

# Bluetooth Low Energy in Android Java

1st Edition

Tony Gaitatzis

BackupBrain Publishing, 2017

ISBN: 978-1-7751280-4-5

[backupbrain.co](http://backupbrain.co)

# **Bluetooth Low Energy in Android Java**

by Tony Gaitatzis

Copyright © 2015 All Rights Reserved

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review. For permission requests, write to the publisher, addressed “Bluetooth Arduino Book Reprint Request,” at the address below.

[backupbrain@gmail.com](mailto:backupbrain@gmail.com)

This book contains code samples available under the MIT License, printed below:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

(This page intentionally left blank)

# Dedication

*To Sierra, for lending me a phone  
and Bruno for being much more thorough than I could ever be.*

(This page intentionally left blank)

# Preface

Thank you for buying this book. I'm excited to have written it and more excited that you are reading it.

I started with Bluetooth Low Energy in 2011 while making portable brain imaging technology. Later, while working on a friend's wearable electronics startup, I ended up working behind teh scenes on the TV show America's Greatest Makers in the Spring of 2016.

Coming from a web programming background, I found the mechanics and nomenclature of BLE confusing and cryptic. After immersing myself in it for a period of time I acclimated to the differences and began to appreciate the power behind this low-power technology.

Unlike other wireless technologies, BLE can be powered from a coin cell battery for months at a time - perfect for a wearable or Internet of Things (IoT) project! Because of its low power and short data transmissions, it is great for transmitting bite size information, but not great for streaming data such as sound or video.

Good luck and enjoy!

# Conventions Used in This Book

Every developer has their own coding conventions. I personally believe that well-written code is self-explanatory. Moreover, consistent and organized coding conventions let developers step into each other's code much more easily, enabling them to reliably predict how the author has likely organized and implemented a feature, thereby making it easier to learn, collaborate, fix bugs and perform upgrades.

The coding conventions I used in this book is as follows:

Inline comments are as follows:

```
// inline comments
```

Multiline comments follow the JavaDoc standard:

```
/**  
 * This is a multiline comment  
 * It features more than one line of comment  
 */
```

Constants are written in all capitals:

```
public static final int CONSTANT_NAME = 0x01;
```

Local variables are written in Camel Case:

```
int MembervariableName = 1;
```

Member variables are written in Camel Case with a lowercase “m” preceding the name.

```
private int mMembervariableName = 1;
```

Function declarations are in Camel Case. In cases where there isn’t enough space to write the whole function on a line, parameters are written on another line:

```
void shortFunction() {  
}  
  
void superLongFunctionName(int parameterOne,  
    int parameterTwo) {  
    ...  
}
```

Class names are in Camel Case with the first character in upper case

```
public className {  
    ...  
}
```

Long lines will be broken with a backslash (\ ) and the next line will be indented:

```
static final String SOME_REALLY_LONG_VARIABE_NAME_WILL_BE_BROKEN = \  
    "onto the next line";
```

# Introduction

In this book you will learn the basics of how to program a Bluetooth Low Energy Peripheral on an nRF microcontroller, culminating in three projects:

- An iBeacon
- An Echo Server
- A Remote Controlled Device

Through the course of the book you will learn important concepts that relate to:

- How Bluetooth Low Energy works,
- How data is sent and received
- Common paradigms for handling data

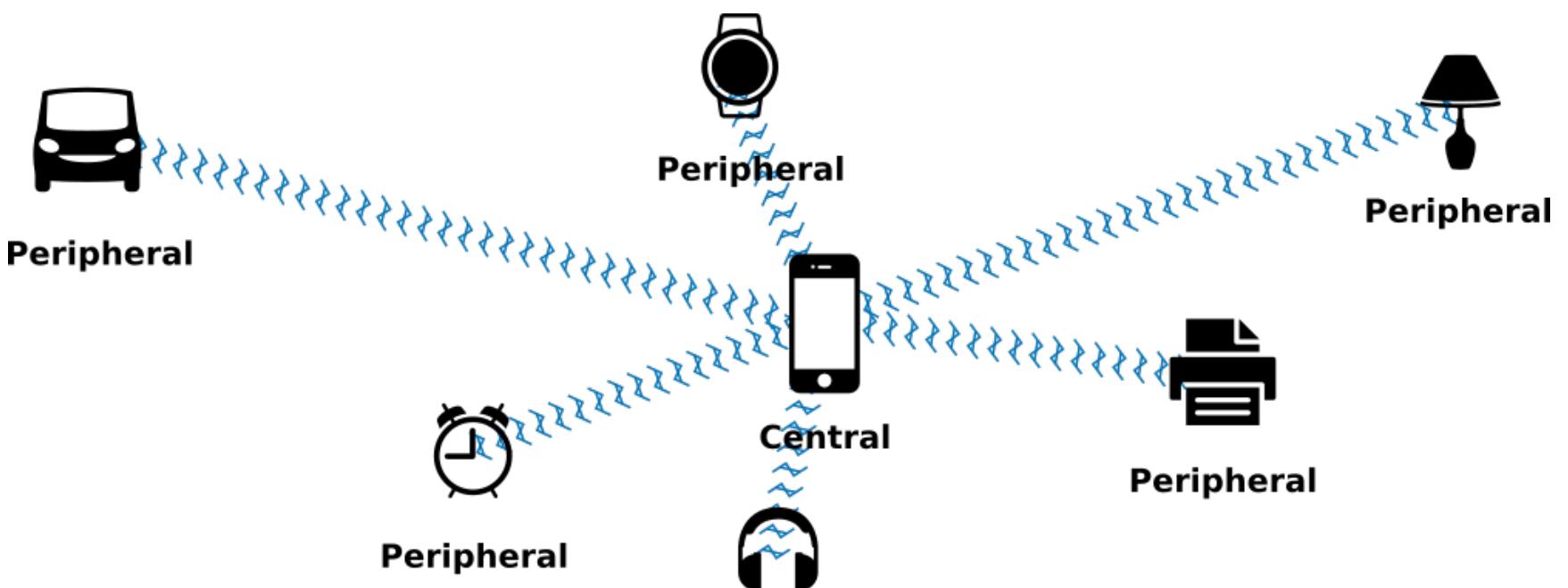
This book is an excellent read for anyone familiar with Arduino or C++ programming, who wants to build an Internet of Things device.

# Overview

Bluetooth Low Energy (BLE) is a digital radio protocol. Very simply, it works by transmitting radio signals from one computer to another.

Bluetooth supports a hub-and-spoke model of connectivity. One device acts as a hub, or “Central” in Bluetooth terminology. Other devices act as “Peripherals.”

A Central may hold several simultaneous connections with a number of peripherals, but a peripheral may only hold one connection at a time ([Figure 1-1](#)). Hence the names Central and Peripheral.



**Figure 1-1. Bluetooth network topology**

For example, your smartphone acts as a Central. It may connect to a Bluetooth speaker, lamp, smartwatch, and fitness tracker. Your fitness tracker and speaker, both Peripherals, can only be connected to one smartphone at a time.

The Central has two modes: scanning and connected. The Peripheral has two modes: advertising and connected. The Peripheral must be advertising for the Central to see it.

# Advertising

A Peripheral advertises by advertising its device name and other information on one radio frequency, then on another in a process known as frequency hopping. In doing so, it reduces radio interference created from reflected signals or other devices.

# Scanning

Similarly, the Central listens for a server's advertisement first on one radio frequency, then on another until it discovers an advertisement from a Peripheral. The process is not unlike that of trying to find a good show to watch on TV.

The time between radio frequency hops of the scanning Central happens at a different speed than the frequency hops of the advertising Peripheral. That way the scan and advertisement will eventually overlap so that the two can connect.

Each device has a unique media access control address (MAC address) that identifies it on the network. Peripherals advertise this MAC address along with other information about the Peripheral's settings.

# Connecting

A Central may connect to a Peripheral after the Central has seen the Peripheral's advertisement. The connection involves some kind of handshaking which is handled by the devices at the hardware or firmware level.

While connected, the Peripheral may not connect to any other device.

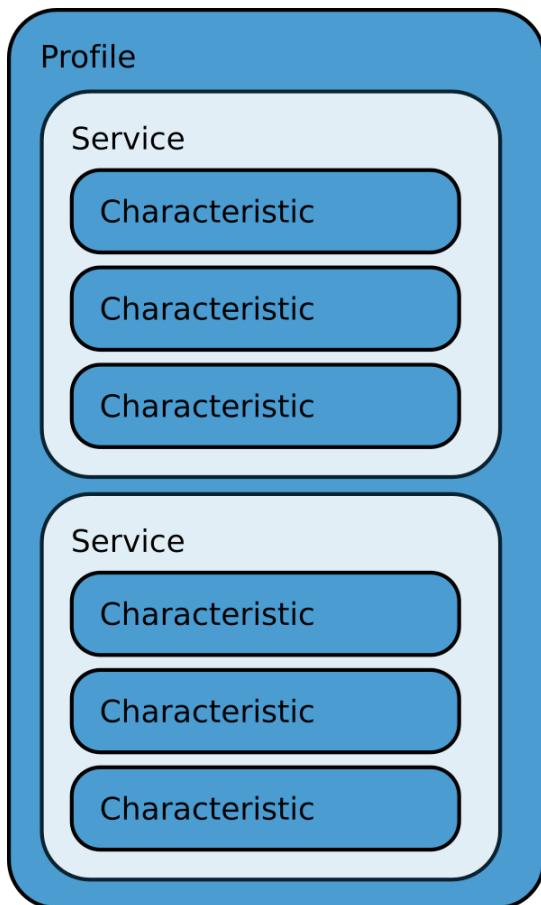
# Disconnecting

A Central may disconnect from a Peripheral at any time. The Peripheral is aware of the disconnection.

# Communication

A Central may send and request data to a Peripheral through something called a “Characteristic.” Characteristics are provided by the Peripheral for the Central to access. A Characteristic may have one or more properties, for example READ or WRITE. Each Characteristic belongs to a Service, which is like a container for Characteristics. This paradigm is called the Bluetooth Generic Attribute Profile (GATT).

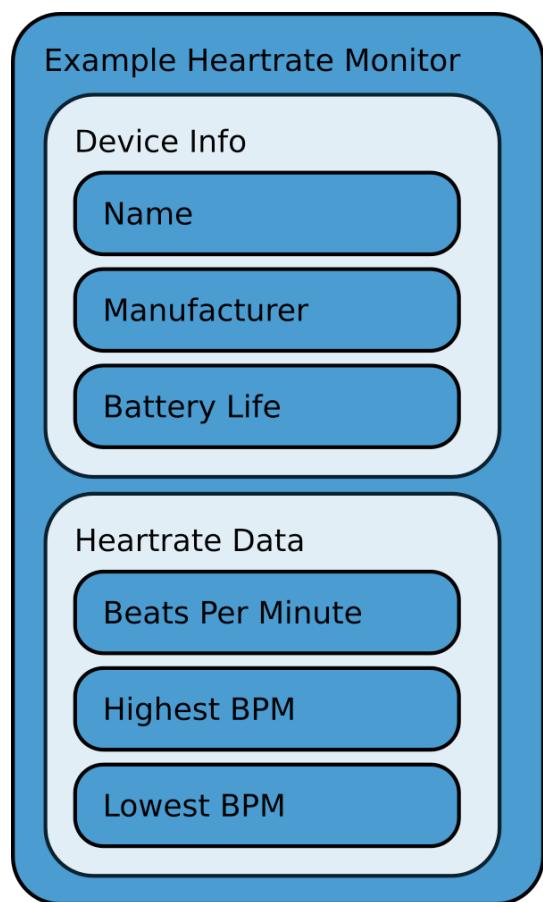
The GATT paradigm is laid out as follows ([Figure 1-2](#)).



**Figure 1-2. Example GATT Structure**

To transmit or request data from a Characteristic, a Central must first connect to the Characteristic’s Service.

For example, a heart rate monitor might have the following GATT profile, allowing a Central to read the beats per minute, name, and battery life of the server ([Figure 1-3](#)).



**Figure 1-3. Example GATT structure for a heart monitor**

In order to retrieve the battery life of the Characteristic, the Central must be connected also to the Peripheral's "Device Info" Service.

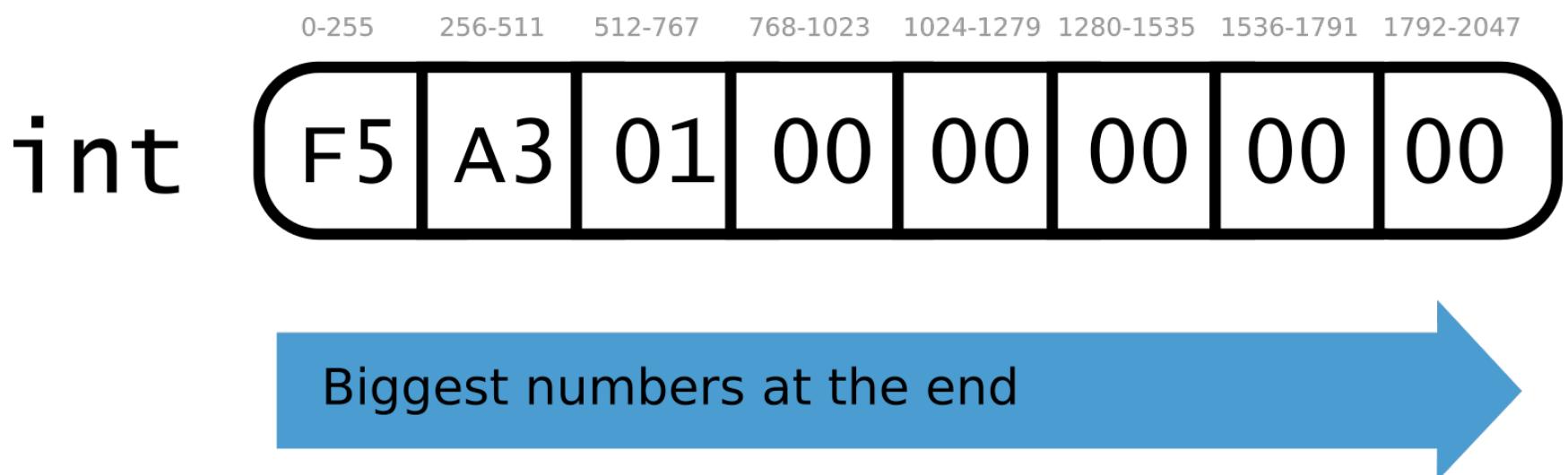
Because a Characteristic is provided by a Peripheral, the terminology refers to what can be done to the Characteristic. A "write" occurs when data is sent to the Characteristic and a "read" occurs when data is downloaded from the Characteristic.

To reiterate, a Characteristic is a field that can be written to or read from. A Service is a container that may hold one or more Characteristics. GATT is the layout of these Services and Characteristics. Characteristic can be written to or read from.

# Byte Order

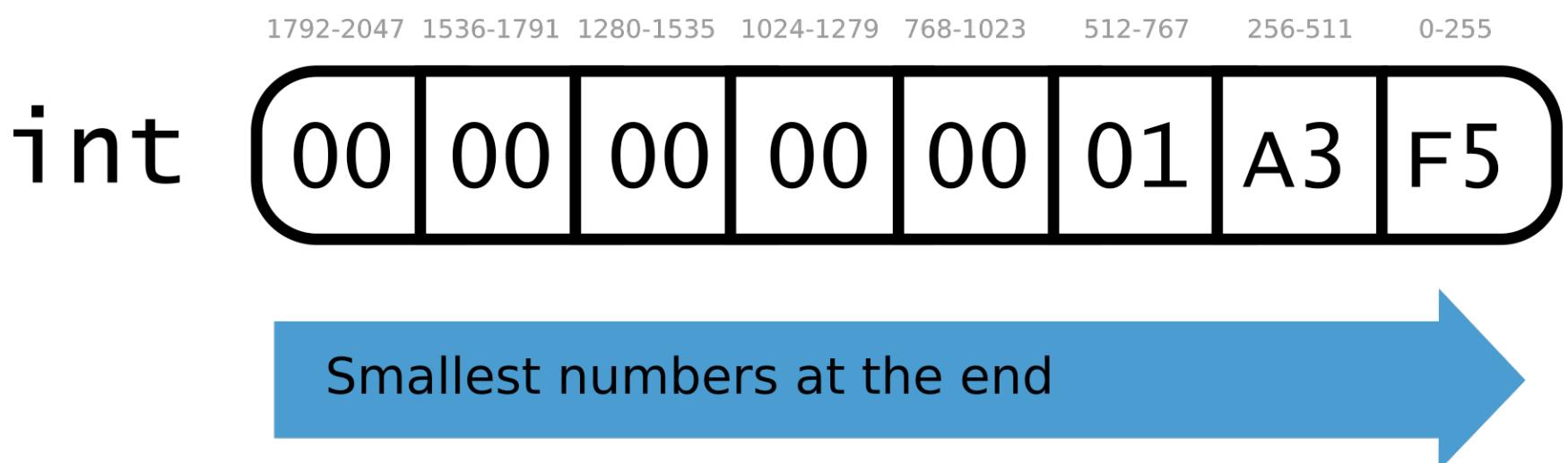
Bluetooth orders data in both Big-Endian and Little-Endian depending on the context.

During advertisement, data is transmitted in Big Endian, with the most significant bytes of a number at the end ([Figure 1-4](#)).



**Figure 1-4. Big Endian byte order**

Data transfers inside the GATT however are transmitted in Little Endian, with the least significant byte at the end ([Figure 1-5](#)).



**Figure 1-5. Little Endian byte order**

# Permissions

A Characteristic grants certain Permissions of the Central. These permissions include the ability to read and write data on the Characteristic, and to subscribe to Notifications.

# Descriptors

Descriptors describe the configuration of a Characteristic. The only one that has been specified so far is the “Notification” flag, which lets a Central subscribe to Notifications.

# UUIDs

A UUID, or Universally Unique IDentifier is a very long identifier that is likely to be unique, no matter when the UUID was created or who created it.

BLE uses UUIDs to label Services and Characteristics so that Services and Characteristics can be identified accurately even when switching devices or when several Characteristics share the same name.

For example, if a Peripheral has two “Temperature” Characteristics - one for Celsius and the other in Fahrenheit, UUIDs allow for the right data to be communicated.

UUIDs are usually 128-bit strings and look like this:

ca06ea56-9f42-4fc3-8b75-e31212c97123

But since BLE has very limited data transmission, 16-bit UUIDs are also supported and can look like this:

0x1815

Each Characteristic and each Service is identified by its own UUID. Certain UUIDs are reserved for specific purposes.

For example, UUID 0x180F is reserved for Services that contain battery reporting Characteristics.

Similarly, Characteristics have reserved UUIDs in the Bluetooth Specification.

For example, UUID 0x2A19 is reserved for Characteristics that report battery levels.

A list of UUIDs reserved for specific Services can be found in ***Appendix IV: Reserved GATT Services***.

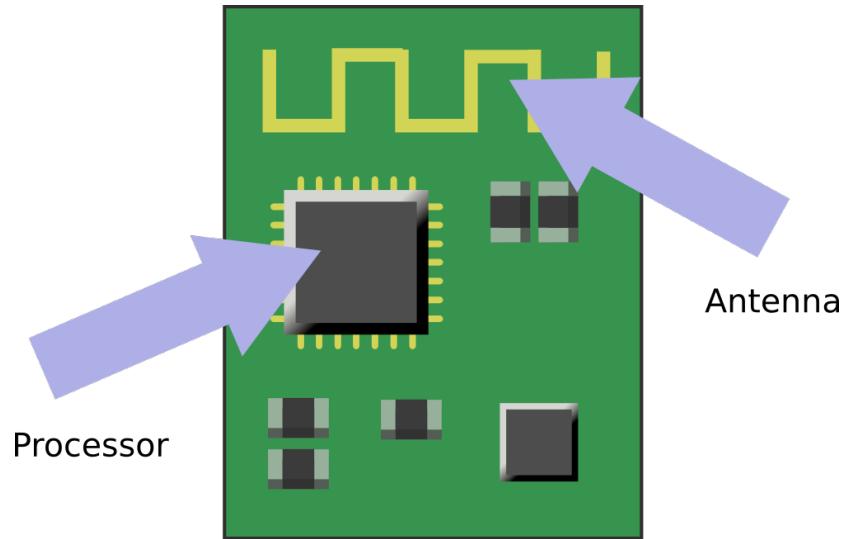
A list of UUIDs reserved for specific Characteristics can be in ***Appendix V: Reserved GATT Characteristics***.

If you are unsure what UUIDs to use for a project, you are safe to choose an unassigned service (e.g. 0x180C) for a Service and generic Characteristic (0x2A56).

Although the possibility of two generated UUIDs being the same are extremely low, programmers are free to arbitrarily define UUIDs which may already exist. So long as the UUIDs defining the Services and Characteristics do not overlap in the a single GATT Profile, there is no issue in using UUIDs that exist in other contexts.

## Bluetooth Hardware

All Bluetooth devices feature at least a processor and an antenna (Figure 1-6).



**Figure 1-6. Parts of a Bluetooth device**

The antenna transmits and receives radio signals. The processor responds to changes from the antenna and controls the antenna's tuning, the advertisement message, scanning, and data transmission of the BLE device.

## Power and Range

BLE has 20x2 Mhz channels, with a maximum 10 mW transmission power, 20 byte packet size, and 1 Mbit/s speed.

As with any radio signal, the quality of the signal drops dramatically with distance, as shown below (Figure 1-7).



**Figure 1-7. Distance versus Bluetooth Signal Strength**

This signal quality is correlated the Received Signal Strength Indicator (RSSI).

If the RSSI is known when the Peripheral and Central are 1 meter apart ( $A$ ), as well as the RSSI at the current distance ( $R$ ) and the radio propagation constant ( $n$ ). The distance between the Central and the Peripheral in meters ( $d$ ) can be approximated with this equation:

$$d \approx 10^{\frac{A-R}{10n}}$$

The radio propagation constant depends on the environment, but it is typically somewhere between 2.7 in a poor environment and 4.3 in an ideal environment.

Take for example a device with an RSSI of 75 at one meter, a current RSSI reading 35, with a propagation constant of 3.5:

$$d \approx 10^{\frac{75-35}{10 \times 3.5}}$$

$$d \approx 10^{\frac{40}{35}}$$

$$d \approx 14$$

Therefore the distance between the Peripheral and Central is approximately 14 meters.

# Introducing Arduino

Android makes it easy for developers to get into making apps because it doesn't have any developer registration costs, and there is no vetting process for publishing apps.

That means developers can produce apps and share them with their friends without a lot of work.

Android, as with all modern smartphones, is designed to support Bluetooth Low Energy.

We will be using Android to learn how to communicate between the Android and the Arduino using Bluetooth Low Energy (BLE). Although the examples in this book are relatively simple, the app potential of this technology is amazing. To program the Android, you will need Android Studio, the Google Android IDE.

Android Studio can be downloaded from <http://developer.android.com/sdk>.

Although Android Studio is easy to download, it can be complicated to install, because it relies on the Java Development Kit (JDK).

At the time of this writing, you must download JDK 7 or higher, available at <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> (Figure 2-1).

<b>Java SE Development Kit 8u73</b>		
<b>You must accept the Oracle Binary Code License Agreement for Java SE to download this software.</b>		
<input type="radio"/> Accept License Agreement	<input checked="" type="radio"/> Decline License Agreement	
<b>Product / File Description</b>	<b>File Size</b>	<b>Download</b>
Linux ARM 32 Hard Float ABI	77.73 MB	<a href="#">jdk-8u73-linux-arm32-vfp-hflt.tar.gz</a>
Linux ARM 64 Hard Float ABI	74.68 MB	<a href="#">jdk-8u73-linux-arm64-vfp-hflt.tar.gz</a>
Linux x86	154.75 MB	<a href="#">jdk-8u73-linux-i586.rpm</a>
Linux x86	174.91 MB	<a href="#">jdk-8u73-linux-i586.tar.gz</a>
Linux x64	152.73 MB	<a href="#">jdk-8u73-linux-x64.rpm</a>
Linux x64	172.91 MB	<a href="#">jdk-8u73-linux-x64.tar.gz</a>
Mac OS X x64	227.25 MB	<a href="#">jdk-8u73-macosx-x64.dmg</a>
Solaris SPARC 64-bit (SVR4 package)	139.7 MB	<a href="#">jdk-8u73-solaris-sparcv9.tar.Z</a>
Solaris SPARC 64-bit	99.08 MB	<a href="#">jdk-8u73-solaris-sparcv9.tar.gz</a>
Solaris x64 (SVR4 package)	140.36 MB	<a href="#">jdk-8u73-solaris-x64.tar.Z</a>
Solaris x64	96.78 MB	<a href="#">jdk-8u73-solaris-x64.tar.gz</a>
Windows x86	181.5 MB	<a href="#">jdk-8u73-windows-i586.exe</a>
Windows x64	186.84 MB	<a href="#">jdk-8u73-windows-x64.exe</a>

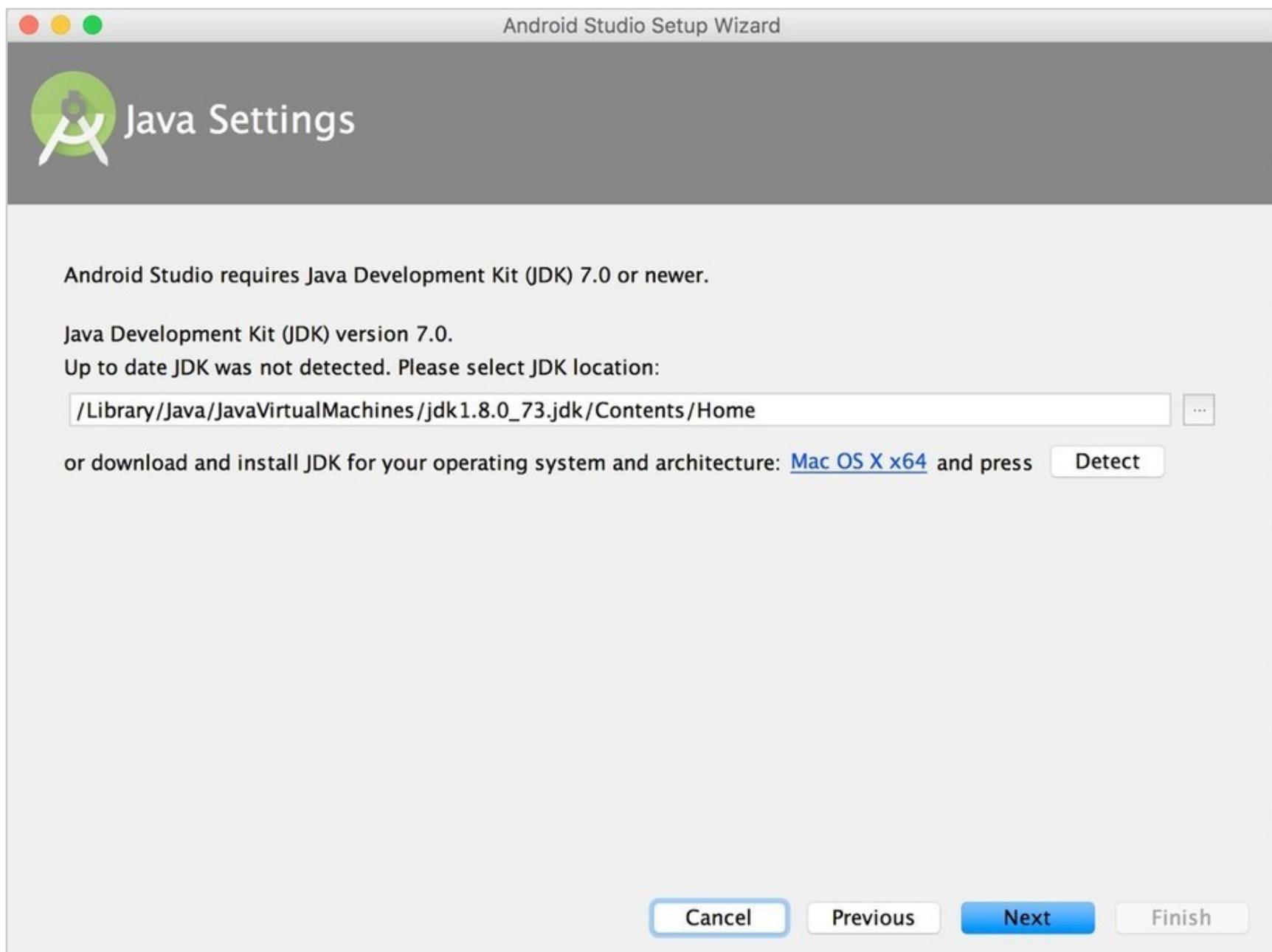
**Figure 2-1. Java Development Kit versions**

Accept the License Agreement and choose the link that matches your system platform.

Install the JDK, then install Android Studio.

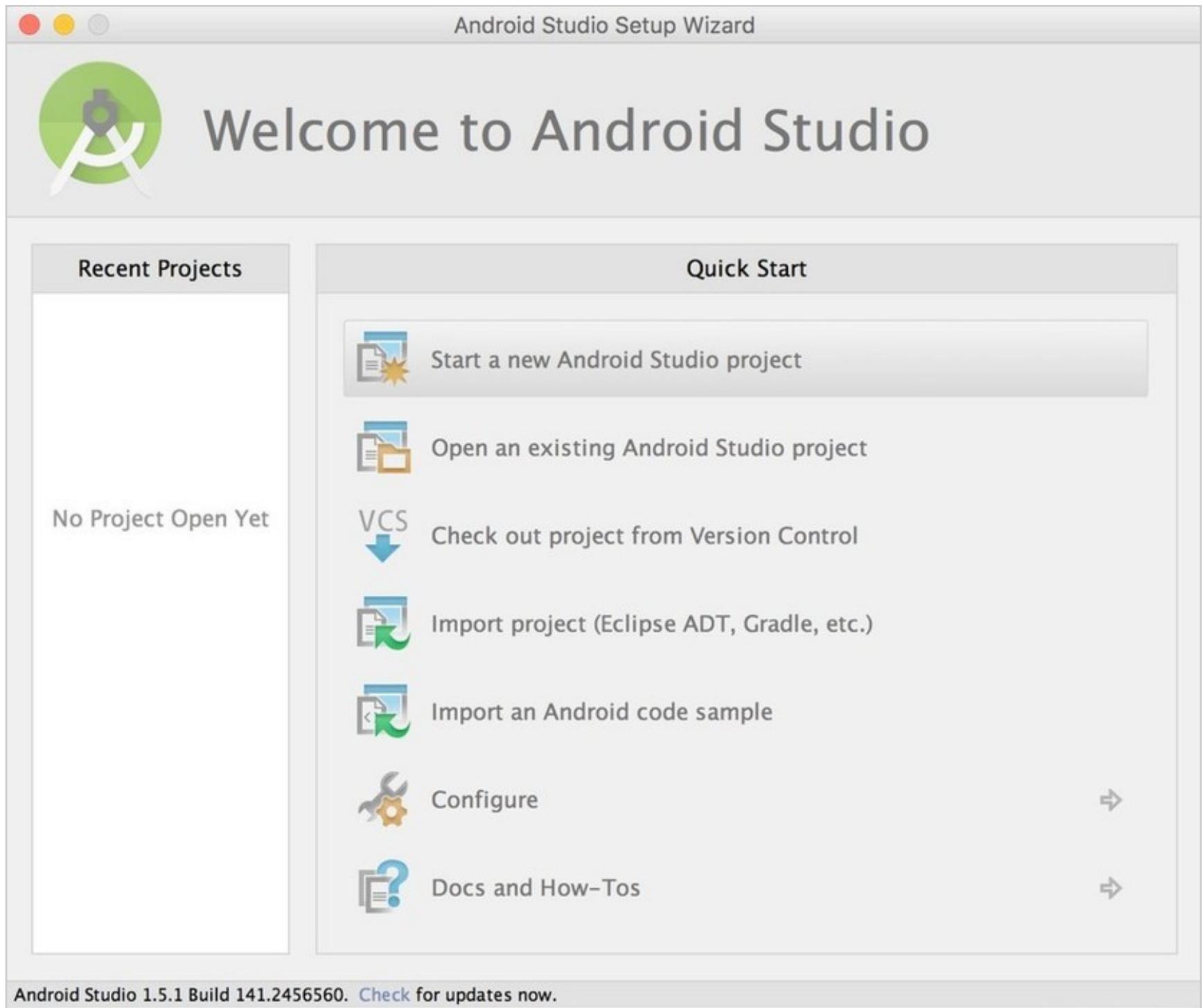
Android Studio will look for the JDK. You must find the installation folder of your newly installed JDK.

From there, continue the Android Studio installation with appropriate answers for your system ([Figure 2-2](#)).



**Figure 2-2. Android Studio installation**

When you are done installing, you will get a screen that looks like this ([Figure 2-3](#)).



**Figure 2-3. Starting a new Android project**

You are now ready to begin programming Bluetooth Low Energy Android Apps.

# API Compatibility

Android phones were first created by Google in 2008. Periodically, Google periodically releases updates that enable newer or more advanced features to be programmed. Each feature release is called an API level. Higher API levels have newer and more improved features but take time before users adopt them. Therefore, not every user has access to the newest features.

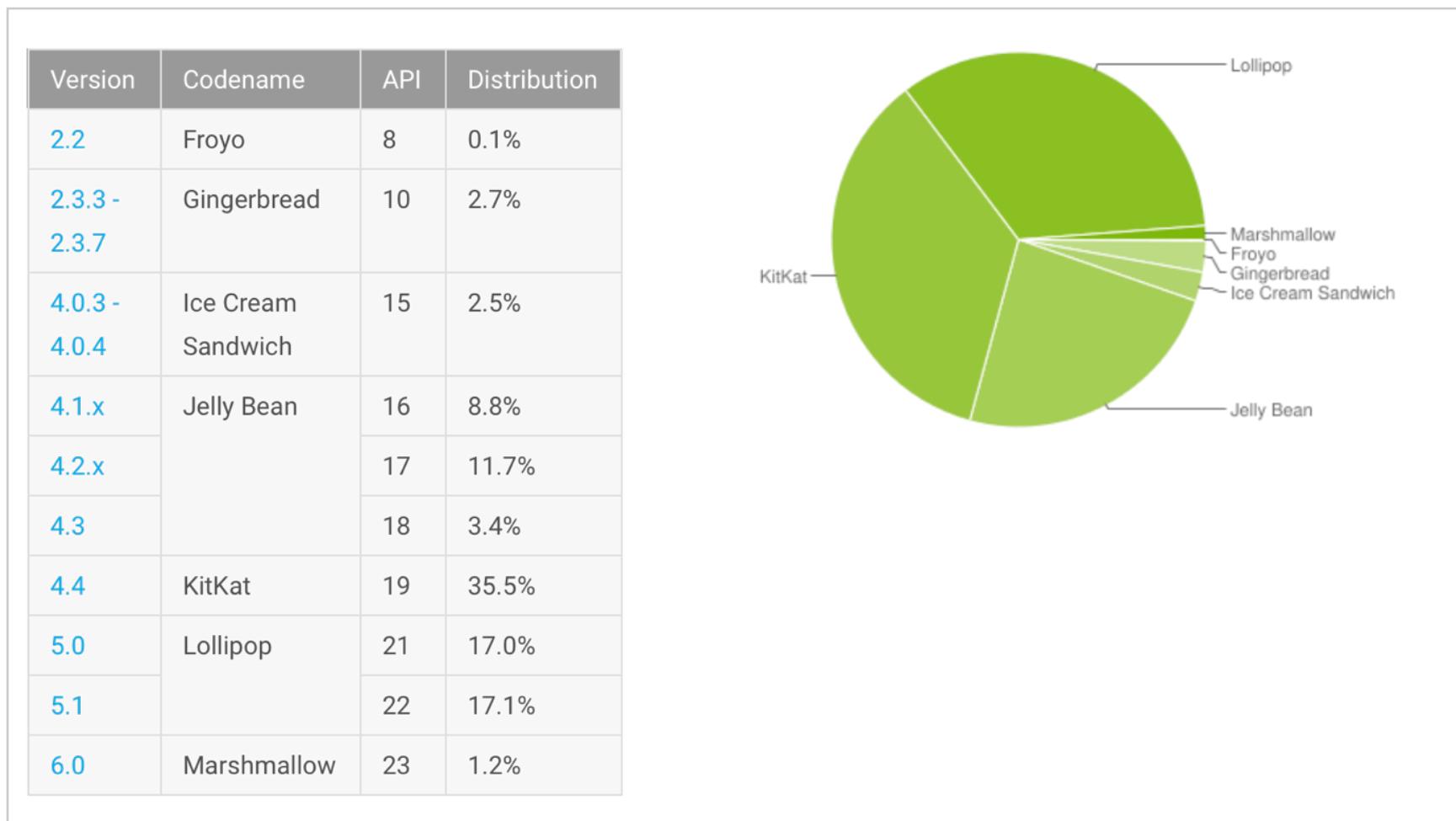
Bluetooth Low Energy was implemented in Android API level 18. At the time of this writing, the latest API level is 23, which corresponds to user version 6, Android “Marshmallow.” Bluetooth Central mode is supported on most devices with API level 18 or higher.

Bluetooth Peripheral mode is only supported in API level 21 (Android 5) or greater. Even with this API level, many phones do not have hardware support. A non-exhaustive list of supported hardware is available in Appendix VII: Android Peripheral Mode Support.

When the latest API level is used for development fewer people can use the program. When an older API level is used, more people can use the program but fewer features are supported. For this reason, choosing an API level is a tension between supporting more users and supporting more features.

We will support Android API level 18 and higher for Central mode, and API level 21 or higher for Peripheral mode. These are the oldest version of the Android API that supports Bluetooth Low Energy but has the most compatible devices.

As you can see from the graph below, nearly 75% of Android users have phones that support API level 18 or higher as of February 2016 ([Figure 2-4](#)).



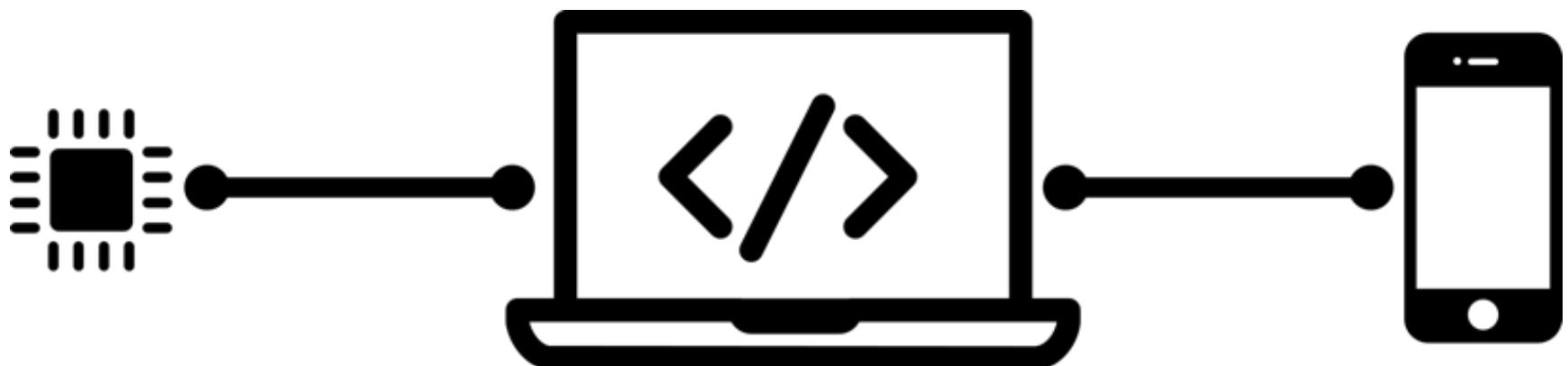
**Figure 2-4. Android IDE Distribution. Source:** [Android Developer Center](#)

# Bootstrapping

The first thing to do in any software project is to become familiar with the environment.

Because we are working with Bluetooth, it's important to learn how to initialize the Bluetooth radio and report what the program is doing.

Both the Central and Peripheral talk to the computer over USB when being programmed. That allows you to report errors and status messages to the computer when the programs are running ([Figure 3-1](#)).



**Figure 3-1. Programming configuration**

## Programming the Central

This chapter details how to create a Central App that turns the Bluetooth radio on. The Bluetooth radio requires permission to use and might be off by default.

The first thing to do before working with Bluetooth Low Energy in Android is to enable it. This is a two-step process:

- Request the Bluetooth Feature
- Enable Bluetooth Hardware

## Request Bluetooth Feature

Every time you access hardware on Android, first request access in the Android Manifest. Put the following two lines in the Manifest between the `<manifest></manifest>` tags.

```
...
<uses-feature
    android:name="android.hardware.bluetooth_le"  android:required="true" />
<uses-permission android:name="android.permission.BLUETOOTH"/>
...
```

## Enable Bluetooth

To turn on Bluetooth:

- Ask the Application if the BLE feature exists;
- Grab the Bluetooth Manager; and then
- Grab the Bluetooth Adapter.

```
...
// Does the BLE feature exist?
if (getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE))
{
```

```
// get the Bluetooth Manager  
final BluetoothManager bluetoothManager =  
    (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);  
// get Bluetooth Adapter  
BluetoothAdapter bluetoothAdapter = bluetoothManager.getAdapter();  
}  
...  
...
```

This can be done in the `onCreate()` method of the Activity. If Bluetooth Low Energy is not supported by the hardware, an error will happen then.

## Check if Bluetooth is Enabled

The user might turn the Bluetooth radio off any time. Therefore, every time the Application resumes, the Activity needs to check if check if Bluetooth is still enabled or has been disabled, using this function.

```
boolean bluetoothEnabled = getBluetoothAdapter().isEnabled();
```

The Activity can alert the user that the Bluetooth Radio is off, or the it can ask the user to to enable it. The Activity can enable Bluetooth programmatically using the `BluetoothAdapter.ACTION_REQUEST_ENABLE` Intent.

```
private final static int REQUEST_ENABLE_BT = 1;  
if (bluetoothEnabled) {  
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);  
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);  
}
```

It takes a moment for Bluetooth to turn on. To prevent trying to access Bluetooth before it's ready, the Activity need to subscribe to a `AdvertiseReceiver`. The `AdvertiseReceiver` will alert the Activity to changes in the Bluetooth radio status.

```

IntentFilter filter = new \
    IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
registerReceiver(mReceiver, filter);

private final AdvertiseReceiver mReceiver = new AdvertiseReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
            final int state = intent.getIntExtra(
                BluetoothAdapter.EXTRA_STATE,
                BluetoothAdapter.ERROR
            );
            switch (state) {
                case BluetoothAdapter.STATE_OFF:
                    // bluetooth radio has been switched off
                    break;
                case BluetoothAdapter.STATE_TURNING_OFF:
                    break;
                case BluetoothAdapter.STATE_ON:
                    // bluetooth radio has switched on
                    break;
                case BluetoothAdapter.STATE_TURNING_ON:
                    break;
            }
        }
    }
};

```

To turn on Bluetooth within the App, the BLUETOOTH\_ADMIN permission must be requested in the Manifest. Do this by putting the following line between <manifest></manifest> in the Manifest.

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

Now the app can turn on Bluetooth when it's been disabled.

# Putting It All Together

Create a new project called ExampleBle. Create packages and classes so that the project structure resembles this:

Your code structure should now look like this ([Figure 3-2](#)).

```
app/
  manifests/
    AndroidManifest.xml
java/
  example.com.exampleble/
    ble/
      BleCommManager
      MainActivity
res/
  layout/
    activity_main.xml
  menu/
    menu_main.xml
  values/
    strings.xml
```

**Figure 3-2. Project Structure**

The BleCommManager is responsible for general Android radio functions such as turning the radio on and off.

Therefore, BleCommManager looks like this:

## Example 3-1. java/example.com.exampleble/BLECommManager.java

```
package example.com.exampleble.ble;
public class BleCommManager {
    private static final String TAG = BleCommManager.class.getSimpleName();
```

```

private BluetoothAdapter mBluetoothAdapter; // Andrdoid's Bluetooth Adapter
/***
 * Initialize the BleCommManager
 *
 * @param context the Activity context
 * @throws Exception Bluetooth Low Energy is not supported
 */
public BleCommManager(final Context context) throws Exception {
    // make sure Android device supports Bluetooth Low Energy
    if (!context.getPackageManager().
        hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
        throw new Exception("Bluetooth Not Supported");
    }
    // get a reference to the Bluetooth Manager class,
    // which allows us to talk to talk to the BLE radio
    final BluetoothManager bluetoothManager = (BluetoothManager)
        context.getSystemService(Context.BLUETOOTH_SERVICE);
    mBluetoothAdapter = bluetoothManager.getAdapter();
}
/***
 * Get the Android Bluetooth Adapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}
}

```

MainActivity will ask BleCommManager to turn the radio on when the app turns on. When the radio turns on, or if it is already on, the text in the mBluetoothStatusTV TextView will be changed to read “Bluetooth Active.”

