties that defines the device's classes and services. Although this information provides a useful hint regarding a device's type, the attributes of this class don't necessarily describe all Bluetooth profiles and services that the device supports.

`BluetoothProfile` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.html)

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 (#Profiles).

`BluetoothHeadset` (https://developer.android.com/reference/android/bluetooth/BluetoothHeadset.html)

Provides support for Bluetooth headsets to be used with mobile phones. This includes both the Bluetooth Headset profile and the Hands-Free (v1.5) profile.

`BluetoothA2dp` (https://developer.android.com/reference/android/bluetooth/BluetoothA2dp.html)

Defines how high-quality audio can be streamed from one device to another over a Bluetooth connection using the Advanced Audio Distribution Profile (A2DP).

`BluetoothHealth` (https://developer.android.com/reference/android/bluetooth/BluetoothHealth.html)

Represents a Health Device Profile proxy that controls the Bluetooth service.

`BluetoothHealthCallback` (https://developer.android.com/reference/android/bluetooth/BluetoothHealthCallback.html)

An abstract class that you use to implement `BluetoothHealth` (https://developer.android.com/reference/android/bluetooth/BluetoothHealth.html) callbacks. You must extend this class and implement the callback methods to receive updates about changes in the application's registration state and Bluetooth channel state.

`BluetoothHealthAppConfiguration` (https://developer.android.com/reference/android/bluetooth/BluetoothHealthAppConfiguration.html)

Represents an application configuration that the Bluetooth Health third-party application registers to communicate with a remote Bluetooth health device.

`BluetoothProfile.ServiceListener` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.ServiceListener.html)

An interface that notifies `BluetoothProfile` (https://developer.android.com/reference/android/bluetooth/BluetoothProfile.html) interprocess communication (IPC) clients when they have been connected to or disconnected from the internal service that runs a particular profile.

Follow @AndroidDev on Twitter

Follow Android Developers on Google+

Check out Android Developers on YouTube