### **Example 3-2. java/example.com.exampleble/ble/MainActivity.java**

```

package example.com.exampleble;
public class MainActivity extends AppCompatActivity {
    /** Constants */

```

```
private static final String TAG = MainActivity.class.getSimpleName();
private static final int REQUEST_ENABLE_BT = 1;
/** Bluetooth stuff */
private BleCommManager mBleCommManager;
/** UI Stuff */
private TextView mBluetoothStatusTV;
/** 
 * Load Activity for the first time
 * @param savedInstanceState
 */
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    loadUI();
}
private void loadUI() {
    mBluetoothStatusTV = (TextView) findViewById(R.id.bluetooth_status);
}
/** 
 * Turn on Bluetooth Radio notifications when App resumes
 */
@Override
public void onResume() {
    super.onResume();
    initializeBluetooth();
}
@Override
public void onPause() {
    super.onPause();
    try {
        unregisterReceiver(mBluetoothAdvertiseReceiver);
    } catch (IllegalArgumentException e) {
        Log.e(TAG, "receiver not registered");
    }
}
```

```

    }

}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu;
    // this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    return false;
}

/**
 * Bluetooth Radio has been turned on. Update UI
 */
private void onBluetoothActive() {
    mBluetoothStatusTV.setText(R.string.bluetooth_active);
}

/**
 * Initialize the Bluetooth Radio
 */
public void initializeBluetooth() {
    // notify when bluetooth is turned on or off
    IntentFilter filter =
        new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
    registerReceiver(mBluetoothAdvertiseReceiver, filter);
    try {
        mBleCommManager = new BleCommManager(this);
    } catch (Exception e) {
        Toast.makeText(this, "Could not initialize bluetooth",
            Toast.LENGTH_SHORT).show();
        Log.e(TAG, e.getMessage());
        finish();
    }
    // should prompt user to open settings if bluetooth is not enabled.
}

```

```

    if (mBleCommManager.getBluetoothAdapter().isEnabled()) {
        onBluetoothActive();
    } else {
        Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    }
}

/**
 * Keep track of changes tho the Bluetooth Radio
 */
private final AdvertiseReceiver mBluetoothAdvertiseReceiver =
    new AdvertiseReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
            final int state =
                intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
                    BluetoothAdapter.ERROR);
            switch (state) {
                case BluetoothAdapter.STATE_OFF:
                    initializeBluetooth();
                    break;
                case BluetoothAdapter.STATE_TURNING_OFF:
                    break;
                case BluetoothAdapter.STATE_ON:
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            onBluetoothActive();
                        }
                    });
                    break;
                case BluetoothAdapter.STATE_TURNING_ON:
                    break;
            }
        }
    }
}

```

```
        }
    }
}
};
```

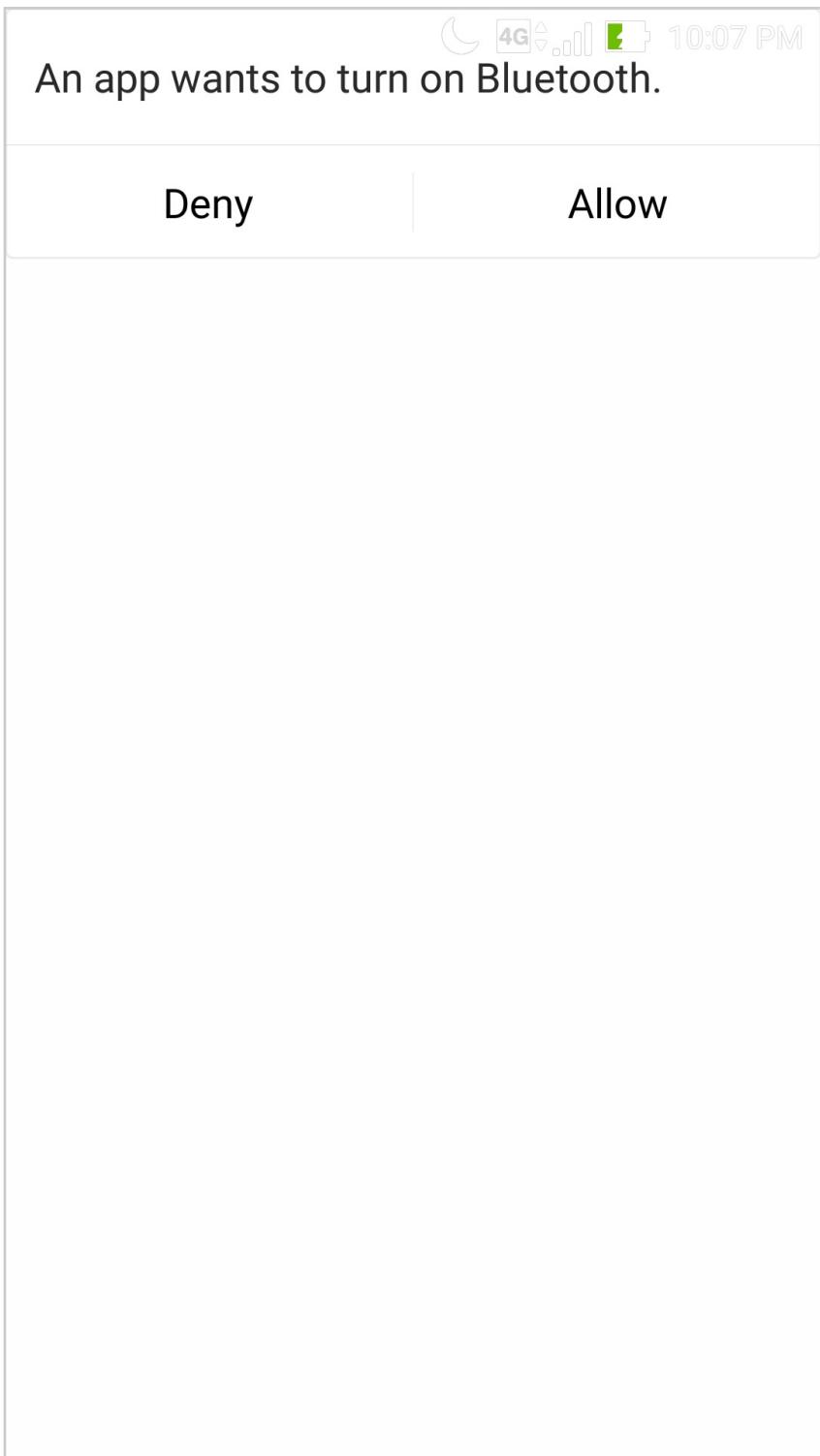
## Manifest

Finally, enable the Bluetooth functionality in the Android Manifest:

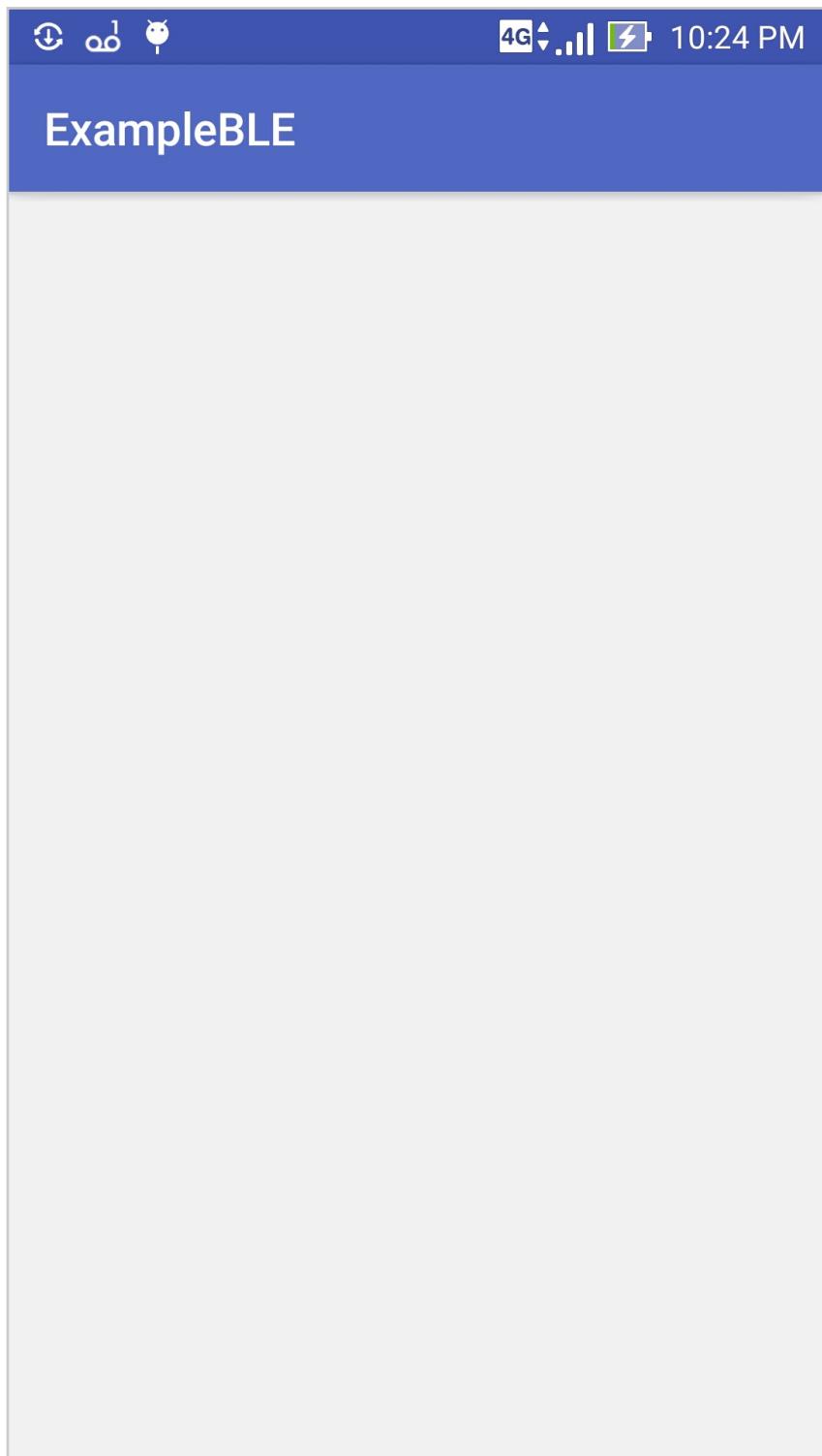
### Example 3-3. manifests/AndroidManifest.xml

```
...
<uses-feature
    android:name="android.hardware.bluetooth_le"  android:required="true" />
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
...
```

The resulting app will ask to turn the Bluetooth Radio on ([Figure 3-3](#)) and will run after the Bluetooth Radio has been turned on ([Figure 3-4](#)).



**Figure 3-3. Dialog to request user's permission to enable Bluetooth**



**Figure 3-4. Main app screen**

# Programming the Peripheral

Peripheral Mode in Android requires API level 21 (Android 5) or greater. The first thing to do before working with Bluetooth Low Energy as a Peripheral in Android is to enable it. This is a two-step process:

- Request the Bluetooth Feature
- Enable Bluetooth Hardware

## Request Bluetooth Feature

Every time you access hardware on Android, first request access in the Android Manifest. Put the following two lines in the Manifest between the `<manifest></manifest>` tags.

Example 3-7. manifests/AndroidManifest.xml

```
...
<uses-feature
    android:name="android.hardware.bluetooth_le" android:required="true" />
<uses-permission android:name="android.permission.BLUETOOTH"/>
...
```

## Enable Bluetooth

To turn on Bluetooth:

- Ask the Application if the BLE feature exists;

- Grab the Bluetooth Manager; and then
- Grab the Bluetooth Adapter.

```
...
// Does the BLE feature exist?
if (getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE))
{
    // get the Bluetooth Manager
    final BluetoothManager bluetoothManager =
        (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
    // get Bluetooth Adapter
    BluetoothAdapter bluetoothAdapter = bluetoothManager.getAdapter();
}
...
...
```

This can be done in the `onCreate()` method of the Activity or when initializing a new Peripheral. If Bluetooth Low Energy is not supported by the hardware, an error will happen then.

## Check if Bluetooth Peripheral Mode is available

Even if the Android device is API level 21 (Android 5) or higher, the hardware may not support Peripheral mode. Therefore, it is important to check if this feature is supported:

```
bool isPeripheralModeSupported = false;

// most devices support this method of checking. Some report true
if (!mBluetoothAdapter.isMultipleAdvertisementSupported()) {
    isPeripheralModeSupported = true;
}
```

```
if (isPeripheralModeSupported) {  
    mBluetoothAdvertiser =  
        mBluetoothAdapter.getBluetoothLeAdvertiser();  
  
    // For devices that incorrectly report true to the above condition  
    // this method works  
    if (mBluetoothAdvertiser == null) {  
        isPeripheralModeSupported = false  
    }  
}
```

The Activity can alert the user that the Bluetooth Radio is off, or it can ask the user to enable it. The Activity can enable Bluetooth programmatically using the BluetoothAdapter.ACTION\_REQUEST\_ENABLE Intent. Bluetooth must be on in order to Advertise a Peripheral

```
private final static int REQUEST_ENABLE_BT = 1;  
if (bluetoothEnabled) {  
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);  
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);  
}
```

It takes a moment for Bluetooth to turn on. To prevent trying to access Bluetooth before it's ready, the Activity need to subscribe to a AdvertiseReceiver. The AdvertiseReceiver will alert the Activity to changes in the Bluetooth radio status.

```
IntentFilter filter = new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);  
registerReceiver(mReceiver, filter);  
private final AdvertiseReceiver mReceiver = new AdvertiseReceiver() {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        final String action = intent.getAction();  
        if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {  
            final int state = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,  
                BluetoothAdapter.ERROR);  
        }  
    }  
}
```

```
        switch (state) {  
            case BluetoothAdapter.STATE_OFF:  
                // bluetooth radio has been switched off  
                break;  
            case BluetoothAdapter.STATE_TURNING_OFF:  
                break;  
            case BluetoothAdapter.STATE_ON:  
                // bluetooth radio has switched on  
                break;  
            case BluetoothAdapter.STATE_TURNING_ON:  
                break;  
        }  
    }  
};
```

To use turn on Bluetooth within the App, the BLUETOOTH\_ADMIN permission must be requested in the Manifest. Do this by putting the following line between <manifest></manifest> in the Manifest.

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

Now the app can turn on Bluetooth when it's been disabled.

## Putting It All Together

Create a new project called ExampleBlePeripheral. Create packages and classes so that the project structure resembles this:

Your code structure should now look like this ([Figure 3-5](#)).

```
app/
  manifests/
    AndroidManifest.xml
java/
  example.com.examplebleperipheral/
    ble/
      BlePeripheral
    MainActivity
res/
  layout/
    activity_main.xml
  menu/
    menu_main.xml
  values/
    strings.xml
```

**Figure 3-5. Project Structure**

## Resources

String Resources will define the text to be used in the App

### Example 3-4. res/values/strings.xml

```
<resources>
  <string name="app_name">ExampleBLEPeripheral</string>
  <string name="bluetooth_on">Bluetooth on</string>
</resources>
```

## Objects

The BlePeripheral is responsible for checking if Peripheral mode is supported.

Therefore, BlePeripheral looks like this:

### Example 3-5. java/example.com.exampleble/BlePeripheral.java

```
package example.com.examplebleperipheral.ble;

public class BlePeripheral {
    /** Constants */
    private static final String TAG = MyBlePeripheral.class.getSimpleName();

    /** Bluetooth stuff */
    private BluetoothAdapter mBluetoothAdapter;
    private BluetoothLeAdvertiser mBluetoothAdvertiser;

    /**
     * Construct a new Peripheral
     *
     * @param context The Application Context
     * @throws Exception Exception thrown if Bluetooth is not supported
     */
    public BlePeripheral(final Context context) throws Exception {
        // make sure Android device supports Bluetooth Low Energy
        if (!context.getPackageManager().hasSystemFeature(
            PackageManager.FEATURE_BLUETOOTH_LE))
        {
            throw new Exception("Bluetooth Not Supported");
        }

        // get a reference to the Bluetooth Manager class,
        // which allows us to talk to talk to the BLE radio
        final BluetoothManager bluetoothManager =
            (BluetoothManager) context.getSystemService(
                Context.BLUETOOTH_SERVICE
            );
        mBluetoothAdapter = bluetoothManager.getAdapter();
    }
}
```

```

// Beware: this function doesn't work on some platforms
if(!mBluetoothAdapter.isMultipleAdvertisementSupported()) {
    throw new Exception ("Peripheral mode not supported");
}

mBluetoothAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();

// Use this method instead for better support
if (mBluetoothAdvertiser == null) {
    throw new Exception ("Peripheral mode not supported");
}

}

/***
 * Get the system Bluetooth Adapter
 *
 * @return BluetoothAdapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}
}

```

## Activities

MainActivity will ask to turn on the Bluetooth Radio, then instantiate a BlePeripheral. When the Bluetooth radio turns on, mBluetoothOnSwitch will switch to the on position.

### **Example 3-6. java/example.com.exampleble/ble/MainActivity.java**

```

package example.com.exampleble;

public class MainActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = MainActivity.class.getSimpleName();
    private static final int REQUEST_ENABLE_BT = 1;

```

```
/** Bluetooth Stuff */
private MyBlePeripheral mMyBlePeripheral;

/** UI Stuff */
private Switch mBluetoothOnswitch;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    // notify when bluetooth is turned on or off
    IntentFilter filter = \
        new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
    registerReceiver(mBleAdvertiseReceiver, filter);
    loadUI();
}

@Override
public void onResume() {
    super.onResume();
    initializeBluetooth();
}

@Override
public void onDestroy() {
    super.onDestroy();
    unregisterReceiver(mBleAdvertiseReceiver);
}

/**
 * Load UI components
 */
```

```

public void loadUI() {
    mBluetoothOnSwitch = (Switch) findViewById(R.id.bluetooth_on);
}

/**
 * Initialize the Bluetooth Radio
 */
public void initializeBluetooth() {
    // reset connection variables
    try {
        mMyBlePeripheral = new MyBlePeripheral(this);
    } catch (Exception e) {
        Toast.makeText(
            this,
            "Could not initialize bluetooth", Toast.LENGTH_SHORT
        ).show();
        Log.e(TAG, e.getMessage());
        finish();
    }
    mBluetoothOnSwitch.setChecked(
        mMyBlePeripheral.getBluetoothAdapter().isEnabled()
    );
    // should prompt user to open settings if Bluetooth is not enabled.
    if (!mMyBlePeripheral.getBluetoothAdapter().isEnabled()) {
        Intent enableBtIntent = \
            new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    }
}

/**
 * when the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mBleAdvertiseReceiver = \
    new AdvertiseReceiver()
{

```

```

@Override
public void onReceive(Context context, Intent intent) {
    final String action = intent.getAction();

    if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
        final int state = intent.getIntExtra(
            BluetoothAdapter.EXTRA_STATE,
            BluetoothAdapter.ERROR
        );
        switch (state) {
            case BluetoothAdapter.STATE_OFF:
                Log.v(TAG, "Bluetooth turned off");
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_OFF:
                break;
            case BluetoothAdapter.STATE_ON:
                Log.v(TAG, "Bluetooth turned on");
                break;
            case BluetoothAdapter.STATE_TURNING_ON:
                break;
        }
    }
}

```

The Main Activity layout features a toggle switch to show the Bluetooth radio state.

### **Example 3-7. res/layout/activity\_main.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"
        tools:showIn="@layout/activity_main" tools:context=".MainActivity">

        <LinearLayout
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
            <Switch
                android:clickable="false"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="@string/bluetooth_on"
                android:id="@+id/bluetooth_on" />
        </LinearLayout>
    </RelativeLayout>
```

```
</android.support.design.widget.CoordinatorLayout>
```

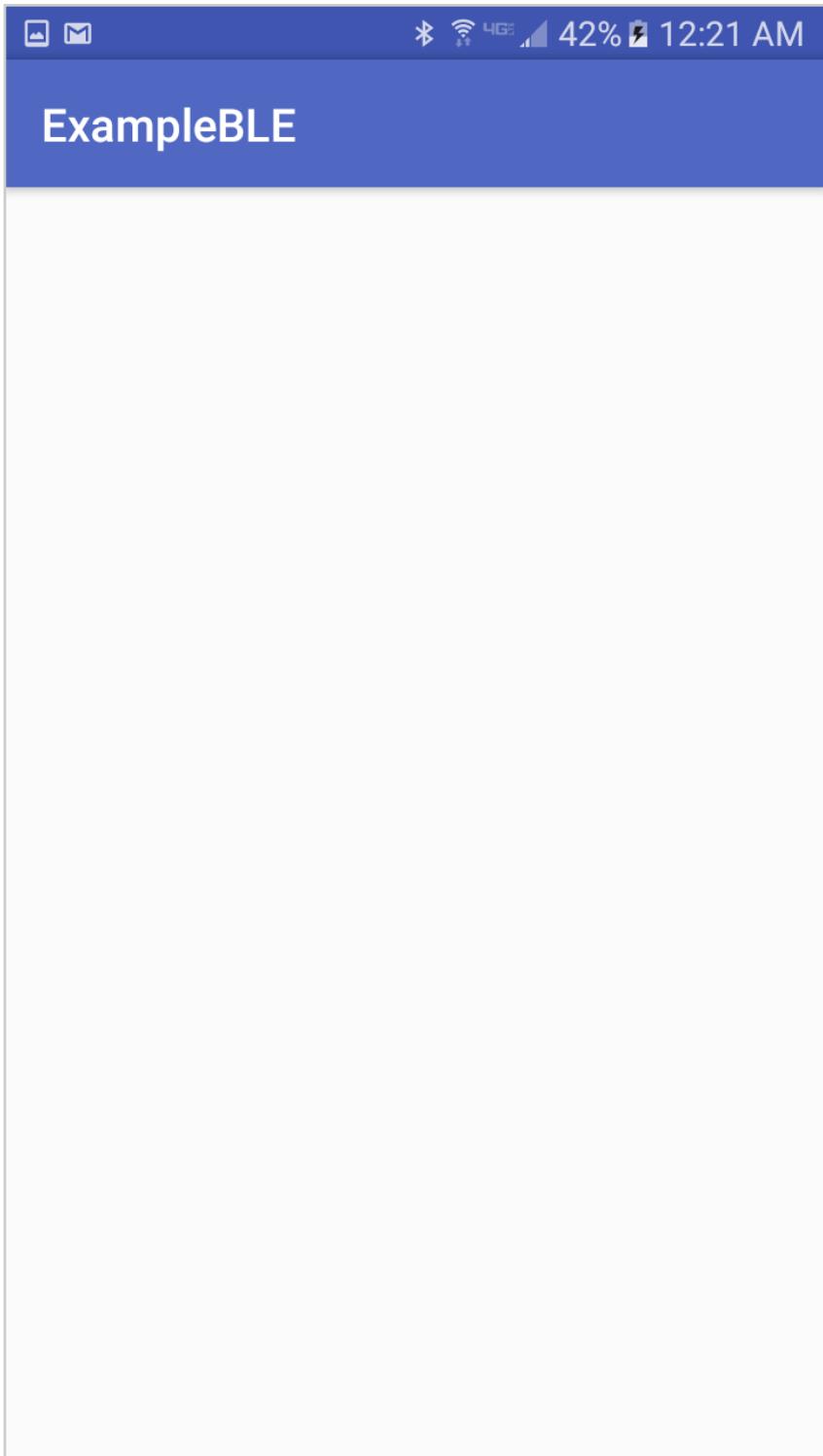
## Manifest

Finally, enable the Bluetooth functionality in the Android Manifest:

### Example 3-8. manifests/AndroidManifest.xml

```
...
<uses-feature
    android:name="android.hardware.bluetooth_le" android:required="true" />
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
...
```

The resulting app will be able to turn the Bluetooth Radio on ([Figure 3-6](#)).



**Figure 3-6. Main app screen**

## Example code

The code for this chapter is available online

at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter03>

# Scanning and Advertising

The first step to any Bluetooth Low Energy interaction is for the Peripheral to make the Central aware of its existence, through a process called Advertising.

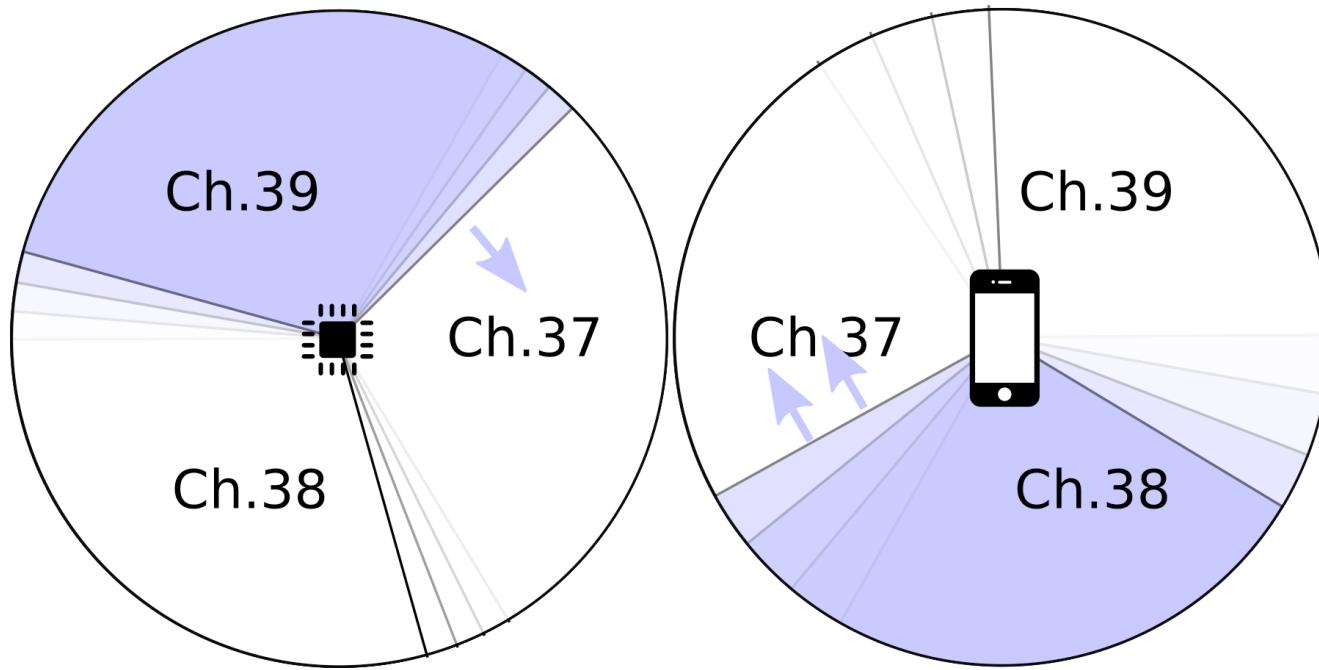
During the Advertising process, a Peripheral Advertises while a Central scans.

Bluetooth devices discover each other when they are tuned to the same radio frequency, also known as a Channel. There are three channels dedicated to device discovery in Bluetooth Low Energy: ([Table 4-1](#)):

**Table 4-1. Bluetooth Low Energy Discovery Radio Channels**

Channel	Radio Frequency
37	2402 Mhz
39	2426 Mhz
39	2480 Mhz

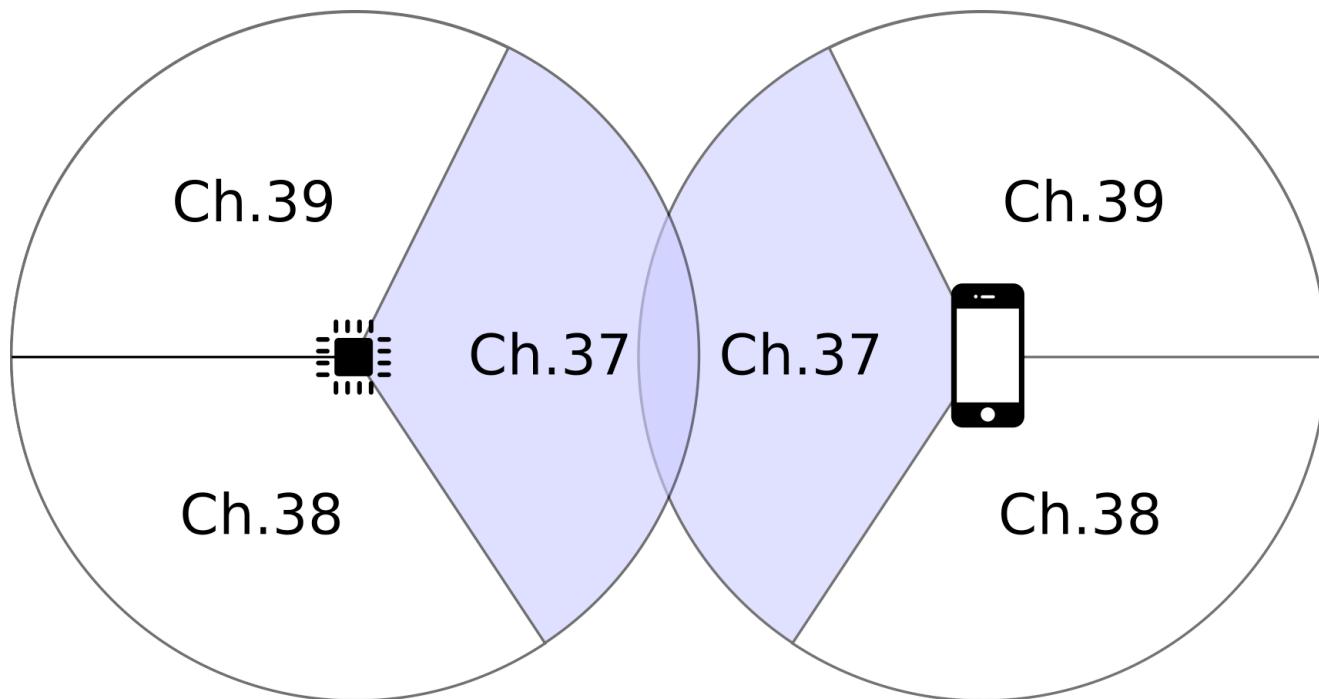
The peripheral will advertise its name and other data over one channel and then another. This is called frequency hopping ([Figure 4-1](#)).



**Figure 4-1. Advertise and scan processes**

Similarly, the Central listens for advertisements first on one channel and then another. The Central hops frequencies faster than the Peripheral, so that the two are guaranteed to be on the same channel eventually.

Peripherals may advertise from 100ms to 100 seconds depending on their configuration, changing channels every 0.625ms ([Figure 4-2](#)).



**Figure 4-2. Scan finds Advertiser**

Scanning settings vary wildly, for example scanning every 10ms for 100ms, or scanning for 1 second for 10 seconds.

Typically when a Central discovers advertising Peripheral, the Central requests a Scan Response from the Peripheral. In some cases, the Scan Response contains useful data. For example, iBeacons use Scan Response data to inform Centrals of each iBeacon's location without the Central needing to connect and download more data.

The first step to any Bluetooth Low Energy interaction is for the Peripheral to make the Central aware of its existence, through a process called Advertising.

Advertising reports the server name and other information one channel at a time until there are no more channels and the server repeats the process again at the first channel.

The Peripheral may start or stop advertising at any time.

## Programming the Central

The previous chapter showed how to access the Bluetooth hardware, specifically the BluetoothAdapter. This chapter will show how to scan for Bluetooth devices. This is done by scanning for Peripherals for a short period of time. During that time any time a Peripheral is discovered, the system will trigger a callback function. From there discovered Peripheral can be inspected.

The BluetoothAdapter gives you access to the BluetoothLeScanner, which lets you scan for nearby Peripherals.

```
// the device scanner allows us to scan for BLE Devices  
BluetoothLeScanner bluetoothScanner = bluetoothAdapter.getBluetoothLeScanner();
```

In Android, it's running CPU-intensive tasks in the main thread prevents the user interface from functioning properly. For this reason, Bluetooth scanning is best done in a separate thread.

In this example, the App will scan for BLE devices for 3-5 seconds. 5 seconds is a reasonable amount of time to assume that most devices will be discovered during the scanning process.

The method for implementing scanning changed in API level 21 (Android Lollipop).

## APIs 18-20

Prior to API 21, scanning was performed like this:

```
// run the scan for SCAN_PERIOD_MS milliseconds inside a thread
new Thread() {
    @Override
    public void run() {
        // Scan for BLE devices in a new thread
        // scan for devices for 5 seconds
        static final int SCAN_PERIOD_MS = 5000;
        // begin the scan
        bluetoothAdapter.startLeScan(scanCallback);
        try {
            // let the scan run for SCAN_PERIOD_MS
            Thread.sleep(SCAN_PERIOD_MS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        bluetoothAdapter.stopLeScan(scanCallback); // stop the scan
    }
}.start();
```

As each new Peripheral is discovered, an event is triggered in the LeScanCallback:

**Table 4-2. LeScanCallback**

Event	Description
onLeScan	Triggered when a new Peripheral is discovered

LeScanCallback can be implemented like this, to get the Peripheral's advertise name and MAC address:

```
private LeScanCallback scanCallback = new LeScanCallback() {  
    // This function executes when a device is found  
    @Override  
    public void onLeScan(final BluetoothDevice device,  
        int rssi, byte[] scanRecord) {  
        String advertiseName = device.getName();  
        String macAddress = device.getAddress();  
    }  
};
```

## APIs 21 and Up

Since API Level 21 (Lollipop), scanning is implemented like this:

```
// set the scan settings filters  
final ScanSettings settings = new ScanSettings.Builder()  
    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY).build();  
final List<ScanFilter> filters = new ArrayList<ScanFilter>();  
// Get the Bluetooth LE Scanner  
bluetoothLeScanner = mBluetoothAdapter.getBluetoothLeScanner();  
// run the scan inside a thread for SCAN_PERIOD milliseconds  
new Thread() {  
    @Override  
    public void run() {  
        // scan for devices for 5 seconds  
        static final int SCAN_PERIOD_MS = 5000;  
        bluetoothLeScanner.startScan(filters, settings, scanCallback);  
    }  
};
```

```

try {
    Thread.sleep(SCAN_PERIOD);
} catch (InterruptedException e) {
    e.printStackTrace();
}
bluetoothLeScanner.stopScan(scanCallback);
}.start();

```

The Scan settings can be changed depending on the desired tradeoff between battery life and scanning quality.

**Table 4-3 . ScanSettings**

Setting	Description
<b>SCAN_MODE_BALANCED</b>	Balance battery efficiency and scan speed
<b>SCAN_MODE_LOW_LATENCY</b>	Prefer scan speed over battery efficiency
<b>SCAN_MODE_LOW_POWER</b>	Prefer battery efficiency to scan speed
<b>SCAN_MODE OPPORTUNISTIC</b>	Eavesdrop for other scanner's scan response results

When a new Peripheral or Peripherals are discovered, or there is an error starting the scan process, corresponding events are triggered in the ScanCallback:

**Table 4-4. ScanCallback**

Event	Description
<b>onScanResult</b>	Triggered when a new Peripheral is discovered
<b>onBatchScanResults</b>	Triggered when several new Peripherals are discovered
<b>onScanFailed</b>	Triggered when scanning did not start successfully

A ScanResult contains information about the BLE device and advertise information. From this it is possible to determine the advertise name, mac address, type of Bluetooth device, and even the Received Signal Strength Indication (RSSI) - useful for determining how far away the Peripheral is.

LeScanCallback can be implemented like this, to get the Peripheral's advertise name and MAC address:

```
private ScanCallback scanCallback = new ScanCallback {  
    /**  
     * New Peripheral found.  
     *  
     * @param callbackType Tells how this callback was triggered  
     * @param scanResult information about the result,  
     *                      including Bluetooth Device  
     */  
  
    @Override  
    public void onScanResult(int callbackType, ScanResult scanResult) {  
        BluetoothDevice bluetoothDevice = result.getDevice();  
        String advertiseName = bluetoothDevice.getName();  
        String macAddress = bluetoothDevice.getAddress();  
        int rssi = result.getRssi();  
    }  
    /**  
     * New Peripherals found.  
     *  
     * @param results List: List of scan results that are previously scanned.  
     */  
  
    @Override  
    public abstract void onBatchScanResults(List<ScanResult> scanResults) {  
        // process each scanResult in the list  
    }  
    /**  
     * Problem initializing the scan. See the error code for reason  
     *  
     * @param errorCode int: Error code (one of SCAN_FAILED_*)  
    }
```

```

        *
        for scan failure.

    */

@Override
public abstract void onScanFailed(int errorCode) {
    switch (errorCode) {
        case SCAN_FAILED_ALREADY_STARTED:
            // Scan with same settings already started by app
            break;
        case SCAN_FAILED_APPLICATION_REGISTRATION_FAILED:
            // App cannot be registered
            break;
        case SCAN_FAILED_FEATURE_UNSUPPORTED:
            // Power optimization setting is not supported on this device
            break;
        default: // SCAN_FAILED_INTERNAL_ERROR
            // Internal error
    }
}
};


```

To support the widest array of Android devices, it may be useful to implement both API 18 and API 21 versions of the scanner. This can be done as follows:

```

static long SCAN_PERIOD_MS = 5000; // 5 seconds of scanning time
// Use BluetoothAdapter.startLeScan() for Android API 18, 19, and 20
if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
    // Scan for SCAN_PERIOD milliseconds.
    // at the end of that time, stop the scan.
    new Thread() {
        @Override
        public void run() {
            mBluetoothAdapter.startLeScan(bleScanCallbackv18);
            try {
                Thread.sleep(SCAN_PERIOD_MS);
            } catch (InterruptedException e) {
                // handle exception
            }
        }
    }.start();
}
};


```

```

        }

        mBluetoothAdapter.stopLeScan(bleScanCallbackv18);

    }.start();

} else { // use BluetoothLeScanner.startScan() for API 21 (Lollipop) or greater
    final ScanSettings settings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY).build();

    final List<ScanFilter> filters = new ArrayList<ScanFilter>();
    bluetoothLeScanner = mBluetoothAdapter.getBluetoothLeScanner();
    new Thread() {

        @Override
        public void run() {
            bluetoothLeScanner.startScan(
                filters,
                settings,
                bleScanCallbackv21
            );
            try {
                Thread.sleep(SCAN_PERIOD_MS);
            } catch (InterruptedException e) {
                // handle exception
            }
            bluetoothLeScanner.stopScan(bleScanCallbackv21);
        }
    }.start();
}

```

The scan callbacks can then be implemented as follows:

```

private LeScanCallback bleScanCallbackv18 = new LeScanCallback{ ... };

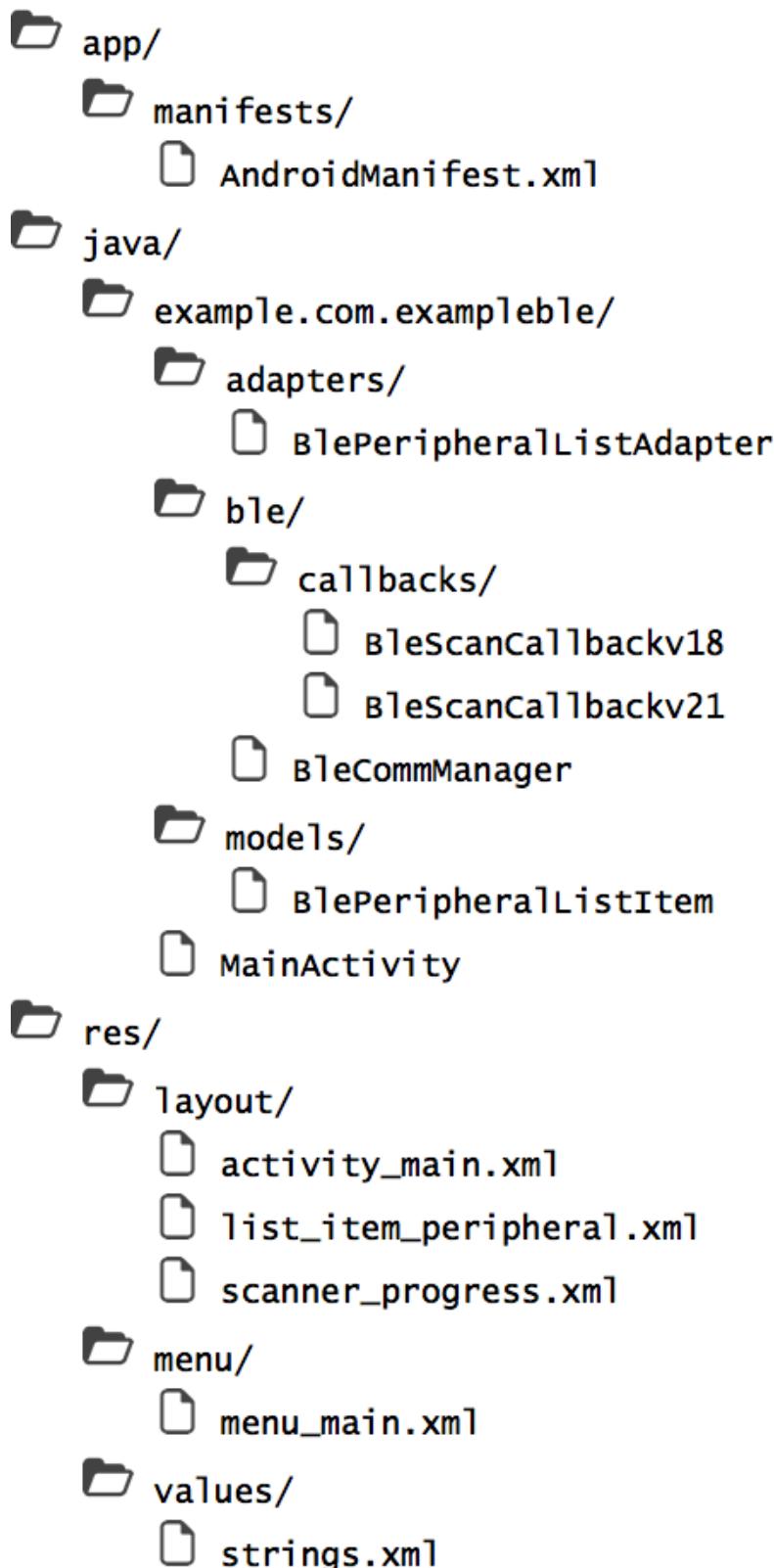
private ScanCallback bleScanCallbackv21 = new ScanCallback { ... };

```

# Putting It All Together

Create a new app called ExampleBleScanner, and copy everything from the previous example.

Create a file structure that looks like this ([Figure 4-3](#)).



**Figure 4-3. Project structure**

This example will feature some user interface elements, including a list adapter. The list adapter will populate a list of discovered Peripherals in the main Activity.

Custom scanner callbacks will be created, which respond to events when scanning has stopped.

## Resources

To begin, define some dimensions and strings that will be used by the rest of the app.

Add a text padding dimension:

### Example 4-1. res/values/dimens.xml

```
<resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
    <dimen name="text_padding">5dp</dimen>
</resources>
```

Add text for buttons and labels:

### Example 4-2. res/values/strings.xml

```
<resources>
    <string name="app_name">ExampleBLEScan</string>
    <string name="action_start_scan">Scan</string>
    <string name="action_stop_scan">Stop</string>
    <string name="scanning">Scanning...</string>
    <string name="peripheral_list_empty">No Peripherals Found</string>
</resources>
```

## Objects

Add the following functions to the BleCommManager to give it functionality to start and stop scanning for BLE devices.

### Example 4-3. java/example.com.exampleble/ble/BLECommManager.java

```
public class BleCommManager {  
    private static final String TAG = BleCommManager.class.getSimpleName();  
    private static final long SCAN_PERIOD = 5000; // 5 seconds of scanning time  
    private BluetoothAdapter mBluetoothAdapter; // Andrdoid's Bluetooth Adapter  
    private BluetoothLeScanner bluetoothLeScanner; // Ble scanner - API >= 21  
    private Timer mTimer = new Timer(); // scan timer  
  
    ...  
  
    /**  
     * Scan for Peripherals  
     *  
     * @param bleScanCallbackv18 APIv18 compatible ScanCallback  
     * @param bleScanCallbackv21 APIv21 compatible ScanCallback  
     * @throws Exception  
     */  
    public void scanForPeripherals(  
        final BleScanCallbackv18 bleScanCallbackv18,  
        final BleScanCallbackv21 bleScanCallbackv21) throws Exception  
    {  
        // Don't proceed if there is already a scan in progress  
        mTimer.cancel();  
        // Use BluetoothAdapter.startLeScan() for Android API 18, 19, and 20  
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {  
            // Scan for SCAN_PERIOD milliseconds.  
            // at the end of that time, stop the scan.  
            new Thread() {  
                @Override  
                public void run() {  
                    mBluetoothAdapter.startLeScan(bleScanCallbackv18);  
                    try {  
                        Thread.sleep(SCAN_PERIOD);  
                        mBluetoothAdapter.stopLeScan(bleScanCallbackv18);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        }  
    }
```

```
.start();

// alert the system that BLE scanning
// has stopped after SCAN_PERIOD milliseconds
mTimer = new Timer();
mTimer.schedule(new TimerTask() {

    @Override
    public void run() {
        stopScanning(bleScanCallbackv18, bleScanCallbackv21);
    }
}, SCAN_PERIOD);

} else { // use BluetoothLeScanner.startScan() for API >21 (Lollipop)
    final ScanSettings settings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
        .build();

    final List<ScanFilter> filters = new ArrayList<ScanFilter>();
    mBluetoothLeScanner = mBluetoothAdapter.getBluetoothLeScanner();
    new Thread() {
        @Override
        public void run() {
            mBluetoothLeScanner.startScan(
                filters,
                settings,
                bleScanCallbackv21
            );
            try {
                Thread.sleep(SCAN_PERIOD);
                mBluetoothLeScanner.stopScan(bleScanCallbackv21);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }.start();
    // alert the system that BLE scanning
    // has stopped after SCAN_PERIOD milliseconds
    mTimer = new Timer();
    mTimer.schedule(new TimerTask() {
```

```

    @Override
    public void run() {
        stopScanning(bleScanCallbackv18, bleScanCallbackv21);
    }
}, SCAN_PERIOD);

}

/***
 * Stop Scanning
 *
 * @param bleScanCallbackv18 APIv18 compatible ScanCallback
 * @param bleScanCallbackv21 APIv21 compatible ScanCallback
 */
public void stopScanning(
    final BleScanCallbackv18 bleScanCallbackv18,
    final BleScanCallbackv21 bleScanCallbackv21)
{
    mTimer.cancel();
    // propagate the onScanComplete through the system
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
        mBluetoothAdapter.stopLeScan(bleScanCallbackv18);
        bleScanCallbackv18.onScanComplete();
    } else {
        mBluetoothLeScanner.stopScan(bleScanCallbackv21);
        bleScanCallbackv21.onScanComplete();
    }
}
}

```

Define the two scan callbacks `BleScanCallbackv18`, and `BleScanCallbackv21`, which will describe how the `BleCommManager` should react to newly discovered Peripherals.

#### **Example 4-4. java/example.com.exampleble/ble/callbacks/BleScanCallbackv18.java**

```
package example.com.exampleble.ble.callbacks;
public abstract class BleScanCallbackv18 implements
    BluetoothAdapter.LeScanCallback {
    /**
     * New Perpheral found.
     *
     * @param bluetoothDevice The Peripheral Device
     * @param rssi The Peripheral's RSSI
     *           indicating how strong the radio signal is
     * @param scanRecord Other information about the scan result
     */
    //Override
    public abstract void onLeScan(final BluetoothDevice bluetoothDevice,
        int rssi, byte[] scanRecord);
    /**
     * BLE Scan complete
     */
    public abstract void onScanComplete();
}
```

#### **Example 4-5. java/example.com.exampleble/ble/callbacks/BleScanCallbackv21.java**

```
package example.com.exampleble.ble.callbacks;
public abstract class BleScanCallbackv21 extends ScanCallback {
    /**
     * New Perpheral found.
     *
     * @param callbackType int: Determines how this callback was triggered
     * @param result a Bluetooth Low Energy Scan Result
     */
    @Override
    public abstract void onScanResult(int callbackType, ScanResult result);
    /**
     * New Perpherals found.
     */
```

```

*
 * @param results List: List of scan results that are previously scanned.
 */
@Override
public abstract void onBatchScanResults(List<ScanResult> results);
/***
 * Problem initializing the scan. See the error code for reason
 *
 * @param errorCode int: Error code (one of SCAN_FAILED_*)
 *                      for scan failure.
 */
@Override
public abstract void onScanFailed(int errorCode);
/***
 * Scan has completed
 */
public abstract void onScanComplete();
}

```

## User Interface

In this app, a custom ListAdapter is used to visually represent each of the discovered Peripherals.

The `list_item_peripheral.xml` defines what each list item in the list of devices looks like. Ours will have three text fields, one for the device name, one for the address, and one for the RSSI (radio signal strength indicator).

### **Example 4-6. res/layout/list\_item\_peripheral.xml**

```

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

```

```

        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="2">
        <TextView
            android:id="@+id/advertise_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:paddingTop="@dimen/text_padding"/>
        <TextView
            android:id="@+id/mac_address"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="10sp"
            android:paddingTop="@dimen/text_padding"/>
    </LinearLayout>
    <TextView
        android:id="@+id/rssi"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="10sp"
        android:paddingTop="@dimen/text_padding"/>
</LinearLayout>

```

The BLEDevicesListItem is the representation of the data in each list\_item.xml list item.

#### **Example 4-7. java/example.com.exampleble/models/BlePeripheralListItem.java**

```

package example.com.exampleble.models;
public class BlePeripheralListItem {
    private int mItemId;
    private int mRssi;
    private BluetoothDevice mBluetoothDevice;
    public BlePeripheralListItem(BluetoothDevice bluetoothDevice) {

```

```

    mBluetoothDevice = bluetoothDevice;
}

public void setItemId(int id) { mItemId = id; }
public void setRssi(int rssi) { mRssi = rssi; }
public int getItemId() { return mItemId; }
public String getAdvertiseName() { return mBluetoothDevice.getName(); }
public String getMacAddress() { return mBluetoothDevice.getAddress(); }
public int getRssi() { return mRssi; }
public BluetoothDevice getDevice() { return mBluetoothDevice; }
}

```

To relay data to the list, the list adapter must be able to process data specific to the discovered Peripherals.

## Example

### 4-8. java/example.com.exampleble/adapters/BlePeripheralsListAdapter.java

```

package example.com.exampleble.adapters;

public class BlePeripheralsListAdapter extends BaseAdapter {
    private static String TAG = \
        BlePeripheralsListAdapter.class.getSimpleName();
    private ArrayList<BlePeripheralListItem> mBluetoothPeripheralListItems = \
        new ArrayList<BlePeripheralListItem>(); // list of Peripherals
    /**
     * How many items are in the ListView
     * @return the number of items in this ListView
     */
    @Override
    public int getCount() {
        return mBluetoothPeripheralListItems.size();
    }
    /**
     * Add a new Peripheral to the ListView
     *
     * @param bluetoothDevice Peripheral device information
     * @param rssi Peripheral's RSSI, indicating its radio signal quality
     */
}

```

```

*/
public void addBluetoothPeripheral(BluetoothDevice bluetoothDevice,
        int rssi) {
    // update UI stuff
    int listItemID = mBluetoothPeripheralListItems.size();
    BlePeripheralListItem listItem = \
        new BlePeripheralListItem(bluetoothDevice);
    listItem.setItemID(listItemID);
    listItem.setRssi(rssi);
    // add to list
    mBluetoothPeripheralListItems.add(listItem);
}

/**
 * Get current state of ListView
 * @return ArrayList of BlePeripheralListItems
 */
public ArrayList<BlePeripheralListItem> getItems() {
    return mBluetoothPeripheralListItems;
}

/**
 * Clear all items from the ListView
 */
public void clear() {
    mBluetoothPeripheralListItems.clear();
}

/**
 * Get the BlePeripheralListItem held at some position in the ListView
 *
 * @param position the position of a desired item in the list
 * @return the BlePeripheralListItem at some position
 */
@Override
public BlePeripheralListItem getItem(int position) {
    return mBluetoothPeripheralListItems.get(position);
}

@Override

```

```

public long getItemId(int position) {
    return mBluetoothPeripheralListItems.get(position).getItemId();
}

/**
 * This viewHolder represents what UI components are in each List Item
 */
public static class viewHolder{
    public TextView mAdvertiseNameTV;
    public TextView mMAddressTV;
    public TextView mRssiTV;
}

/**
 * Generate a new ListItem for some known position in the ListView
 *
 * @param position the position of the ListItem
 * @param convertView An existing List Item
 * @param parent The Parent ViewGroup
 * @return The List Item
 */
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    View v = convertView;
    viewHolder peripheralListView;
    // if this ListItem does not exist yet, generate it
    // otherwise, use it
    if(convertview == null) {
        // convert list_item_peripheral.xml to a view
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        v = inflater.inflate(R.layout.list_item_peripheral, null);
        // match the UI stuff in the list Item to what's in the xml file
        peripheralListView = new viewHolder();
        peripheralListView.mAdvertiseNameTV =
            (TextView) v.findViewById(R.id.advertise_name);
        peripheralListView.mMacAddressTV = (TextView)
            v.findViewById(R.id.mac_address);
        peripheralListView.mRssiTV = (TextView)

```

```

        v.findViewById(R.id.rssi);
        v.setTag( peripheralListItemView );
    } else {
        peripheralListItemView = (ViewHolder) v.getTag();
    }

    Log.v(TAG, "ListItem size: "+ mBluetoothPeripheralListItems.size());
    // if there are known Peripherals, create a ListItem that says so
    // otherwise, display a ListItem with Bluetooth Peripheral information
    if (mBluetoothPeripheralListItems.size() <= 0) {
        peripheralListItemView.mAdvertiseNameTV.setText(
            R.string.peripheral_list_empty);
    } else {
        BlePeripheralListItem item =
            mBluetoothPeripheralListItems.get(position);
        peripheralListItemView.mAdvertiseNameTV.setText(
            item.getAdvertiseName());
        peripheralListItemView.mMacAddressTV.setText(item.getMacAddress());
        peripheralListItemView.mRssiTV.setText(
            String.valueOf(item.getRssi()));
    }
    return v;
}
}

```

A progress spinner will be used to inform the user of the status of a scan:

#### **Example 4-9. res/layout/progress\_scanner.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<ProgressBar xmlns:android="http://schemas.android.com/apk/res/android"
    style="@android:style/widget.ProgressBar.Small"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/scan_progress" />

```

## Activities

The Activity must be able to initialize a scan when the user clicks the Scan button. It will scan for Peripherals for 5 seconds, an arbitrarily reasonable scanning time. The ListView is updated with each new Peripheral as discovered.

If no Peripherals are discovered, the ListView will display the text, “No Peripherals Found.”

The scan button and the progress spinner are defined in menu\_main.xml:

### Example 4-10. res/menu/menu\_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity">

    <item android:id="@+id/action_start_scan"
        android:title="@string/action_start_scan"
        android:orderInCategory="100" app:showAsAction="always" />
    <item android:id="@+id/action_stop_scan"
        android:title="@string/action_stop_scan"
        android:orderInCategory="100" app:showAsAction="always" />
    <item
        android:id="@+id/scan_progress_item"
        android:title="@string/scanning"
        android:visible="true"
        android:orderInCategory="100"
        app:showAsAction="always"
        app:actionLayout="@layout/progress_scanner"
        android:layout_marginRight="@dimen/activity_horizontal_margin" />
</menu>
```

The List View and the main menu are contained in activity\_main.xml:

## Example 4-11. res/layout/activity\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" android:fitsSystemWindows="true"
    tools:context=".MainActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
    <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"
        tools:showIn="@layout/activity_main" tools:context=".MainActivity">
        <ListView
            android:id="@+id/peripherals_list"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
        <TextView android:id="@+id/peripheral_list_empty"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="@string/peripheral_list_empty"/>
```

```
</RelativeLayout>  
</android.support.design.widget.CoordinatorLayout>
```

Tie it all together in the MainActivity class:

#### Example 4-12. java/example.com.exampleble/MainActivity.java

```
public class MainActivity extends AppCompatActivity {  
    /** Constants */  
    private static final String TAG = MainActivity.class.getSimpleName();  
    private static final int REQUEST_ENABLE_BT = 1;  
    /** Bluetooth Stuff */  
    private BleCommManager mBleCommManager;  
    /** Activity State */  
    private boolean mScanningActive = false;  
    /** UI Stuff */  
    private MenuItem mScanProgressSpinner;  
    private MenuItem mStartScanItem, mStopScanItem;  
    private ListView mBlePeripheralsListView;  
    private TextView mPeripheralsListEmptyTV;  
    private BlePeripheralsListAdapter mBlePeripheralsListAdapter;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);  
        setSupportActionBar(toolbar);  
        loadUI();  
    }  
    public void loadUI() {  
        // Load UI components, set up the Peripheral list  
        mPeripheralsListEmptyTV =  
            (TextView) findViewById(R.id.peripheral_list_empty);  
        mBlePeripheralsListView = (ListView) findViewById(  
            R.id.peripherals_list  
        );
```

```

        mBlePeripheralsListAdapter = new BlePeripheralsListAdapter();
        mBlePeripheralsListView.setAdapter(mBlePeripheralsListAdapter);
        mBlePeripheralsListView.setEmptyView(mPeripheralsListEmptyTV);
    }

@Override
public void onResume() {
    super.onResume();
    initializeBluetooth();
}

@Override
public void onPause() {
    super.onPause();
    // stop scanning when the activity pauses
    mBleCommManager.stopScanning(mBleScanCallbackV18, mScanCallbackV21);
}

/**
 * Create a menu
 * @param menu The menu
 * @return <b>true</b> if processed successfully
 */
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu;
    // this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    mStartScanItem = menu.findItem(R.id.action_start_scan);
    mStopScanItem = menu.findItem(R.id.action_stop_scan);
    mScanProgressSpinner = menu.findItem(R.id.scan_progress_item);
    return true;
}

/**
 * Handle a menu item click
 *
 * @param item the MenuItem
 * @return <b>true</b> if processed successfully
 */

```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Start a BLE scan when a user clicks the "start scanning" menu button
    // and stop a BLE scan
    // when a user clicks the "stop scanning" menu button
    switch (item.getItemId()) {
        case R.id.action_start_scan:
            // User chose the "Scan" item
            startScan();
            return true;
        case R.id.action_stop_scan:
            // User chose the "Stop" item
            stopScan();
            return true;
        default:
            // If we got here, the user's action was not recognized.
            // Invoke the superclass to handle it.
            return super.onOptionsItemSelected(item);
    }
}
/***
 * Initialize the Bluetooth Radio
 */
public void initializeBluetooth() {
    try {
        mBleCommManager = new BleCommManager(this);
    } catch (Exception e) {
        Toast.makeText(this, "Could not initialize bluetooth",
                Toast.LENGTH_SHORT).show();
        Log.e(TAG, e.getMessage());
        finish();
    }
    // should prompt user to open settings if Bluetooth is not enabled.
    if (!mBleCommManager.getBluetoothAdapter().isEnabled()) {
        Intent enableBtIntent =
            new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    }
}
```

```

        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    }

}

/***
 * Start scanning for Peripherals
 *
 * New in this chapter
 */
public void startScan() {
    // update UI components
    mStartScanItem.setVisible(false);
    mStopScanItem.setVisible(true);
    mScanProgressSpinner.setVisible(true);
    // clear the list of Peripherals and start scanning
    mBlePeripheralsListAdapter.clear();
    try {
        mScanningActive = true;
        mBleCommManager.scanForPeripherals(mBleScanCallbackv18,
            mScanCallbackv21);
    } catch (Exception e) {
        Log.e(TAG, "Could not open Ble Device Scanner");
    }
}
/***
 * Stop scanning for Peripherals
 *
 * New in this chapter
 */
public void stopScan() {
    mBleCommManager.stopScanning(mBleScanCallbackv18, mScanCallbackv21);
}
/***
 * Event trigger when BLE Scanning has stopped
 *
 * New in this chapter
*/

```

```

public void onBleScanStopped() {
    // update UI components to reflect that a BLE scan has stopped
    // Possible for this method to be called before the menu has been created
    // Check to see if menu items are initialized, or Activity will crash
    mScanningActive = false;
    if (mStopScanItem != null) mStopScanItem.setVisible(false);
    if (mScanProgressSpinner != null) {
        mScanProgressSpinner.setVisible(false);
    }
    if (mStartScanItem != null) mStartScanItem.setVisible(true);
}
/***
 * Event trigger when new Peripheral is discovered
 *
 * New in this chapter
 */
public void onBlePeripheralDiscovered(BluetoothDevice bluetoothDevice,
    int rssi) {
    Log.v(TAG, "Found "+bluetoothDevice.getName()+", "
        + bluetoothDevice.getAddress());
    // only add the peripheral if
    // - it has a name, or
    // - doesn't already exist in our list, or
    // - is transmitting at a higher power (is closer)
    // than a similar peripheral
    boolean addPeripheral = true;
    if (bluetoothDevice.getName() == null) {
        addPeripheral = false;
    }
    for(BlePeripheralListItem listItem :
        mBlePeripheralsListAdapter.getItems()) {
        if (listItem.getAdvertiseName().equals(
            bluetoothDevice.getName()))
        {
            addPeripheral = false;
        }
    }
}

```

```

    }

    if (addPeripheral) {
        mBlePeripheralsListAdapter.addBluetoothPeripheral(blueoothDevice,
            rssi);
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                mBlePeripheralsListAdapter.notifyDataSetChanged();
            }
        });
    }
}

/**
 * Use this callback for Android API 21 (Lollipop) or greater
 *
 * New in this chapter
 */
private final BleScanCallbackV21 mScanCallbackV21 =
    new BleScanCallbackV21() {
        @Override
        public void onScanResult(int callbackType, ScanResult result) {
            BluetoothDevice blueoothDevice = result.getDevice();
            int rssi = result.getRssi();
            onBlePeripheralDiscovered(blueoothDevice, rssi);
        }
        @Override
        public void onBatchScanResults(List<ScanResult> results) {
            for (ScanResult result : results) {
                BluetoothDevice blueoothDevice = result.getDevice();
                int rssi = result.getRssi();
                onBlePeripheralDiscovered(blueoothDevice, rssi);
            }
        }
        @Override
        public void onScanFailed(int errorCode) {
            switch (errorCode) {

```

```

        case SCAN_FAILED_ALREADY_STARTED:
            Log.e(TAG, "Scan with the same settings already started");
            break;

        case SCAN_FAILED_APPLICATION_REGISTRATION_FAILED:
            Log.e(TAG, "App cannot be registered.");
            break;

        case SCAN_FAILED_FEATURE_UNSUPPORTED:
            Log.e(TAG, "Power optimized scan is not supported.");
            break;

        default: // SCAN_FAILED_INTERNAL_ERROR
            Log.e(TAG, "Fails to start scan due an internal error");
    }

    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

public void onScanComplete() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

};

/***
 * Use this callback for Android API 18, 19, and 20 (before Lollipop)
 *
 * New in this chapter
 */
public final BleScanCallbackV18 mBleScanCallbackV18 =
    new BleScanCallbackV18() {
        @Override

```

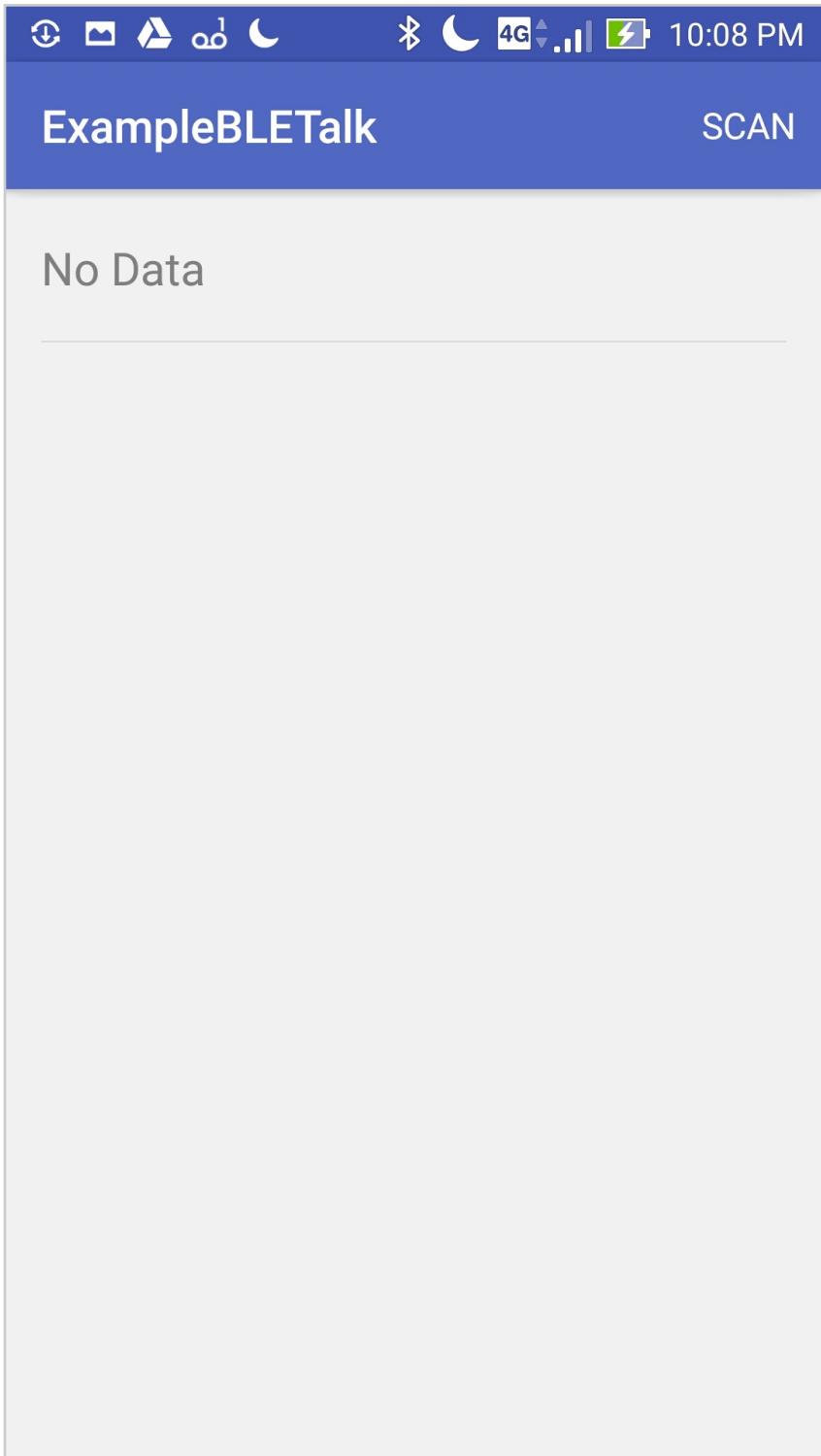
```
public void onLeScan(BluetoothDevice bluetoothDevice, int rssi,
        byte[] scanRecord) {
    onBlePeripheralDiscovered(bluetoothDevice, rssi);
}

@Override
public void onScanComplete() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

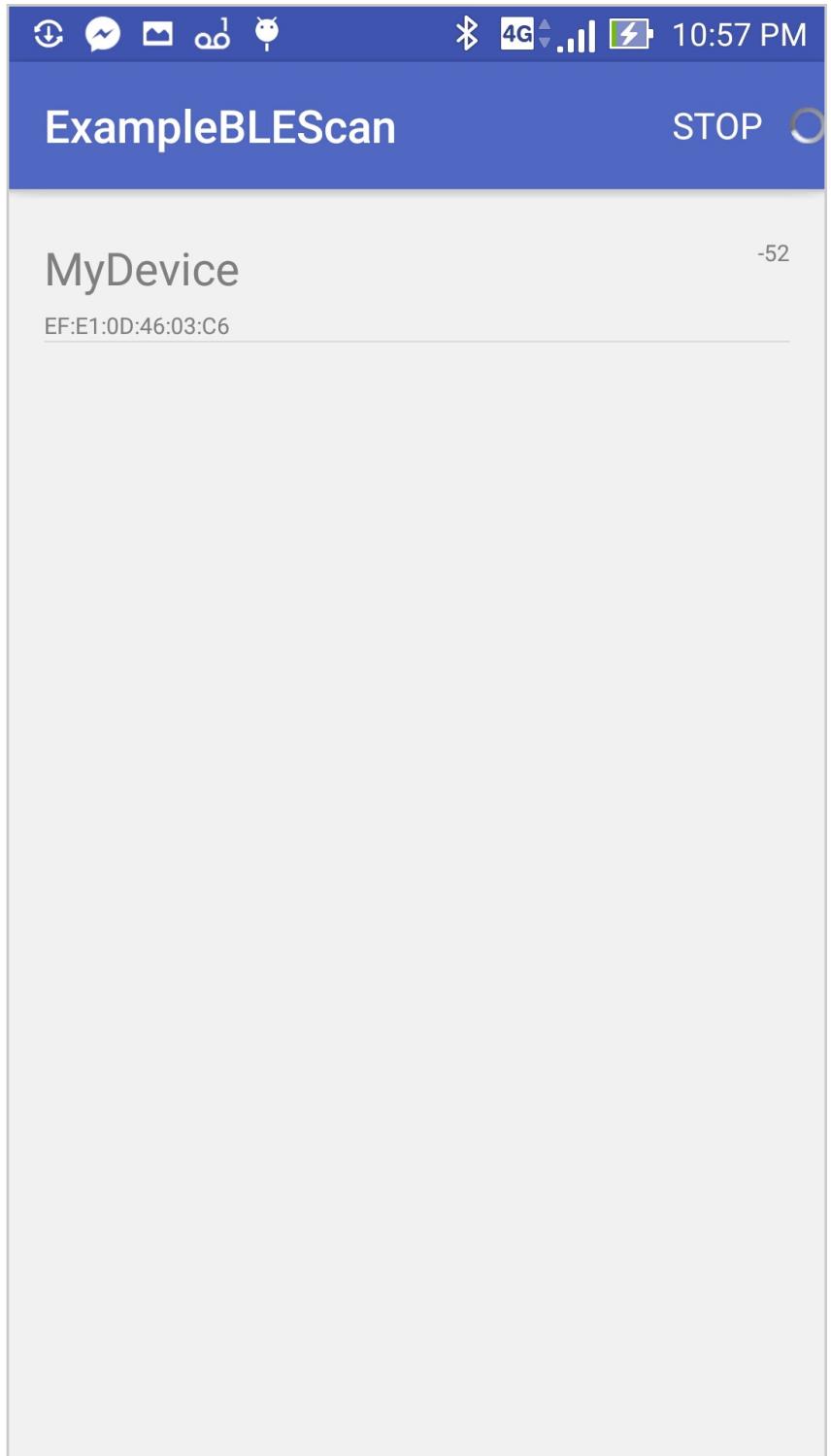
};

}
```

Compile and run the app. When it runs, you will see a screen with a scan button ([Figure 4-4](#)). When the scan button is clicked, it locates your BLE device ([Figure 4-5](#)).



**Figure 4-4. App screen prior to a Bluetooth scan**



**Figure 4-5. App screen after discovering a Bluetooth Peripheral**

## Programming the Peripheral

The previous Chapter showed how to turn on the Bluetooth radio and detect if Peripheral is supported by the Android hardware

This chapter will show how to advertise a Bluetooth Low Energy Peripheral.

The BluetoothAdapter gives access to the BluetoothLeScanner, which can initiate a scan for nearby Peripherals.

```
// the device scanner allows us to scan for BLE Devices  
BluetoothLeScanner bluetoothScanner = bluetoothAdapter.getBluetoothLeScanner();
```

In Android, it's running CPU-intensive tasks in the main thread prevents the user interface from functioning properly. For this reason, Bluetooth scanning is best done in a separate thread.

In this example, the App will scan for BLE devices for 3-5 seconds. 5 seconds is a reasonable amount of time to assume that most devices will be discovered during the scanning process.

The method for implementing scanning changed in API level 21 (Android Lollipop).

## Advertising

To begin advertising, set the Advertise settings and Advertisement Data.

The Advertisement data can contain the Peripheral name, GATT Profile, and other information. The Peripheral name is set like this:

```
static public final String ADVERTISING_NAME = "MyDevice";  
  
// Set the advertised name  
mBluetoothAdapter.setName(ADVERTISING_NAME);
```

The Advertisement data and settings are set using the BluetoothLeAdvertiser class:

```
BluetoothLeAdvertiser mBluetoothAdvertiser = \  
    mBluetoothAdapter.getBluetoothLeAdvertiser();  
  
// Build Advertise settings with transmission power and advertise speed
```

```
AdvertiseSettings advertiseSettings = new AdvertiseSettings.Builder()
    .setAdvertiseMode(mAdvertisingMode)
    .setTxPowerLevel(mTransmissionPower)
    .setConnectable(false)
    .build();

AdvertiseData.Builder advertiseBuilder = new AdvertiseData.Builder();
// set advertising name
advertiseBuilder.setIncludeDeviceName(true);

AdvertiseData advertiseData = advertiseBuilder.build();

// begin advertising
try {
    mBluetoothAdvertiser.startAdvertising(
        advertiseSettings,
        advertiseData,
        mAdvertiseCallback
);
} catch (Exception e) {
    Log.e(TAG, "could not start advertising");
    Log.e(TAG, e.getMessage());
    e.printStackTrace();
}
```

The Advertising mode can be changed depending on the desired tradeoff between battery life and discoverability.

**Table 4-5. Advertising Mode**

Setting	Description
<b>ADVERTISE_MODE_BALANCED</b>	Balance battery efficiency and discoverability
<b>ADVERTISE_MODE_LOW_LATENCY</b>	Prefer discoverability over battery efficiency
<b>ADVERTISE_MODE_LOW_POWER</b>	Prefer battery efficiency to discoverability

The Transmission Power can also be set. A larger Transmission Power results longer-range communication but poor battery life, whereas a smaller Transmission Power results in better battery life but shorter communication range.

**Table 4-6. Transmission Power**

Setting	Description
<b>ADVERTISE_TX_POWER_HIGH</b>	-56 dBm at 1 meter distance
<b>ADVERTISE_TX_POWER_MEDIUM</b>	-66 dBm at 1 meter distance
<b>ADVERTISE_TX_POWER_LOW</b>	-75 dBm at 1 meter distance
<b>ADVERTISE_TX_POWER_ULTRA_LOW</b>	Invisible at 1 meter distance

When Advertising begins or fails, corresponding events are triggered in the AdvertiseCallback:

**Table 4-7. AdvertiseCallback**

Event	Description
<b>onStartSuccess</b>	Triggered when Advertising begins
<b>onStartFailure</b>	Triggered if Advertising fails

The AdvertiseCallback contains information about the Advertise settings or reason for failure

AdvertiseCallback can be implemented like this, to get the Peripheral's advertise name and MAC address:

```
private AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {  
    /**  
     * Advertising Started  
     *  
     * @param settingsInEffect The AdvertiseSettings that worked  
     */  
    @Override  
    public void onStartSuccess(AdvertiseSettings settingsInEffect) {  
        super.onStartSuccess(settingsInEffect);  
    }  
  
    /**  
     * Advertising Failed  
     *  
     * @param errorCode the reason for failure  
     */  
    @Override  
    public void onStartFailure(int errorCode) {  
        super.onStartFailure(errorCode);  
        switch (errorCode) {  
            case AdvertiseCallback.ADVERTISE_FAILED_ALREADY_STARTED:  
                Log.e(TAG, "Failed to start; advertising is already started.");  
                break;  
            case AdvertiseCallback.ADVERTISE_FAILED_DATA_TOO_LARGE:  
                Log.e(  
                    TAG,  
                    "Failed to start; advertised data is larger than 31 bytes."  
                );  
                break;  
            case AdvertiseCallback.ADVERTISE_FAILED_FEATURE_UNSUPPORTED:  
                Log.e(TAG, "This feature is not supported on this platform.");  
                break;  
            case AdvertiseCallback.ADVERTISE_FAILED_INTERNAL_ERROR:  
                Log.e(TAG, "An internal error occurred while advertising.");  
                break;  
        }  
    }  
};
```

```

        Log.e(TAG, "Operation failed due to an internal error.");
        break;

    case AdvertiseCallback.ADVERTISE_FAILED_TOO_MANY_ADVERTISERS:
        Log.e(
            TAG,
            "Failed to start; no advertising instance is available."
        );
        break;

    default:
        Log.e(TAG, "unknown problem");
    }
}
};


```

Advertising may fail for a number of reasons, from the Advertising packet being too big to the feature being unsupported.

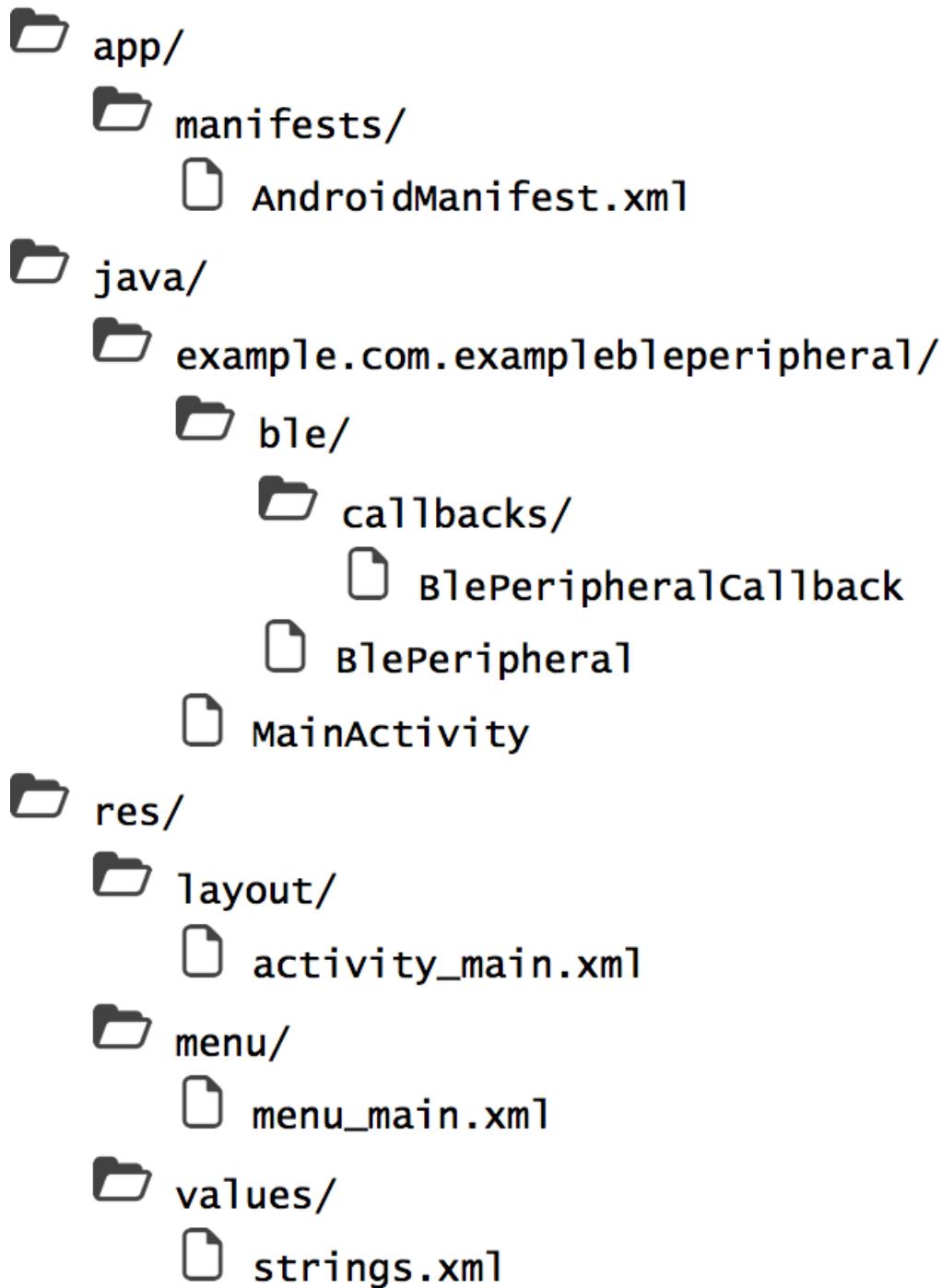
**Table 4-8. Reasons for Advertising failure**

Setting	Description
ADVERTISE_FAILED_ALREADY_STARTED	Peripheral is already advertising
ADVERTISE_FAILED_DATA_TOO_LARGE	AdvertisingData is greater than 31 bytes long
ADVERTISE_FAILED_FEATURE_UNSUPPORTED	Advertising is unsupported
ADVERTISE_FAILED_INTERNAL_ERROR	Unknown error occurred
ADVERTISE_FAILED_TOO_MANY_ADVERTISERS	Advertising instance is unavailable

## Putting It All Together

Create a new app called ExampleBlePeripheral, and copy everything from the previous example.

Create a file structure that looks like this ([Figure 4-6](#)).



**Figure 4-6. Project structure**

This example will feature some user interface elements, including a list adapter. The list adapter will populate a list of discovered Peripherals in the main Activity.

Custom scanner callbacks will be created, which respond to events when scanning has stopped.

## Resources

To begin, define some dimensions and strings that will be used by the rest of the app.

Add text for buttons and labels:

### Example 4-13. res/values/strings.xml

```
<resources>
    <string name="app_name">ExampleBLEPeripheral</string>
    <string name="advertising_name_label">Advertising Name: </string>
    <string name="bluetooth_on">Bluetooth on</string>
    <string name="advertising">Advertising</string>
</resources>
```

## Objects

Add the following functions to the BlePeripheral to give it functionality to start and stop Advertising. Notice the change to the constructor, which now requires a BlePeripheralCallback to be passed.

### Example 4-14. java/example.com.exampleble/ble/BlePeripheral.java

```
public class BlePeripheral {
    /** Constants */
    private static final String TAG = MyBlePeripheral.class.getSimpleName();

    /** Peripheral and GATT Profile */
    public static final String ADVERTISING_NAME = "MyDevice";

    /** Advertising settings */

    // advertising mode can be one of:
    // - ADVERTISE_MODE_BALANCED,
    // - ADVERTISE_MODE_LOW_LATENCY,
    // - ADVERTISE_MODE_LOW_POWER
```

```

int mAdvertisingMode = AdvertiseSettings.ADVERTISE_MODE_LOW_LATENCY;

// transmission power mode can be one of:
// - ADVERTISE_TX_POWER_HIGH
// - ADVERTISE_TX_POWER_MEDIUM
// - ADVERTISE_TX_POWER_LOW
// - ADVERTISE_TX_POWER_ULTRA_LOW
int mTransmissionPower = AdvertiseSettings.ADVERTISE_TX_POWER_HIGH;

/** Callback Handlers */
public BlePeripheralCallback mBlePeripheralCallback;

/** Bluetooth Stuff */
private BluetoothAdapter mBluetoothAdapter;
private BluetoothLeAdvertiser mBluetoothAdvertiser;
/** Constants */
private static final String TAG = MyBlePeripheral.class.getSimpleName();

/** Peripheral and GATT Profile */
public static final String ADVERTISING_NAME = "MyDevice";

/** Advertising settings */

// advertising mode can be one of:
// - ADVERTISE_MODE_BALANCED,
// - ADVERTISE_MODE_LOW_LATENCY,
// - ADVERTISE_MODE_LOW_POWER
int mAdvertisingMode = AdvertiseSettings.ADVERTISE_MODE_LOW_LATENCY;

// transmission power mode can be one of:
// - ADVERTISE_TX_POWER_HIGH
// - ADVERTISE_TX_POWER_MEDIUM
// - ADVERTISE_TX_POWER_LOW
// - ADVERTISE_TX_POWER_ULTRA_LOW
int mTransmissionPower = AdvertiseSettings.ADVERTISE_TX_POWER_HIGH;

```

```

/** Callback Handlers */
public BlePeripheralCallback mBlePeripheralCallback;

/** Bluetooth Stuff */
private BluetoothAdapter mBluetoothAdapter;
private BluetoothLeAdvertiser mBluetoothAdvertiser;

/**
 * Construct a new Peripheral
 *
 * @param context The Application Context
 * @param blePeripheralCallback The callback handler
 *                      that interfaces with this Peripheral
 * @throws Exception Exception thrown if Bluetooth is not supported
 */
public MyBlePeripheral(
    final Context context,
    BlePeripheralCallback blePeripheralCallback) throws Exception
{
    mBlePeripheralCallback = blePeripheralCallback;
    // make sure Android device supports Bluetooth Low Energy
    if (!context.getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_BLUETOOTH_LE))
    {
        throw new Exception("Bluetooth Not Supported");
    }
    // get a reference to the Bluetooth Manager class,
    // which allows us to talk to talk to the BLE radio
    final BluetoothManager bluetoothManager = \
        (BluetoothManager) context.getSystemService(
            Context.BLUETOOTH_SERVICE
        );
    mBluetoothAdapter = bluetoothManager.getAdapter();
    // Beware: this function doesn't work on some systems
    if(!mBluetoothAdapter.isMultipleAdvertisementSupported()) {

```

```

        throw new Exception ("Peripheral mode not supported");
    }

    mBluetoothAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();
    // Use this method instead for better support
    if (mBluetoothAdvertiser == null) {
        throw new Exception ("Peripheral mode not supported");
    }
}

/***
 * Get the system Bluetooth Adapter
 *
 * @return BluetoothAdapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}

/***
 * Start Advertising
 *
 * @throws Exception Exception thrown
 *                  if Bluetooth Peripheral mode is not supported
 */
public void startAdvertising() {
    mBluetoothAdapter.setName(ADVERTISING_NAME);
    // Build Advertise settings with transmission power and advertise speed
    AdvertiseSettings advertiseSettings = new AdvertiseSettings.Builder()
        .setAdvertiseMode(mAdvertisingMode)
        .setTxPowerLevel(mTransmissionPower)
        .setConnectable(false)
        .build();

    AdvertiseData.Builder advertiseBuilder = new AdvertiseData.Builder();
    // set advertising name
    advertiseBuilder.setIncludeDeviceName(true);
    AdvertiseData advertiseData = advertiseBuilder.build();
}

```

```

// begin advertising
try {
    mBluetoothAdvertiser.startAdvertising(
        advertiseSettings,
        advertiseData,
        mAdvertiseCallback
    );
} catch (Exception e) {
    Log.e(TAG, "could not start advertising");
    Log.e(TAG, e.getMessage());
    e.printStackTrace();
}
}

/**
 * Stop advertising
 */
public void stopAdvertising() {
    if (mBluetoothAdvertiser != null) {
        mBluetoothAdvertiser.stopAdvertising(mAdvertiseCallback);
        mBlePeripheralCallback.onAdvertisingStopped();
    }
}

/**
 * Handle callbacks from the Bluetooth Advertiser
 */
private AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
    /**
     * Advertising Started
     *
     * @param settingsInEffect The AdvertiseSettings that worked
     */
    @Override
    public void onStartSuccess(AdvertiseSettings settingsInEffect) {
        super.onStartSuccess(settingsInEffect);
    }
}

```

```

        mBlePeripheralCallback.onAdvertisingStarted();

    }

    /**
     * Advertising Failed
     *
     * @param errorCode the reason for failure
     */
    @Override
    public void onStartFailure(int errorCode) {
        super.onStartFailure(errorCode);
        mBlePeripheralCallback.onAdvertisingFailed(errorCode);
    }
}

```

Define a BlePeripheralCallback, which will describe how the BlePeripheral will react to state changes and other events.

## Example

### 4-15. [java/example.com.exampleble/ble/callbacks/BlePeripheralCallback.java](#)

```

package example.com.exampleble.ble.callbacks;
public abstract class BlePeripheralCallback {
    /**
     * Advertising Started
     */
    public abstract void onAdvertisingStarted();

    /**
     * Advertising Could not Start
     */
    public abstract void onAdvertisingFailed(int errorCode);

    /**
     * Advertising Stopped
     */
}

```

```
 */  
 public abstract void onAdvertisingStopped();  
}
```

## Activities

MainActivity will begin advertising as soon as the Bluetooth radio turns on. The BlePeripheralCallback will alert MainActivity of changes to the Peripheral state.

The Activity will display the Advertising name of the Peripheral in a TextView. It will also show the state of the Bluetooth radio and of the Advertising Peripheral using Switches:

### Example 4-16. res/layout/activity\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>  
<android.support.design.widget.CoordinatorLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:fitsSystemWindows="true"  
    tools:context=".MainActivity">  
    <android.support.design.widget.AppBarLayout  
        android:layout_height="wrap_content"  
        android:layout_width="match_parent"  
        android:theme="@style/AppTheme.AppBarOverlay">  
        <android.support.v7.widget.Toolbar  
            android:id="@+id/toolbar"  
            android:layout_width="match_parent"  
            android:layout_height="?attr/actionBarSize"  
            android:background="?attr/colorPrimary"  
            app:popupTheme="@style/AppTheme.PopupOverlay" />  
    </android.support.design.widget.AppBarLayout>  
    <RelativeLayout
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main" tools:context=".MainActivity">
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/advertising_name_label" />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/advertising_name" />
</LinearLayout>
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bluetooth_on"
    android:id="@+id/bluetooth_on" />
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/advertising"
```

```
        android:id="@+id/advertising" />
    </LinearLayout>
</RelativeLayout>
</android.support.design.widget.CoordinatorLayout>
```

Tie it all together in the MainActivity class:

#### **Example 4-17. java/example.com.exampleble/MainActivity.java**

```
public class MainActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = MainActivity.class.getSimpleName();
    private static final int REQUEST_ENABLE_BT = 1;

    /** Bluetooth Stuff */
    private MyBlePeripheral mMyBlePeripheral;

    /** UI Stuff */
    private TextView mAdvertisingNameTV;
    private Switch mBluetoothOnSwitch,
                  mAdvertisingSwitch;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        // notify when bluetooth is turned on or off
        IntentFilter filter = new IntentFilter(
            BluetoothAdapter.ACTION_STATE_CHANGED
        );
        registerReceiver(mBleAdvertiseReceiver, filter);
        loadUI();
    }
}
```

```
@Override
public void onPause() {
    super.onPause();
    // stop advertising when the activity pauses
    mMyBlePeripheral.stopAdvertising();
}

@Override
public void onResume() {
    super.onResume();
    initializeBluetooth();
}

@Override
public void onDestroy() {
    super.onDestroy();
    unregisterReceiver(mBleAdvertiseReceiver);
}

/**
 * Load UI components
 */
public void loadUI() {
    mAdvertisingNameTV = (TextView) findViewById(R.id.advertising_name);
    mBluetoothOnSwitch = (Switch) findViewById(R.id.bluetooth_on);
    mAdvertisingSwitch = (Switch) findViewById(R.id.advertising);
    mAdvertisingNameTV.setText(MyBlePeripheral.ADVERTISING_NAME);
}

/**
 * Initialize the Bluetooth Radio
 */
public void initializeBluetooth() {
    // reset connection variables
    try {
        mMyBlePeripheral = new MyBlePeripheral()
```

```

        this,
        mBlePeripheralCallback
    );
} catch (Exception e) {
    Toast.makeText(
        this,
        "Could not initialize bluetooth", Toast.LENGTH_SHORT
    ).show();
    Log.e(TAG, e.getMessage());
    finish();
}

mBluetoothOnSwitch.setChecked(
    mMyBlePeripheral.getBluetoothAdapter().isEnabled()
);
// should prompt user to open settings if Bluetooth is not enabled.
if (!mMyBlePeripheral.getBluetoothAdapter().isEnabled()) {
    Intent enableBtIntent = new Intent(
        BluetoothAdapter.ACTION_REQUEST_ENABLE
    );
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
} else {
    startAdvertising();
}
}

/**
 * Start advertising Peripheral
 */
public void startAdvertising() {
    Log.v(TAG, "starting advertising...");
    try {
        mMyBlePeripheral.startAdvertising();
    } catch (Exception e) {
        Log.e(TAG, "problem starting advertising");
    }
}

```

```

/**
 * Event trigger when BLE Advertising has stopped
 */
public void onBleAdvertisingStarted() {
    mAdvertisingSwitch.setChecked(true);
}

/**
 * Advertising Failed to start
 */
public void onBleAdvertisingFailed() {
    mAdvertisingSwitch.setChecked(false);
}

/**
 * Event trigger when BLE Advertising has stopped
 */
public void onBleAdvertisingStopped() {
    mAdvertisingSwitch.setChecked(false);
}

/**
 * when the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mBleAdvertiseReceiver = \
    new AdvertiseReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();

        if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
            final int state = intent.getIntExtra(
                BluetoothAdapter.EXTRA_STATE,
                BluetoothAdapter.ERROR

```

```
);

        switch (state) {
            case BluetoothAdapter.STATE_OFF:
                Log.v(TAG, "Bluetooth turned off");
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_OFF:
                break;
            case BluetoothAdapter.STATE_ON:
                Log.v(TAG, "Bluetooth turned on");
                startAdvertising();
                break;
            case BluetoothAdapter.STATE_TURNING_ON:
                break;
        }
    }

};

/***
 * Respond to changes to the Bluetooth Peripheral state
 */
private final BlePeripheralCallback mBlePeripheralCallback = \
    new BlePeripheralCallback()
{
    public void onAdvertisingStarted() {
        Log.v(TAG, "Advertising started");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleAdvertisingStarted();
            }
        });
    }

    public void onAdvertisingFailed(int errorCode) {
```

```
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        onBleAdvertisingFailed();
    }
});

switch (errorCode) {
    case AdvertiseCallback.ADVERTISE_FAILED_ALREADY_STARTED:
        Log.e(
            TAG,
            "Failed to start; advertising is already started."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_DATA_TOO_LARGE:
        Log.e(
            TAG,
            "Failed to start; advertised data > 31 bytes."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_FEATURE_UNSUPPORTED:
        Log.e(
            TAG,
            "This feature is not supported on this platform."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_INTERNAL_ERROR:
        Log.e(
            TAG,
            "Operation failed due to an internal error."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_TOO_MANY_ADVERTISERS:
        Log.e(
            TAG,
            "Failed to start; advertising instance is available."
        );
}
```

```
        break;

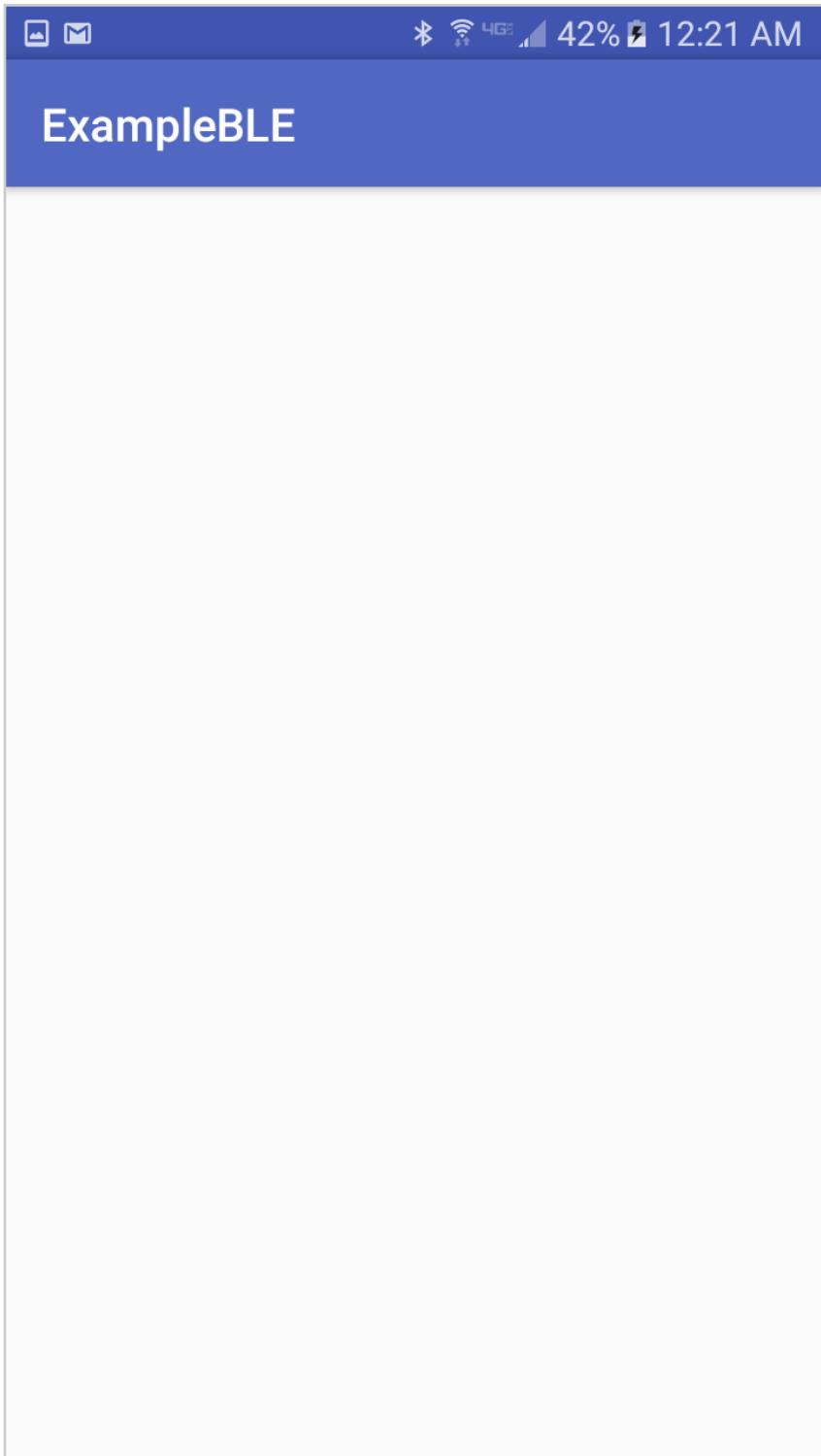
    default:
        Log.e(TAG, "unknown problem");
    }

}

public void onAdvertisingStopped() {
    Log.v(TAG, "Advertising stopped");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleAdvertisingStopped();
        }
    });
}

};
```

Compile and run the app. When it runs, a Bluetooth Peripheral will be advertising (Figure 4-7).



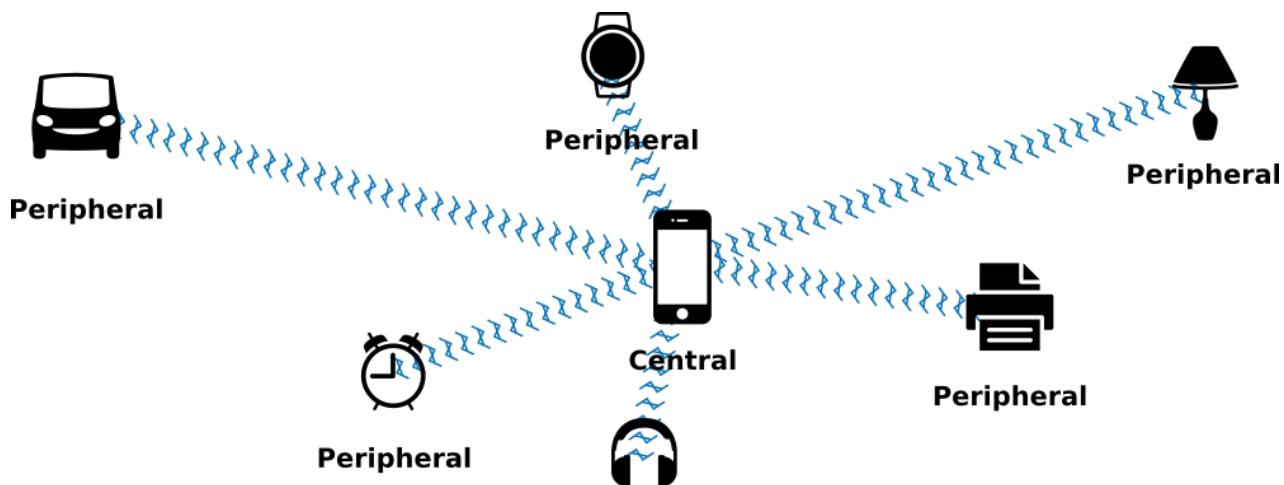
**Figure 4-7. App screen showing advertising Peripheral**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter04>

# Connecting

Once a Central has discovered a Peripheral, the central can attempt to connect. This must be done before data can be passed between the Central and Peripheral. A Central may hold several simultaneous connections with a number of peripherals, but a Peripheral may only hold one connection at a time. Hence the names Central and Peripheral (Figure 5-1).



**Figure 5-1. Bluetooth network topology**

Bluetooth supports data 37 data channels ranging from 2404 MHz to 2478 MHz.

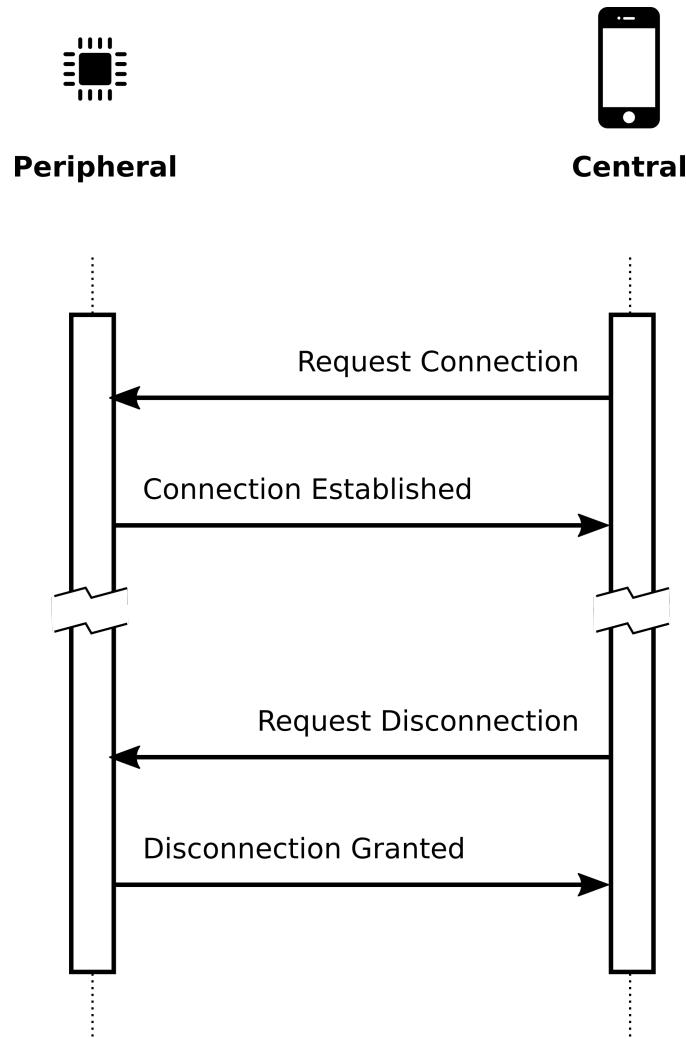
Once the connection is established, the Central and Peripheral negotiate which of these channels to begin communicating over. As part of this, a unique Media Access Control address (MAC) of the Central is sent to the Peripheral.

A MAC address is a 48-bit address given to every network device. It is typically represented in a hexadecimal format, similar to this:

08:00:27:0E:25:B8

Because the Peripheral can only hold one connection at a time, it must disconnect from the Central before a new connection can be made.

The connection and disconnection process works like this ([Figure 5-2](#)).



**Figure 5-2. Connection and disconnection process**

## Programming the Central

The previous chapter's App showed how to discover nearby Peripherals. Using the MAC address of a known Peripheral, Android can connect to that Peripheral when in range.

Connecting to the device is as simple as grabbing the Peripheral's MAC address and sending a connection request, like this:

```
String deviceAddress = "00:A0:C9:14:C8:3A";
BluetoothDevice device = bluetoothAdapter.getRemoteDevice(deviceAddress);
device.connectGatt(context, false, gattCallback);
```

Connecting to the Bluetooth Device will expose a lot of potential events, from connection and disconnection, to read and write events. These are packaged in the BluetoothGattCallback class, which must be implemented to respond to events triggered by the Bluetooth device.

Those events include:

**Table 4-1. BluetoothGattCallback**

Event	Description
<b>onConnectionStateChange</b>	Triggered when a BLE Peripheral connects or disconnects
<b>onServicesDiscovered</b>	Triggered when GATT Services are discovered
<b>onCharacteristicRead</b>	Triggered when data has been downloaded from a GATT Characteristic
<b>onCharacteristicWrite</b>	Triggered when data has been uploaded to a GATT Characteristic
<b>onCharacteristicChanged</b>	Triggered when a GATT Characteristic's data has changed

These events are implemented in code like this:

```
private final BluetoothGattCallback mGattCallback = \  
    new BluetoothGattCallback() {  
  
        /**  
         * Called whenever a connection or disconnection occurs  
         */  
  
        @Override  
        public void onConnectionStateChange(  
            BluetoothGatt bluetoothGatt, int status, int newState) {  
            // store the bluetoothGatt connection here  
            this.bluetoothGatt = bluetoothGatt;  
            if (newState == BluetoothProfile.STATE_CONNECTED) {  
                Log.d(TAG, "Connected to device");  
            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {  
                Log.d(TAG, "Disconnected from device");  
            }  
        }  
    }
```

```

}

/**
 * called when services are discovered on the connected device
 */
@Override
public void onServicesDiscovered(BluetoothGatt gatt, int status) {
}

/**
 * called any time a characteristic's data is downloaded for reading
 */
@Override
public void onCharacteristicRead(
    BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic,
    int status)
{
}

/**
 * called any time data has been uploaded to a remote characteristic
 */
@Override
public void onCharacteristicWrite (BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic, int status) {
}

/**
 * called any time a remote characteristic's data has changed
 */
@Override
public void onCharacteristicChanged(BluetoothGatt gatt,
    final BluetoothGattCharacteristic characteristic) {
}

};


```

By saving the BluetoothGatt reference returned in the onConnectionStateChange() function, disconnecting the Peripheral later is done like this:

```
bluetoothGatt.disconnect();
bluetoothGatt.close(); // call this when disconnection is confirmed
```

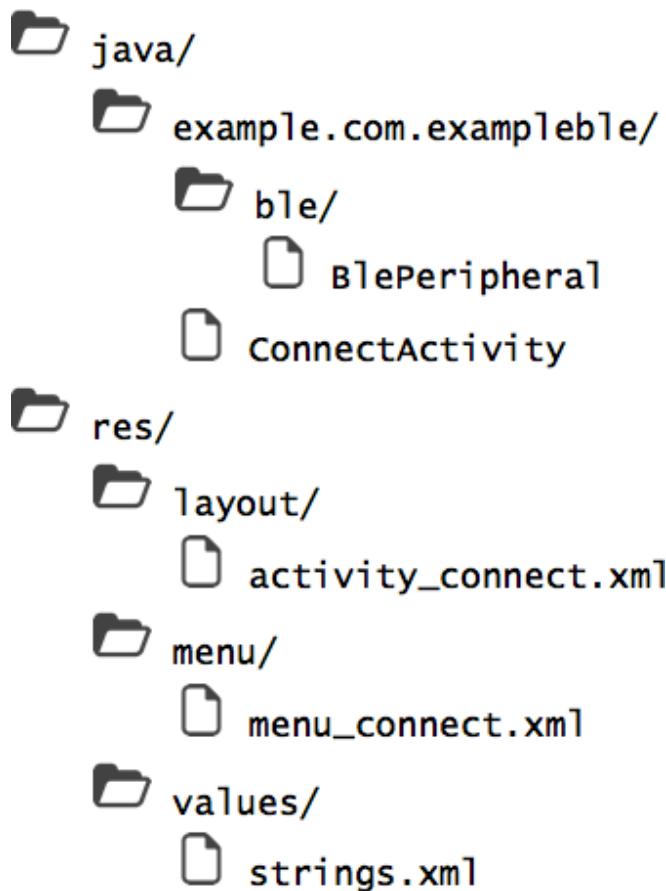
Disconnecting is important. The Peripheral can only be connected to one device at a time. Sometimes, closing an Activity without disconnecting the Peripheral can leave the Peripheral in a connected state - unable to advertise or connect to a Central again in the future.

## Putting It All Together

Create a new project called ExampleBLEConnect and copy everything from the previous example. In this example we will create an app that looks for a “MyDevice” BLE advertisement, and connects to it.

BlePeripheral represents a remote Peripheral. ConnectActivity will be where to connect to the Peripheral and list the Peripheral properties.

Create new classes and new layout resource and menu files in res/with the following names ([Figure 5-3](#)).



**Figure 5-3. Added project structure**

## Resources

Define the text for the new buttons to connect, disconnect, and status texts to alert the user to changes in the connection state in res/values/strings.xml:

### Example 5-1. res/values/strings.xml

```
...
<string name="action_connect">Connect</string>
<string name="action_disconnect">Disconnect</string>
<string name="connecting">Connecting...</string>
<string name="loading">Loading...</string>
...
```

## Objects

BlePeripheral represents the Peripheral. It will respond to connection and disconnections.

### Example 5-2. java/example.com.exampleble/ble/BlePeripheral.java

```
package example.com.exampleble.ble;

public class BlePeripheral {
    private static final String TAG = BlePeripheral.class.getSimpleName();
    private BluetoothDevice mBluetoothDevice;
    private BluetoothGatt mBluetoothGatt;
    public BlePeripheral() {
    }
    /**
     * Connect to a Peripheral
     *
     * @param bluetoothDevice the Bluetooth Device
     * @param callback The connection callback
     * @param context The Activity that initialized the connection
     * @return a connection to the BluetoothGatt
     * @throws Exception if no device is given
     */
    public BluetoothGatt connect(BluetoothDevice bluetoothDevice,
                                BluetoothGattCallback callback,
                                final Context context) throws Exception {
        if (bluetoothDevice == null) {
            throw new Exception("No bluetooth device provided");
        }
        mBluetoothDevice = bluetoothDevice;
        mBluetoothGatt = bluetoothDevice.connectGatt(context, false, callback);
        return mBluetoothGatt;
    }
    /**
     * Disconnect from a Peripheral
     */
    public void disconnect() {
        if (mBluetoothGatt != null) {
            mBluetoothGatt.disconnect();
        }
    }
}
```

```

    }

    /**
     * A connection can only close after a successful disconnect.
     * Be sure to use the BluetoothGattCallback.onConnectionStateChanged event
     * to notify of a successful disconnect
     */
    public void close() {
        if (mBluetoothGatt != null) {
            mBluetoothGatt.close(); // close connection to Peripheral
            mBluetoothGatt = null; // release from memory
        }
    }

    public BluetoothDevice getBluetoothDevice() {
        return mBluetoothDevice;
    }
}

```

## Activities

The previous app was able to detect and list nearby BLE Peripherals. This app will allow connecting to a Peripheral when a user selects that Peripheral from the ListView.

Add the click functionality to MainActivity.

### **Example 5-3. java/example.com.exampleble/MainActivity.java**

```

...
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    loadUI();
    attachCallbacks();
}

```

```

... loadUI()
    /**
     * Attach callback listeners to UI elements
     */
    public void attachCallbacks() {
        // if a list item is clicked,
        // open corresponding Peripheral in the ConnectActivity
        mBlePeripheralsListView.setOnItemClickListener(
            new AdapterView.OnItemClickListener() {
                @Override
                public void onItemClick(AdapterView<?> parent, View view,
                        int position, long id) {
                    // only open if the selected item represents a Peripheral
                    BlePeripheralListItem selectedPeripheralListItem =
                            (BlePeripheralListItem)
                    mBlePeripheralsListView.getItemAtPosition(position);
                    Log.v(TAG, "Click at position: " + position + ", id: " + id);
                    BlePeripheralListItem listItem =
                            mBlePeripheralsListAdapter.getItem(position);
                    connectToPeripheral(listItem.getMacAddress());
                    stopScan();
                }
            });
    }
...

```

The Connect Activity keeps track of the BLE Peripheral and connection status.

When a connection is detected, the user interface is updated.

#### **Example 5-4. java/example.com.exampleble/ConnectActivity.java**

```

package example.com.exampleble;
public class ConnectActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = ConnectActivity.class.getSimpleName();

```

```
public static final String PERIPHERAL_MAC_ADDRESS_KEY = \
    "com.example.com.exampleble.PERIPHERAL_MAC_ADDRESS";\n\n/** Bluetooth Stuff */\nprivate BleCommManager mBleCommManager;\nprivate BlePeripheral mBlePeripheral;\n\n/** Functional stuff */\nprivate String mPeripheralMacAddress;\nprivate String mBlePeripheralName;\n\n/** Activity State */\nprivate boolean mBleConnected = false;\nprivate boolean mLeaveActivity = false;\n\n/** UI Stuff */\nprivate MenuItem mProgressSpinner;\nprivate MenuItem mConnectItem, mDisconnectItem;\nprivate TextView mPeripheralAdvertiseNameTV, mPeripheralAddressTV;\n\n@Override\nprotected void onCreate(Bundle savedInstanceState) {\n    // grab information passed to the savedInstanceState,\n    // from when the user clicked on the list in MainActivty\n    if (savedInstanceState == null) {\n        Bundle extras = getIntent().getExtras();\n        if (extras != null) {\n            mPeripheralMacAddress = \
                extras.getString(PERIPHERAL_MAC_ADDRESS_KEY);\n        }\n    } else {\n        mPeripheralMacAddress = \
            savedInstanceState.getString(PERIPHERAL_MAC_ADDRESS_KEY);\n    }\n    super.onCreate(savedInstanceState);\n    setContentView(R.layout.activity_connect);
```

```

        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        loadUI();
        mBlePeripheral = new BlePeripheral();
    }

    /**
     * Prepare the UI elements
     */
    public void loadUI() {
        mPeripheralAdvertiseNameTV = \
            (TextView)findViewById(R.id.advertise_name);
        mPeripheralAddressTV = (TextView)findViewById(R.id.mac_address);
    }

    @Override
    public void onResume() {
        super.onResume();
        // connect the bluetooth device
        initializeBluetooth();
    }

    @Override
    public void onPause() {
        super.onPause();
        disconnect();
    }

    /**
     * Create the menu
     *
     * @param menu
     * @return true if successful
     */
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {

```

```
// Inflate the menu;
// this adds items to the action bar if it is present.
getMenuInflater().inflate(R.menu.menu_connect, menu);
mConnectItem = menu.findItem(R.id.action_connect);
mDisconnectItem = menu.findItem(R.id.action_disconnect);
mProgressSpinner = menu.findItem(R.id.scan_progress_item);
initializeBluetooth();
connect();
return true;
}

/**
 * User clicked a menu button
 */
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_connect:
            // User chose the "Scan" item
            connect();
            return true;
        case R.id.action_disconnect:
            // User chose the "Stop" item
            mLeaveActivity = true;
            quitActivity();
            return true;
        default:
            // The user's action was not recognized.
            // Invoke the superclass to handle it.
            return super.onOptionsItemSelected(item);
    }
}

/**
 * Turn on Bluetooth radio
*/
```

```

public void initializeBluetooth() {
    try {
        mBleCommManager = new BleCommManager(this);
    } catch (Exception e) {
        Toast.makeText(
            this,
            "Could not initialize bluetooth", Toast.LENGTH_SHORT
        ).show();
        Log.e(TAG, e.getMessage());
        finish();
    }
}

/**
 * Connect to Peripheral
 */
public void connect() {
    // grab the Peripheral Device address and attempt to connect
    BluetoothDevice bluetoothDevice = \
        mBleCommManager.getBluetoothAdapter().getRemoteDevice(
            mPeripheralMacAddress);
    mProgressSpinner.setVisibility(true);
    try {
        mBlePeripheral.connect(
            bluetoothDevice,
            mGattCallback,
            getApplicationContext()
        );
    } catch (Exception e) {
        mProgressSpinner.setVisibility(false);
        Log.e(TAG, "Error connecting to peripheral");
    }
}

/**
 * Disconnect from Peripheral

```

```

*/
public void disconnect() {
    // disconnect from the Peripheral.
    mProgressSpinner.setVisibility(true);
    mBlePeripheral.disconnect();
}

/**
 * Bluetooth Peripheral connected. Update UI
 */
public void onBleConnected() {
    // update UI to reflect a connection
    BluetoothDevice bluetoothDevice = mBlePeripheral.getBluetoothDevice();
    mBlePeripheralName = bluetoothDevice.getName();
    mPeripheralAdvertiseNameTV.setText(bluetoothDevice.getName());
    mPeripheralAddressTV.setText(bluetoothDevice.getAddress());
    mConnectItem.setVisibility(false);
    mDisconnectItem.setVisibility(true);
    mProgressSpinner.setVisibility(false);

}

/**
 * Quit the activity if the Peripheral is disconnected.
 * otherwise disconnect and try again
 */
public void quitActivity() {
    if (!mBleConnected) {
        finish();
    } else {
        disconnect();
    }
}

private BluetoothGattCallback mGattCallback = new BluetoothGattCallback() {
    /**

```

```

        * Characteristic value changed
    */
    @Override
    public void onCharacteristicChanged(
        BluetoothGatt gatt,
        final BluetoothGattCharacteristic characteristic)
    {
    }

    /**
     * Peripheral connected or disconnected
    */
    @Override
    public void onConnectionStateChange(
        BluetoothGatt gatt,
        int status,
        int newState)
    {
        // There has been a connection or a disconnection
        // with a Peripheral.
        // If this is a connection, update the UI to reflect the change
        // and discover the GATT profile of the connected Peripheral
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            Log.v(TAG, "Connected to peripheral");
            mBleConnected = true;
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    onBleConnected();
                }
            });
            gatt.discoverServices();
        } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
            mBlePeripheral.close();
            mBleConnected = false;
            if (mLeaveActivity) quitActivity();
        }
    }
}

```

```

        }

    /**
     * Gatt Profile discovered
     */
    @Override
    public void onServicesDiscovered(
            BluetoothGatt bluetoothGatt,
            int status)
    {
    }
};

}

```

The Connect Activity layout defines two text views: one for the device name and one for the MAC address:

#### **Example 5-5. res/layout/activity\_connect.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".ConnectActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr actionBarSize"

```

```

        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"
        tools:showIn="@layout/activity_main" tools:context=".MainActivity">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceLarge"
            android:layout_marginBottom="@dimen/activity_vertical_margin"
            android:text="@string/loading"
            android:id="@+id/advertise_name"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="@dimen/activity_vertical_margin"
            android:id="@+id/mac_address"/>
    </LinearLayout>
</android.support.design.widget.CoordinatorLayout>

```

The Connect activity also requires its own menu to enable connection and disconnection requests.

### **Example 5-6. res/menu/menu\_connect.xml**

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"

```

```
xmlns:tools="http://schemas.android.com/tools"
tools:context=".ConnectActivity">
<item android:id="@+id/action_connect"
    android:title="@string/action_connect"
    android:orderInCategory="100" app:showAsAction="ifRoom"
    android:visible="true" />
<item android:id="@+id/action_disconnect"
    android:title="@string/action_disconnect"
    android:orderInCategory="100" app:showAsAction="ifRoom"
    android:visible="false" />
<item
    android:id="@+id/scan_progress_item"
    android:title="@string/connecting"
    android:visible="false"
    android:orderInCategory="100"
    app:showAsAction="ifRoom"
    app:actionLayout="@layout/scanner_progress"
    android:layout_marginRight="@dimen/activity_horizontal_margin" />
</menu>
```

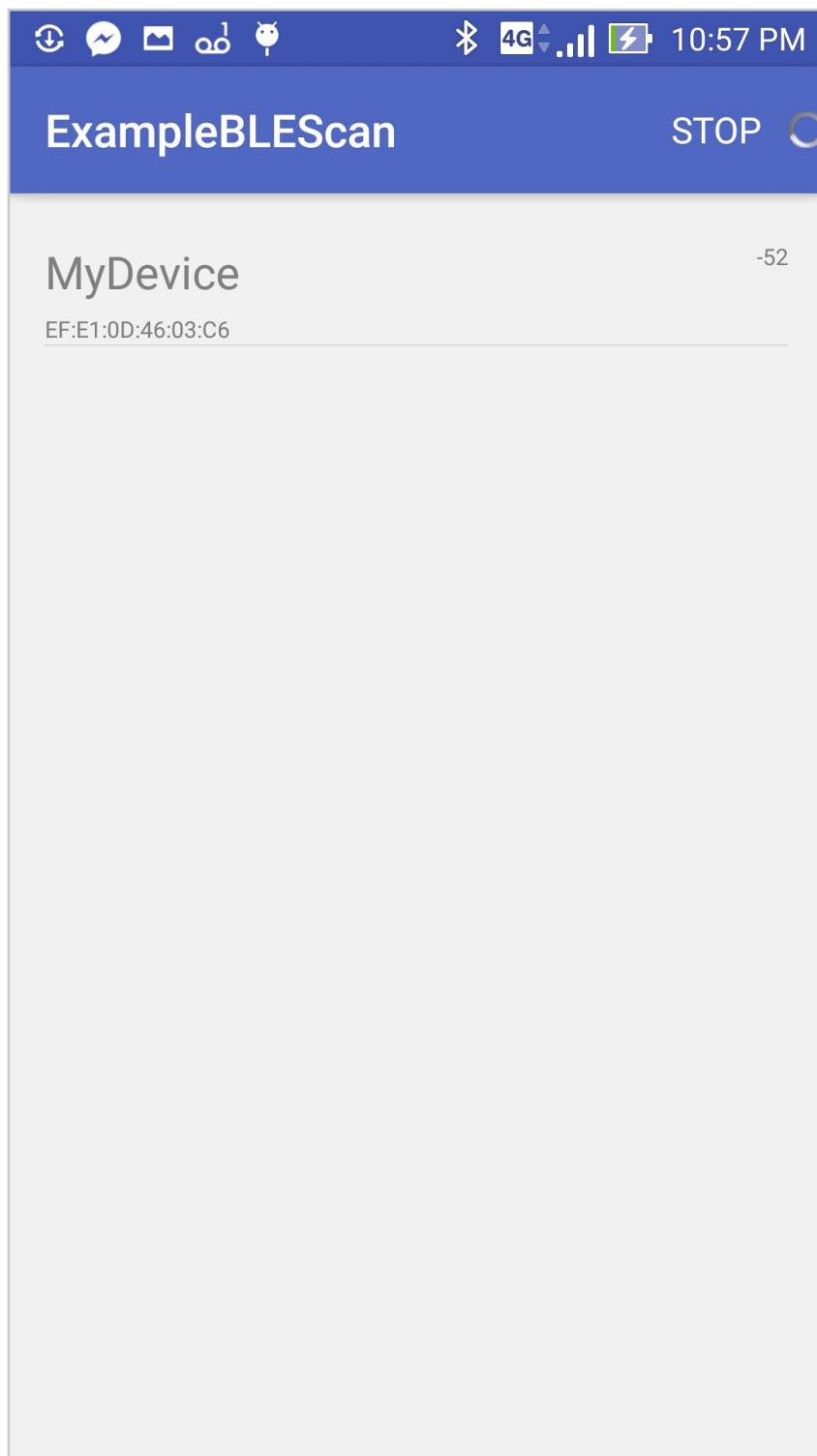
## Manifest

The Connect activity must also be described in the Android Manifest, so that it has permissions to run:

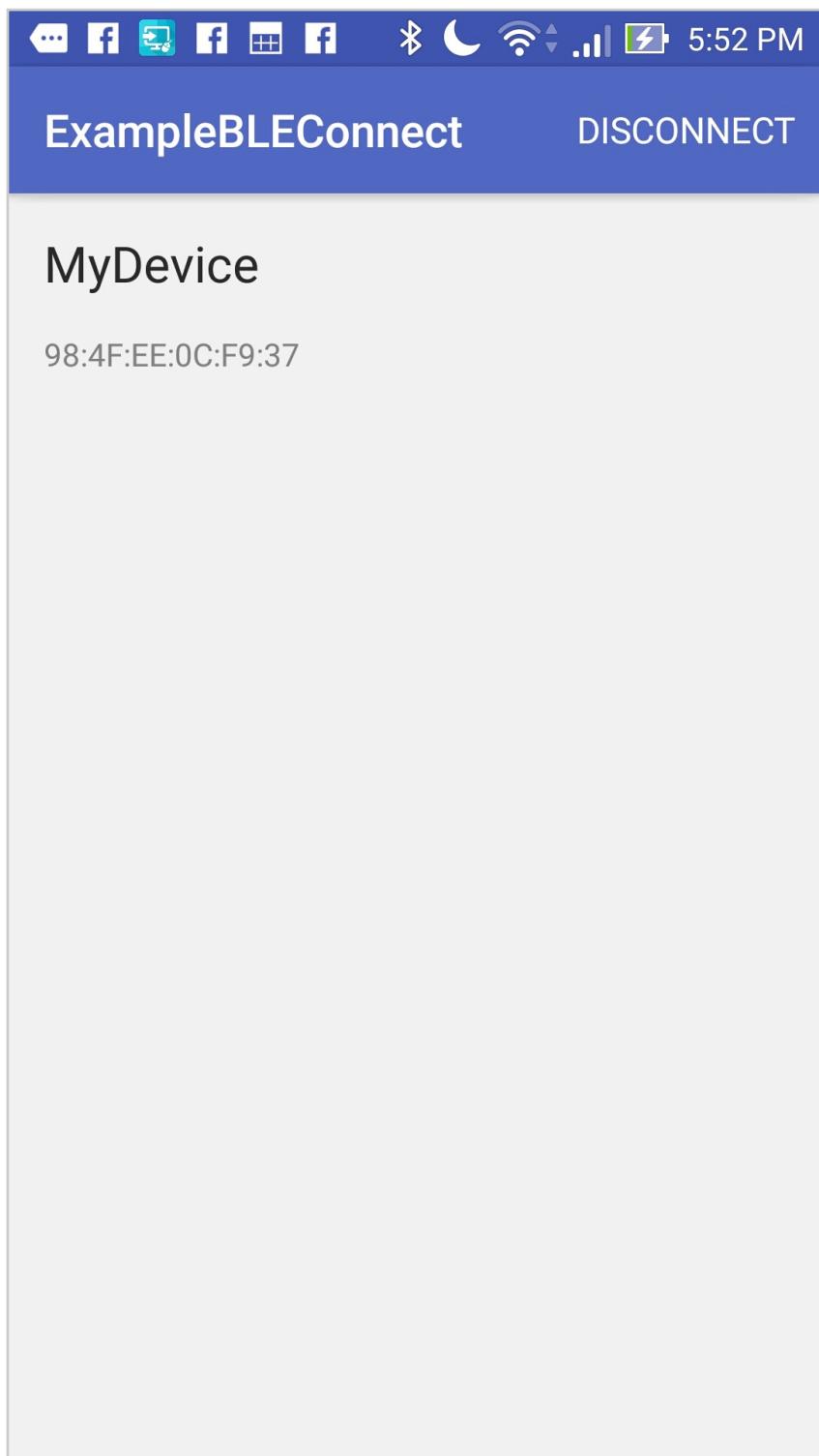
### Example 5-7. manifests/AndroidManifest.xml

```
...
<application ... >
...
    <activity
        android:name=".ConnectActivity"
        android:theme="@style/AppTheme.NoActionBar" />
</application>
...
```

The resulting App will be one that can scan ([Figure 5-4](#)) and connect to a advertising Peripheral ([Figure 5-5](#)).



**Figure 5-4. App screen after discovering Peripheral**



**Figure 5-5. App screen after connecting to Peripheral**

# Programming the Peripheral

This chapter will show how to handle connections from a nearby Central. When a Central connects or disconnects, the Peripheral is notified through an event callback.

It is important to know that Android peripherals have a bug that causes the MAC address to change between connections.

This callback exposes a lot of potential events, from connection and disconnection, to read and write events. These are packaged in the `BluetoothGattServerCallback` class, which must be implemented to respond to events triggered by the Bluetooth device.

Those events include:

**Table 4-2. BluetoothGattServerCallback**

Event	Description
<code>onConnectionStateChange</code>	Central connects or disconnects
<code>onCharacteristicReadRequest</code>	Central attempted to read a value from a Characteristic
<code>onCharacteristicWriteRequest</code>	Central attempted to write a value to a Characteristic
<code>onDescriptorWriteRequest</code>	Central attempted to change a Descriptor in a Characteristic
<code>onNotificationSent</code>	Central was notified of a change to a Characteristic

These events are implemented in code like this:

```
private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onConnectionStateChange(
        BluetoothDevice device,
        final int status,
```

```
    int newState)
{
    super.onConnectionStateChange(device, status, newState);
    Log.v(TAG, "Connected");
    if (status == BluetoothGatt.GATT_SUCCESS) {
        if (newState == BluetoothGatt.STATE_CONNECTED) {
            Log.v(TAG, "Central connected");
        } else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
            Log.v(TAG, "Central disconnected");
        }
    }
}

@Override
public void onCharacteristicReadRequest(
    BluetoothDevice device,
    int requestId,
    int offset,
    BluetoothGattCharacteristic characteristic)
{
    super.onCharacteristicReadRequest(
        device,
        requestId,
        offset,
        characteristic
    );
}

@Override
public void onNotificationSent(BluetoothDevice device, int status) {
    super.onNotificationSent(device, status);
}

@Override
public void onCharacteristicWriteRequest(
    BluetoothDevice device,
```

```
    int requestId,
    BluetoothGattCharacteristic characteristic,
    boolean preparedwrite,
    boolean responseNeeded,
    int offset,
    byte[] value)

{
    super.onCharacteristicWriteRequest(
        device, requestId,
        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
}
```

```
@Override
public void onDescriptorWriteRequest(
    BluetoothDevice device,
    int requestId,
    BluetoothGattDescriptor descriptor,
    boolean preparedwrite,
    boolean responseNeeded,
    int offset,
    byte[] value)
{
    super.onDescriptorWriteRequest(
        device,
        requestId,
        descriptor,
        preparedwrite,
        responseNeeded,
        offset, value
    );
}
```

```
};
```

In order for the Peripheral to handle these callbacks, it must open the BluetoothGattServer class from the BluetoothManager.

```
BluetoothGattServer mGattServer = \  
    bluetoothManager.openGattServer(context, mGattServerCallback);
```

## Putting It All Together

Create a new project called ExampleBLEPeripheral and copy everything from the previous example. This example will create an app that advertises a Bluetooth Low Energy Peripheral named "MyDevice" and allows connections from a Central.

The App will toggle switches when changes to the Bluetooth radio and the connectivity state of the Bluetooth Peripheral are changed.

## Resources

Define the text for the switch that shows the connectivity state, as well as the text from the previous project res/values/strings.xml:

### Example 5-8. res/values/strings.xml

```
<resources>  
    <string name="app_name">ExampleBLEPeripheral</string>  
    <string name="advertising_name_label">Advertising Name: </string>  
    <string name="bluetooth_on">Bluetooth On</string>  
    <string name="advertising">Advertising</string>  
    <string name="central_connected">Central Connected</string>  
</resources>
```

## Objects

BlePeripheral represents a Peripheral. It will handle connection and disconnections

### Example 5-9. java/example.com.exampleble/ble/BlePeripheral.java

```
package example.com.exampleble.ble;
public class BlePeripheral {
    /** Constants */
    private static final String TAG = MyBlePeripheral.class.getSimpleName();

    /** Peripheral and GATT Profile */
    public static final String ADVERTISING_NAME = "MyDevice";

    /** Advertising settings */

    // advertising mode can be one of:
    // - ADVERTISE_MODE_BALANCED,
    // - ADVERTISE_MODE_LOW_LATENCY,
    // - ADVERTISE_MODE_LOW_POWER
    int mAdvertisingMode = AdvertiseSettings.ADVERTISE_MODE_LOW_LATENCY;

    // transmission power mode can be one of:
    // - ADVERTISE_TX_POWER_HIGH
    // - ADVERTISE_TX_POWER_MEDIUM
    // - ADVERTISE_TX_POWER_LOW
    // - ADVERTISE_TX_POWER_ULTRA_LOW
    int mTransmissionPower = AdvertiseSettings.ADVERTISE_TX_POWER_HIGH;

    /** Callback Handlers */
    public BlePeripheralCallback mBlePeripheralCallback;

    /** Bluetooth Stuff */
    private BluetoothAdapter mBluetoothAdapter;
    private BluetoothLeAdvertiser mBluetoothAdvertiser;
    private BluetoothGattServer mGattServer;
```

```

/**
 * Construct a new Peripheral
 *
 * @param context The Application Context
 * @param blePeripheralCallback The callback handler
 *                      that interfaces with this Peripheral
 * @throws Exception Exception thrown if Bluetooth is not supported
 */
public MyBlePeripheral(
    final Context context,
    BlePeripheralCallback blePeripheralCallback) throws Exception
{
    mBlePeripheralCallback = blePeripheralCallback;
    // make sure Android device supports Bluetooth Low Energy
    if (!context.getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_BLUETOOTH_LE))
    {
        throw new Exception("Bluetooth Not Supported");
    }
    // get a reference to the Bluetooth Manager class,
    // which allows us to talk to talk to the BLE radio
    final BluetoothManager bluetoothManager =
        (BluetoothManager) context.getSystemService(
            Context.BLUETOOTH_SERVICE);
    mGattServer = bluetoothManager.openGattServer(
        context,
        mGattServerCallback
    );
    mBluetoothAdapter = bluetoothManager.getAdapter();
    // Beware: this function doesn't work on some platforms
    if(!mBluetoothAdapter.isMultipleAdvertisementSupported()) {
        throw new Exception ("Peripheral mode not supported");
    }
    mBluetoothAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();
    // Use this method instead for better support
    if (mBluetoothAdvertiser == null) {

```

```
        throw new Exception ("Peripheral mode not supported");
    }

}

/***
 * Get the system Bluetooth Adapter
 *
 * @return BluetoothAdapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}

/***
 * Start Advertising
 */
public void startAdvertising() {
    // set the Advertised name
    mBluetoothAdapter.setName(ADVERTISING_NAME);

    // Build Advertise settings with transmission power
    // and advertise speed
    AdvertiseSettings advertiseSettings = new AdvertiseSettings.Builder()
        .setAdvertiseMode(mAdvertisingMode)
        .setTxPowerLevel(mTransmissionPower)
        .setConnectable(true)
        .build();

    AdvertiseData.Builder advertiseBuilder = new AdvertiseData.Builder();
    // set advertising name
    advertiseBuilder.setIncludeDeviceName(true);
    AdvertiseData advertiseData = advertiseBuilder.build();
    // begin advertising
    mBluetoothAdvertiser.startAdvertising(
        advertiseSettings,
        advertiseData,
        mAdvertiseCallback
    );
}
```

```

    );
}

/***
 * Stop advertising
 */
public void stopAdvertising() {
    if (mBluetoothAdvertiser != null) {
        mBluetoothAdvertiser.stopAdvertising(mAdvertiseCallback);
        mBlePeripheralCallback.onAdvertisingStopped();
    }
}

/***
 * Peripheral State callbacks
 */
private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onConnectionStateChange(
        BluetoothDevice device,
        final int status,
        int newState)
    {
        super.onConnectionStateChange(device, status, newState);
        Log.v(TAG, "Connected");
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothGatt.STATE_CONNECTED) {
                mBlePeripheralCallback.onCentralConnected(device);
                stopAdvertising();
            } else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
                mBlePeripheralCallback.onCentralDisconnected(device);
                try {
                    startAdvertising();
                } catch (Exception e) {

```

```

        Log.e(TAG, "error starting advertising");
    }
}

}

};

/***
 * Advertisement Success/Failure Callbacks
 */
private AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
    @Override
    public void onStartSuccess(AdvertiseSettings settingsInEffect) {
        super.onStartSuccess(settingsInEffect);
        mBlePeripheralCallback.onAdvertisingStarted();
    }

    @Override
    public void onStartFailure(int errorCode) {
        super.onStartFailure(errorCode);
        mBlePeripheralCallback.onAdvertisingFailed(errorCode);
    }
};
}

```

BlePeripheralCallback communicates state changes from BlePeripheral to MainActivity.

## Example

### 5-10. java/example.com.exampleble/ble/callbacks/BlePeripheralCallback.java

```

package example.com.exampleble.ble.callbacks;
public abstract class BlePeripheralCallback {

    /***
     * Advertising Started

```

```
 */
public abstract void onAdvertisingStarted();

/**
 * Advertising Could not Start
 */
public abstract void onAdvertisingFailed(int errorCode);

/**
 * Advertising Stopped
 */
public abstract void onAdvertisingStopped();

/**
 * Central Connected
 *
 * @param bluetoothDevice the BluetoothDevice
 *                         representing the connected Central
 */
public abstract void onCentralConnected(
    final BluetoothDevice bluetoothDevice
);

/**
 * Central Disconnected
 *
 * @param bluetoothDevice the BluetoothDevice
 *                         representing the disconnected Central
 */
public abstract void onCentralDisconnected(
    final BluetoothDevice bluetoothDevice
);
}
```

## Activities

The previous app was able to advertise. This app will allow connections from a Central. When a Central connects, the mCentralConnectedSwitch will switch on.

Add the Switch and BlePeripheralCallbacks to MainActivity.

### Example 5-11. java/example.com.exampleble/MainActivity.java

```
...
/** UI Stuff */
private TextView mAdvertisingNameTV;
private Switch mBluetoothOnSwitch,
    mAdvertisingSwitch,
    mCentralConnectedSwitch;
...

/**
 * Load UI components
 */
public void loadUI() {
    mAdvertisingNameTV = (TextView) findViewById(R.id.advertising_name);
    mBluetoothOnSwitch = (Switch) findViewById(R.id.bluetooth_on);
    mAdvertisingSwitch = (Switch) findViewById(R.id.advertising);
    mCentralConnectedSwitch = (Switch) findViewById(R.id.central_connected);
    mAdvertisingNameTV.setText(MyBlePeripheral.ADVERTISING_NAME);
}

...
/**
 * Initialize the Bluetooth Radio
 */
public void initializeBluetooth() {
    // reset connection variables
    try {
        mMyBlePeripheral = new MyBlePeripheral(
            this,
            mBlePeripheralCallback
        );
    }
}
```

```

        } catch (Exception e) {
            Toast.makeText(
                this,
                "Could not initialize bluetooth", Toast.LENGTH_SHORT
            ).show();
            Log.e(TAG, e.getMessage());
            finish();
        }

        mBluetoothOnSwitch.setChecked
            mMyBlePeripheral.getBluetoothAdapter().isEnabled()
        );

        // should prompt user to open settings if Bluetooth is not enabled.
        if (!mMyBlePeripheral.getBluetoothAdapter().isEnabled()) {
            Intent enableBtIntent = new Intent(
                BluetoothAdapter.ACTION_REQUEST_ENABLE
            );
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        } else {
            startAdvertising();
        }

    }

    ...
}

/**
 * Event trigger when Central has connected
 *
 * @param bluetoothDevice
 */
public void onBleCentralConnected(final BluetoothDevice bluetoothDevice) {
    mCentralConnectedSwitch.setChecked(true);
}

/**
 * Event trigger when Central has disconnected
 * @param bluetoothDevice
 */

```

```
public void onBleCentralDisconnected(
    final BluetoothDevice bluetoothDevice)
{
    mCentralConnectedSwitch.setChecked(false);
}

/**
 * When the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mBleAdvertiseReceiver = \
    new AdvertiseReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();

        if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
            final int state = intent.getIntExtra(
                BluetoothAdapter.EXTRA_STATE,
                BluetoothAdapter.ERROR
            );
            switch (state) {
                case BluetoothAdapter.STATE_OFF:
                    Log.v(TAG, "Bluetooth turned off");
                    initializeBluetooth();
                    break;
                case BluetoothAdapter.STATE_TURNING_OFF:
                    break;
                case BluetoothAdapter.STATE_ON:
                    Log.v(TAG, "Bluetooth turned on");
                    startAdvertising();
                    break;
                case BluetoothAdapter.STATE_TURNING_ON:
                    break;
            }
        }
    }
}
```

```

        }

    };

}

/***
 * Respond to changes to the Bluetooth Peripheral state
 */
private final BlePeripheralCallback mBlePeripheralCallback = \
    new BlePeripheralCallback()
{
    public void onAdvertisingStarted() {
        Log.v(TAG, "Advertising started");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleAdvertisingStarted();
            }
        });
    }

    public void onAdvertisingFailed(int errorCode) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleAdvertisingFailed();
            }
        });
    }

    switch (errorCode) {
        case AdvertiseCallback.ADVERTISE_FAILED_ALREADY_STARTED:
            Log.e(
                TAG,
                "Failed to start; advertising is already started."
            );
            break;

        case AdvertiseCallback.ADVERTISE_FAILED_DATA_TOO_LARGE:
            Log.e(
                TAG,

```

```
        "Failed to start; advertised data is > 31 bytes."
    );
    break;
case AdvertiseCallback.ADVERTISE_FAILED_FEATURE_UNSUPPORTED:
    Log.e(
        TAG,
        "This feature is not supported on this platform."
    );
    break;
case AdvertiseCallback.ADVERTISE_FAILED_INTERNAL_ERROR:
    Log.e(
        TAG,
        "Operation failed due to an internal error."
    );
    break;
case AdvertiseCallback.ADVERTISE_FAILED_TOO_MANY_ADVERTISERS:
    Log.e(
        TAG,
        "Failed to start; advertising instance is unavailable."
    );
    break;
default:
    Log.e(TAG, "unknown problem");
}
}

public void onAdvertisingStopped() {
    Log.v(TAG, "Advertising stopped");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleAdvertisingStopped();
        }
    });
}

public void onCentralConnected(final BluetoothDevice bluetoothDevice) {
    Log.v(TAG, "Central connected");
}
```

```

        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleCentralConnected(blueoothDevice);
            }
        });
    }

    public void onCentralDisconnected(
        final BluetoothDevice blueoothDevice)
    {
        Log.v(TAG, "Central disconnected");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleCentralDisconnected(blueoothDevice);
            }
        });
    }
}

```

The Main Activity layout defines a new Switch that shows the Connectivity state of the Peripheral:

### **Example 5-12. res/layout/activity\_connect.xml**

```

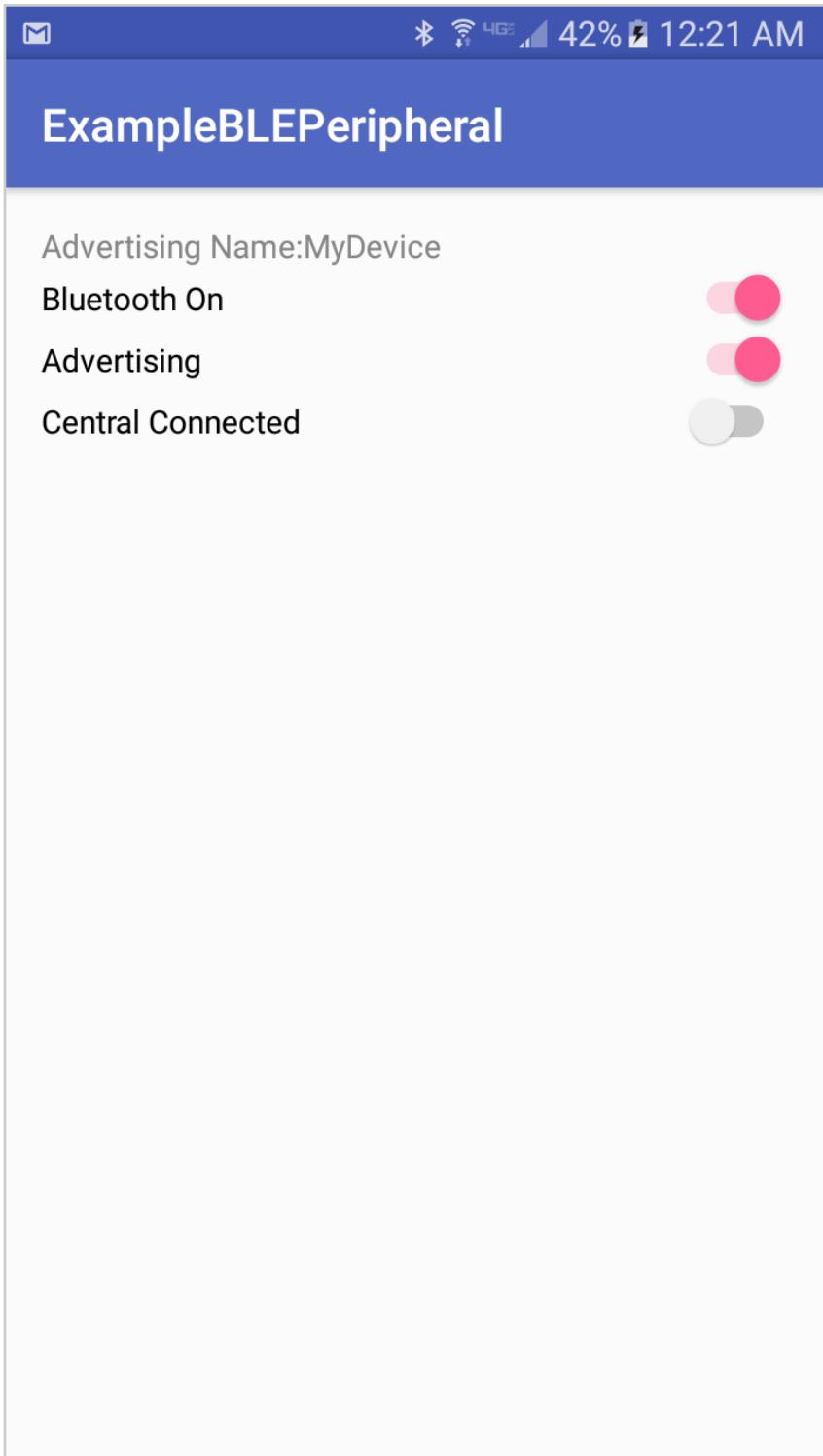
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity">
    <android.support.design.widget.AppBarLayout

```

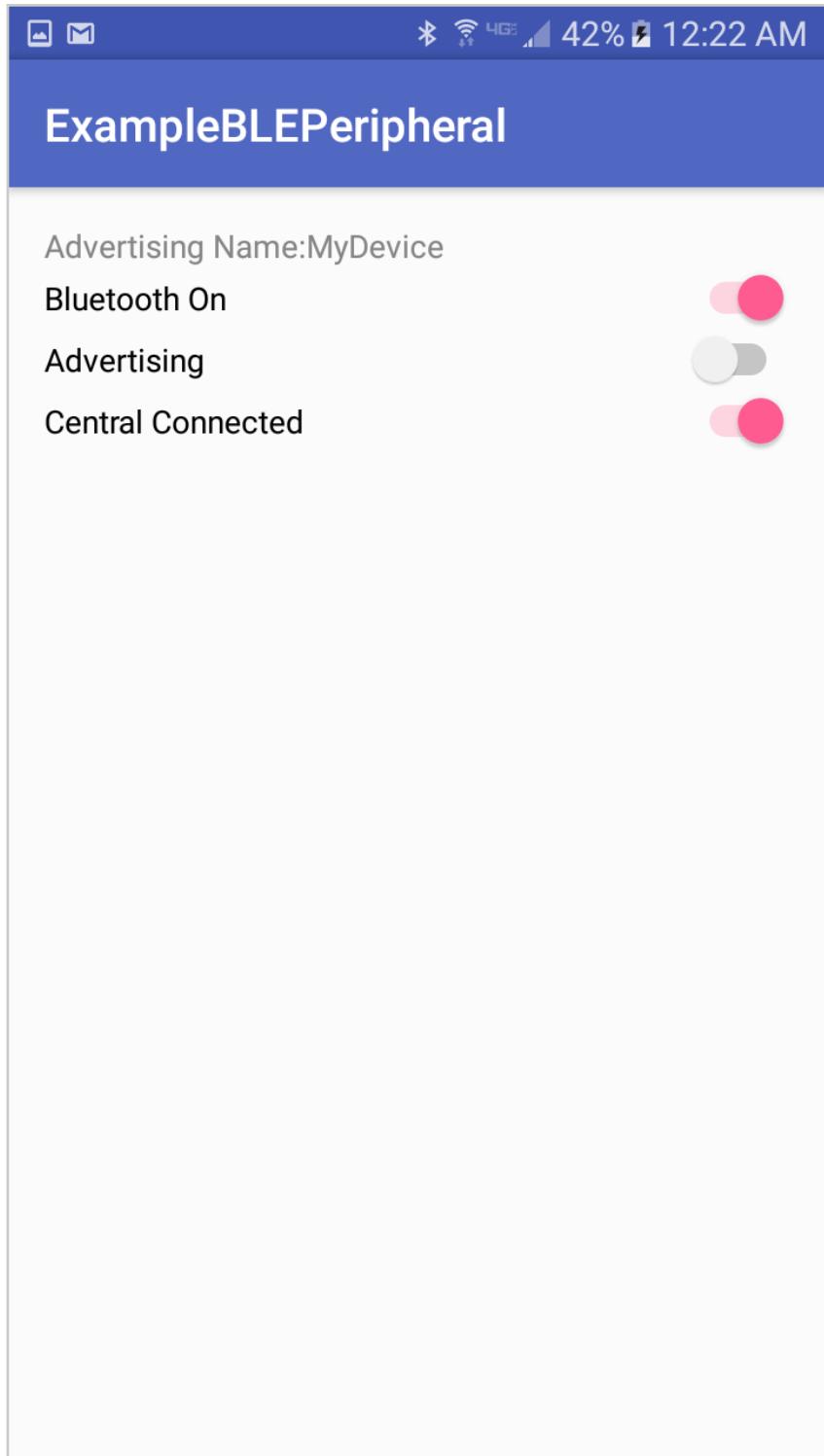
```
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:theme="@style/AppTheme.AppBarOverlay">
    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />
</android.support.design.widget.AppBarLayout>
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main" tools:context=".MainActivity">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <LinearLayout
            android:orientation="horizontal"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/advertising_name_label" />
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:id="@+id/advertising_name" />
        </LinearLayout>
    </LinearLayout>
```

```
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bluetooth_on"
    android:id="@+id/bluetooth_on" />
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/advertising"
    android:id="@+id/advertising" />
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/central_connected"
    android:id="@+id/central_connected" />
</LinearLayout>
</RelativeLayout>
</android.support.design.widget.CoordinatorLayout>
```

The resulting App will be one that Advertises a Peripheral ([Figure 5-6](#)) and allows connections from a Central ([Figure 5-7](#))



**Figure 5-6. App screen after before  
Central connected**



**Figure 5-7. App screen after Central  
connected**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter05>

# Services and Characteristics

Before data can be transmitted back and forth between a Central and Peripheral, the Peripheral must host a GATT Profile. That is, the Peripheral must have Services and Characteristics.

## Identifying Services and Characteristics

Each Service and Characteristic is identified by a Universally Unique Identifier (UUID). The UUID follows the pattern 0000XXXX-0000-1000-8000-00805f9b34fb, so that a 32-bit UUID 00002a56-0000-1000-8000-00805f9b34fb can be represented as 0x2a56.

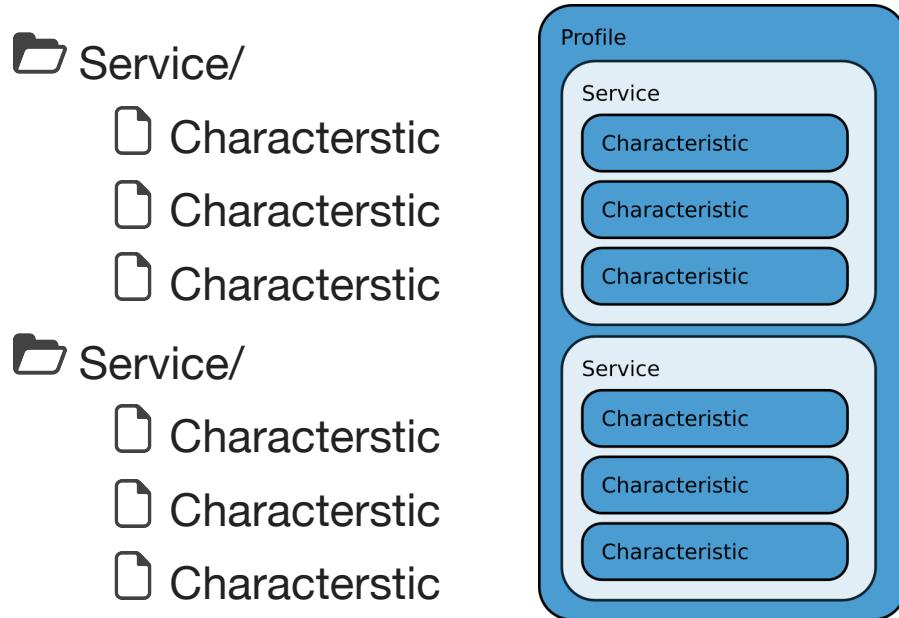
Some UUIDs are reserved for specific use. For instance any Characteristic with the 16-bit UUID 0x2a35 (or the 32-bit UUID 00002a35-0000-1000-8000-00805f9b34fb) is implied to be a blood pressure reading.

For a list of reserved Service UUIDs, see **Appendix IV: Reserved GATT Services**.

For a list of reserved Characteristic UUIDs, see **Appendix V: Reserved GATT Characteristics**.

## Generic Attribute Profile

Services and Characteristics describe a tree of data access points on the peripheral. The tree of Services and Characteristics is known as the Generic Attribute (GATT) Profile. It may be useful to think of the GATT as being similar to a folder and file tree (Figure 6-1).



**Figure 6-1. GATT Profile filesystem metaphor**

Characteristics act as channels that can be communicated on, and Services act as containers for Characteristics. A top level Service is called a Primary service, and a Service that is within another Service is called a Secondary Service.

## Permissions

Characteristics can be configured with the following attributes, which define what the Characteristic is capable of doing ([Table 6-1](#)):

**Table 6-1. Characteristic Permissions**

Descriptor	Description
<b>Read</b>	Central can read this Characteristic, Peripheral can set the value.
<b>Write</b>	Central can write to this Characteristic, Peripheral will be notified when the Characteristic value changes and Central will be notified when the write operation has occurred.
<b>Notify</b>	Central will be notified when Peripheral changes the value.

Because the GATT Profile is hosted on the Peripheral, the terms used to describe a Characteristic's permissions are relative to how the Peripheral accesses that Characteristic. Therefore, when a Central uploads data to the Peripheral, the Peripheral can "read" from the Characteristic. The Peripheral "writes" new data to the Characteristic, and can "notify" the Central that the data is altered.

## Data Length and Speed

It is worth noting that Bluetooth Low Energy has a maximum data packet size of 20 bytes, with a 1 Mbit/s speed.

## Programming the Central

The Central can be programmed to read the GATT Profile of the Peripheral after connection.

This is done by attaching a callback function to the connection gets called when the Central successfully connects to the Peripheral. When that happens, the Central re-

quests the Service listing from the connected Peripheral, using the following command:

```
mBluetoothGatt.discoverServices();
```

This will prompt the Central to begin scanning the Peripheral for Services. The same BluetoothGattCallback used to handle the connection state change offers a callback specifically for discovered Services. This callback is executed when the Services are discovered.

**Table 6-2. BluetoothGattCallback**

Event	Description
<b>onConnectionStateChange</b>	Triggered when a BLE Peripheral connects or disconnects
<b>onServicesDiscovered</b>	Triggered when GATT Services are discovered
<b>onCharacteristicRead</b>	Triggered when data has been downloaded form a GATT Characteristic
<b>onCharacteristicWrite</b>	Triggered when data has been uploaded to a GATT Characteristic
<b>onCharacteristicChanged</b>	Triggered when a GATT Characteristic's data has changed

There are Primary Services and Secondary services. Secondary Services are contained within other Services; Primary Services are not. The type of Service can be discovered by inspecting the SERVICE\_TYPE\_PRIMARY flag.

```
int type = item.getType();
boolean isPrimary = (type == BluetoothGattService.SERVICE_TYPE_PRIMARY);
```

Each Service may contain one or more characteristics, which the Central can use to communicate with the Peripheral. Once the Peripheral's Services have been discovered, a map of each Service's Characteristics can be listed:

```

public void onServicesDiscovered(BluetoothGatt gatt, int status) {
    // iterate through discovered services
    List<BluetoothGattService> gattServices = mBluetoothGatt.getServices();
    for (BluetoothGattService gattService : gattServices) {
        // iterate through service's characteristics
        List<BluetoothGattCharacteristic> characteristics =
            gattService.getCharacteristics();
        for (BluetoothGattCharacteristic characteristic : characteristics) {
            // Do something with each characteristic
        }
    }
}

```

Each Characteristic has certain permission properties that allow the Central to read, write, or receive notifications from it.

**Table 6-3. BluetoothGattCharacteristic Permissions**

Attribute	Permission	Description
<b>PROPERTY_READ</b>	Read	Central can read data altered by the Peripheral
<b>PROPERTY_WRITE_NO_RESPONSE</b>	Write	Central can send data. No response from Peripheral
<b>PROPERTY_WRITE</b>	Write	Central can send data, Peripheral reads
<b>PROPERTY_NOTIFY</b>	Notify	Central is notified as a result of a change

In Android, these properties are expressed as a binary integer. Permissions can be extracted like this:

```

int permissions = characteristic.getProperties(); // fetch permissions
boolean iswritable = permissions & BluetoothGattCharacteristic.PROPERTY_WRITE;
boolean iswritableNoResponse = permissions &
    BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE;
boolean isReadable = permissions & BluetoothGattCharacteristic.PROPERTY_READ;

```

```
boolean isNotifiable = permissions & \  
    BluetoothGattCharacteristic.PROPERTY_NOTIFY;
```

## A Note on Caching

Because Bluetooth was designed to be a low-power protocol, measures are taken to limit redundancy and power consumption through radio and CPU usage. As a result, a Peripheral's GATT Profile is cached on Android. This is not a problem for normal use, but when you are developing, it can be confusing to change Characteristic permissions and not see the updates reflected on Android.

To get around this, the device cache can be refreshed:

```
// use introspection to access hidden refresh method  
// in the Bluetooth connection  
Method localMethod = bluetoothGatt.getClass().getMethod(  
    "refresh",  
    new Class[0]  
);  
// if the refresh method exists, run it.  
if (localMethod != null) localMethod.invoke(bluetoothGatt, new Object[0]);
```

## Putting It All Together

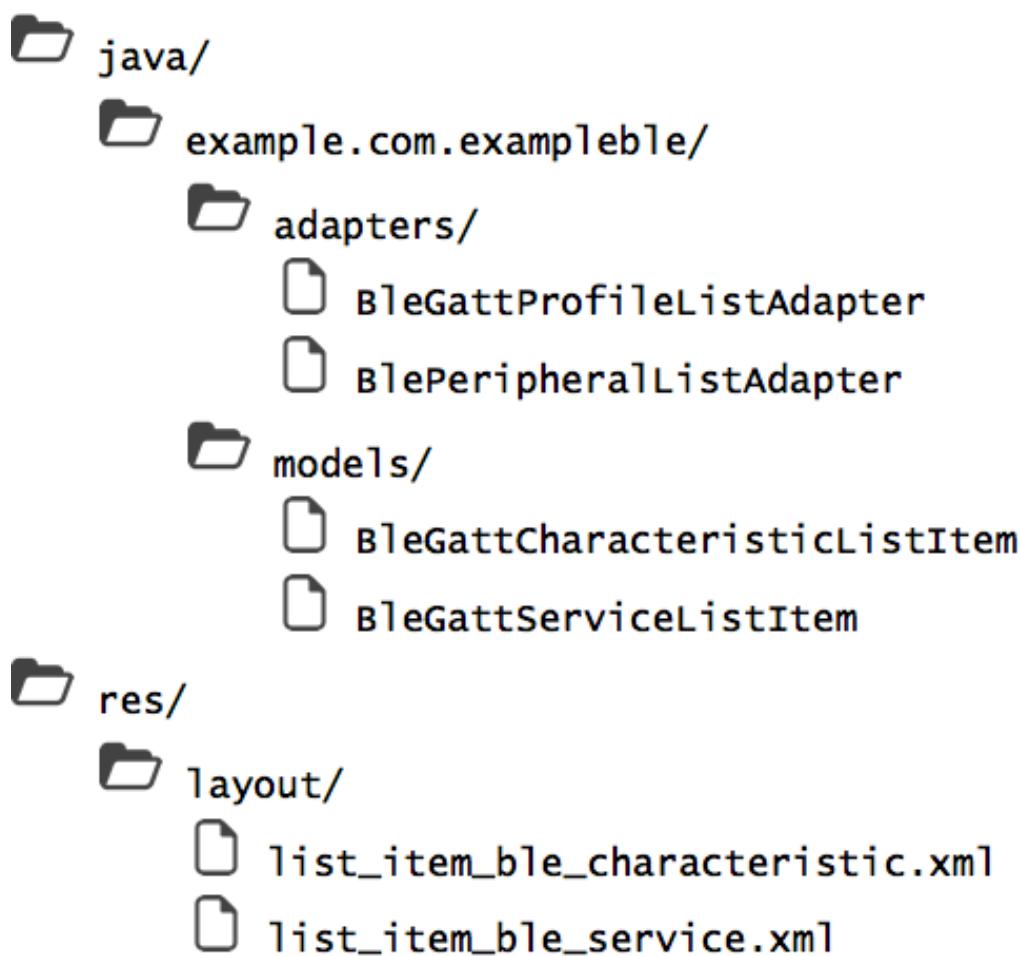
Create a new project called ExampleBleServices and copy everything from the previous example.

This app will work like the one from the previous chapter, except that once it connects to the Peripheral, it will also list the GATT Profile for that peripheral. The GATT Profile will be displayed in an expandable list ([Figure 6-3](#)).

00001800-0000-1000-8000-00805f9b34fb	Primary
00002a00-0000-1000-8000-00805f9b34fb	Readable
00002a01-0000-1000-8000-00805f9b34fb	Readable
00002a04-0000-1000-8000-00805f9b34fb	Readable
00001801-0000-1000-8000-00805f9b34fb	Primary

**Figure 6-3. GATT Profile downloaded from Peripheral**

Create new classes and new layout resource and menu files in res/with the following names ([Figure 6-4](#)).



**Figure 6-4. Project Structure**

## Resources

The Connect activity will now list the permissions of each characteristic. Add some definitions to res/values/strings.xml.

### Example 6-1. res/values/strings.xml

```
...
<string name="service_type_primary">Primary</string>
<string name="service_type_secondary">Secondary</string>
<string name="gatt_profile_list_empty">No GATT Profile Discovered</string>
<string name="property_read">Readable</string>
<string name="property_write">Writable</string>
<string name="property_notify">Notifiable</string>
<string name="property_none">No Access</string>
...
...
```

## Objects

Modify BlePeripheral.java to inspect the permissions of a Characteristic and to disable device caching.

### Example 6-2. java/example.com.exampleble/ble/BlePeripheral

```
...
public boolean refreshDeviceCache(BluetoothGatt gatt) {
    try {
        BluetoothGatt localBluetoothGatt = gatt;
        Method localMethod =
        localBluetoothGatt.getClass().getMethod("refresh", new Class[0]);
        if (localMethod != null) {
            boolean bool = ((Boolean) localMethod.invoke(
                localBluetoothGatt, new Object[0])).booleanValue();
            return bool;
        }
    }
    catch (Exception localException) {
```

```

        Log.e(TAG, "An exception occurred while refreshing device");
    }

    return false;
}

// true if characteristic is writable
public static boolean isCharacteristicWritable(
    BluetoothGattCharacteristic pchar) {
    return (pChar.getProperties() &
        (BluetoothGattCharacteristic.PROPERTY_WRITE |
        BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE)) != 0;
}

// true if characteristic is readable
public static boolean isCharacteristicReadable(
    BluetoothGattCharacteristic pchar) {
    return ((pChar.getProperties() &
        BluetoothGattCharacteristic.PROPERTY_READ) != 0);
}

// true if characteristic can send notifications
public static boolean isCharacteristicNotifiable(
    BluetoothGattCharacteristic pchar) {
    return (pChar.getProperties() &
        BluetoothGattCharacteristic.PROPERTY_NOTIFY) != 0;
}

...

```

## Views

The GATT Profile will be represented as an Expandable List View, with BleGattServiceListItem as the parent nodes and BleGattCharacteristicListItems as the child nodes.

Each BleGattServiceListItem will display the UUID and type of a Service.

### Example 6-3. java/example.com.exampleble/models/BleGattServiceListItem

```

package example.com.exampleble.models;
public class BleGattServiceListItem {

```

```

private final int mItemId;
private final BluetoothGattService mService;
public BleGattServiceListItem(BluetoothGattService gattService,
    int serviceItemID) {
    mItemId = serviceItemID;
    mService = gattService;
}
public int getItemId() { return mItemId; }
public UUID getUuid() { return mService.getUuid(); }
public int getType() { return mService.getType(); }
public BluetoothGattService getService() { return mService; }
}

```

The corresponding view will display details of a Service. It has two text fields: one for the UUID and one for the type (Primary or Secondary).

#### **Example 6-4. res/layout/list\_item\_ble\_service.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:background="@color/white">
    <TextView
        android:id="@+id/uuid"
        android:layout_width="276dp"
        android:layout_height="wrap_content"
        android:textSize="15sp"
        android:paddingTop="@dimen/text_padding"/>
    <TextView
        android:id="@+id/service_type"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="10sp"
        android:paddingTop="@dimen/text_padding"/>

```

```
</LinearLayout>
```

BleGattCharacteristicListItem is similar. It displays the UUID and the permissions of a Characteristic.

#### Example 6-5. java/example.com.exampleble/models/BleGattCharacteristicListItem

```
package example.com.exampleble.models;
public class BleGattCharacteristicListItem {
    private int mItemId;
    private BluetoothGattCharacteristic mCharacteristic;
    public BleGattCharacteristicListItem(
        BluetoothGattCharacteristic characteristic, int itemId) {
        mCharacteristic = characteristic;
        mItemId = itemId;
    }
    public int getItemId() { return mItemId; }
    public BluetoothGattCharacteristic getCharacteristic() {
        return mCharacteristic;
    }
}
```

To connect these into a collapsable list, extend a BaseExpandableListAdapter, which allows list items to be grouped and child list items to be created. The “parent” list items will represent Services and the “child” list items will represent Characteristics.

#### Example 6-6. java/example.com.exampleble/adapters/BLEServiceListAdapter

```
package example.com.exampleble.adapters;
public class BleGattProfileListAdapter extends BaseExpandableListAdapter {
    private final static String TAG = \
        BleGattProfileListAdapter.class.getSimpleName();

    private ArrayList<BleGattServiceListItem> mBleGattServiceListItems = \
        new ArrayList<>(); // list of services
```

```

// list of characteristics
private Map<Integer, ArrayList<BleGattCharacteristicListItem>> \
    mBlecharacteristicListItems = \
    new HashMap<Integer, ArrayList<BleGattCharacteristicListItem>>();

/**
 * Instantiate the class
 */
public BleGattProfileListAdapter() {
}

/** PARENT (GATT SERVICE) METHODS FOR COLLAPSIBLE TREE STRUCTURE **/

/**
 * Get the Service ListItem listed in a groupPosition
 * @param groupPosition the position of the ListItem
 * @return the ServiceListItem
 */
@Override
public BleGattServiceListItem getGroup(int groupPosition) {
    return mBleGattServiceListItems.get(groupPosition);
}

/**
 * How many Services are listed
 * @return number of Services
 */
@Override
public int getGroupCount() {
    return mBleGattServiceListItems.size();
}

/**
 * Add a new Service to be listed in the Listview
 * @param service the GATT Service to add
 */

```

```

public void addService(BluetoothGattService service) {
    int serviceItemID = mBleGattServiceListItems.size();
    BleGattServiceListItem serviceListItem = \
        new BleGattServiceListItem(service, serviceItemID);
    mBleGattServiceListItems.add(serviceListItem);
    mBleCharacteristicListItems.put(
        serviceItemID,
        new ArrayList<BleGattCharacteristicListItem>()
    );
}

/**
 * Add a new characteristic to be listed in the ListView
 *
 * @param service the Service that this characteristic belongs to
 * @param characteristic the Gatt Characteristic to add
 * @throws Exception if such a service does not exist
 */
public void addCharacteristic(
    BluetoothGattService service,
    BluetoothGattCharacteristic characteristic) throws Exception
{
    // find the Service in this listview with
    // matching UUID from the input service
    int serviceItemId = -1;
    for (BleGattServiceListItem bleGattServiceListItem :
        mBleGattServiceListItems)
    {
        //serviceItemId++;
        if (bleGattServiceListItem.getService().getUuid().equals(
            service.getUuid()))
        {
            Log.v(
                TAG,
                "Service found with UUID: "+service.getUuid().toString()
            );

```

```

        serviceItemId = bleGattServiceListItem.getItemId();
    }

}

// Throw an exception if no such service exists
if (serviceItemId < 0) throw new Exception(
    "Service not found with UUID: "+service.getUuid().toString()
);

// add characteristic to the end of the sub-list for the parent service
if (!mBleCharacteristicListItems.containsKey(serviceItemId)) {
    mBleCharacteristicListItems.put(
        serviceItemId,
        new ArrayList<BleGattCharacteristicListItem>()
    );
}

int characteristicItemId = mBleCharacteristicListItems.size();
BleGattCharacteristicListItem characteristicListItem = \
    new BleGattCharacteristicListItem(
        characteristic,
        characteristicItemId
    );
mBleCharacteristicListItems.get(
    serviceItemId
).add(characteristicListItem);
}

/***
 * Clear all ListItems from ListView
 */
public void clear() {
    mBleGattServiceListItems.clear();
    mBleCharacteristicListItems.clear();
}

/***
 * Get the Service ListItem at some position
*

```

```

    * @param position the position of the ListItem
    * @return BleGattServiceListItem at position
    */
@Override
public long getGroupId(int position) {
    return mBleGattServiceListItems.get(position).getitemId();
}

/**
 * This GroupViewHolder represents what UI components
 * are in each Service ListItem in the ListView
 */
public static class GroupViewHolder{
    public TextView mUuidTV;
    public TextView mServiceTypeTV;
}

/**
 * Generate a new Service ListItem for some known position in the ListView
 *
 * @param position the position of the Service ListItem
 * @param convertView An existing List Item
 * @param parent The Parent ViewGroup
 * @return The Service List Item
 */
@Override
public View getGroupView(
    int position,
    boolean isExpanded,
    View convertView,
    ViewGroup parent)
{
    View v = convertView;
    GroupViewHolder serviceListView;
    // if this ListItem does not exist yet, generate it
    // otherwise, use it
}

```

```

if(convertview == null) {
    // convert list_item_peripheral.xml.xml to a view
    LayoutInflater inflater = LayoutInflater.from(parent.getContext());
    v = inflater.inflate(
        R.layout.list_item_ble_service,
        parent,
        false
    );
    // match the UI stuff in the list Item to what's in the xml file
    serviceListItemView = new GroupViewHolder();
    serviceListItemView.mUuidTV = (TextView) v.findViewById(R.id.uuid);
    serviceListItemView.mServiceTypeTV = (TextView) v.findViewById(
        R.id.service_type
    );
    v.setTag( serviceListItemView );
} else {
    serviceListItemView = (GroupViewHolder) v.getTag();
}
// if there are known Services, create a ListItem that says so
// otherwise, display a ListItem with Bluetooth Service information
if (getGroupCount() <= 0) {
    serviceListItemView.mUuidTV.setText(
        R.string.peripheral_list_empty
    );
} else {
    BleGattServiceListItem item = getGroup(position);
    serviceListItemView.mUuidTV.setText(item.getUuid().toString());
    int type = item.getType();
    // Is this a primary or secondary service
    if (type == BluetoothGattService.SERVICE_TYPE_PRIMARY) {
        serviceListItemView.mServiceTypeTV.setText(
            R.string.service_type_primary
        );
    } else {
        serviceListItemView.mServiceTypeTV.setText(
            R.string.service_type_secondary
        );
    }
}

```

```

        );
    }

    return v;
}

/***
 * The IDs for this ListView do not change
 * @return <b>true</b>
 */
@Override
public boolean hasStableIds() {
    return true;
}

/** CHILD (GATT CHARACTERISTIC) METHODS FOR COLLAPSIBLE TREE STRUCTURE **/


/***
 * Get the characteristic ListItem at some position in the ListView
 * @param groupPosition the position of the Service ListItem in the ListView
 * @param childPosition the sub-position of the characteristic ListItem under
 * the service
 * @return BleGattCharacteristicListItem at some groupPosition, childPosition
 */
@Override
public BleGattCharacteristicListItem getChild(
    int groupPosition,
    int childPosition
) {
    return mBlecharacteristicListItems.get(
        groupPosition
    ).get(childPosition);
}

/***
 * Get the ID of a Charactersitic ListItem
 *

```

```

    * @param groupPosition the position of a Service ListItem
    * @param childPosition The sub-position of a Characteristic ListItem
    * @return the ID of the Characteristic ListItem
    */
}

@Override
public long getChildId(int groupPosition, int childPosition) {
    return mBleCharacteristicListItems.get(
        groupPosition
    ).get(childPosition).getItemId();
}

/**
 * How many Characteristics exist under a Service
 *
 * @param groupPosition The position of the Service ListItem in the ListView
 * @return the number of Characteristics in this Service
 */
@Override
public int getChildrenCount(int groupPosition) {
    return mBleCharacteristicListItems.get(groupPosition).size();
}

/**
 * Characteristics are selectable
 * because the user can click on them to open the TalkActivity
 *
 * @param groupPosition The Service ListItem position
 * @param childPosition The Characteristic ListItem sub-position
 * @return <b>true</b>
 */
@Override
public boolean isChildSelectable(int groupPosition, int childPosition) {
    return true;
}

/**

```

```

* This ChildViewHolder represents what
* UI components are in each characteristic List Item in the Listview
*/
public static class ChildViewHolder {
    // displays UUID
    public TextView mUuidTV;
    // displays when characteristic is readable
    public TextView mPropertyReadableTV;
    // displays when characteristic is writeable
    public TextView mPropertyWritableTV;
    // displays when characteristic is Notifiable
    public TextView mPropertyNotifiableTV;
    // displays when no access is given
    public TextView mPropertyNoneTV;
}

/**
 * Generate a new Characteristic ListItem
 * for some known position in the Listview
 *
 * @param groupPosition the position of the Service ListItem
 * @param childPosition the position of the Characterstic ListItem
 * @param convertView An existing List Item
 * @param parent The Parent ViewGroup
 * @return The Characteristic ListItem
 */
@Override
public View getChildView(
    final int groupPosition,
    final int childPosition,
    boolean isLastChild,
    View convertView,
    ViewGroup parent)
{
    View v = convertView;
    ChildViewHolder characteristicListView;

```

```

BleGattCharacteristicListItem item = getChild(
    groupPosition,
    childPosition
);
BluetoothGattCharacteristic characteristic = item.getCharacteristic();
// if this ListItem does not exist yet, generate it
// otherwise, use it
if (convertView == null) {
    LayoutInflater inflater = LayoutInflater.from(
        parent.getContext()
    );
    v = inflater.inflate(R.layout.list_item_ble_characteristic, null);
    // match the UI stuff in the list Item to what's in the xml file
    characteristicListView = new ChildViewHolder();
    characteristicListView.mUuidTV = \
        (TextView) v.findViewById(R.id.uuid);
    characteristicListView.mPropertyReadableTV = \
        (TextView)v.findViewById(R.id.property_read);
    characteristicListView.mPropertywritableTV = \
        (TextView)v.findViewById(R.id.property_write);
    characteristicListView.mPropertyNotifiableTV = \
        (TextView)v.findViewById(R.id.property_notify);
    characteristicListView.mPropertyNoneTV = \
        (TextView)v.findViewById(R.id.property_none);
} else {
    characteristicListView = (ChildViewHolder) v.getTag();
}
if (characteristicListView != null) {
    // display the UUID of the characteristic
    characteristicListView.mUuidTV.setText(
        characteristic.getUuid().toString()
    );
    // Display the read/write/notify attributes of the Characteristic
    if (BlePeripheral.isCharacteristicReadable(characteristic)) {
        characteristicListView.mPropertyReadableTV.setVisibility(
            View.VISIBLE

```

```
    );
} else {
    characteristicListView.mPropertyReadableTV.setVisibility(
        View.GONE
    );
}
if (BlePeripheral.isCharacteristicWritable(characteristic)) {
    characteristicListView.mPropertyWritableTV.setVisibility(
        View.VISIBLE
    );
} else {
    characteristicListView.mPropertyWritableTV.setVisibility(
        View.GONE
    );
}
if (BlePeripheral.isCharacteristicNotifiable(characteristic)) {
    characteristicListView.mPropertyNotifiableTV.setVisibility(
        View.VISIBLE
    );
} else {
    characteristicListView.mPropertyNotifiableTV.setVisibility(
        View.GONE
    );
}
if (!BlePeripheral.isCharacteristicNotifiable(characteristic) && \
    !BlePeripheral.isCharacteristicWritable(characteristic) && \
    !BlePeripheral.isCharacteristicReadable(characteristic))
{
    characteristicListView.mPropertyNoneTV.setVisibility(
        View.VISIBLE
    );
} else {
    characteristicListView.mPropertyNoneTV.setVisibility(
        View.GONE
    );
}
```

```
    }

    return v;
}

}
```

## Activities

Add functionality in the Connect Activity to scan for Services when a Bluetooth Peripheral is connected, and to list those Services when they are discovered.

### Example 6-7. java/example.com.exampleble/ConnectActivity.java

```
...

private ExpandableListView mGattProfileListView;
private BleGattProfileListAdapter mGattProfileListAdapter;
private TextView mGattProfileListEmptyTV;
... // onCreate, onResume, onPause
public void loadUI() {
    mPeripheralAdvertiseNameTV = \
        (TextView) findViewById(R.id.advertise_name);
    mPeripheralAddressTV = (TextView) findViewById(R.id.mac_address);
    mGattProfileListEmptyTV = \
        (TextView) findViewById(R.id.gatt_profile_list_empty);
    mGattProfileListView = \
        (ExpandableListView) findViewById(R.id.peripherals_list);
    mGattProfileListAdapter = new BleGattProfileListAdapter();
    mGattProfileListView.setAdapter(mGattProfileListAdapter);
    mGattProfileListView.setEmptyView(mGattProfileListEmptyTV);
}

...
// connect, disconnect, onBleConnected, etc
/***
 * Bluetooth Peripheral GATT Profile discovered. Update UI
 *
 * New in this chapter
*/
```

```
public void onBleServiceDiscoveryStopped() {
    // update UI to reflect the GATT profile of the connected Peripheral
    mProgressSpinner.setVisibility(false);
    mConnectItem.setVisibility(false);
    mDisconnectItem.setVisibility(true);
}

private BluetoothGattCallback mGattCallback = new BluetoothGattCallback() {
    /**
     * Characteristic value changed
     */
    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
                                        final BluetoothGattCharacteristic characteristic) {
        // we don't care about this here as we aren't communicating yet
    }

    /**
     * Peripheral connected or disconnected
     */
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt,
                                       int status, int newState) {
        // There has been a connection or a disconnection
        // with a Peripheral.
        // If this is a connection, update the UI to reflect the change
        // and discover the GATT profile of the connected Peripheral
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            Log.v(TAG, "Connected to peripheral");
            mBleConnected = true;
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    onBleConnected();
                    onBleServiceDiscoveryStarted();
                }
            });
        }
    }
}
```

```

        gatt.discoverServices();

    } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        mBlePeripheral.close();
        mBleConnected = false;
        if (mLeaveActivity) quitActivity();
    }
}

/**
 * Gatt Profile discovered
 */
@Override
public void onServicesDiscovered(
    BluetoothGatt bluetoothGatt,
    int status)
{
    // if services were discovered, iterate through and display them
    if (status == BluetoothGatt.GATT_SUCCESS) {
        List<BluetoothGattService> services =
            bluetoothGatt.getServices();
        for (BluetoothGattService service : services) {
            if (service != null) {
                Log.v(TAG, "Service uuid: " + service.getUuid());
                // add the gatt service to our list
                mGattProfileListAdapter.addService(service);
                // update the UI to reflect the new Service
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        mGattProfileListAdapter.notifyDataSetChanged();
                    }
                });
                // ask for this service's characteristics:
                List<BluetoothGattCharacteristic> characteristics =
                    service.getCharacteristics();
                for (BluetoothGattCharacteristic characteristic :
                    characteristics) {

```

```
        if (characteristic != null) {
            // add characteristics to the Service's list
            try {
                mGattProfileListAdapter.addCharacteristic(
                    service, characteristic);
                // update the ListView UI
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        mGattProfileListAdapter.
                            notifyDataSetChanged();
                    }
                });
            } catch (Exception e) {
                Log.e(TAG, e.getMessage());
            }
        }
    }

    disconnect(); // disconnect from the Peripheral
} else {
    Log.e(TAG, "Problem discovering GATT services");
}

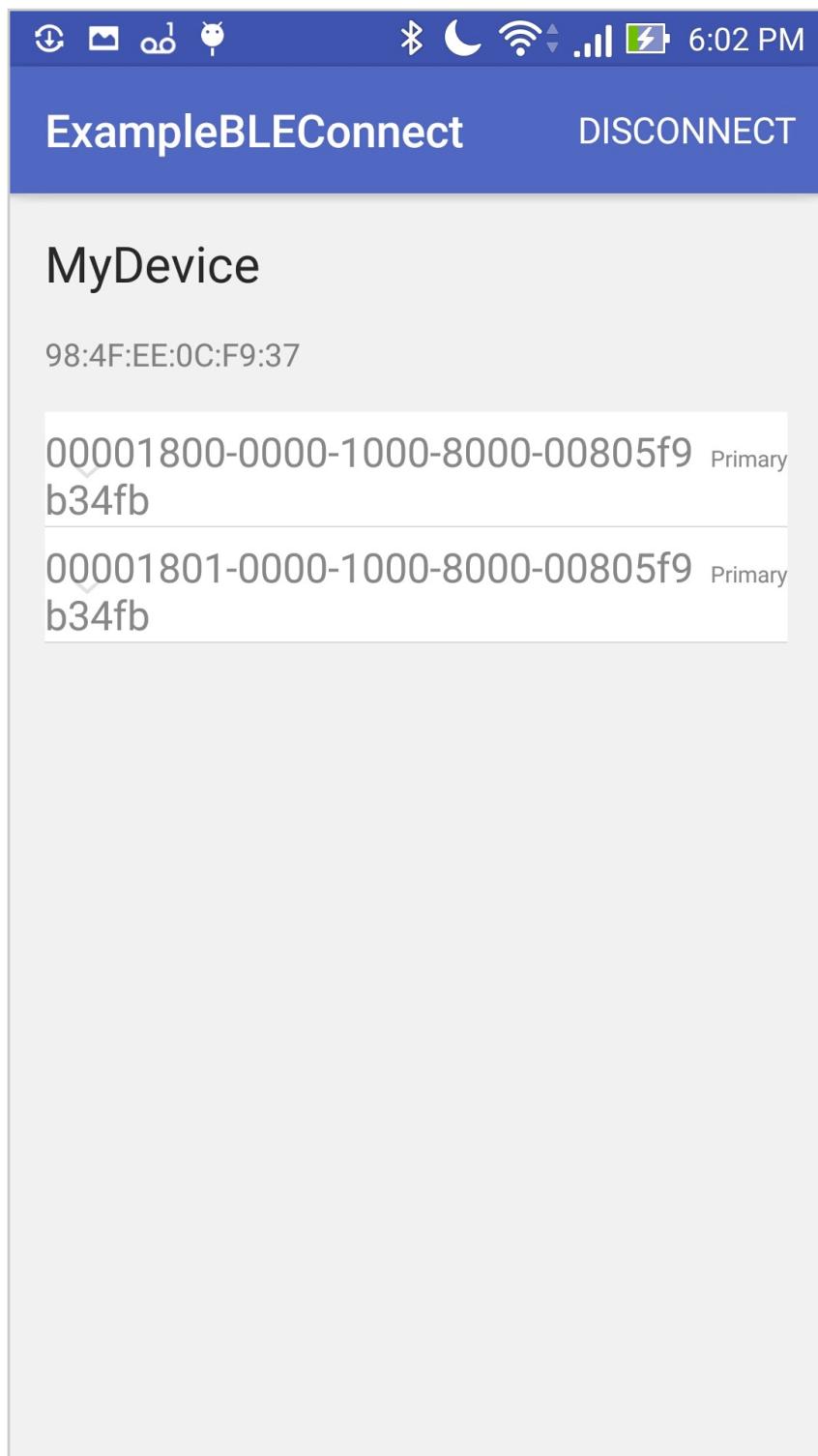
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        onBleServiceDiscoveryStopped();
    }
});
}

};

...

```

The resulting app will be able to connect to a Peripheral ([Figure 6-5](#)) and list the Services and Characteristics ([Figure 6-6](#)).



**Figure 6-5.** App screen showing primary Services hosted by a connected Peripheral



**Figure 6-6.** App screen showing an expanded primary Service, revealing several Characteristics

# Programming the Peripheral

The Peripheral can be programmed to host a GATT Profile - the tree structure of Services and Characteristics that a connected Central will use to communicate with the Peripheral.

Services are created and added to the Peripheral's GATT Server via the BluetoothGattServer class. Instantiation requires the definition of a BluetoothGattServerCallback, which triggers callbacks when Characteristics are read from or written to.

New services are created and added to the Peripheral like this:

```
// Service UUID
UUID mServiceUuid = UUID.fromString("0000180c-0000-1000-8000-00805f9b34fb");

// Create a new Service
mService = new BluetoothGattService(
    mServiceUuid,
    BluetoothGattService.SERVICE_TYPE_PRIMARY);

// Add Service to the GATT Server
mGattServer.addService(mService);

// add the Service UUID to the Advertising Packet
advertiseBuilder.addServiceUuid(new ParcelUuid(serviceUuid));
```

Secondary (nested) Services are created like this:

```
// Service UUID
UUID mSecondaryServiceUuid = \
    UUID.fromString("0000180d-0000-1000-8000-00805f9b34fb");

// Create a new Secondary Service
mSecondaryService = new BluetoothGattService(
    mSecondaryServiceUuid,
    BluetoothGattService.SERVICE_TYPE_SECONDARY);
```

```
// Add Secondary Service to an existing Service  
mService.addService(mSecondaryService);
```

New Characteristics are created and added to a Service like this:

```
// Create a new Characteristic  
mCharacteristic = new BluetoothGattCharacteristic(  
    UUID.fromString("00002a00-0000-1000-8000-00805f9b34fb") ,  
    BluetoothGattCharacteristic.PROPERTY_READ ,  
    BluetoothGattCharacteristic.PERMISSION_READ) ;  
  
// Add the characteristic to the Service  
mService.addCharacteristic(mCharacteristic);  
Characteristics must have both Properties and Permissions.
```

Properties are the access level that Centrals have to data, such as PROPERTY\_READ, PROPERTY\_WRITE, or PROPERTY\_NOTIFY.

Some common Properties are given below:

**Table 6-4. Common BluetoothGattCharacteristic Properties**

Property	Description
PROPERTY_READ	Characteristic may be read from
PROPERTY_WRITE_NO_RESPONSE	Characteristic supports notifications
PROPERTY_WRITE	Characteristic may be written to
PROPERTY_NOTIFY	Characteristic may be written to but will not respond with a confirmation

Permissions are the ability for a Central to change a Descriptor within the Characteristic. For example, if the Central needs to be able to subscribe to a Characteristic, it will need to have PERMISSION\_WRITE permission to that Characteristic.

**Table 6-5. Common BluetoothGattCharacteristic Permissions**

Permission	Description
PERMISSION_READ	Characteristic Descriptors may be read
PERMISSION_WRITE	Characteristic Descriptors may be modified

Properties and Permissions can be chained using the binary OR (|) operator.

For example, a read-only Characteristic that for which notifications can be subscribed to:

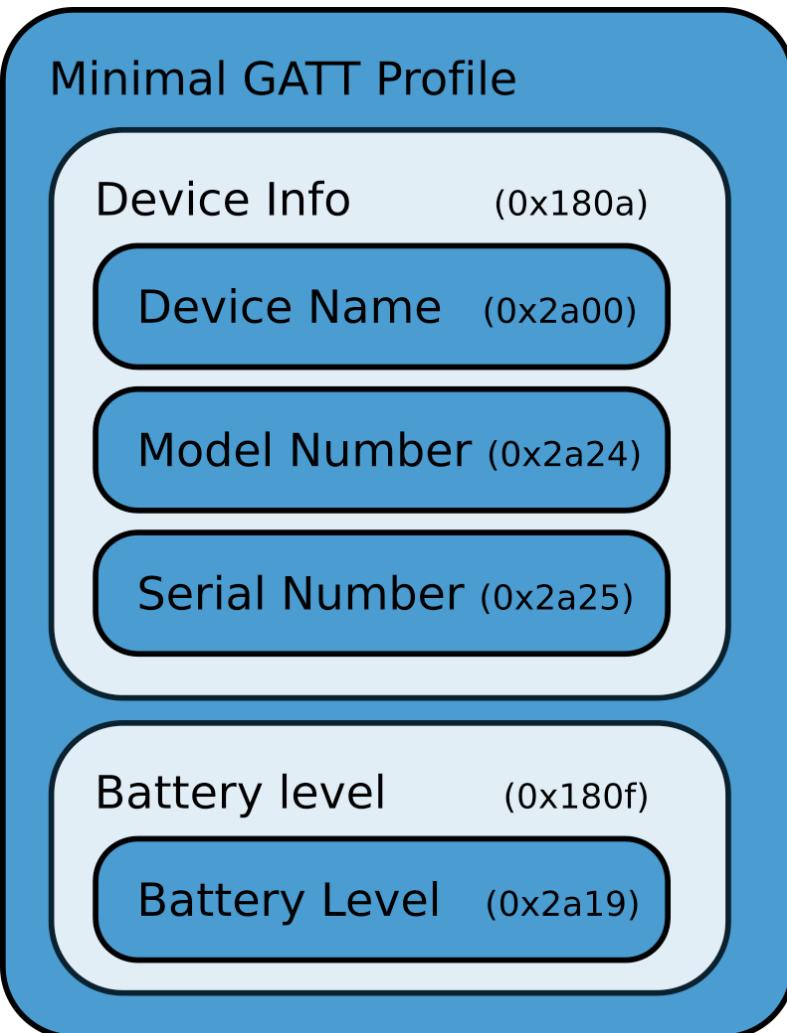
```
mCharacteristic = new BluetoothGattCharacteristic(  
    UUID.fromString("00002a00-0000-1000-8000-00805f9b34fb") ,  
    BluetoothGattCharacteristic.PROPERTY_READ | \  
        BluetoothGattCharacteristic.PROPERTY_NOTIFY,  
    BluetoothGattCharacteristic.PERMISSION_READ | \  
        BluetoothGattCharacteristic.PERMISSION_WRITE);
```

Or a write-only Characteristic without notifications:

```
mCharacteristic = new BluetoothGattCharacteristic(  
    UUID.fromString("00002a00-0000-1000-8000-00805f9b34fb") ,  
    BluetoothGattCharacteristic.PROPERTY_WRITE,  
    BluetoothGattCharacteristic.PERMISSION_READ);
```

## A Note on GATT Profile Best Practices

All Peripherals should contain Device information and a Battery Service, resulting in a minimal GATT profile for any Peripheral that resembles this ([Figure 6-7](#)):



**Figure 6-7. Minimal GATT**

This provides Central software, surveying tools, and future developers to better understand what each Peripheral is, how to interact with it, and what the battery capabilities are.

For pedagogical reasons, many of the examples will not include this portion of the GATT Profile.

## Putting It All Together

Create a new project called `ExampleBlePeripheral` and copy everything from the previous example.

This app will work like the one from the previous chapter, except that it will host a minimal GATT profile ([Figure 6-8](#)).

- Service: 000180a-000-1000-8000-00805f9-b34fb
  - └ Charateristic: 0002a00-000-1000-8000-00805f9-b34fb
  - └ Charateristic: 0002a24-000-1000-8000-00805f9-b34fb
  - └ Charateristic: 0002a25-000-1000-8000-00805f9-b34fb
- Service: 000180f-000-1000-8000-00805f9-b34fb
  - └ Charateristic: 0002a19-000-1000-8000-00805f9-b34fb

**Figure 6-8. update image for minimal Gatt Profile**

## Objects

Modify BlePeripheral.java to build a minimal Gatt Services profile, including mock data to populate the Characteristic values.

### Example 6-8. java/example.com.exampleble/ble/BlePeripheral

```
...
public static final String CHARSET = "ASCII";

// fake data for GATT Profile
private static final String MODEL_NUMBER = "1AB2";
private static final String SERIAL_NUMBER = "1234";

/** Peripheral and GATT Profile */
public static final String ADVERTISING_NAME = "MyDevice";

public static final UUID DEVICE_INFORMATION_SERVICE_UUID = \
    UUID.fromString("0000180a-0000-1000-8000-00805f9b34fb");
public static final UUID BATTERY_LEVEL_SERVICE = \
    UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb");

public static final UUID DEVICE_NAME_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a00-0000-1000-8000-00805f9b34fb");
public static final UUID MODEL_NUMBER_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a24-0000-1000-8000-00805f9b34fb");
```

```

public static final UUID SERIAL_NUMBER_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a04-0000-1000-8000-00805f9b34fb");

public static final UUID BATTERY_LEVEL_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb");

...
private BluetoothGattServer mGattServer;
private BluetoothGattService mDeviceInformationService;
private BluetoothGattService mBatteryLevelService;
private BluetoothGattCharacteristic mDeviceNameCharacteristic,
    mModelNumberCharacteristic,
    mSerialNumberCharacteristic,
    mBatteryLevelCharacteristic;

/**
 * Construct a new Peripheral
 *
 * @param context The Application Context
 * @param blePeripheralCallback The callback handler
 * that interfaces with this Peripheral
 * @throws Exception Exception thrown if Bluetooth is not supported
 */
public MyBlePeripheral(
    final Context context,
    BlePeripheralCallback blePeripheralCallback) throws Exception
{
    mBlePeripheralCallback = blePeripheralCallback;
    // make sure Android device supports Bluetooth Low Energy
    if (!context.getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_BLUETOOTH_LE))
    {
        throw new Exception("Bluetooth Not Supported");
    }
    // get a reference to the Bluetooth Manager class,
    // which allows us to talk to talk to the BLE radio
    final BluetoothManager bluetoothManager = (BluetoothManager) \

```

```

        context.getSystemService(Context.BLUETOOTH_SERVICE);
        mGattServer = bluetoothManager.openGattServer(
            context,
            mGattServerCallback
        );
        mBluetoothAdapter = bluetoothManager.getAdapter();
        if(!mBluetoothAdapter.isMultipleAdvertisementSupported()) {
            throw new Exception ("Peripheral mode not supported");
        }
        mBluetoothAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();
        // Use this method instead for better support
        if (mBluetoothAdvertiser == null) {
            throw new Exception ("Peripheral mode not supported");
        }
        setupDevice();
    }

    /**
     * Get the battery level
     */
    public int getBatteryLevel() {
        BatteryManager batteryManager = \
            (BatteryManager)mContext.getSystemService(BATTERY_SERVICE);
        return batteryManager.getIntProperty(
            BatteryManager.BATTERY_PROPERTY_CAPACITY
        );
    }

    /**
     * Set up the GATT profile
     */
    private void setupDevice() throws Exception {
        // set the device name
        mBluetoothAdapter.setName(ADVERTISING_NAME);

```

```

// build characteristics
mDeviceInformationService = new BluetoothGattService(
    DEVICE_INFORMATION_SERVICE_UUID,
    BluetoothGattService.SERVICE_TYPE_PRIMARY
);
mDeviceNameCharacteristic = new BluetoothGattCharacteristic(
    DEVICE_NAME_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_READ,
    BluetoothGattCharacteristic.PERMISSION_READ);
mModelNumberCharacteristic = new BluetoothGattCharacteristic(
    MODEL_NUMBER_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_READ,
    BluetoothGattCharacteristic.PERMISSION_READ);
mSerialNumberCharacteristic = new BluetoothGattCharacteristic(
    SERIAL_NUMBER_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_READ,
    BluetoothGattCharacteristic.PERMISSION_READ);
mBatteryLevelService = new BluetoothGattService(
    BATTERY_LEVEL_SERVICE,
    BluetoothGattService.SERVICE_TYPE_PRIMARY
);
mBatteryLevelCharacterstic = new BluetoothGattCharacteristic(
    BATTERY_LEVEL_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_READ,
    BluetoothGattCharacteristic.PERMISSION_READ);

// Add Characteristics to Services
mDeviceInformationService.addCharacteristic(
    mDeviceNameCharacteristic
);
mDeviceInformationService.addCharacteristic(
    mModelNumberCharacteristic
);
mDeviceInformationService.addCharacteristic(
    mSerialNumberCharacteristic
);

```

```

mBatteryLevelService.addCharacteristic(mBatteryLevelCharacterstic);
// put in fake values for the Characteristic
mDeviceNameCharacteristic.setValue(ADVERTISING_NAME.getBytes(CHARSET));
mModelNumberCharacteristic.setValue(MODEL_NUMBER.getBytes(CHARSET));
mSerialNumberCharacteristic.setValue(SERIAL_NUMBER.getBytes(CHARSET));
// add Services to Peripheral
mGattServer.addService(mDeviceInformationService);
mGattServer.addService(mBatteryLevelService);
// update the battery level
// every BATTERY_STATUS_CHECK_TIME_MS milliseconds
TimerTask updateBatteryTask = new TimerTask() {
    @Override
    public void run() {
        mBatteryLevelCharacterstic.setValue(
            getBatteryLevel(),
            BluetoothGattCharacteristic.FORMAT_UINT8,
            0
        );
    }
};

Timer randomStringTimer = new Timer();
// schedule the battery update and run it once immediately
randomStringTimer.schedule(
    updateBatteryTask,
    0,
    BATTERY_STATUS_CHECK_TIME_MS
);
}

...
/***
 * Start Advertising
 *
 * @throws Exception Exception thrown if Bluetooth Peripheral
 * mode is not supported
 */
public void startAdvertising() throws Exception {

```

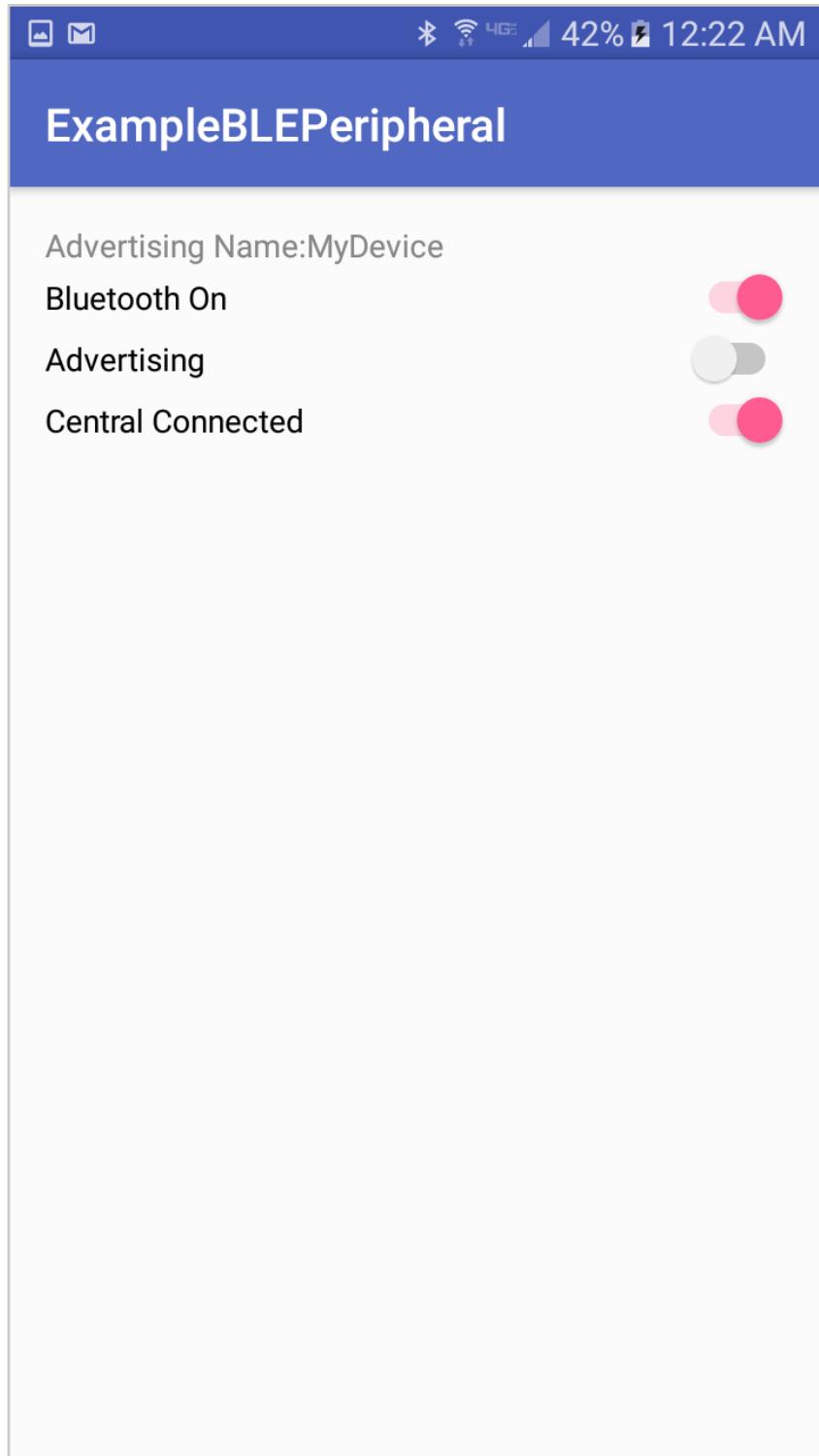
```
// set the device name
mBluetoothAdapter.setName(ADVERTISING_NAME);

// Build Advertise settings with transmission power and advertise speed
AdvertiseSettings advertiseSettings = new AdvertiseSettings.Builder()
    .setAdvertiseMode(mAdvertisingMode)
    .setTxPowerLevel(mTransmissionPower)
    .setConnectable(true)
    .build();

AdvertiseData.Builder advertiseBuilder = new AdvertiseData.Builder();
// set advertising name
advertiseBuilder.setIncludeDeviceName(true);
// add Services to Advertising Data
advertiseBuilder.addServiceUuid(new ParcelUuid(
    DEVICE_INFORMATION_SERVICE_UUID
));
advertiseBuilder.addServiceUuid(new ParcelUuid(BATTERY_LEVEL_SERVICE));
AdvertiseData advertiseData = advertiseBuilder.build();
// begin advertising
mBluetoothAdvertiser.startAdvertising(
    advertiseSettings,
    advertiseData,
    mAdvertiseCallback
);
}

...
}
```

The resulting app will be able to host a minimal GATT Profile ([Figure 6-9](#)).



**Figure 6-9. App screen showing Services in GATT Profile**

## Example code

The code for this chapter is available online

at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter06>

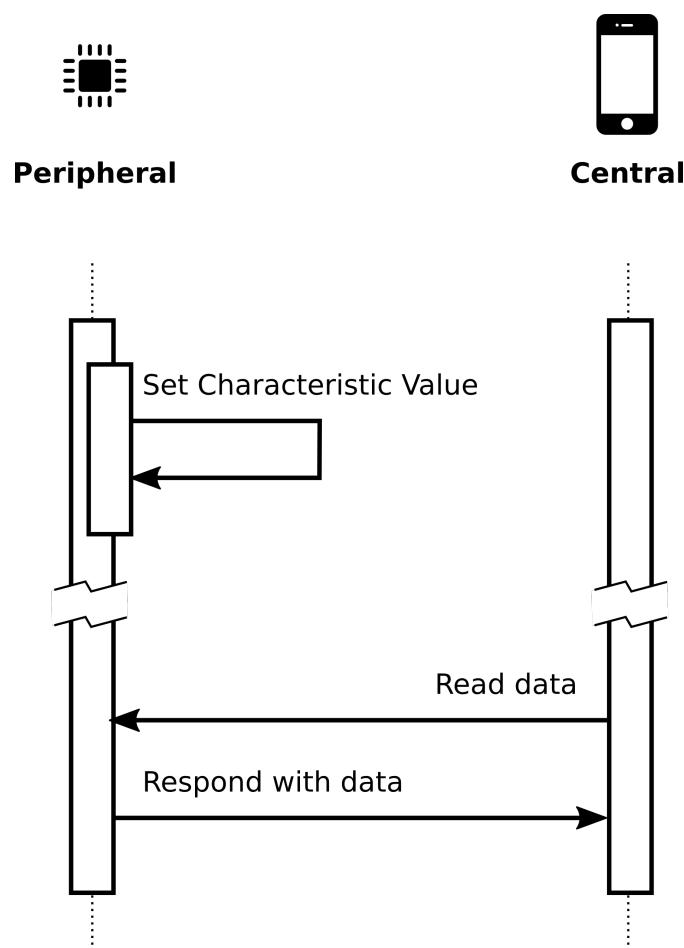
# Reading Data from a Peripheral

The real value of Bluetooth Low Energy is the ability to transmit data wirelessly.

Bluetooth Peripherals are passive, so they don't push data to a connected Central. Instead, Centrals make a request to read data from a Characteristic. This can only happen if the Characteristic enables the Read Attribute.

This is called "reading a value from a Characteristic."

Therefore, if a Peripheral changes the value of a Characteristic, then later a Central downloads data from the Peripheral, the process looks like this ([Figure 7-1](#)):



**Figure 7-1. The process of a Central reading data from a Peripheral**

A Central can read a Characteristic repeatedly, regardless if Characteristic's value has changed.

# Programming the Central

Before reading data from a connected Peripheral, it may be useful to know if a Characteristic provides read permission. Read permission can be read by getting the Characteristic property bit map and isolating the read property from it, like this:

```
int permissions = characteristic.getProperties(); // fetch permissions  
boolean isReadable = permissions & BluetoothGattCharacteristic.PROPERTY_READ;
```

Once the Central has a Bluetooth GATT connection and has access to a Characteristic with which to communicate with a connected Peripheral, the Central can request to read data from that Characteristic like this:

```
bluetoothGatt.readCharacteristic(characteristic);
```

This will initiate a read request from the Central to the Peripheral.

When the Central finishes reading data from the Peripheral's Characteristic, the onCharacteristicRead method is triggered the BluetoothGattCallback object.

**Table 7-1. BluetoothGattCallback**

Event	Description
<b>onConnectionStateChange</b>	Triggered when a BLE Peripheral connects or disconnects
<b>onServicesDiscovered</b>	Triggered when GATT Services are discovered
<b>onCharacteristicRead</b>	Triggered when data has been downloaded form a GATT Characteristic
<b>onCharacteristicWrite</b>	Triggered when data has been uploaded to a GATT Characteristic
<b>onCharacteristicChanged</b>	Triggered when a GATT Characteristic's data has changed

In this callback, the Characteristic's byte value can be read using the characteristic.getValue() method.

```
private final BluetoothGattCallback gattCallback = new BluetoothGattCallback() {  
    @Override  
    public void onCharacteristicRead(  
        BluetoothGatt gatt,  
        BluetoothGattCharacteristic characteristic,  
        int status)  
    {  
        byte[] data = characteristic.getValue();  
    }  
};
```

From here the data can be converted into any format, including a String or an Integer.

```
byte[] data = characteristic.getValue();  
  
// convert to string  
String string = new String(data, "ASCII");  
  
// convert to signed integer  
int intValue = ByteBuffer.wrap(data, ByteOrder.LITTLE_ENDIAN).getInt();  
  
// convert to float  
float floatValue = ByteBuffer.wrap(data, ByteOrder.LITTLE_ENDIAN).getFloat();
```

One caveat about transmitting numeric data using Android is that Bluetooth Low Energy transmits data in Little-Endian format and Java uses Big-Endian to process data.

# Putting It All Together

Create new classes and new layout resource and menu files in res/with the following names ([Figure 7-2](#)).



**Figure 7-2. Added project files**

## Resources

Add strings to support a “Read” button and a “Response” label for the Connect Activity.

### Example 7-1. res/values/strings.xml

```
...
<string name="read_button">Read</string>
<string name="response_label">Response</string>
```

```
<string name="property_read">Readable</string>
```

```
...
```

## Objects

Modify the BlePeripheral class to include a method that checks if a Characteristic is readable, and one that initiates a read request from a Characteristic.

### Example 7-2. java/example.com.exampleble/ble/BlePeripheral

```
...  
    ...  
  
    /**  
     * Check if a characteristic has read permissions  
     *  
     * @return Returns <b>true</b> if property is Readable  
     */  
  
    public static boolean isCharacteristicReadable(  
        BluetoothGattCharacteristic characteristic) {  
        return ((characteristic.getProperties() &  
            BluetoothGattCharacteristic.PROPERTY_READ) != 0);  
    }  
  
    /**  
     * Request a data/value read from a BLE characteristic  
     *  
     * New in this chapter  
     *  
     * @param characteristic  
     */  
  
    public void readValueFromCharacteristic(  
        final BluetoothGattCharacteristic characteristic) {  
        // Reading a characteristic requires both requesting the read and  
        // handling the callback that is sent when the read is successful  
        mBluetoothGatt.readCharacteristic(characteristic);  
    }  
...
```

Modify the BleGattProfileListAdapter to display a TextView when a Characteristic is writable:

### Example 7-3. java/example.com.exampleble/adapters/BleGattProfileListAdapter

```
...
/***
 * These UI components are in each characteristic List Item in the ListView
 */
public static class ChildViewHolder{
    public TextView mUuidTV; // displays UUID
    public TextView mPropertyReadableTV; // characteristic is readable
    public TextView mPropertyNoneTV; // displays when no access is given
}
/***
 * Generate a new characteristic ListItem
 * for some known position in the ListView
 *
 * @param groupPosition the position of the Service ListItem
 * @param childPosition the position of the characteristic ListItem
 * @param convertView An existing ListItem
 * @param parent The Parent ViewGroup
 * @return The characteristic ListItem
 */
@Override
public View getChildView(final int groupPosition, final int childPosition,
    boolean isLastChild, View convertView, ViewGroup parent) {
    View v = convertView;
    ChildViewHolder characteristicListView;
    BleGattCharacteristicListItem item =
        getChild(groupPosition, childPosition);
    BluetoothGattCharacteristic characteristic = item.getCharacteristic();
    // if this ListItem does not exist yet, generate it
    // otherwise, use it
    if (convertView == null) {
        LayoutInflator inflater = LayoutInflator.from(
            parent.getContext())
        ...
    }
}
```

```

);
v = inflater.inflate(
    R.layout.list_item_ble_characteristic,
    null
);
// match the UI stuff in the list Item to what's in the xml file
characteristicListView = new ChildViewHolder();
characteristicListView.mUuidTV =
    (TextView) v.findViewById(R.id.uuid);
characteristicListView.mPropertyReadableTV =
    (TextView)v.findViewById(R.id.property_read);
characteristicListView.mPropertyNoneTV =
    (TextView)v.findViewById(R.id.property_none);
} else {
    characteristicListView = (ChildViewHolder) v.getTag();
}
if (characteristicListView != null) {
    // display the UUID of the characteristic
    characteristicListView.mUuidTV.setText(
        characteristic.getUuid().toString());
    // Display the read/write/notify attributes of the Characteristic
    if (BlePeripheral.isCharacteristicReadable(characteristic)) {
        characteristicListView.mPropertyReadableTV.
            setVisibility(View.VISIBLE);
    } else {
        characteristicListView.mPropertyReadableTV.
            setVisibility(View.GONE);
    }
    if (!BlePeripheral.isCharacteristicReadable(characteristic)) {
        characteristicListView.mPropertyNoneTV.
            setVisibility(View.VISIBLE);
    } else {
        characteristicListView.mPropertyNoneTV.
            setVisibility(View.GONE);
    }
}

```

```
    return v;
}

...

```

Create a DataConverter to converted byte arrays and other data into various formats, useful for debugging or representing Characteristic values in as an int, float, or a String:

#### Example 7-4. `java/example.com.exampleble/utilities/DataConverter`

```
package example.com.exampleble.utilities;

public class DataConverter {

    /**
     * convert bytes to hexadecimal for debugging purposes
     *
     * @param bytes
     * @return Hexadecimal String representation of the byte array
     */
    public static String bytesToHex(byte[] bytes) {
        if (bytes.length <=0) return "";
        char[] hexArray = "0123456789ABCDEF".toCharArray();
        char[] hexChars = new char[bytes.length * 3];
        for ( int j = 0; j < bytes.length; j++ ) {
            int v = bytes[j] & 0xFF;
            hexChars[j * 3] = hexArray[v >>> 4];
            hexChars[j * 3 + 1] = hexArray[v & 0x0F];
            hexChars[j * 3 + 2] = 0x20; // space
        }
        return new String(hexChars);
    }

    /**
     * convert bytes to an integer in Little Endian for debugging purposes
     *
     * @param bytes a byte array
     * @return int integer representation of byte array
     */
}
```

```
 */
public static int bytesToInt(byte[] bytes) {
    if ((bytes.length % 2) == 1) {
        return ByteBuffer.wrap(bytes).getInt();
    } else {
        return 0;
    }
}

/**
 * convert bytes to a Float
 * @param bytes
 * @return float
 */
public static float bytesToFloat(byte[] bytes) {
    return ByteBuffer.wrap(bytes).getFloat();
}

/**
 * convert bytes to an integer in Little Endian for debugging purposes
 *
 * @param bytes a byte array
 * @return integer integer representation of byte array
 */
public static long bytesToLong(byte[] bytes) {
    long val = 0;
    for (int i=0; i < bytes.length; i++) {
        val += ((bytes[i] & 0xff) << (8*i));
    }
    return val;
}
}
```

## Views

Modify the list\_item\_ble\_characteristic file to display a TextView if a Characteristic is writable:

### Example 7-5. res/layout/ble/list\_item\_ble\_characteristic.xml

```
...
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_weight="2">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_read"
        android:visibility="gone"
        android:id="@+id/property_read"
        android:paddingLeft="@dimen/activity_horizontal_margin" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_none"
        android:visibility="gone"
        android:id="@+id/property_none"
        android:paddingLeft="@dimen/activity_horizontal_margin" />
</LinearLayout>
...
```

## Activities

Modify the Connect Activity to launch the Talk Activity when the user clicks on a Characteristic in the List View.

### Example 7-6. java/example.com.exampleble/ConnectActivity.java

```
...
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    loadUI();
    attachCallbacks();
    ...
}

/**
 * Attach callback handlers to UI elements
 */
public void attachCallbacks() {
    // When a user clicks on a Service
    // expand the Characteristics belonging to that Service.
    // When a user clicks on a Characteristic, open in TalkActivity
    mGattProfileListView.setOnChildClickListener(
        new ExpandableListView.OnChildClickListener() {
            @Override
            public boolean onChildClick(ExpandableListView parent,
                View v, int groupPosition, int childPosition, long id) {
                Log.v(TAG, "List view click: groupPosition: " + groupPosition +
                    ", childPosition: " + childPosition);
                BluetoothGattService service =
                    mGattProfileListAdapter.getGroup(
                        groupPosition
                    ).getService();
                BluetoothGattCharacteristic characteristic =
                    mGattProfileListAdapter.getChild(groupPosition,
                        childPosition).getCharacteristic();
                // start the Talk Activity and connect
            }
        }
    );
}
```

```

        Intent intent = new Intent(
            getBaseContext(), TalkActivity.class
        );
        intent.putExtra(
            TalkActivity.PERIPHERAL_NAME,
            mBlePeripheralName
        );
        intent.putExtra(TalkActivity.PERIPHERAL_MAC_ADDRESS_KEY,
            mPeripheralMacAddress);
        intent.putExtra(TalkActivity.CHARACTERISTIC_KEY,
            characteristic.getUuid().toString());
        intent.putExtra(TalkActivity.SERVICE_KEY,
            service.getUuid().toString());
        Log.v(
            TAG,
            "Setting intent: " + \
            TalkActivity.CHARACTERISTIC_KEY + \
            ": " + characteristic.getUuid().toString()
        );
        startActivity(intent);
        return false;
    }
}

...

```

The new Talk activity will connect to a Peripheral and enable the user to read data from a Characteristic on that Peripheral.

### **Example 7-7. java/example.com.exampleble/TalkActivity.java**

```

package example.com.exampleble;
public class TalkActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = TalkActivity.class.getSimpleName();
    private static final String CHARACTER_ENCODING = "ASCII";

```

```

public static final String PERIPHERAL_NAME = \
    "com.example.com.exampleble.PERIPHERAL_NAME";
public static final String PERIPHERAL_MAC_ADDRESS_KEY = \
    "com.example.com.exampleble.PERIPHERAL_MAC_ADDRESS";
public static final String CHARACTERISTIC_KEY = \
    "com.example.com.exampleble.CHARACTERISTIC_UUID";
public static final String SERVICE_KEY = \
    "com.example.com.exampleble.SERVICE_UUID";

/** Bluetooth Stuff */
private BleCommManager mBleCommManager;
private BlePeripheral mBlePeripheral;
private BluetoothGattCharacteristic mCharacteristic;

/** Functional stuff */
private String mPeripheralMacAddress;
private String mBlePeripheralName;
private boolean mScanningActive = false;
private UUID mCharacteristicUUID, mServiceUUID;

/** UI Stuff */
private MenuItem mProgressSpinner;
private TextView mResponseText;
private TextView mPeripheralAdvertiseNameTV;
private TextView mPeripheralAddressTV;
private TextView mServiceUUIDTV;
private Button mReadButton;

@Override
protected void onCreate(Bundle savedInstanceState) {
    // grab a Characteristic from the savedInstanceState,
    // passed when a user clicked on a Characteristic in the Connect Activity
    if (savedInstanceState == null) {
        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            mBlePeripheralName = extras.getString(PERIPHERAL_NAME);
        }
    }
}

```

```

        mPeripheralMacAddress = extras.getString(
            PERIPHERAL_MAC_ADDRESS_KEY
        );
        mCharacteristicUUID = UUID.fromString(
            extras.getString(CHARACTERISTIC_KEY)
        );
        mServiceUUID = UUID.fromString(extras.getString(SERVICE_KEY));
    }
} else {
    mBlePeripheralName = savedInstanceState.getString(PERIPHERAL_NAME);
    mPeripheralMacAddress = savedInstanceState.getString(
        PERIPHERAL_MAC_ADDRESS_KEY
    );
    mCharacteristicUUID = \
        UUID.fromString(savedInstanceState.getString(
            CHARACTERISTIC_KEY
        ));
    mServiceUUID = \
        UUID.fromString(
            savedInstanceState.getString(SERVICE_KEY)
        );
}
Log.v(TAG, "Incoming mac address: "+ mPeripheralMacAddress);
super.onCreate(savedInstanceState);
setContentview(R.layout.activity_talk);
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
mBlePeripheral = new BlePeripheral();
loadUI();
}

/**
 * Load UI components
 */
public void loadUI() {
    mResponseText = (TextView) findViewById(R.id.response_text);

```

```

mPeripheralAdvertiseNameTV = \
    (TextView) findViewById(R.id.advertise_name);
mPeripheralAddressTV = (TextView) findViewById(R.id.mac_address);
mServiceUUIDTV = (TextView) findViewById(R.id.service_uuid);
mReadButton = (Button) findViewById(R.id.read_button);
mPeripheralAdvertiseNameTV.setText(R.string.connecting);
Log.v(TAG, "Incoming Service UUID: " + mServiceUUID.toString());
Log.v(
    TAG,
    "Incoming Characteristic UUID: " + mCharacteristicUUID.toString()
);
mServiceUUIDTV.setText(mCharacteristicUUID.toString());
mReadButton.setVisibility(View.GONE);
mResponseText.setVisibility(View.GONE);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu;
    // this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_talk, menu);
    return true;
}

@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    mProgressSpinner = menu.findItem(R.id.scan_progress_item);
    initializeBluetooth();
    //connect(); // when working with non-Android peripherals
    startScan(); // when working with Android Peripherals
    return super.onPrepareOptionsMenu(menu);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {

```

```

        switch (item.getItemId()) {
            case R.id.action_disconnect:
                // User chose the "Stop" item
                disconnect();
                return true;
            default:
                // If we got here, the user's action was not recognized.
                // Invoke the superclass to handle it.
                return super.onOptionsItemSelected(item);
        }
    }

public void initializeBluetooth() {
    try {
        mBleCommManager = new BleCommManager(this);
    } catch (Exception e) {
        Toast.makeText(
            this,
            "Could not initialize bluetooth", Toast.LENGTH_SHORT
        ).show();
        Log.e(TAG, e.getMessage());
        finish();
    }
}

public void connect() {
    // grab the Peripheral Device address and attempt to connect
    BluetoothDevice bluetoothDevice = \
        mBleCommManager.getBluetoothAdapter().getRemoteDevice(
            mPeripheralMacAddress
        );
    mProgressSpinner.setVisibility(true);
    try {
        mBlePeripheral.connect(
            bluetoothDevice,

```

```

        mGattCallback,
        getApplicationContext()
    );
} catch (Exception e) {
    mProgressSpinner.setVisibility(false);
    Log.e(TAG, "Error connecting to peripheral");
}
}

/**
 * Peripheral has connected. Update UI
 */
public void onBleConnected() {
    BluetoothDevice bluetoothDevice = mBlePeripheral.getBluetoothDevice();
    mPeripheralAdvertiseNameTV.setText(bluetoothDevice.getName());
    mPeripheralAddressTV.setText(bluetoothDevice.getAddress());
    mProgressSpinner.setVisibility(false);
}

}

/**
 * Characteristic supports reads. Update UI
 */
public void onCharacteristicReadable() {
    Log.v(TAG, "Characteristic is readable");
    mReadButton.setVisibility(View.VISIBLE);
    mResponseText.setVisibility(View.VISIBLE);
    mReadButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Log.v(TAG, "Read button clicked");
            mBlePeripheral.readValueFromCharacteristic(mCharacteristic);
        }
    });
}
}

```

```

/**
 * Update TextView when a new message is read from a characteristic
 * Also scroll to the bottom so that new messages are always in view
 *
 * @param message the characteristic value to display in the UI as text
 */
public void updateResponseText(String message) {
    mResponseText.append(message + "\n");
    final int scrollAmount = \
        mResponseText.getLayout().getLineTop(
            mResponseText.getLineCount()
        ) - mResponseText.getHeight();
    // if there is no need to scroll, scrollAmount will be <=0
    if (scrollAmount < 0) {
        mResponseText.scrollTo(0, scrollAmount);
    } else {
        mResponseText.scrollTo(0, 0);
    }
}

/**
 * BluetoothGattCallback handles connections, state changes,
 * reads, writes, and GATT profile listings to a Peripheral
 *
 */
private final BluetoothGattCallback mGattCallback = \
    new BluetoothGattCallback()
{
    /**
     * Characteristic successfully read
     *
     * @param gatt connection to GATT
     * @param characteristic The characteristic that was read
     * @param status the status of the operation
     */
    @Override

```

```
public void onCharacteristicRead(
    final BluetoothGatt gatt,
    final BluetoothGattCharacteristic characteristic,
    int status)
{
    // characteristic was read. Convert the data to something usable
    // on Android and display it in the UI
    if (status == BluetoothGatt.GATT_SUCCESS) {
        final byte[] data = characteristic.getValue();
        String m = "";
        try {
            m = new String(data, CHARACTER_ENCODING);
        } catch (Exception e) {
            Log.e(
                TAG,
                "Could not convert message byte array to String"
            );
        }
        final String message = m;
        Log.v(
            TAG,
            "Characteristic read hex value: " + \
            DataConverter.bytesToHex(data)
        );
        Log.v(
            TAG,
            "Characteristic read int value: " + \
            DataConverter.bytesToInt(data)
        );
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                updateResponseText(message);
            }
        });
    }
}
```

```
}

/**
 * Characteristic was written successfully. update the UI
 *
 * @param gatt Connection to the GATT
 * @param characteristic The characteristic that was written
 * @param status write status
 */
@Override
public void onCharacteristicWrite(
    BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic,
    int status)
{
}

/**
 * Characteristic value changed. Read new value.
 * @param gatt Connection to the GATT
 * @param characteristic The characteristic
 */
@Override
public void onCharacteristicChanged(
    BluetoothGatt gatt,
    final BluetoothGattCharacteristic
    characteristic)
{
}

/**
 * Peripheral connected or disconnected. Update UI
 * @param bluetoothGatt Connection to GATT
 * @param status status of the operation
 * @param newState new connection state
 */

```

```

@Override
public void onConnectionStateChange(
    BluetoothGatt bluetoothGatt,
    int status,
    int newState)
{
    if (newState == BluetoothProfile.STATE_CONNECTED) {
        Log.v(TAG, "Connected to peripheral");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleConnected();
            }
        });
        bluetoothGatt.discoverServices();
    } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        Log.v(TAG, "Disconnected from peripheral");
        disconnect();
        mBlePeripheral.close();
    }
}

/**
 * GATT Profile discovered. Update UI
 * @param bluetoothGatt connection to GATT
 * @param status status of operation
 */
@Override
public void onServicesDiscovered(
    BluetoothGatt bluetoothGatt,
    int status)
{
    // if services were discovered,
    // then let's iterate through them and display them on screen
    if (status == BluetoothGatt.GATT_SUCCESS) {

```

```
// connect to a specific service

BluetoothGattService gattService = \
    bluetoothGatt.getService(mServiceUUID);

// while we are here, let's ask for this service's characteristics:

List<BluetoothGattCharacteristic> characteristics = \
    gattService.getCharacteristics();
for (BluetoothGattCharacteristic characteristic : \
    characteristics)

{
    if (characteristic != null) {
        Log.v(
            TAG,
            "found characteristic: " + \
            characteristic.getUuid().toString()
        );
    }
}

// determine the read/write/notify permissions
// of the Characterstic
Log.v(TAG, "desired service is: " + mServiceUUID.toString());
Log.v(
    TAG,
    "desired charactersitic is: " + \
    mCharacteristicUUID.toString()
);
Log.v(
    TAG,
    "this service: " + \
    bluetoothGatt.getService(mServiceUUID).getUuid().toString()
);
Log.v(
    TAG,
    "this characteristic: " + \
    bluetoothGatt.getService(
```

```

        mServiceUUID
    ).getCharacteristic(
        mCharacteristicUUID
    ).getUuid().toString()
);

mCharacteristic = bluetoothGatt.getService(
    mServiceUUID
).getCharacteristic(mCharacteristicUUID);
if (BlePeripheral.isCharacteristicReadable(mCharacteristic)) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onCharacteristicReadable();
        }
    });
} else {
    Log.e(
        TAG,
        "Problem occurred while discovering " +
        "GATT services from this peripheral"
    );
}
};

/***
 * Disconnect
 */
private void disconnect() {
    // close the Activity when disconnecting.
    // No actions can be done without a connection
    mBlePeripheral.disconnect();
    finish();
}

```

```

/**
 * Android-based Peripherals change MAC addresses between connections
 * As a result we can't simply connect to the last known MAC address
 * Instead, we must scan for matching advertised names again
 */

/**
 * Start scanning for Peripherals
 */
public void startScan() {
    try {
        mScanningActive = true;
        mBleCommManager.scanForPeripherals(
            mBleScanCallbackv18,
            mScanCallbackv21
        );
    } catch (Exception e) {
        Log.e(TAG, "Could not open Ble Device Scanner");
    }
}

/**
 * Stop scanning for Peripherals
 */
public void stopScan() {
    mBleCommManager.stopScanning(mBleScanCallbackv18, mScanCallbackv21);
}

/**
 * Event trigger when BLE Scanning has stopped
 */
public void onBleScanStopped() {
    // update UI components to reflect that a BLE scan has stopped
    // it's possible that this method
    // will be called before the menu has been instantiated
}

```

```

    // Check to see if menu items are initialized, or Activity will crash
    mScanningActive = false;
}

/**
 * Event trigger when new Peripheral is discovered
 */
public void onBlePeripheralDiscovered(
    BluetoothDevice bluetoothDevice,
    int rssi)
{
    Log.v(
        TAG,
        "Found "+mBlePeripheralName+": "+ \
            bluetoothDevice.getName() + ", " + \
            bluetoothDevice.getAddress()
    );
    // only add the peripheral if
    // - it has a name, or
    // - doesn't already exist in our list, or
    // - is transmitting at a higher power (is closer)
    // than an existing peripheral
    if (bluetoothDevice.getName().equals(mBlePeripheralName)) {
        Log.v(TAG, "desired device found. connecting");
        stopScan();
        mPeripheralMacAddress = bluetoothDevice.getAddress();
        connect();
    }
}

/**
 * Use this callback for Android API 21 (Lollipop) or greater
 */
private final BleScanCallbackV21 mScanCallbackV21 = \
    new BleScanCallbackV21()

```

```

{

    /**
     * New Peripheral discovered
     *
     * @param callbackType int: Determines how this callback
     * was triggered. Could be one of CALLBACK_TYPE_ALL_MATCHES,
     * CALLBACK_TYPE_FIRST_MATCH or CALLBACK_TYPE_MATCH_LOST
     * @param result a Bluetooth Low Energy Scan Result,
     * containing the Bluetooth Device, RSSI,
     * and other information
     */
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        BluetoothDevice bluetoothDevice = result.getDevice();
        int rssi = result.getRssi();
        onBlePeripheralDiscovered(bluetoothDevice, rssi);
    }

    /**
     * Several peripherals discovered when scanning in low power mode
     *
     * @param results List: List of previously scanned results.
     */
    @Override
    public void onBatchScanResults(List<ScanResult> results) {
        for (ScanResult result : results) {
            BluetoothDevice bluetoothDevice = result.getDevice();
            int rssi = result.getRssi();
            onBlePeripheralDiscovered(bluetoothDevice, rssi);
        }
    }

    /**
     * Scan failed to initialize
     *
     * @param errorCode int: Error code (one of SCAN_FAILED_*)
     */
}

ure.

```

```

*/
@Override
public void onScanFailed(int errorCode) {
    switch (errorCode) {
        case SCAN_FAILED_ALREADY_STARTED:
            Log.e(
                TAG,
                "Fails to start scan as BLE scan with the " + \
                "same settings is already started by the app. "
            );
            break;
        case SCAN_FAILED_APPLICATION_REGISTRATION_FAILED:
            Log.e(
                TAG,
                "Fails to start scan as app cannot be registered."
            );
            break;
        case SCAN_FAILED_FEATURE_UNSUPPORTED:
            Log.e(
                TAG,
                "Fails to start; this feature is not supported."
            );
            break;
        default: // SCAN_FAILED_INTERNAL_ERROR
            Log.e(TAG, "Fails to start scan due an internal error");
    }
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

/**
 * Scan completed

```

```

        */

    public void onScanComplete() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleScanStopped();
            }
        });
    }

};

/***
 * Use this callback for Android API 18, 19, and 20 (before Lollipop)
 */
public final BleScanCallbackV18 mBleScanCallbackV18 = \
    new BleScanCallbackV18()
{
    /**
     * New Peripheral discovered
     * @param bluetoothDevice The Peripheral Device
     * @param rssi The Peripheral's RSSI indicating
     *             how strong the radio signal is
     * @param scanRecord Other information about the scan result
     */
    @Override
    public void onLeScan(
        BluetoothDevice bluetoothDevice,
        int rssi,
        byte[] scanRecord)
    {
        onBlePeripheralDiscovered(bluetoothDevice, rssi);
    }

    /**
     * Scan completed
     */
}

```

```
    @Override
    public void onScanComplete() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleScanStopped();
            }
        });
    }
}
```

The Talk activity will read data from the connected Peripheral when the user clicks a Read button in the user interface.

### Example 7-8. res/layout/activity\_talk.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" android:fitsSystemWindows="true"
    tools:context=".ConnectActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:orientation="vertical"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:layout_width="match_parent"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main" tools:context=".MainActivity"
    android:weightSum="1">
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:layout_marginBottom="@dimen/activity_vertical_margin"
    android:text="@string/loading"
    android:id="@+id/advertise_name"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="@dimen/activity_vertical_margin"
    android:text="@string/loading"
    android:id="@+id/mac_address"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="@dimen/activity_vertical_margin"
    android:text="@string/loading"
    android:id="@+id/service_uuid"/>
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/read_button"
    android:id="@+id/read_button" />
<TextView
    android:layout_width="match_parent"
```

```
        android:layout_height="172dp"
        android:text=""
        android:background="#ffffffff"
        android:id="@+id/response_text"
        android:layout_weight=".5" />
    </LinearLayout>
</android.support.design.widget.CoordinatorLayout>
```

The Talk activity has a menu item that lets user to exit from the Characteristic and return to the Peripheral's GATT profile in the Connect Activity:

### Example 7-9. res/menu/menu\_talk.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".ConnectActivity">
    <item android:id="@+id/action_disconnect"
        android:title="@string/action_disconnect"
        android:orderInCategory="100" app:showAsAction="always" />
    <item
        android:id="@+id/scan_progress_item"
        android:title="@string/connecting"
        android:visible="true"
        android:orderInCategory="100"
        app:showAsAction="always"
        app:actionLayout="@layout/scanner_progress"
        android:layout_marginRight="@dimen/activity_horizontal_margin" />
</menu>
```

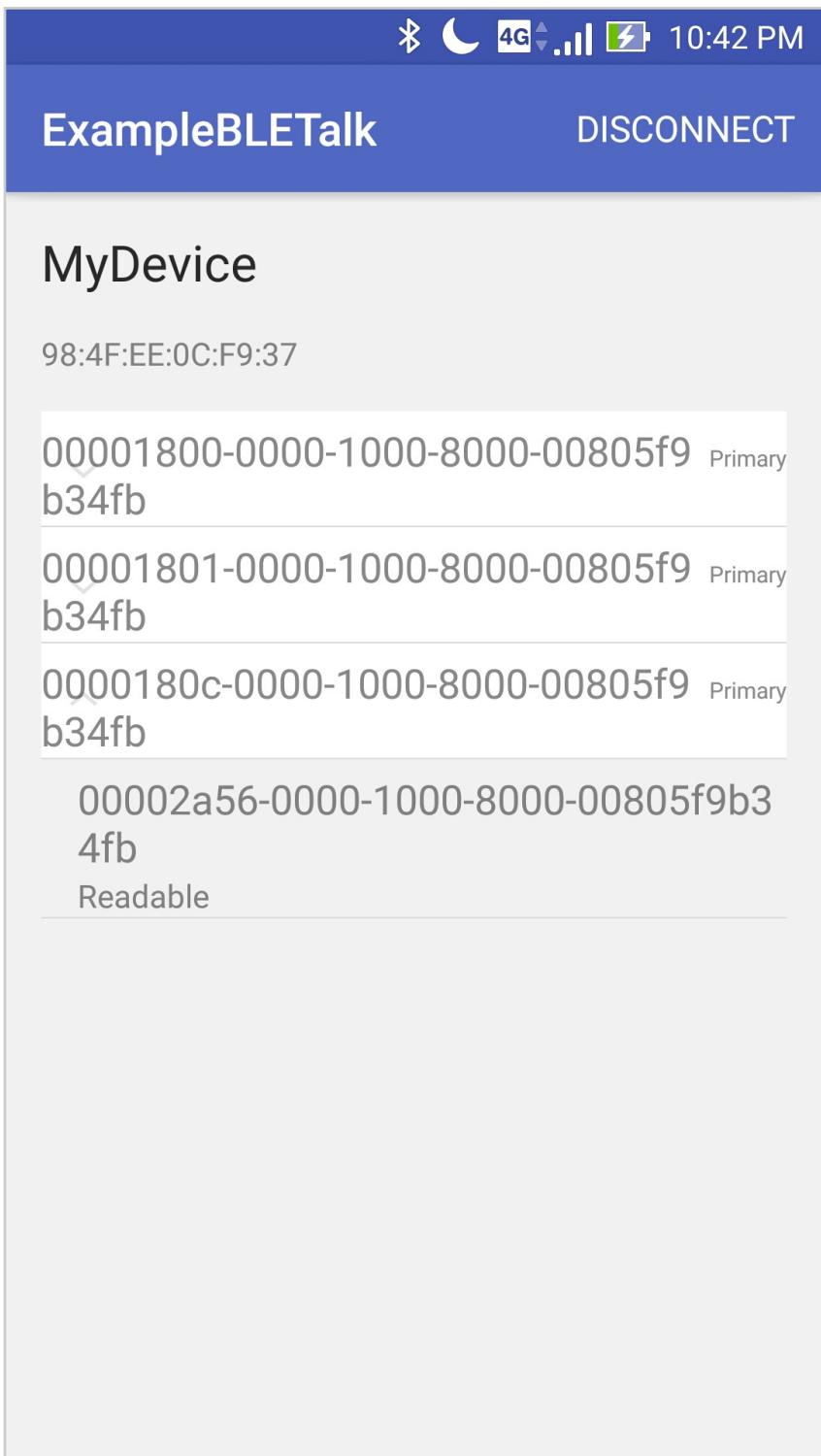
## Manifest

The Talk activity must also be described in the Android Manifest, so that it has permissions to run:

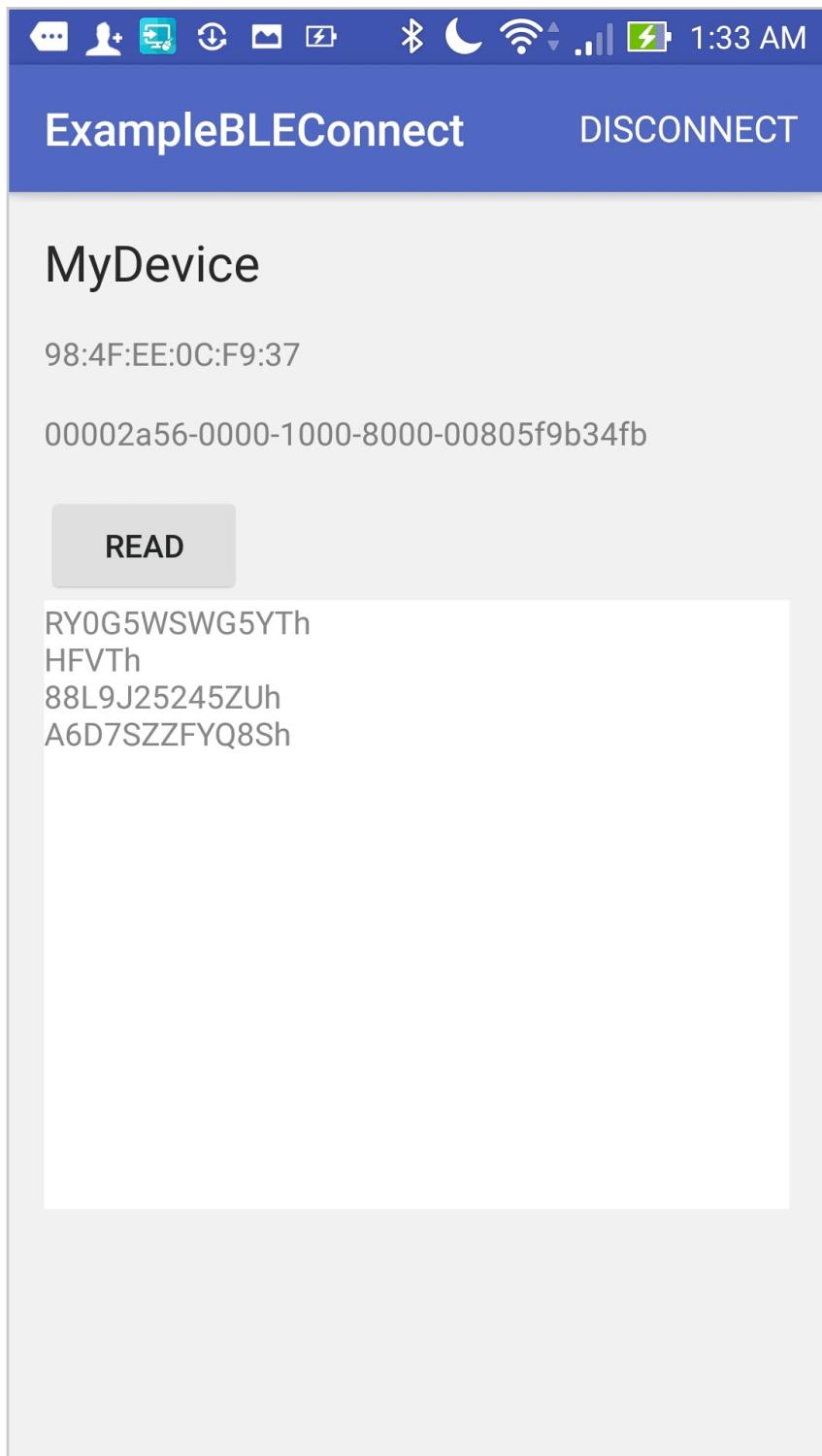
## Example 7-10. manifests/AndroidManifest.xml

```
...
<application ... >
...
<activity
    android:name=".TalkActivity"
    android:theme="@style/AppTheme.NoActionBar" />
</application>
...
```

When run, the App will be able to scan for and connect to a Peripheral. Once connected, it can list the GATT Profile - the Services and Characteristics hosted on the Peripheral ([Figure 7-3](#)). A Characteristic can be selected and values can be read from that Characteristic ([Figure 7-4](#)).



**Figure 7-3. App screen showing GATT Profile for connected Peripheral**



**Figure 7-4. App screen showing values read from Characteristic on a connected Peripheral**

## Programming the Peripheral

This chapter will show how to create a Characteristic with read access.

A read-only Characteristic is created and added to a Primary Service like this:

```

public static final UUID CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");

// create a read-only Characteristic
mCharacteristic = new BluetoothGattCharacteristic(
    CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_READ | \
        BluetoothGattCharacteristic.PROPERTY_WRITE,
    BluetoothGattCharacteristic.PERMISSION_READ
);

// add the Characteristic to the Service
mService.addCharacteristic(mCharacteristic);

```

When the Central requests to read data from the Peripheral's Characteristic, the `onCharacteristicReadRequest` method is triggered the `BluetoothGattServerCallback` object.

**Table 7-2. BluetoothGattServerCallback**

Event	Description
<code>onConnectionStateChange</code>	Central connects or disconnects
<code>onCharacteristicReadRequest</code>	Characteristic supports notifications
<code>onCharacteristicWriteRequest</code>	Central attempted to read a value from a Characteristic
<code>onDescriptorWriteRequest</code>	Central attempted to change a Descriptor in a Characteristic
<code>onNotificationSent</code>	Central was notified of a change to a Characteristic

In order for the Central to read a value, a value must be sent using the `BluetoothGattServer.sendResponse()` method. It can be implemented like this:

```

private final BluetoothGattServerCallback gattCallback = \
    new BluetoothGattServerCallback()
{

```

```
@Override
public void onCharacteristicReadRequest(
    BluetoothDevice device,
    int requestId,
    int offset,
    BluetoothGattCharacteristic characteristic)
{
    // bubble the event to the parent object
    super.onCharacteristicReadRequest(
        device,
        requestId,
        offset,
        characteristic
    );
    // send the value to the Connected Central
    mGattServer.sendResponse(
        device, requestId,
        BluetoothGatt.GATT_SUCCESS,
        offset,
        characteristic.getValue()
    );
}
};
```

It is important to note that Bluetooth Low Energy transmits data in Little-Endian format and Java uses Big-Endian to process data.

## Putting It All Together

Copy the previous chapter's project into a new project and modify the files below.

## Objects

Modify the BlePeripheral class to include a method that checks if a Characteristic is readable, and one that initiates a read request from a Characteristic.

### Example 7-11. java/example.com.exampleble/ble/BlePeripheral

```
...
/** Peripheral and GATT Profile */
public static final String ADVERTISING_NAME = "MyDevice";
public static final UUID SERVICE_UUID = \
    UUID.fromString("0000180c-0000-1000-8000-00805f9b34fb");
public static final UUID CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
private static final int CHARACTERISTIC_LENGTH = 20;
...

private BluetoothGattServer mGattServer;
private BluetoothGattService mService;
private BluetoothGattCharacteristic mCharacteristic;
...

/**
 * Set up the Advertising name and GATT profile
 */
private void setupDevice() {
    mService = new BluetoothGattService(
        SERVICE_UUID,
        BluetoothGattService.SERVICE_TYPE_PRIMARY
    );
    mCharacteristic = new BluetoothGattCharacteristic(
        CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ,
        BluetoothGattCharacteristic.PERMISSION_READ);

    mService.addCharacteristic(mCharacteristic);
    mGattServer.addService(mService);
    // write random characters to the
    // Read Characteristic every timerInterval_ms
```

```
int timerInterval_ms = 1000;
TimerTask updateReadCharacteristicTask = new TimerTask() {
    @Override
    public void run() {
        int stringLength = (int) (
            Math.random() % CHARACTERISTIC_LENGTH
        );
        String randomString = DataConverter.getRandomString(
            stringLength
        );
        try {
            mCharacteristic.setValue(randomString);
        } catch (Exception e) {
            Log.e(TAG, "Error converting String to byte array");
        }
    }
};

Timer randomStringTimer = new Timer();
randomStringTimer.schedule(
    updateReadCharacteristicTask,
    0,
    timerInterval_ms
);
}

...
private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onConnectionStateChange(
        BluetoothDevice device,
        final int status,
        int newState)
    {
        super.onConnectionStateChange(device, status, newState);
    }
}
```

```
    Log.v(TAG, "Connected");

    if (status == BluetoothGatt.GATT_SUCCESS) {
        if (newState == BluetoothGatt.STATE_CONNECTED) {
            mBlePeripheralCallback.onCentralConnected(device);
            stopAdvertising();
        } else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
            mBlePeripheralCallback.onCentralDisconnected(device);
            try {
                startAdvertising();
            } catch (Exception e) {
                Log.e(TAG, "error starting advertising");
            }
        }
    }

@Override
public void onCharacteristicReadRequest(
    BluetoothDevice device,
    int requestId,
    int offset,
    BluetoothGattCharacteristic characteristic
) {
    super.onCharacteristicReadRequest(
        device,
        requestId,
        offset,
        characteristic
    );
    Log.d(
        TAG,
        "Device tried to read characteristic: " + \
        characteristic.getUuid()
    );
    Log.d(
        TAG,
```

```
"value: " + Arrays.toString(characteristic.getValue())
);

if (characteristic.getUuid() == mCharacteristic.getUuid()) {
    if (offset < CHARACTERISTIC_LENGTH) {
        if (offset != 0) {
            mGattServer.sendResponse(
                device,
                requestId,
                BluetoothGatt.GATT_INVALID_OFFSET,
                offset,
                characteristic.getValue()
            );
        } else {
            mGattServer.sendResponse(
                device,
                requestId,
                BluetoothGatt.GATT_SUCCESS,
                offset,
                characteristic.getValue()
            );
        }
    } else {
        Log.d(
            TAG,
            "invalid offset when trying to read characteristic"
        );
    }
}

};

...

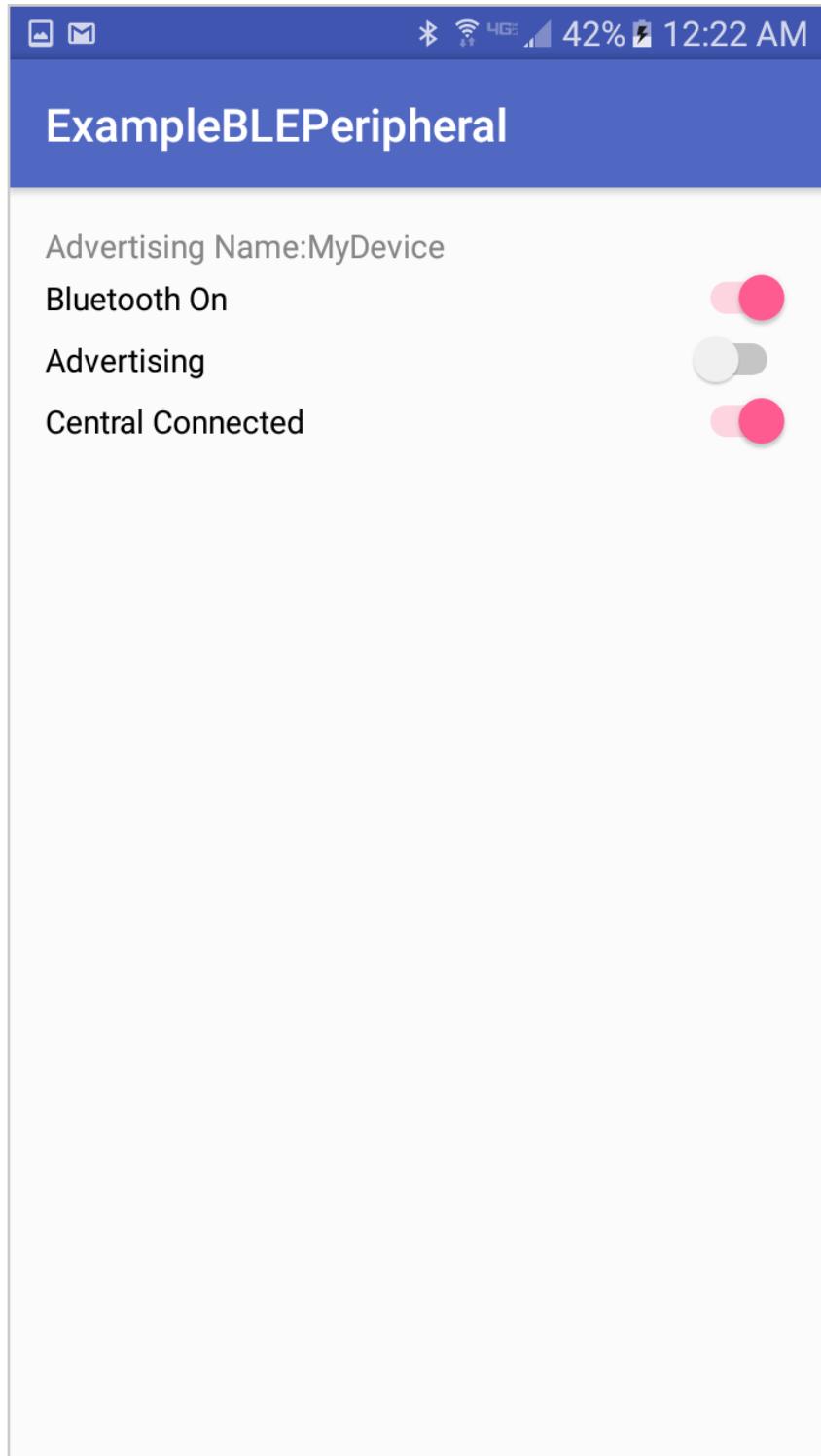
```

Create a DataConverter class that generates a random String:

## Example 7-12. java/example.com.exampleble/utilities/DataConverter

```
package example.com.exampleble.utilities;
public class DataConverter {
    /**
     * Generate a random String
     *
     * @param int length of resulting String
     * @return random string
     */
    private static final String ALLOWED_CHARACTERS = \
        "0123456789abcdefghijklmnopqrstuvwxyz";
    public static String getRandomString(final int length) {
        final Random random = new Random();
        final StringBuilder sb=new StringBuilder(length);
        for(int i=0;i<length;++i)
            sb.append(ALLOWED_CHARACTERS.charAt(
                random.nextInt(ALLOWED_CHARACTERS.length())));
        return sb.toString();
    }
}
```

When run, the App will be able to host a simple GATT profile with a single Characteristic that sets the Characteristic to a random String every 5 seconds ([Figure 7-5](#)).



**Figure 7-5. App screen showing GATT Profile for connected Peripheral**

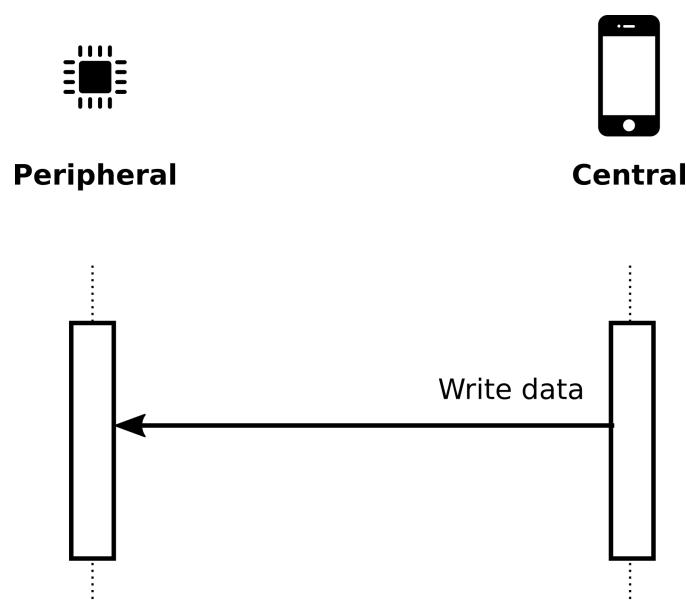
## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter07>

# Writing Data to a Peripheral

Data is sent from the Central to a Peripheral when the Central writes a value in a Characteristic hosted on the Peripheral, presuming that Characteristic has write permissions.

The process looks like this ([Figure 8-1](#)):



**Figure 8-1. The process of a Central writing data to a Peripheral**

## Programming the Central

Before attempting to write data a Characteristic, it is useful to know if the Characteristic has “write” permissions. Write permissions can be read by accessing a Characteristic’s properties and isolating the Write properties:

```
int permissions = characteristic.getProperties(); // fetch permissions
boolean iswriteable = permissions &
    (BluetoothGattCharacteristic.PROPERTY_WRITE |
    BluetoothCharacteristic.PROPERTY_WRITE_NO_RESPONSE);
```

To send data to the peripheral, the Central must set the local Characteristic cache, then push the update to the Peripheral:

```
characteristic.setValue(message); // set local cache  
bluetoothGatt.writeCharacteristic(characteristic); // update remote Peripheral
```

The message can be an int, String, or byte array. When working with complex or long data, it may be easier to convert the data to a byte array. As a trivial example, a String can be converted to a byte array like this:

```
byte[] temp = message.getBytes();
```

Just as when reading numbers from a Characteristic, the endianness of the number must be changed to Little-Endian before transmitting over Bluetooth. For example, this is how to write an Integer to a Characteristic:

```
ByteBuffer byteBuffer = ByteBuffer.wrap(bytes, ByteOrder.BIG_ENDIAN);  
byteBuffer.order(ByteOrder.LITTLE_ENDIAN); // switch to little-endian  
int message = byteBuffer.getInt();  
characteristic.setValue(message); // set local cache  
bluetoothGatt.writeCharacteristic(characteristic); // update remote Peripheral
```

When the Central has been successfully written to, the onCharacteristicWrite event gets triggered in the BluetoothGattCallback.

**Table 8-1. BluetoothGattCallback**

Event	Description
<b>onConnectionStateChange</b>	Triggered when a BLE Peripheral connects or disconnects
<b>onServicesDiscovered</b>	Triggered when GATT Services are discovered

The implementation looks like this:

```
private final BluetoothGattCallback gattCallback = new BluetoothGattCallback()
{
    @Override
    public void onCharacteristicWrite(BluetoothGatt gatt,
                                      BluetoothGattCharacteristic characteristic,
                                      int status) {
        // make note of the transmission completion
    }
};
```

## Putting It All Together

The BlePeripheral will be modified to write data to a Characteristic on a connected Peripheral, and TalkActivity will be modified to display a send button.

## Resources

The Talk Activity will need a send button to enable users to trigger a Characteristic write event. The send button text will be defined in res/values/strings.xml.

### Example 8-1. res/values/strings.xml

```
...
<string name="write_button">Write</string>
<string name="property_write">Writable</string>
...
```

## Objects

Modify the BlePeripheral class to include two new methods:

- isCharacteristicWritable is true when a Characteristic is writeable
- writeValueToCharacteristic initializes a Characteristic write event.

### Example 8-2. java/example.com.exampleble/ble/BleDevice

```
...
public static boolean isCharacteristicWritable(
    BluetoothGattCharacteristic characteristic) {
    return (characteristic.getProperties() &
        (BluetoothGattCharacteristic.PROPERTY_WRITE |  

         BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE)) != 0;
}  

/**  

 * Write a value to a characteristic  

 *  

 * New in this chapter  

 *  

 * @param message The message being written  

 * @param characteristic The characteristic being written to  

 * @throws Exception  

 */
public void writeValueToCharacteristic(String message,
    BluetoothGattCharacteristic characteristic) throws Exception {
    byte[] messageBytes = message.getBytes();
    Log.v(TAG, "Writing message: '" +
        new String(messageBytes, "ASCII") + "' to " +
        characteristic.getUuid().toString());
    characteristic.setValue(messageBytes);
    mBluetoothGatt.writeCharacteristic(characteristic);
}
...
```

Modify the BleGattProfileListAdapter to display a TextView if a Characteristic has write permissions:

### Example 8-3. java/example.com.exampleble/ble/adapters/BleGattProfileListAdapter

```
...
/***
 * These UI components are in each characteristic List Item in the ListView
 */
public static class ChildviewHolder{
    public TextView mUuidTV; // displays UUID
    public TextView mPropertyReadableTV; // characteristic is readable
    public TextView mPropertyWritableTV; // characteristic is writeable
    public TextView mPropertyNoneTV; // displays when no access is given
}

@Override
public View getChildView(final int groupPosition, final int childPosition,
    boolean isLastChild, View convertView, ViewGroup parent) {
    View v = convertView;
    ChildviewHolder characteristicListView;
    BleGattCharacteristicListItem item =
        getChild(groupPosition, childPosition);
    BluetoothGattCharacteristic characteristic = item.getCharacteristic();
    // if this ListItem does not exist yet, generate it
    // otherwise, use it
    if (convertView == null) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        v = inflater.inflate(R.layout.list_item_ble_characteristic, null);
        // match the UI stuff in the list Item to what's in the xml file
        characteristicListView = new ChildviewHolder();
        characteristicListView.mUuidTV =
            (TextView) v.findViewById(R.id.uuid);
        characteristicListView.mPropertyReadableTV =
            (TextView)v.findViewById(R.id.property_read);
```

```

characteristicListView.mPropertyWritableTV =
        (TextView)v.findViewById(R.id.property_write);
characteristicListView.mPropertyNoneTV =
        (TextView)v.findViewById(R.id.property_none);
} else {
    characteristicListView = (ChildViewHolder) v.getTag();
}
if (characteristicListView != null) {
    // display the UUID of the characteristic
    characteristicListView.mUuidTV.setText(
            characteristic.getUuid().toString());
    // Display the read/write/notify attributes of the characteristic
    if (BlePeripheral.isCharacteristicReadable(characteristic)) {
        characteristicListView.mPropertyReadableTV.
                setVisibility(View.VISIBLE);
    } else {
        characteristicListView.mPropertyReadableTV.
                setVisibility(View.GONE);
    }
    if (BlePeripheral.isCharacteristicwritable(characteristic)) {
        characteristicListView.mPropertywritableTV.
                setVisibility(View.VISIBLE);
    } else {
        characteristicListView.mPropertywritableTV.
                setVisibility(View.GONE);
    }
    if (!BlePeripheral.isCharacteristicwritable(characteristic) && \\
        !BlePeripheral.isCharacteristicReadable(characteristic))
    {
        characteristicListView.mPropertyNoneTV.
                setVisibility(View.VISIBLE);
    } else {
        characteristicListView.mPropertyNoneTV.
                setVisibility(View.GONE);
    }
}

```

```
    return v;  
}  
  
...
```

## Views

Modify the list\_item\_ble\_characteristic.xml file to hold a TextView for when a Characteristic is writable:

### Example 8-4. res/layout/list\_item\_ble\_characteristic.xml

```
...  
  
<LinearLayout  
    android:orientation="horizontal"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_vertical"  
    android:layout_weight="2">  
    <TextView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:textAppearance="?android:attr/textAppearanceSmall"  
        android:text="@string/property_read"  
        android:visibility="gone"  
        android:id="@+id/property_read"  
        android:paddingLeft="@dimen/activity_horizontal_margin" />  
    <TextView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:textAppearance="?android:attr/textAppearanceSmall"  
        android:text="@string/property_write"  
        android:visibility="gone"  
        android:id="@+id/property_write"  
        android:paddingLeft="@dimen/activity_horizontal_margin" />  
    <TextView  
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_none"
        android:visibility="gone"
        android:id="@+id/property_none"
        android:paddingLeft="@dimen/activity_horizontal_margin" />
    </LinearLayout>
...

```

## Activities

Add two new methods to the Talk activity:

- `onCharacteristicWritable` shows user interface elements for sending messages,
- `onMessageSent` is triggered when a message has been sent out over Bluetooth

### Example 8-5. `java/example.com.exampleble/TalkActivity.java`

```
public class TalkActivity extends AppCompatActivity {
    /** Constants */
    private static final String CHARACTER_ENCODING = "ASCII";
    ...
    private TextView mSendText;
    private Button mSendButton;
    // onCreate, onResume, onPause, disconnect
    public void loadUI() {
        mResponseText = (TextView) findViewById(R.id.response_text);
        mSendText = (TextView) findViewById(R.id.write_text);
        mPeripheralAdvertiseNameTV =
            (TextView) findViewById(R.id.advertise_name);
        mPeripheralAddressTV = (TextView) findViewById(R.id.mac_address);
        mServiceUUIDTV = (TextView) findViewById(R.id.service_uuid);
        mSendButton = (Button) findViewById(R.id.write_button);
        mReadButton = (Button) findViewById(R.id.read_button);
        mPeripheralAdvertiseNameTV.setText(R.string.connecting);
        mServiceUUIDTV.setText(mCharacteristicUUID.toString());
    }
}
```

```

        mSendButton.setVisibility(View.GONE);
        mSendText.setVisibility(View.GONE);
        mReadButton.setVisibility(View.GONE);
        mResponseText.setVisibility(View.GONE);
    }

/***
 * characteristic supports writes.  Update UI
 */
public void onCharacteristicWritable() {
    Log.v(TAG, "Characteristic is writable");
    // send features
    // attach callbacks to the button and other stuff
    mSendButton.setVisibility(View.VISIBLE);
    mSendText.setVisibility(View.VISIBLE);
    mSendButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // write to the characteristic when the send button is clicked
            Log.v(TAG, "Send button clicked");
            String message = mSendText.getText().toString();
            Log.v(TAG, "Attempting to send message: "+message);
            try {
                mBlePeripheral.writeValueToCharacteristic(message,
                    mCharacteristic);
            } catch (Exception e) {
                Log.e(TAG, "problem sending message through bluetooth");
            }
        }
    });
}

/***
 * Clear the input TextView
 * when a characteristic is successfully written to.
*/

```

```
public void onBleCharacteristicValueWritten() {
    mSendText.setText("");
}

/**
 * BluetoothGattCallback handles connections, state changes, reads,
 * writes, and GATT profile listings to a Peripheral
 *
 */
private final BluetoothGattCallback mGattCallback = \
    new BluetoothGattCallback()
{
    ...
    /**
     * Characteristic was written successfully. update the UI
     *
     * New in this chapter
     *
     * @param gatt Connection to the GATT
     * @param characteristic The characteristic that was written
     * @param status write status
     */
    @Override
    public void onCharacteristicWrite(
        BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic,
        int status)
    {
        Log.v(TAG, "characteristic written");
        if (status == BluetoothGatt.GATT_SUCCESS) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    onBleCharacteristicValueWritten();
                }
            });
        }
    }
}
```

```

        }

    }

    ...

    /**
     * GATT Profile discovered. Update UI
     * @param bluetoothGatt connection to GATT
     * @param status status of operation
     */
    @Override
    public void onServicesDiscovered(
            BluetoothGatt bluetoothGatt,
            int status)
    {
        // if services were discovered, iterate through them and display
        if (status == BluetoothGatt.GATT_SUCCESS) {
            // connect to a specific service
            BluetoothGattService gattService =
                    bluetoothGatt.getService(mServiceUUID);
            // ask for this service's characteristics:
            List<BluetoothGattCharacteristic> characteristics =
                    gattService.getCharacteristics();
            for (BluetoothGattCharacteristic characteristic :
                    characteristics) {
                if (characteristic != null) {
                    Log.v(TAG, "found characteristic: " +
                            characteristic.getUuid().toString());
                }
            }
            // determine the permissions of the characteristic
            Log.v(
                    TAG,
                    "desired service is: " + mServiceUUID.toString()
            );
            Log.v(
                    TAG,
                    "desired characteristic is: " + \

```

```

        mCharacteristicUUID.toString()
    );
    Log.v(
        TAG,
        "this service: " + \
        bluetoothGatt.getService(
            mServiceUUID
        ).getUuid().toString());
    Log.v(
        TAG,
        "this characteristic: " + \
        bluetoothGatt.getService(
            mServiceUUID
        ).getCharacteristic(
            mCharacteristicUUID
        ).getUuid().toString()
    );
    mCharacteristic = bluetoothGatt.getService(mServiceUUID).
        getCharacteristic(mCharacteristicUUID);
    if (BlePeripheral.isCharacteristicReadable(mCharacteristic)) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onCharacteristicReadable();
            }
        });
    }
    if (BlePeripheral.isCharacteristicwritable(mCharacteristic)) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onCharacteristicwritable();
            }
        });
    }
} else {

```

```
        Log.e(TAG, "Something went wrong discovering GATT profile");
    }
}
};

...
}
```

Modify the Talk activity's layout file to include a Send button and an EditText to input a message.

#### Example 8-6. res/layout/activity\_talk.xml

```
...
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/read_button"
        android:id="@+id/read_button" />
</LinearLayout>
<TextView
    android:layout_width="match_parent"
    android:layout_height="172dp"
    android:text=""
    android:background="#ffffffff"
    android:id="@+id/response_text"
    android:layout_weight=".5" />
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <EditText
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:id="@+id/write_text"
        android:inputType="textShortMessage"
        android:layout_weight="2" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/write_button"
        android:id="@+id/write_button" />
</LinearLayout>
</LinearLayout>
</android.support.design.widget.CoordinatorLayout>
```

## Views

Modify item\_list\_ble\_characteristic.xml to include a TextView that displays when a Characteristic is writable.

### Example 8-7. res/layout/list\_item\_characteristic.xml

```
...
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_weight="2">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_read"
        android:visibility="gone"
        android:id="@+id/property_read"
        android:paddingLeft="@dimen/activity_horizontal_margin" />
    <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_write"
        android:visibility="gone"
        android:id="@+id/property_write"
        android:paddingLeft="@dimen/activity_horizontal_margin" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_none"
        android:visibility="gone"
        android:id="@+id/property_none"
        android:paddingLeft="@dimen/activity_horizontal_margin" />

```

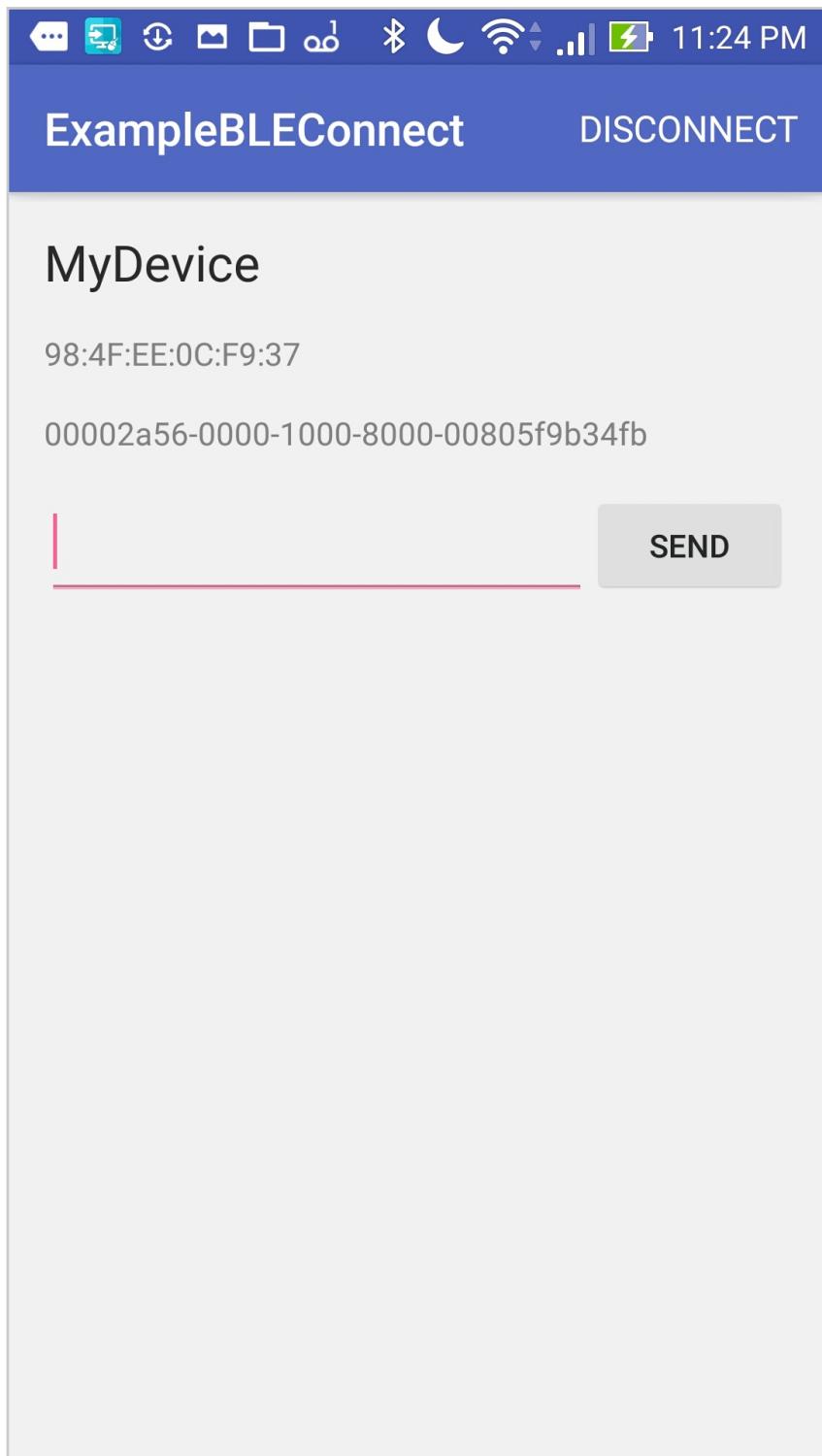
</LinearLayout>

...

Compile and run. The updated app will be able to scan for and connect to a Peripheral ([Figure 8-2](#)). Once connected, it lists services and characteristics, connect to Characteristics and send and data ([Figure 8-3](#)).



**Figure 8-2.** App screen showing GATT Profile of connected Peripheral



**Figure 8-3.** App screen allowing user to write text to a connected Peripheral's Characteristic

## Programming the Peripheral

This chapter will show how to create a Characteristic with read access.

A writeable Characteristic is created and added to a Primary Service like this:

```

public static final UUID CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");

// create a read-only Characteristic
mCharacteristic = new BluetoothGattCharacteristic(
    CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_WRITE,
    BluetoothGattCharacteristic.PERMISSION_READ);

// add the characteristic to the Service
mService.addCharacteristic(mCharacteristic);

```

When the Central requests to write data to the Peripheral's Characteristic, the `onCharacteristicWriteRequest` method is triggered the `BluetoothGattServerCallback` object.

**Table 8-2. BluetoothGattServerCallback**

Event	Description
<code>onConnectionStateChange</code>	Central connects or disconnects
<code>onCharacteristicReadRequest</code>	Characteristic supports notifications
<code>onCharacteristicWriteRequest</code>	Central attempted to read a value from a Characteristic
<code>onDescriptorWriteRequest</code>	Central attempted to change a Descriptor in a Characteristic
<code>onNotificationSent</code>	Central was notified of a change to a Characteristic

The implementation looks like this:

```

private final BluetoothGattServerCallback gattCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onCharacteristicWriteRequest(
        BluetoothDevice device,
        int requestId,

```

```

        BluetoothGattCharacteristic characteristic,
        boolean preparedwrite,
        boolean responseNeeded,
        int offset, byte[] value
    ) {
        // bubble the event up to the parent object
        super.onCharacteristicWriteRequest(
            device,
            requestId,
            characteristic,
            preparedwrite,
            responseNeeded,
            offset,
            value
        );
        // do something with the data
        mValue = value;
        // if the characteristic is BluetoothGattCharacteristic.PROPERTY_WRITE
        // send the confirmation response
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            0,
            null
        );
    }
}

```

## Putting It All Together

The BlePeripheral will be modified to accept incoming write operations and to log the written Characteristic values in a TextView.

## Resources

The Talk Activity will need a send button to enable users to trigger a Characteristic write event. The send button text will be defined in res/values/strings.xml.

### Example 8-8. res/values/strings.xml

```
...  
    <string name="characteristic_subscribed">Characteritic Subscribed</string>  
    <string name="characteristic_log">Characteritic Log:</string>  
...
```

## Objects

Modify the BlePeripheral class to create a writeable Characteristic and to handle incoming write requests in the BluetoothGattServerCallback:

### Example 8-9. java/example.com.exampleble/ble/BlePeripheral

```
...  
    /** Constants **/  
    private static final String TAG = BlePeripheral.class.getSimpleName();  
    public static final String CHARSET = "ASCII";  
  
    /** Peripheral and GATT Profile **/  
    public static final String ADVERTISING_NAME = "MyDevice";  
    public static final UUID SERVICE_UUID = \  
        UUID.fromString("0000180c-0000-1000-8000-00805f9b34fb");  
    public static final UUID CHARACTERISTIC_UUID = \  
        UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");  
    private static final int CHARACTERISTIC_LENGTH = 20;  
...  
    /**  
     * Set up the Advertising name and GATT profile  
     */  
    private void setupDevice() {  
        // create a Service
```

```

mService = new BluetoothGattService(
    SERVICE_UUID,
    BluetoothGattService.SERVICE_TYPE_PRIMARY
);
// create a write-only characteristic
mCharacteristic = new BluetoothGattCharacteristic(
    CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_WRITE,
    BluetoothGattCharacteristic.PERMISSION_READ);
// add the characteristic to the service
mService.addCharacteristic(mCharacteristic);
// add the service to the GATT Profile
mGattServer.addService(mService);
}

...
/***
 * Check if a characteristic supports write with permissions
 * @return Returns <b>true</b> if property is writable
 */
public static boolean isCharacteristicWritableWithResponse(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        BluetoothGattCharacteristic.PROPERTY_WRITE) != 0;
}

/**
 * Respond to GATT events
 */
private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onConnectionStateChange(

```

```

        BluetoothDevice device,
        final int status,
        int newState)

    {
        super.onConnectionStateChange(device, status, newState);
        Log.v(TAG, "Connected");
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothGatt.STATE_CONNECTED) {
                mBlePeripheralCallback.onCentralConnected(device);
                stopAdvertising();
            } else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
                mBlePeripheralCallback.onCentralDisconnected(device);
                try {
                    startAdvertising();
                } catch (Exception e) {
                    Log.e(TAG, "error starting advertising");
                }
            }
        }
    }

    @Override
    public void onCharacteristicWriteRequest(
        BluetoothDevice device, int requestId,
        BluetoothGattCharacteristic characteristic,
        boolean preparedwrite,
        boolean responseNeeded,
        int offset,
        byte[] value
    ) {
        super.onCharacteristicWriteRequest(
            device,
            requestId,
            characteristic,
            preparedwrite,
            responseNeeded,

```

```

        offset,
        value
    );
    Log.v(
        TAG,
        "Characteristic write request: " + \
        Arrays.toString(value)
    );
    // notify the BlePeripheralCallback of a write
    mBlePeripheralCallback.onCharacteristicWritten(
        characteristic,
        value
    );
    // set the characteristic value and respond if necessary
    if (isCharacteristicWritableWithResponse(characteristic)) {
        characteristic.setValue(value);
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            0,
            null
        );
    }
}
};

...

```

Add an onCharacteristicWritten method to the BlePeripheralCallback, notifying subscribers to Characteristic write events:

#### **Example 8-10. java/example.com.exampleble/ble/callbacks/BlePeripheralCallback**

```

...
/**
 * Characteristic written to

```

```

*
 * @param characteristic The characteristic that was written to
 * @param value the byte value that was written
 */
public abstract void onCharacteristicWritten(
    final BluetoothGattCharacteristic characteristic,
    final byte[] value
);
...

```

Add a function to the DataConverter class to convert a byte array to a hexadecimal string, for debugging purposes:

### **Example 8-11. java/example.com.exampleble/utilities/DataConverter**

```

...
/** 
 * convert bytes to hexadecimal for debugging purposes
 *
 * @param bytes
 * @return Hexadecimal string representation of the byte array
 */
public static String bytesToHex(byte[] bytes) {
    if (bytes.length <=0) return "";
    char[] hexArray = "0123456789ABCDEF".toCharArray();
    char[] hexChars = new char[bytes.length * 3];
    for ( int j = 0; j < bytes.length; j++ ) {
        int v = bytes[j] & 0xFF;
        hexChars[j * 3] = hexArray[v >>> 4];
        hexChars[j * 3 + 1] = hexArray[v & 0x0F];
        hexChars[j * 3 + 2] = 0x20; // space
    }
    return new String(hexChars);
}
...

```

## Activities

Add new functionality to the Main Activity to handle Characteristic write events coming from BlePeripheralCallback:

### Example 8-12. java/example.com.exampleble/MainActivity.java

```
public class MainActivity extends AppCompatActivity {  
    ...  
    /** UI Stuff */  
    private TextView mAdvertisingNameTV, mCharacteristicLogTV;  
    private Switch mBluetoothOnSwitch,  
        mAdvertisingSwitch,  
        mCentralConnectedSwitch;  
    ...  
    /**  
     * Load UI components  
     */  
    public void loadUI() {  
        mAdvertisingNameTV = (TextView) findViewById(R.id.advertising_name);  
        mBluetoothOnSwitch = (Switch) findViewById(R.id.bluetooth_on);  
        mAdvertisingSwitch = (Switch) findViewById(R.id.advertising);  
        mCentralConnectedSwitch = (Switch) findViewById(R.id.central_connected);  
        mAdvertisingNameTV.setText(BlePeripheral.ADVERTISING_NAME);  
        mCharacteristicLogTV = (TextView) findViewById(R.id.characteristic_log);  
    }  
    ...  
    /**  
     * Event trigger when characteristic has been written to  
     *  
     * @param characteristic the Characteristic being written to  
     * @param value the byte value being written  
     */  
    public void onBleCharacteristicWritten(  
        final BluetoothGattCharacteristic characteristic,  
        final byte[] value  
    ) {
```

```

mCharacteristicLogTV.append("\n");
try {
    mCharacteristicLogTV.append(
        new String(value, BlePeripheral.CHARSET)
    );
} catch (Exception e) {
    Log.d(TAG, "could not convert value to String");
}

// scroll to bottom of TextView
final int scrollAmount = mCharacteristicLogTV.getLayout().getLineTop(
    mCharacteristicLogTV.getLineCount()) - \
    mCharacteristicLogTV.getHeight();
if (scrollAmount > 0) {
    mCharacteristicLogTV.scrollTo(0, scrollAmount);
} else {
    mCharacteristicLogTV.scrollTo(0, 0);
}
}

...
/***
 * Respond to changes to the Bluetooth Peripheral state
 */
private final BlePeripheralCallback mBlePeripheralCallback = \
    new BlePeripheralCallback()
{
    public void onAdvertisingStarted() {
        Log.v(TAG, "Advertising started");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleAdvertisingStarted();
            }
        });
    }

    public void onAdvertisingFailed(int errorCode) {

```

```
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        onBleAdvertisingFailed();
    }
});

switch (errorCode) {
    case AdvertiseCallback.ADVERTISE_FAILED_ALREADY_STARTED:
        Log.e(
            TAG,
            "Failed to start; advertising is already started."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_DATA_TOO_LARGE:
        Log.e(
            TAG,
            "Failed to; advertised data is larger than 31 bytes."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_FEATURE_UNSUPPORTED:
        Log.e(
            TAG,
            "This feature is not supported on this platform."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_INTERNAL_ERROR:
        Log.e(
            TAG,
            "Operation failed due to an internal error."
        );
        break;
    case AdvertiseCallback.ADVERTISE_FAILED_TOO_MANY_ADVERTISERS:
        Log.e(
            TAG,
            "Failed to start; advertising instance is unavailable."
        );
}
```

```
        break;
    default:
        Log.e(TAG, "unknown problem");
    }
}

public void onAdvertisingStopped() {
    Log.v(TAG, "Advertising stopped");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleAdvertisingStarted();
        }
    });
}

public void onCentralConnected(final BluetoothDevice bluetoothDevice) {
    Log.v(TAG, "Central connected");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCentralConnected(bluetoothDevice);
        }
    });
}

public void onCentralDisconnected(
    final BluetoothDevice bluetoothDevice
) {
    Log.v(TAG, "Central disconnected");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCentralDisconnected(bluetoothDevice);
        }
    });
}
```

```

    }

    public void onCharacteristicWritten(
        final BluetoothGattCharacteristic characteristic,
        final byte[] value
    ) {
        Log.v(
            TAG,
            "Characteristic written: "+ DataConverter.bytesToHex(value)
        );
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleCharacteristicWritten(characteristic, value);
            }
        });
    }
}

```

Modify the Main Activity's layout file to include a TextView Characteristic log and a TextView label:

#### **Example 8-13. res/layout/activity\_main.xml**

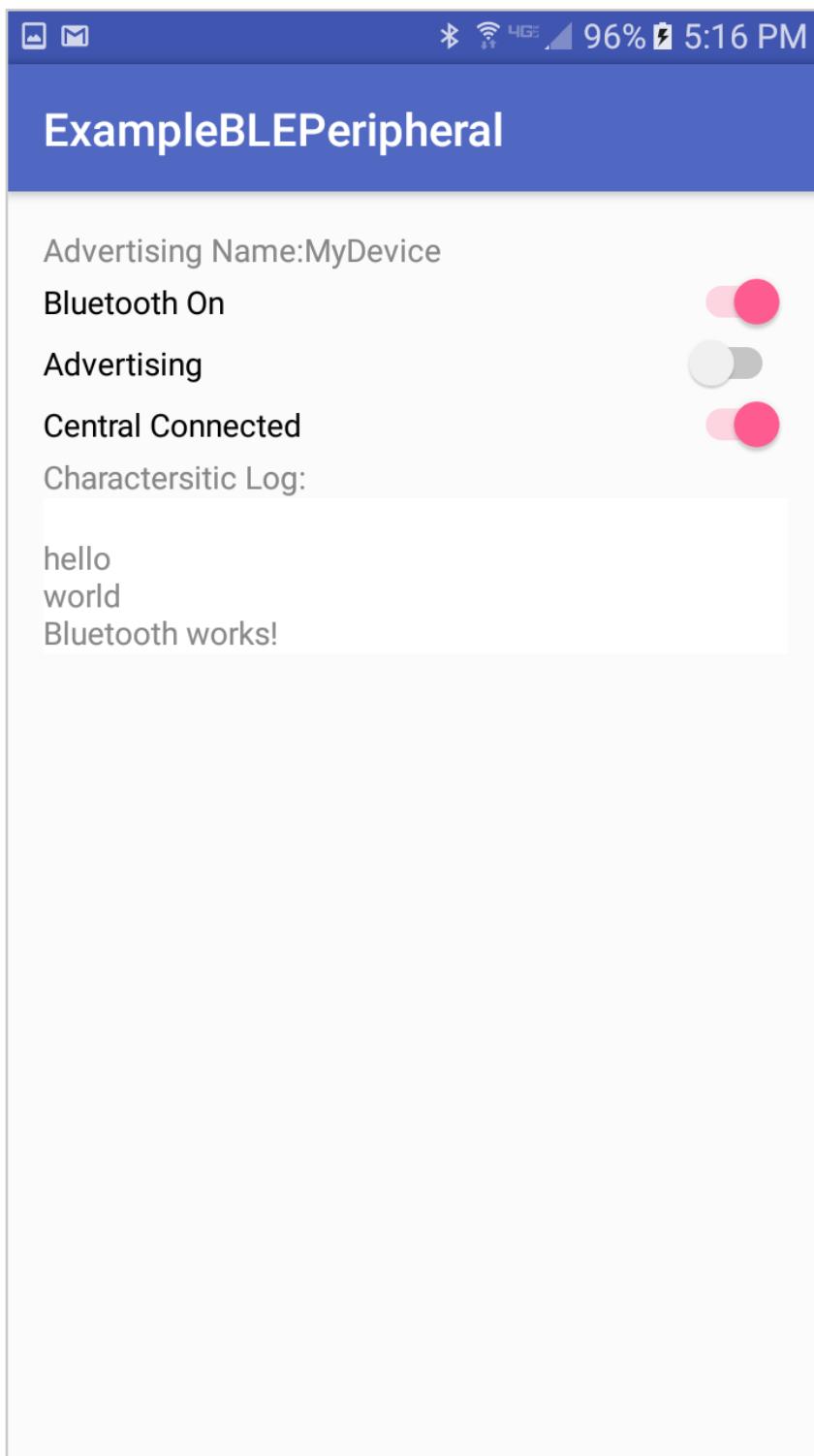
```

...
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/central_connected"
    android:id="@+id/central_connected" />
<TextView
    android:text="@string/characteristic_log"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
<TextView

```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#ffffffff"
        android:id="@+id/characteristic_log" />
    </LinearLayout>
</RelativeLayout>
</android.support.design.widget.CoordinatorLayout>
```

Compile and run. The updated app will host a simple GATT Profile featuring a write-only Characteristic ([Figure 8-4](#)).



**Figure 8-4.** App screen showing GATT Profile of connected Peripheral

## Example code

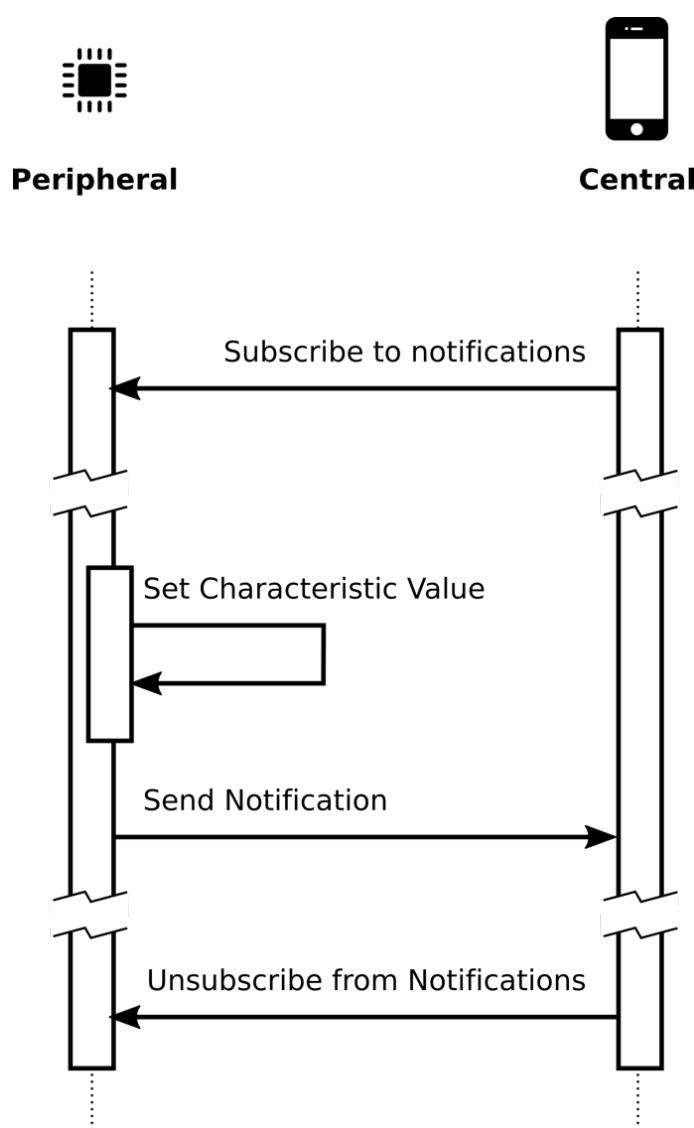
The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter08>

# Using Notifications

Being able to read from the Central has limited value if the Central does not know when new data is available.

Notifications solve this problem. A Characteristic can issue a notification when its value has changed. A Central that subscribes to these notifications will know when the Characteristic's value has changed, but not what that new value is. The Central can then read the latest data from the Characteristic.

The whole process looks something like this ([Figure 9-1](#)).



**Figure 9-1. The process of a Peripheral notifying a connected Central of changes to a Characteristic**

In order to support notifications, a Characteristics must have the Client Characteristic Configuration (0x2902) Descriptor, which must be writeable. Centrals can subscribe to notifications by setting the Descriptor value:

**Table 9-1. Client Characteristic Configuration Descriptor values**

Value	Description
0x0100	Enable notifications
0x0000	Disable notifications

## Programming the Central

Before the Central can subscribe to the Characteristic's notifications, it is useful to know if the Characteristic supports notifications. To determine if notifications are enabled, get the properties of the Characteristic and isolate the `notify` property.

```
int permissions = characteristic.getProperties(); // fetch permissions  
boolean isReadable = permissions & BluetoothGattCharacteristic.PROPERTY_NOTIFY;
```

The Central can subscribe to a Central's notifications in two steps:

First, enable the Characteristic notification flag in the local Characteristic cache:

```
mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);
```

Second, manually set certain bits in the Characteristic descriptor. Because this is a remote operation, it takes a small amount of time between steps to execute. An easy way to accommodate the time delay is to use a delay function:

```

UUID CLIENT_CHARACTERISTIC_CONFIGURATION_UUID = \
    UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");
final Handler handler = new Handler(Looper.getMainLooper());
handler.postDelayed(new Runnable() {
    @Override
    public void run() {
        BluetoothGattDescriptor descriptor =
characteristic.getDescriptor(CLIENT_CHARACTERISTIC_CONFIGURATION_UUID);
        if (descriptor != null) {
            descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
            mBluetoothGatt.writeDescriptor(descriptor);
        }
    }
}, 10);

```

When a Characteristic's value has been changed on the Peripheral, The Peripheral sends a notification. If the Central has subscribed to notifications, the `onCharacteristicChanged` method in `BluetoothGattCallback` is called.

**Table 9-2. BluetoothGattCallback**

Event	Description
<code>onConnectionStateChange</code>	Triggered when a BLE Peripheral connects or disconnects
<code>onServicesDiscovered</code>	Triggered when GATT Services are discovered
<code>onCharacteristicRead</code>	Triggered when data has been downloaded from a GATT Characteristic
<code>onCharacteristicWrite</code>	Triggered when data has been uploaded to a GATT Characteristic
<code>onCharacteristicChanged</code>	Triggered when a GATT Characteristic's data has changed

Triggered when a GATT Characteristic's data has changed

This method is implemented like this:

```
private final BluetoothGattCallback gattCallback = new BluetoothGattCallback()
{
    @Override
    public void onCharacteristicWrite(BluetoothGatt gatt,
                                      BluetoothGattCharacteristic characteristic,
                                      int status) {
        // respond to the change, probably ask to read the data
    }
};
```

## Putting It All Together

These features will be added to BlePeripheral to subscribe to notifications, and we will alter TalkActivity to display a checkbox to subscribe and unsubscribe from the notifications and respond to notification alerts.

## Resources

The Talk activity will me modified so that the user can subscribe to and unsubscribe from the Characteristic's notifications by clicking a check box. Set the checkbox and Notifiable permissions text:

### Example 9-1. res/values/strings.xml

```
...
<string name="notify_checkbox">Subscribe to this characteristic</string>
<string name="property_notify">Notifiable</string>
...
```

## Objects

Add two methods to BlePeripheral:

- isCharacteristicNotifiable is true when a Characteristic supports notifications
- setCharacteristicNotification subscribes or unsubscribes from a Characteristic's notifications

### Example 9-2. java/example.com.exampleble/ble/BlePeripheral

```
...
// UUID of the descriptor used to enable and disable notifications
// Client characteristic Configuration Descriptor
public static final UUID NOTIFY_DESCRIPTOR_UUID = \
    UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");
...

public static boolean isCharacteristicNotifiable(
    BluetoothGattCharacteristic pChar) {
    return (pChar.getProperties() &
        BluetoothGattCharacteristic.PROPERTY_NOTIFY) != 0;
}

/**
 * Subscribe or unsubscribe from Characteristic Notifications
 *
 * New in this chapter
 *
 * @param characteristic
 * @param enabled <b>true</b> for "subscribe" <b>false</b>
 *               for "unsubscribe"
 */
public void setCharacteristicNotification(
    final BluetoothGattCharacteristic characteristic,
    final boolean enabled) {
    // This is a 2-step process
    // Step 1: set the Characteristic Notification parameter locally
    mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);
}
```

```

// Step 2: Write a descriptor to the Bluetooth GATT
// A delay is needed between setCharacteristicNotification
// and setValue.
final Handler handler = new Handler(Looper.getMainLooper());
handler.postDelayed(new Runnable() {
    @Override
    public void run() {
        BluetoothGattDescriptor descriptor =
            characteristic.getDescriptor(characteristic.getUuid());
        if (descriptor != null) {
            if (enabled) {
                descriptor.setValue(
                    BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
            } else {
                descriptor.setValue(
                    BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE
                );
            }
        }
        mBluetoothGatt.writeDescriptor(descriptor);
    }
}, 10);
}
...

```

Modify the BleGattProfileListAdapter class to activity to handle subscription and unsubscription requests. Also add functionality to handle incoming notifications by requesting a to read the Characteristic.

### **Example 9-3. java/example.com.exampleble/adapters/BleGattProfileListAdapter**

```

...
/**
 * These UI components are in each characteristic List Item in the ListView
 */

```

```

public static class ChildViewHolder{
    public TextView mUuidTV; // displays UUID
    public TextView mPropertyReadableTV; // Characteristic is readable
    public TextView mPropertyWritableTV; // Characteristic is writeable
    public TextView mPropertyNotifiableTV; // Characteristic is Notifiable
    public TextView mPropertyNoneTV; // displays when no access is given
}
/***
 * Generate a new Characteristic ListItem for some known position
 *
 * @param groupPosition the position of the Service ListItem
 * @param childPosition the position of the Characterstic ListItem
 * @param convertView An existing List Item
 * @param parent The Parent ViewGroup
 * @return The Characteristic ListItem
 */
@Override
public View getChildView(final int groupPosition, final int childPosition,
        boolean isLastChild, View convertView, ViewGroup parent) {
    View v = convertView;
    ChildViewHolder characteristicListView;
    BleGattCharacteristicListItem item =
            getChild(groupPosition, childPosition);
    BluetoothGattCharacteristic characteristic = item.getCharacteristic();
    // if this ListItem does not exist yet, generate it
    // otherwise, use it
    if (convertView == null) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        v = inflater.inflate(R.layout.list_item_ble_characteristic, null);
        // match the UI stuff in the list Item to what's in the xml file
        characteristicListView = new ChildViewHolder();
        characteristicListView.mUuidTV =
                (TextView) v.findViewById(R.id.uuid);
        characteristicListView.mPropertyReadableTV =
                (TextView)v.findViewById(R.id.property_read);
        characteristicListView.mPropertyWritableTV =

```

```

        (TextView)v.findViewById(R.id.property_write);
characteristicListItemView.mPropertyNotifiableTV =
        (TextView)v.findViewById(R.id.property_notify);
characteristicListItemView.mPropertyNoneTV =
        (TextView)v.findViewById(R.id.property_none);
} else {
    characteristicListItemView = (ChildViewHolder) v.getTag();
}
if (characteristicListItemView != null) {
    // display the UUID of the characteristic
    characteristicListItemView.mUuidTV.setText(
        characteristic.getUuid().toString());
    // Display the read/write/notify attributes of the characteristic
    if (BlePeripheral.isCharacteristicReadable(characteristic)) {
        characteristicListItemView.mPropertyReadableTV.
            setVisibility(View.VISIBLE);
    } else {
        characteristicListItemView.mPropertyReadableTV.
            setVisibility(View.GONE);
    }
    if (BlePeripheral.isCharacteristicWritable(characteristic)) {
        characteristicListItemView.mPropertyWritableTV.
            setVisibility(View.VISIBLE);
    } else {
        characteristicListItemView.mPropertyWritableTV.
            setVisibility(View.GONE);
    }
    if (BlePeripheral.isCharacteristicNotifiable(characteristic)) {
        characteristicListItemView.mPropertyNotifiableTV.
            setVisibility(View.VISIBLE);
    } else {
        characteristicListItemView.mPropertyNotifiableTV.
            setVisibility(View.GONE);
    }
    if (!BlePeripheral.isCharacteristicNotifiable(characteristic) &&
        !BlePeripheral.isCharacteristicWritable(characteristic) &&

```

```
        !BlePeripheral.isCharacteristicReadable(characteristic)) {  
            characteristicListView.mPropertyNoneTV.  
                setVisibility(View.VISIBLE);  
        } else {  
            characteristicListView.mPropertyNoneTV.  
                setVisibility(View.GONE);  
        }  
    }  
    return v;  
}  
...
```

## Views

Modify item\_list\_ble\_characteristic.xml to include a TextView that displays when a Characteristic is notifiable

### Example 9-4. res/layout/list\_item\_characteristic.xml

```
...  
  
<LinearLayout  
    android:orientation="horizontal"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_vertical"  
    android:layout_weight="2">  
    <TextView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:textAppearance="?android:attr/textAppearanceSmall"  
        android:text="@string/property_read"  
        android:visibility="gone"  
        android:id="@+id/property_read"  
        android:paddingLeft="@dimen/activity_horizontal_margin" />  
    <TextView  
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_write"
        android:visibility="gone"
        android:id="@+id/property_write"
        android:paddingLeft="@dimen/activity_horizontal_margin" />
<TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_notify"
        android:visibility="gone"
        android:id="@+id/property_notify"
        android:paddingLeft="@dimen/activity_horizontal_margin" />
<TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="@string/property_none"
        android:visibility="gone"
        android:id="@+id/property_none"
        android:paddingLeft="@dimen/activity_horizontal_margin" />
</LinearLayout>
...

```

## Activities

Modify the Talk activity to handle subscription and unsubscription requests. Also add functionality to handle incoming notifications by requesting a to read the Characteristic.

### Example 9-5. java/example.com.exampleble/TalkActivity.java

```
public class TalkActivity extends AppCompatActivity {
...
    private CheckBox mSubscribeCheckbox;
```

```

// onCreate, onResume, onPause
/***
 * Load UI components
 */
public void loadUI() {
    mResponseText = (TextView) findViewById(R.id.response_text);
    mSendText = (TextView) findViewById(R.id.write_text);
    mPeripheralAdvertiseNameTV = \
        (TextView) findViewById(R.id.advertise_name);
    mPeripheralAddressTV = (TextView) findViewById(R.id.mac_address);
    mServiceUUIDTV = (TextView) findViewById(R.id.service_uuid);
    mSubscribeCheckbox = (CheckBox) findViewById(R.id.subscribe_checkbox);
    mSendButton = (Button) findViewById(R.id.write_button);
    mReadButton = (Button) findViewById(R.id.read_button);
    mPeripheralAdvertiseNameTV.setText(R.string.connecting);
    mServiceUUIDTV.setText(mCharacteristicUUID.toString());
    mSendButton.setVisibility(View.GONE);
    mSendText.setVisibility(View.GONE);
    mReadButton.setVisibility(View.GONE);
    mResponseText.setVisibility(View.GONE);
    mSubscribeCheckbox.setVisibility(View.GONE);
}
// onCreateOptionsMenu, onPrepareOptionsMenu, onOptionsItemSelected
// onCharacteristicWritable, onBleMessageSent
// onCharacteristicReadable, onUpdateResponseText
// disconnect
/***
 * Characteristic supports notifications. Update UI
 *
 * New in this chapter
 */
public void onCharacteristicNotifiable() {
    mSubscribeCheckbox.setVisibility(View.VISIBLE);
    mSubscribeCheckbox.setOnCheckedChangeListener(
        new CompoundButton.OnCheckedChangeListener() {
            @Override

```

```

        public void onCheckedChanged(
                CompoundButton buttonview, boolean isChecked) {
            // subscribe to notifications from this channel
            mBlePeripheral.setCharacteristicNotification(
                    mCharacteristic, isChecked);
        }
    }

}

/**
 * BluetoothGattCallback handles connections, state changes, reads,
 * writes, and GATT profile listings to a Peripheral
 *
 */
private final BluetoothGattCallback mGattCallback =
        new BluetoothGattCallback() {

    ...

    /**
     * Characteristic value changed. Read new value.
     *
     * @param gatt Connection to the GATT
     * @param characteristic The Characterstic
     */
    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
            final BluetoothGattCharacteristic characteristic) {
        mBlePeripheral.readValueFromCharacteristic(characteristic);
    }

    ...

    /**
     * GATT Profile discovered. Update UI
     * @param bluetoothGatt connection to GATT
     * @param status status of operation
     */
    @Override
    public void onServicesDiscovered(BluetoothGatt bluetoothGatt,

```

```

    int status) {

        // iterate through services and display them on screen
        if (status == BluetoothGatt.GATT_SUCCESS) {
            // connect to a specific service
            BluetoothGattService gattService =
                bluetoothGatt.getService(mServiceUUID);
            // ask for this service's characteristics:
            List<BluetoothGattCharacteristic> characteristics =
                gattService.getCharacteristics();
            for (BluetoothGattCharacteristic characteristic :
                    characteristics) {
                if (characteristic != null) {
                    Log.v(TAG, "found characteristic: " +
                        characteristic.getUuid().toString());
                }
            }
            mCharacteristic = bluetoothGatt.getService(mServiceUUID).
                getCharacteristic(mCharacteristicUUID);
            if (BlePeripheral.isCharacteristicReadable(mCharacteristic)) {
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        onCharacteristicReadable();
                    }
                });
            }
            if (BlePeripheral.isCharacteristicWritable(mCharacteristic)) {
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        onCharacteristicWritable();
                    }
                });
            }
            if (BlePeripheral.isCharacteristicNotifiable(mCharacteristic))
            {

```

```
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onCharacteristicNotifiable();
            }
        });
    } else {
        Log.e(TAG, "Problem discovering GATT services");
    }
}
...
}
```

Add a checkbox to the Talk activity layout file.

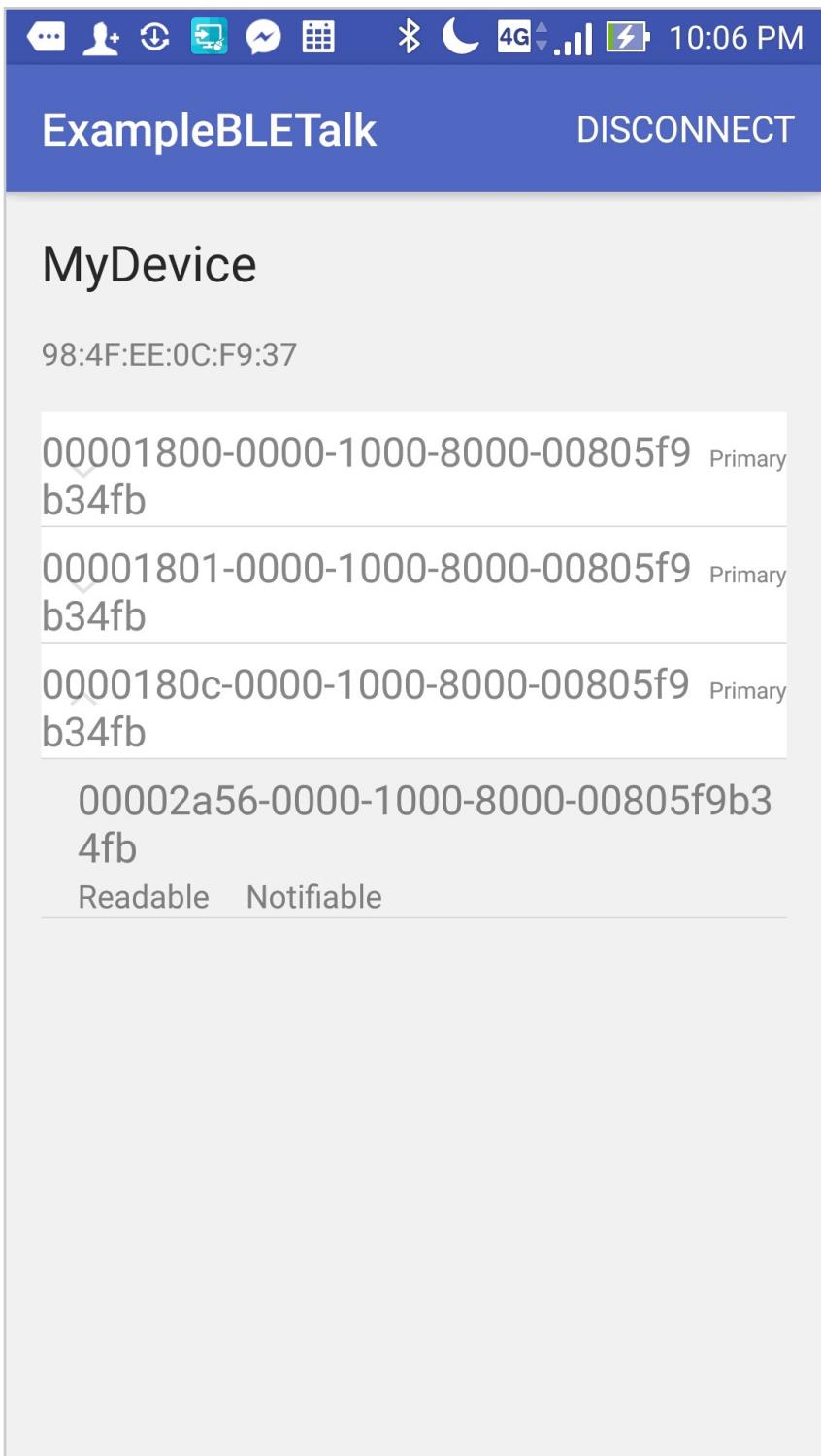
#### Example 9-6. res/layout/activity\_talk.xml

```
...
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/read_button"
        android:id="@+id/read_button" />
    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/notify_checkbox"
        android:id="@+id/subscribe_checkbox"
        android:layout_weight="2"
        android:checked="false" />
```

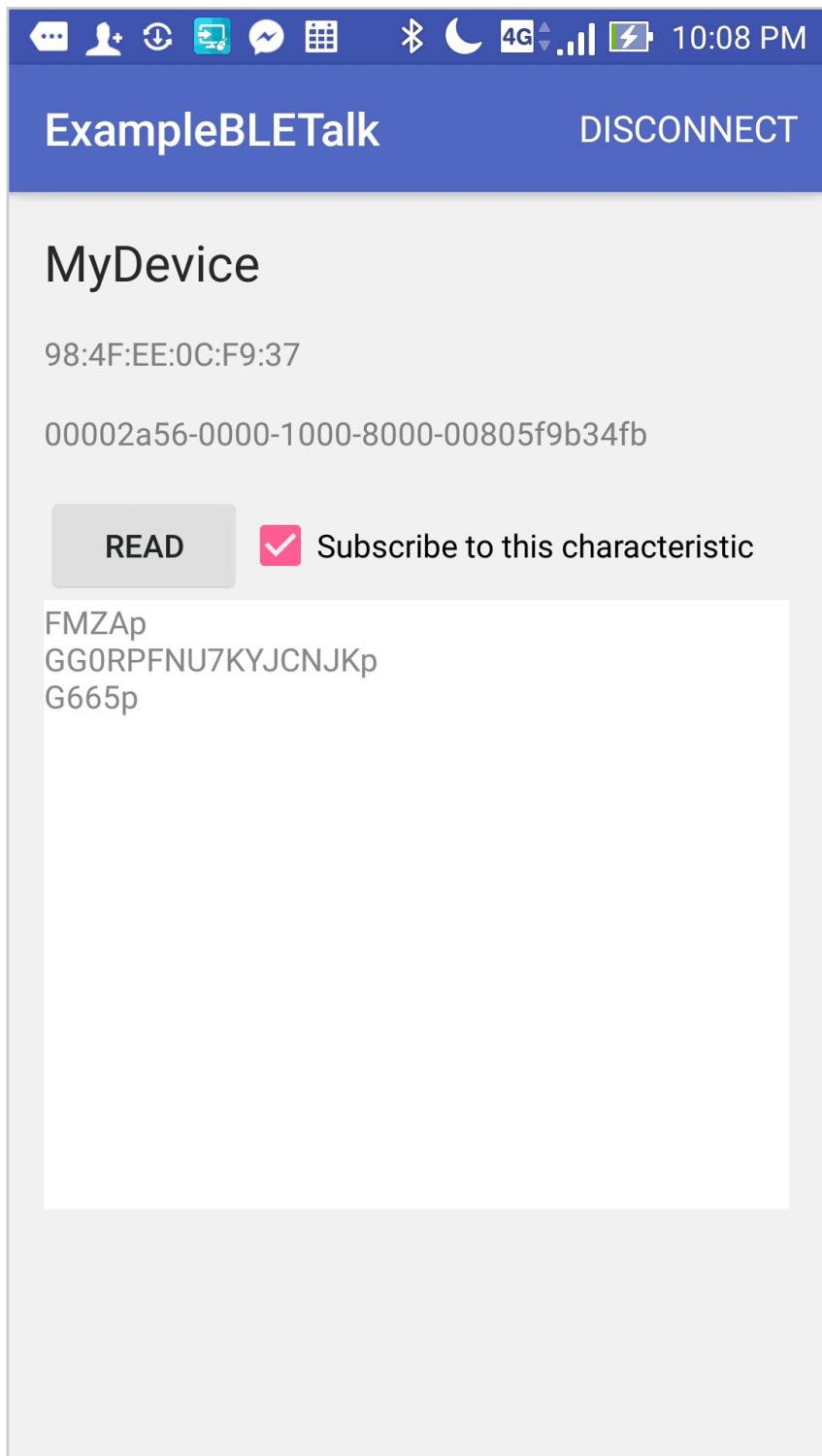
```
</LinearLayout>  
...
```

Compile and run. The app can scan for and connect to a Peripheral. Once connected, it can display the GATT profile of the Peripheral. It can connect to Characteristics, Subscribe to notifications, and receive data without polling ([Figure 9-2](#)).

When you hit the “Subscribe to this characteristic” button, the text field should begin populating with random text generated on the Peripheral ([Figure 9-3](#)).



**Figure 9-2. App screen showing GATT Profile of connected Peripheral**



**Figure 9-3. App screen showing value changes to a Characteristic on a connected Peripheral**

## Programming the Peripheral

Often, battery life is at a premium on Bluetooth Peripherals. For this reason, it is useful to notify a connected Central when a Characteristic's value has changed, but not

send the new data to the Central. Waking up the Bluetooth radio to send one byte consumes less battery than sending 20 or more bytes.

Notifications can be enabled in a Characteristic by creating the Client Characteristic Configuration (0x2902) Descriptor and giving the Characteristic Descriptors PERMISSION\_WRITE permission.

For example, this Characteristic supports both read access and notifications

```
public static final UUID CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");

// Give permission to write to the characteristic
// so that notifications can be enabled or disabled
mCharacteristic = new BluetoothGattCharacteristic(
    CHARACTERISTIC_UUID,
    // Characteristic has Read and notification properties
    BluetoothGattCharacteristic.PROPERTY_READ | \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY,
    // Descriptors can be written to or read
    BluetoothGattCharacteristic.PERMISSION_READ | \
        BluetoothGattCharacteristic.PERMISSION_WRITE);
```

When a Central attempts to change a Descriptor, the `onDescriptorWriteRequest()` method is fired in the `BluetoothGattServerCallback`. This method provides details about the Descriptor and what value is being set.

**Table 9-3. BluetoothGattServerCallback**

Event	Description
<code>onConnectionStateChange</code>	Central connects or disconnects
<code>onCharacteristicReadRequest</code>	Characteristic supports notifications
<code>onCharacteristicWriteRequest</code>	Central attempted to read a value from a Characteristic

This method is implemented like this:

```
private final BluetoothGattCallback gattCallback = new BluetoothGattCallback()
{
    @Override
    public void onDescriptorWriteRequest(
        BluetoothDevice device, int requestId,
        BluetoothGattDescriptor descriptor, boolean preparedWrite,
        boolean responseNeeded,
        int offset,
        byte[] value
    ) {
        // bubble the event up to the parent object
        super.onDescriptorWriteRequest(
            device,
            requestId,
            descriptor,
            preparedWrite,
            responseNeeded,
            offset,
            value
        );
        // determine which characteristic is being requested
        BluetoothGattCharacteristic characteristic = \
            descriptor.getCharacteristic();

        // if the descriptor is writeable, change the value
        if (isDescriptorWriteable(descriptor)) {
            descriptor.setValue(value);
            // if the Descriptor is readable, send a success notification
            if(isDescriptorReadable(descriptor)) {
                mGattServer.sendResponse(
                    device,
                    requestId,
                    BluetoothGatt.GATT_SUCCESS,
                    offset,
                    value
                );
            }
        }
    }
}
```

```

    );
}

} else {
    // if the Descriptor is readable, send an error notification
    if (isDescriptorReadable(descriptor)) {
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_WRITE_NOT_PERMITTED,
            offset,
            value
        );
    }
}
};

}

```

A Descriptor's permissions can be inspected to determine if it is readable or writeable, like this:

```

public static boolean isDescriptorReadable(
    BluetoothGattDescriptor descriptor
) {
    return (descriptor.getPermissions() & \
        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
}

public static boolean isDescriptorWriteable(
    BluetoothGattDescriptor descriptor
) {
    return (descriptor.getPermissions() & \
        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
}

```

# Putting It All Together

These features will be added to BlePeripheral to allow notification subscriptions from Characteristics.

## Resources

The Main activity will be modified to display the subscription status of the single Characteristic in the App using a Switch. Set the Switch label text:

### Example 9-7. res/values/strings.xml

```
...  
    <string name="characteristic_subscribed">characteristic subscribed</string>  
...
```

## Objects

The BlePeripheral will be modified to:

- Identify the Client Characteristic Configuration (0x2902)
- Create a Characteristic with read and notify Properties to allow notifications, and read and write Permissions to allow changes to the Descriptor, and change the Characteristic's value to a random String every 5 seconds
- isDescriptorReadable() and isDescriptorWriteable() methods to determine the read and write capabilities of a Descriptor
- Include the onDescriptorWriteRequest() in the BluetoothGattServerCallback so that the Peripheral can respond to subscription requests

## Example 9-8. java/example.com.exampleble/ble/BlePeripheral

```
...
// Client Characteristic Configuration Descriptor
public static final UUID NOTIFY_DESCRIPTOR_UUID = \
    UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");
...

/**
 * Set up the Advertising name and GATT profile
 */
private void setupDevice() {
    mService = new BluetoothGattService(
        SERVICE_UUID,
        BluetoothGattService.SERVICE_TYPE_PRIMARY
    );
    // Give permission to write to the Characteristic
    // so that notifications can be enabled or disabled
    mCharacteristic = new BluetoothGattCharacteristic(
        CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ | \
            BluetoothGattCharacteristic.PROPERTY_NOTIFY,
        BluetoothGattCharacteristic.PERMISSION_READ | \
            BluetoothGattCharacteristic.PERMISSION_WRITE);
    // add Notification support to Characteristic
    BluetoothGattDescriptor notifyDescriptor = \
        new BluetoothGattDescriptor(
            NOTIFY_DESCRIPTOR_UUID,
            BluetoothGattDescriptor.PERMISSION_WRITE | \
                BluetoothGattDescriptor.PERMISSION_READ
        );
    mCharacteristic.addDescriptor(notifyDescriptor);
    mService.addCharacteristic(mCharacteristic);
    mGattServer.addService(mService);
    // write random characters to the Read Characteristic
    // every timerInterval_ms
    int timerInterval_ms = 1000;
    TimerTask updateReadCharacteristicTask = new TimerTask() {
```

```

@Override
public void run() {
    int stringLength = (int) (
        Math.random() % CHARACTERISTIC_LENGTH
    );
    String randomString = DataConverter.getRandomString(
        stringLength
    );
    mCharacteristic.setValue(randomString);
}

};

Timer randomStringTimer = new Timer();
randomStringTimer.schedule(
    updateReadCharacteristicTask,
    0,
    timerInterval_ms
);
}

...
/***
 * Check if a Characteristic supports Notifications
 *
 * @return Returns <b>true</b> if property is supports notification
 */
public static boolean isCharacteristicNotifiable(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY) != 0;
}

/***
 * Check if a Descriptor can be read
 *
 * @param descriptor a descriptor to check
 * @return Returns <b>true</b> if descriptor is readable
*/

```

```

*/
public static boolean isDescriptorReadable(
    BluetoothGattDescriptor descriptor
) {
    return (descriptor.getPermissions() & \
        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
}

/**
 * Check if a Descriptor can be written
 *
 * @param descriptor a descriptor to check
 * @return Returns <b>true</b> if descriptor is writeable
 */
public static boolean isDescriptorWriteable(
    BluetoothGattDescriptor descriptor
) {
    return (descriptor.getPermissions() & \
        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
}

private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onConnectionStateChange(
        BluetoothDevice device,
        final int status,
        int newState
    ) {
        super.onConnectionStateChange(device, status, newState);
        Log.v(TAG, "Connected");
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothGatt.STATE_CONNECTED) {
                mBlePeripheralCallback.onCentralConnected(device);
                stopAdvertising();
            }
        }
    }
}

```

```
        } else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
            mBlePeripheralCallback.onCentralDisconnected(device);
            try {
                startAdvertising();
            } catch (Exception e) {
                Log.e(TAG, "error starting advertising");
            }
        }
    }

@Override
public void onCharacteristicReadRequest(
    BluetoothDevice device,
    int requestId,
    int offset,
    BluetoothGattCharacteristic characteristic
) {
    super.onCharacteristicReadRequest(
        device,
        requestId,
        offset,
        characteristic
    );
    Log.d(
        TAG,
        "Device tried to read characteristic: " + \
        characteristic.getUuid()
    );
    Log.d(
        TAG,
        "value: " + \
        Arrays.toString(characteristic.getValue())
    );
    if (offset != 0) {
        mGattServer.sendResponse(

```

```

        device,
        requestId,
        BluetoothGatt.GATT_INVALID_OFFSET,
        offset,
        /* value (optional) */ null
    );
    return;
}

mGattServer.sendResponse(
    device,
    requestId,
    BluetoothGatt.GATT_SUCCESS,
    offset,
    characteristic.getValue()
);
}

@Override
public void onNotificationSent(BluetoothDevice device, int status) {
    super.onNotificationSent(device, status);
    Log.v(TAG, "Notification sent. Status: " + status);
}

@Override
public void onCharacteristicWriteRequest(
    BluetoothDevice device,
    int requestId,
    BluetoothGattCharacteristic characteristic,
    boolean preparedwrite,
    boolean responseNeeded,
    int offset,
    byte[] value
) {
    super.onCharacteristicWriteRequest(
        device,
        requestId,

```

```

        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
    // do nothing; no characteristic in this Peripheral supports writes
}

// User tried to change a descriptor -
// check for Notification flag being set
@Override
public void onDescriptorWriteRequest(
    BluetoothDevice device,
    int requestId,
    BluetoothGattDescriptor descriptor,
    boolean preparedwrite,
    boolean responseNeeded,
    int offset,
    byte[] value
) {
    Log.v(
        TAG,
        "Descriptor Write Request " + descriptor.getUuid() + \
        " " + Arrays.toString(value)
    );
    super.onDescriptorWriteRequest(
        device,
        requestId,
        descriptor,
        preparedwrite,
        responseNeeded,
        offset, value
    );
    // determine which characteristic is being requested
    BluetoothGattCharacteristic characteristic = \

```

```
        descriptor.getCharacteristic();

        if (isDescriptorWriteable(descriptor)) {
            descriptor.setValue(value);
        }

        // was this a notification subscription?
        if (descriptor.getUuid().equals(NOTIFY_DESCRIPTOR_UUID)) {
            if (isCharacteristicNotifiable(characteristic)) {
                if (value == \
                    BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE
                ) {
                    mBlePeripheralCallback.onCharacteristicSubscribedTo(
                        characteristic
                    );
                } else if (value == \
                    BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE
                ) {
                    mBlePeripheralCallback.\
                        onCharacteristicUnsubscribedFrom(
                            characteristic
                        );
                }
            }
        }

        if(isDescriptorReadable(descriptor)) {
            mGattServer.sendResponse(
                device,
                requestId,
                BluetoothGatt.GATT_SUCCESS,
                offset,
                value
            );
        }
    }
};

...

```

Modify the BlePeripheralCallback to include functionality to relay Characteristic subscription and unsubscription events.

### Example 9-9. java/example.com.exampleble/ble/callbacks/BlePeripheralCallback

```
...
/** 
 * Characteristic subscribed to
 *
 * @param characteristic The characteristic that was subscribed to
 */
public abstract void onCharacteristicSubscribed(
    final BluetoothGattCharacteristic characteristic
);

/** 
 * Characteristic unsubscribed from
 *
 * @param characteristic The characteristic that was unsubscribed from
 */
public abstract void onCharacteristicUnsubscribed(
    final BluetoothGattCharacteristic characteristic
);

...
```

## Activities

Modify the Main activity to display the subscription status.

### Example 9-10. java/example.com.exampleble/MainActivity.java

```
...
/** UI Stuff */
private TextView mAdvertisingNameTV, mCharacteristicLogTV;
private Switch mBluetoothOnSwitch,
    mAdvertisingSwitch,
```

```

    mCentralConnectedSwitch,
    mCharacteristicSubscribedSwitch;
...

/***
 * Load UI components
 */
public void loadUI() {
    mAdvertisingNameTV = (TextView) findViewById(R.id.advertising_name);
    mCharacteristicLogTV = (TextView) findViewById(R.id.characteristic_log);
    mBluetoothOnSwitch = (Switch) findViewById(R.id.bluetooth_on);
    mAdvertisingSwitch = (Switch) findViewById(R.id.advertising);
    mCentralConnectedSwitch = (Switch) findViewById(R.id.central_connected);
    mCharacteristicSubscribedSwitch = \
        (Switch) findViewById(R.id.characteristic_subscribed);
    mAdvertisingNameTV.setText(MyBlePeripheral.ADVERTISING_NAME);
}

...
/***
 * Event trigger when characteristic has been subscribed to
 *
 * @param characteristic the Characteristic being subscribed to
 */
public void onBleCharacteristicSubscribed(
    final BluetoothGattCharacteristic characteristic
) {
    mCharacteristicSubscribedSwitch.setChecked(true);
}

/***
 * Event trigger when characteristic has been unsubscribed from
 *
 * @param characteristic The Characteristic being unsubscribed from
 */
public void onBleCharacteristicUnsubscribed(
    final BluetoothGattCharacteristic characteristic
)

```

```

        mCharacteristicSubscribedSwitch.setChecked(false);
    }

...
}

/**
 * Respond to changes to the Bluetooth Peripheral state
 */
private final BlePeripheralCallback mBlePeripheralCallback = \
    new BlePeripheralCallback()
{
    public void onAdvertisingStarted() {
        Log.v(TAG, "Advertising started");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleAdvertisingStarted();
            }
        });
    }

    public void onAdvertisingFailed(int errorCode) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleAdvertisingFailed();
            }
        });
    }

    switch (errorCode) {
        case AdvertiseCallback.ADVERTISE_FAILED_ALREADY_STARTED:
            Log.e(
                TAG,
                "Failed to start; advertising has already started."
            );
            break;
        case AdvertiseCallback.ADVERTISE_FAILED_DATA_TOO_LARGE:
            Log.e(
                TAG,

```

```
        "Failed to start; advertised data is > 31 bytes."
    );
    break;
case AdvertiseCallback.ADVERTISE_FAILED_FEATURE_UNSUPPORTED:
    Log.e(
        TAG,
        "This feature is not supported on this platform."
    );
    break;
case AdvertiseCallback.ADVERTISE_FAILED_INTERNAL_ERROR:
    Log.e(
        TAG,
        "Operation failed due to an internal error."
    );
    break;
case AdvertiseCallback.ADVERTISE_FAILED_TOO_MANY_ADVERTISERS:
    Log.e(
        TAG,
        "Failed to start; advertising instance is unavailable."
    );
    break;
default:
    Log.e(TAG, "unknown problem");
}
}

public void onAdvertisingStopped() {
    Log.v(TAG, "Advertising stopped");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleAdvertisingStopped();
        }
    });
}
```

```
public void onCentralConnected(final BluetoothDevice bluetoothDevice) {
    Log.v(TAG, "Central connected");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCentralConnected(bluetoothDevice);
        }
    });
}

public void onCentralDisconnected(
    final BluetoothDevice bluetoothDevice
) {
    Log.v(TAG, "Central disconnected");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCentralDisconnected(bluetoothDevice);
        }
    });
}

public void onCharacteristicWritten(
    final BluetoothGattCharacteristic characteristic,
    final byte[] value
) {
    Log.v(TAG, "Characterstic written to");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBlecharacteristicwritten(characteristic, value);
        }
    });
}

public void onCharacteristicSubscribedTo(
```

```
    final BluetoothGattCharacteristic characteristic
) {
    Log.v(TAG, "Characteristic subscribed to");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCharacteristicSubscribed(characteristic);
        }
    });
}
public void onCharacteristicUnsubscribed(
    final BluetoothGattCharacteristic characteristic
) {
    Log.v(TAG, "Characteristic unsubscribed from");
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCharacteristicUnsubscribed(characteristic);
        }
    });
}
};

...

```

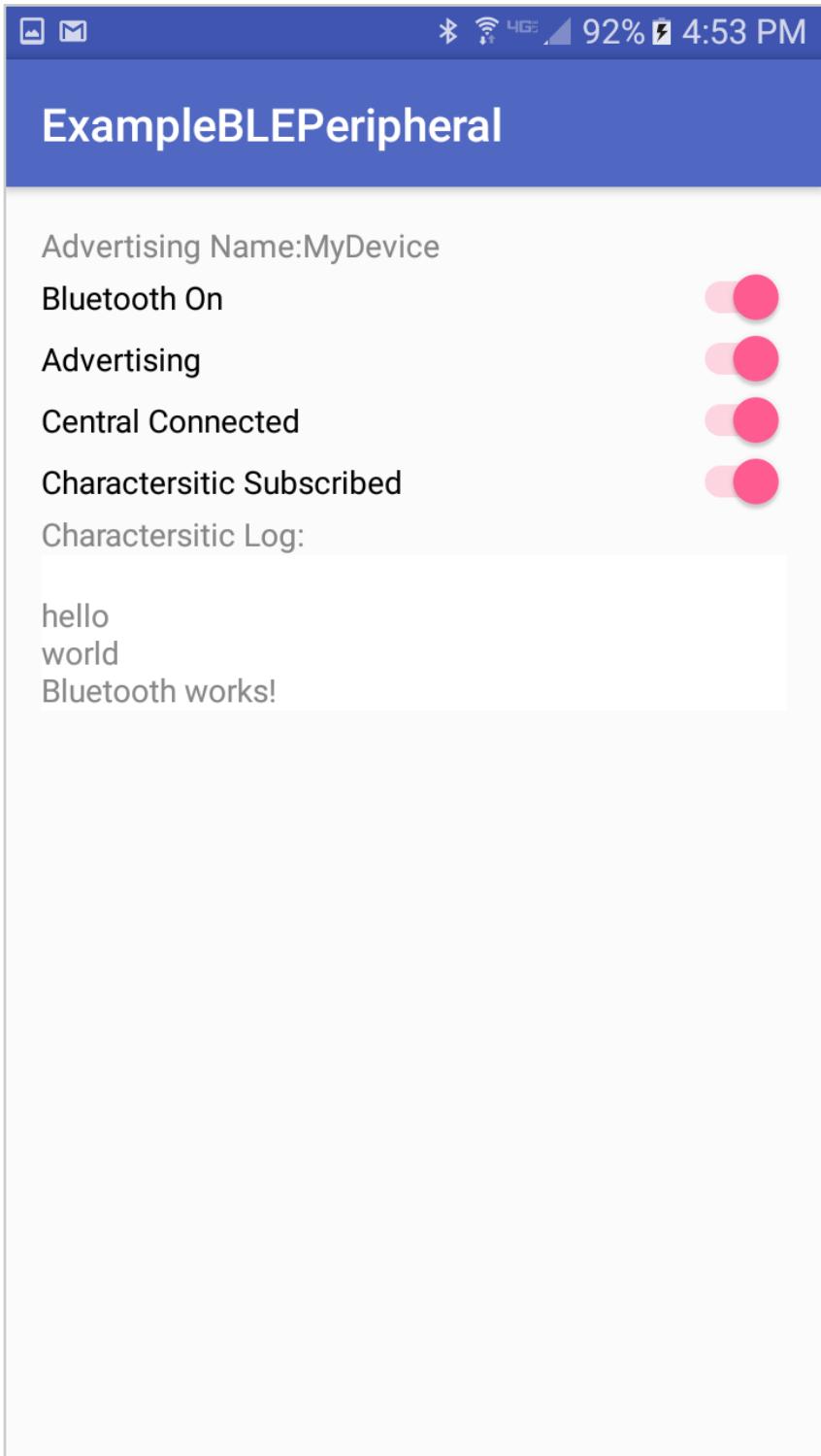
Add a Switch to the Main activity layout file that shows the subscription status of the Characteristic.

### Example 9-11. res/layout/activity\_main.xml

```
...
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/central_connected"
    android:id="@+id/central_connected" />
```

```
<Switch  
    android:clickable="false"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="@string/characteristic_subscribed"  
    android:id="@+id/characteristic_subscribed" />  
  
<TextView  
    android:text="@string/characteristic_log"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content" />  
  
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:background="#ffffff"  
    android:id="@+id/characteristic_log" />  
/</LinearLayout>  
</RelativeLayout>  
</android.support.design.widget.CoordinatorLayout>
```

Compile and run. The app can handle subscriptions to a Characteristic and send notifications when the Characteristic's value has changed ([Figure 9-4](#)).



**Figure 9-4. App screen showing GATT Profile of connected Peripheral**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter09>

# Streaming Data

The maximum packet size you can send over Bluetooth Low Energy is 20 bytes. More data can be sent by dividing a message into packets of 20 bytes or smaller, and sending them one at a time

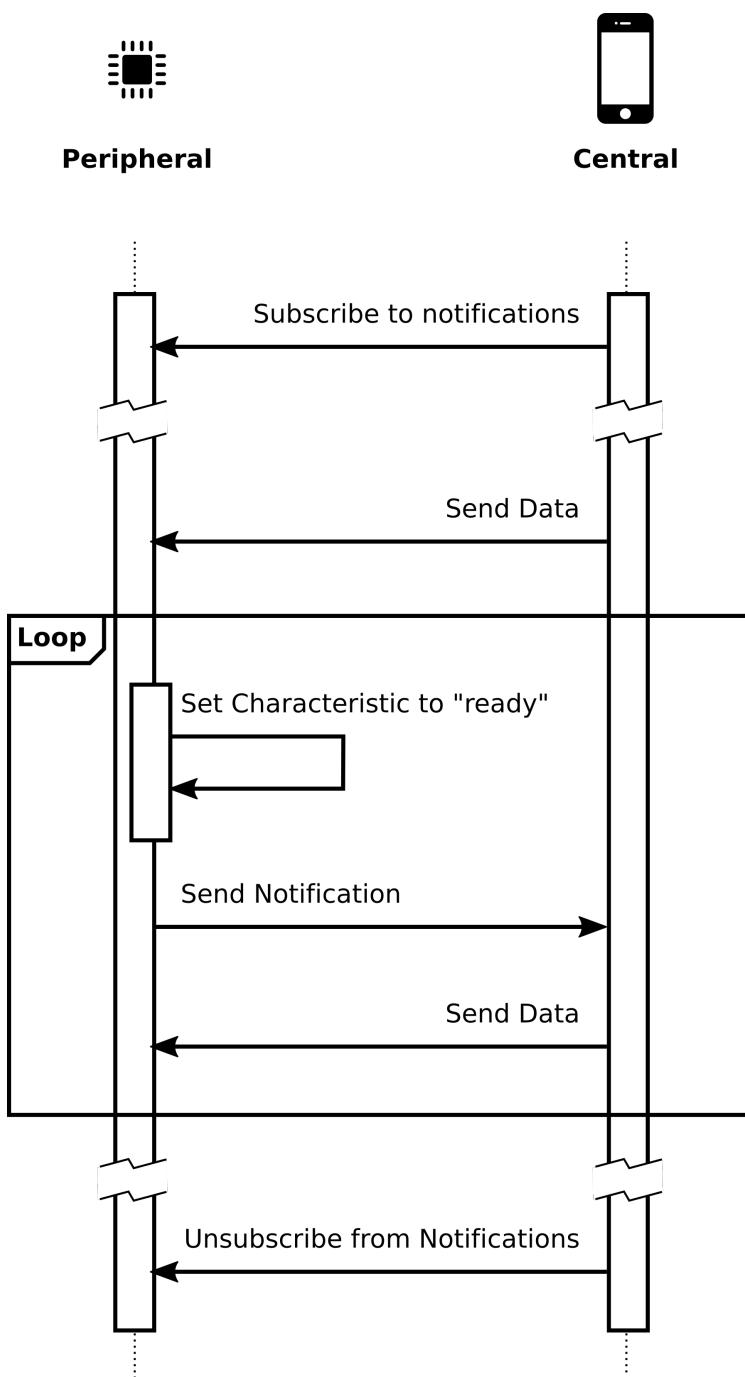
These packets can be sent at a certain speed.

Bluetooth Low Energy transmits at 1 Mb/s. Between the data transmission time and the time it may take for a Peripheral to process incoming data, there is a time delay between when one packet is sent and when the next one is ready to be sent.

To send several packets of data, a queue/notification system must be employed, which alerts the Central when the Peripheral is ready to receive the next packet.

There are many ways to do this. One way is to set up a Characteristic with read, write, and notify permissions, and to flag the Characteristic as “ready” after a write has been processed by the Peripheral. This sends a notification to the Central, which sends the next packet. That way, only one Characteristic is required for a single data transmission.

This process can be visualized like this ([Figure 10-1](#)).



**Figure 10-1. The process of using notifications to handle flow control on a multi-packed data transfer**

The maximum packet size you can send over Bluetooth Low Energy is 20 bytes. More data can be sent by dividing a message into packets of 20 bytes or smaller, and sending them one at a time

These packets can be sent at a certain speed.

Bluetooth Low Energy transmits at 1 Mb/s. Between the data transmission time and the time it may take for a Peripheral to process incoming data, there is a time delay between when one packet is sent and when the next one is ready to be sent.

To send several packets of data, a queue/notification system must be employed, which alerts the Central when the Peripheral is ready to receive the next packet.

There are many ways to do this. One way is to set up a Characteristic with read, write, and notify permissions, and to flag the Characteristic as “ready” after a write has been processed by the Peripheral. This sends a notification to the Central, which sends the next packet. That way, only one Characteristic is required for a single data transmission.

## Programming the Central

The Central in this example waits for the Peripheral to change the Characteristic value to the “ready” flow control message and to send a notification indicating the change. This process signals to the Central that the Characteristic is ready to receive the next packet of data.

**Table 10-1. BluetoothGattCallback**

Event	Description
<b>onConnectionStateChange</b>	Triggered when a BLE Peripheral connects or disconnects
<b>onServicesDiscovered</b>	Triggered when GATT Services are discovered
<b>onCharacteristicRead</b>	Triggered when data has been downloaded from a GATT Characteristic
<b>onCharacteristicWrite</b>	Triggered when data has been uploaded to a GATT Characteristic
<b>onCharacteristicChanged</b>	Triggered when a GATT Characteristic’s data has changed

Set up the parameters of the flow control and packet queue like this:

```
public static final String FLOW_CONTROL_MESSAGE = "ready";
private String outboundMessage; // keep track of the current outbound message
```

```
private int packetSize = 20; // packet size must match the characteristic size
private int numPacketsTotal; // total number of packets in the current message
private int numPacketsSent; // how many packets have been sent so far
```

The flow control works by requesting a Characteristic read event when a notification callback is triggered:

```
private final BluetoothGattCallback mGattcallback = \
    new BluetoothGattCallback()
{
    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
                                         final BluetoothGattCharacteristic characteristic) {
        gatt.readCharacteristic(characteristic);
    }

    @Override
    public void onCharacteristicRead(final BluetoothGatt gatt,
                                     final BluetoothGattCharacteristic characteristic,
                                     int status) {
        final byte[] data = characteristic.getValue();
        final String message = new String(data, "ASCII");
        if (message.equals(FLOW_CONROL_MESSAGE)) {
            // treat as a flow control
            sendNextPacket(
                outboundMessage,
                numPacketsSent,
                characteristic
            );
        }
        // handle read normally
    }
...
}
```

The first packet of data is initialized and sent:

```

public void sendMessage(
    String message,
    BluetoothGattCharacteristic characteristic) throws Exception
{
    outboundMessage = message;
    numPacketsSent = 0;
    byte[] messageBytes = message.getBytes();
    numPacketsTotal = (int) Math.ceil(
        (float) messageBytes.length / packetSize
    );
    sendNextPacket(message, numPacketsSent, characteristic);
}

```

Subsequent packets are sent one at a time until there are no more left:

```

public void sendNextPacket(
    String message, int offset,
    BluetoothGattCharacteristic characteristic)
{
    if (numPacketsSent >= numPacketsTotal) return; // don't send empty data
    byte[] temp = message.getBytes();
    numPacketsTotal = (int) Math.ceil((float) temp.length / packetSize);
    int remainder = temp.length % packetSize;
    int dataLength = packetSize;
    if (offset >= numPacketsTotal) {
        dataLength = remainder;
    }
    byte[] packet = new byte[dataLength];
    // copy a section of data into a packet
    for (int localIndex = 0; localIndex < packet.length; localIndex++) {
        int index = (offset * dataLength) + localIndex;
        if (index < temp.length) {
            packet[localIndex] = temp[index];
        } else {
            packet[localIndex] = 0x00; // terminate string
        }
    }
}

```

```

    }

    // write the data to the characteristic
    characteristic.setvalue(packet);
    mBluetoothGatt.writeCharacteristic(characteristic);
    numPacketsSent++; // keep track of how many packets were sent
}

```

## Putting It All Together

### Objects

Add two methods to BlePeripheral:

- `writeValueToCharacteristic` initializes the transmission of a new message.
- `writePartialValueToCharacteristic` sends a single packet and tracks how many packets have been sent
- `hasMorePackets` says if all packets have been sent
- `getCurrentOffset` says how many packets have been sent
- `getCurrentMessage` returns the current message in queue to be written

### Example 10-1. `java/example.com.exampleble/ble/BlePeripheral`

```

...
public class BlePeripheral {
    private int mNumPacketsTotal;
    private int mNumPacketsSent;
    private int mCharacteristicLength = 20;
    private String mQueuedCharactersticValue;
    public static final String FLOW_CONROL_VALUE = "ready";
    ... object properties
}

```

```

... BlePeripheral, connect, disconnect, refreshDeviceCache, etc.

/**
 * Write a value to the characteristic
 *
 * @param value
 * @param characteristic
 * @throws Exception
 */
public void writeValueToCharacteristic(
    String value,
    BluetoothGattCharacteristic characteristic) throws Exception
{
    // reset the queue counters,
    // prepare the message to be written, and write
    mQueuedCharacteristicValue = value;
    mNumPacketsSent = 0;
    byte[] byteValue = value.getBytes();
    mNumPacketsTotal = (int) Math.ceil((float) byteValue.length /
        mCharacteristicLength);
    writePartialValueToCharacteristic(value, mNumPacketsSent,
        characteristic);
}

/**
 * Write a portion of a larger message to a characteristic
 *
 * @param message The message being written
 * @param offset The current packet index in queue to be written
 * @param characteristic The characteristic being written to
 * @throws Exception
 */
public void writePartialValueToCharacteristic(String message, int offset,
    BluetoothGattCharacteristic characteristic) throws Exception {
    byte[] temp = message.getBytes();
    mNumPacketsTotal = (int) Math.ceil((float) temp.length /
        mCharacteristicLength);
    int remainder = temp.length % mCharacteristicLength;
}

```

```

int packetLength = mCharacteristicLength;
if (offset >= mNumPacketsTotal) {
    packetLength = remainder;
}
byte[] packet = new byte[packetLength];
for (int localIndex = 0; localIndex < packet.length; localIndex++) {
    int index = (offset * packetLength) + localIndex;
    if (index < temp.length) {
        packet[localIndex] = temp[index];
    } else {
        packet[localIndex] = 0x00;
    }
}
Log.v(TAG, "Writing value: '" + new String(packet, "ASCII") +
        "' to " + characteristic.getUuid().toString());
characteristic.setValue(packet);
mBluetoothGatt.writeCharacteristic(characteristic);
mNumPacketsSent++;
}

/**
 * Determine if a message has been completely written to a Characteristic
 *
 * @return <b>false</b> if all of a message is has been written
 */
public boolean morePacketsAvailableInQueue() {
    boolean morePacketsAvailable = mNumPacketsSent < mNumPacketsTotal;
    Log.v(TAG, mNumPacketsSent + " of " +
            mNumPacketsTotal + " packets sent: "+morePacketsAvailable);
    return morePacketsAvailable;
}

/**
 * Determine how much of a message has been written to a Characteristic
 *
 * @return integer representing how many packets have been written so far
 */
public int getCurrentOffset() {

```

```

        return mNumPacketsSent;
    }

    /**
     * Get the current message being written to a characteristic
     *
     * @return the message in queue for writing to a characteristic
     */
    public String getCurrentMessage() {
        return mQueuedCharacteristicValue;
    }
}

```

## Activities

Modify the Talk activity to handle new user interface elements and to handle flow control notifications.

### **Example 10-2. java/example.com.exampleble/TalkActivity.java**

```

...
/** 
 * On a multi-part message, send the next packet of a message
 * when a write operation is successful
 *
 * @param characteristic the characteristic being written to
 */
public void onBleCharacteristicReady(
    final BluetoothGattCharacteristic characteristic) {
    Log.v(TAG, "Flow control message received by server");
    if (mBlePeripheral.morePacketsAvailableInQueue()) {
        try {
            mBlePeripheral.writePartialValueToCharacteristic(
                mBlePeripheral.getCurrentMessage(),
                mBlePeripheral.getCurrentOffset(), characteristic);
        } catch (Exception e) {
            Log.e(TAG, "Unable to send next chunk of message");
        }
    }
}

```

```

        }
    }

private final BluetoothGattCallback mGattCallback =
    new BluetoothGattCallback() {
    /**
     * Characteristic successfully read
     *
     * @param gatt connection to GATT
     * @param characteristic The characteristic that was read
     * @param status the status of the operation
     */
    @Override
    public void onCharacteristicRead(final BluetoothGatt gatt,
                                    final BluetoothGattCharacteristic characteristic,
                                    int status) {
        // characteristic was read. Convert the data to something usable
        // on Android and display it in the UI
        if (status == BluetoothGatt.GATT_SUCCESS) {
            final byte[] data = characteristic.getValue();
            String m = "";
            try {
                m = new String(data, BlePeripheral.CHARACTER_ENCODING);
            } catch (Exception e) {
                Log.e(
                    TAG,
                    "Could not convert message byte array to String"
                );
            }
            final String message = m;
            if (message.equals(BlePeripheral.FLOW_CONTROL_VALUE)) {
                onBleCharacteristicReady(characteristic);
            }
            runOnUiThread(new Runnable() {
                @Override
                public void run() {

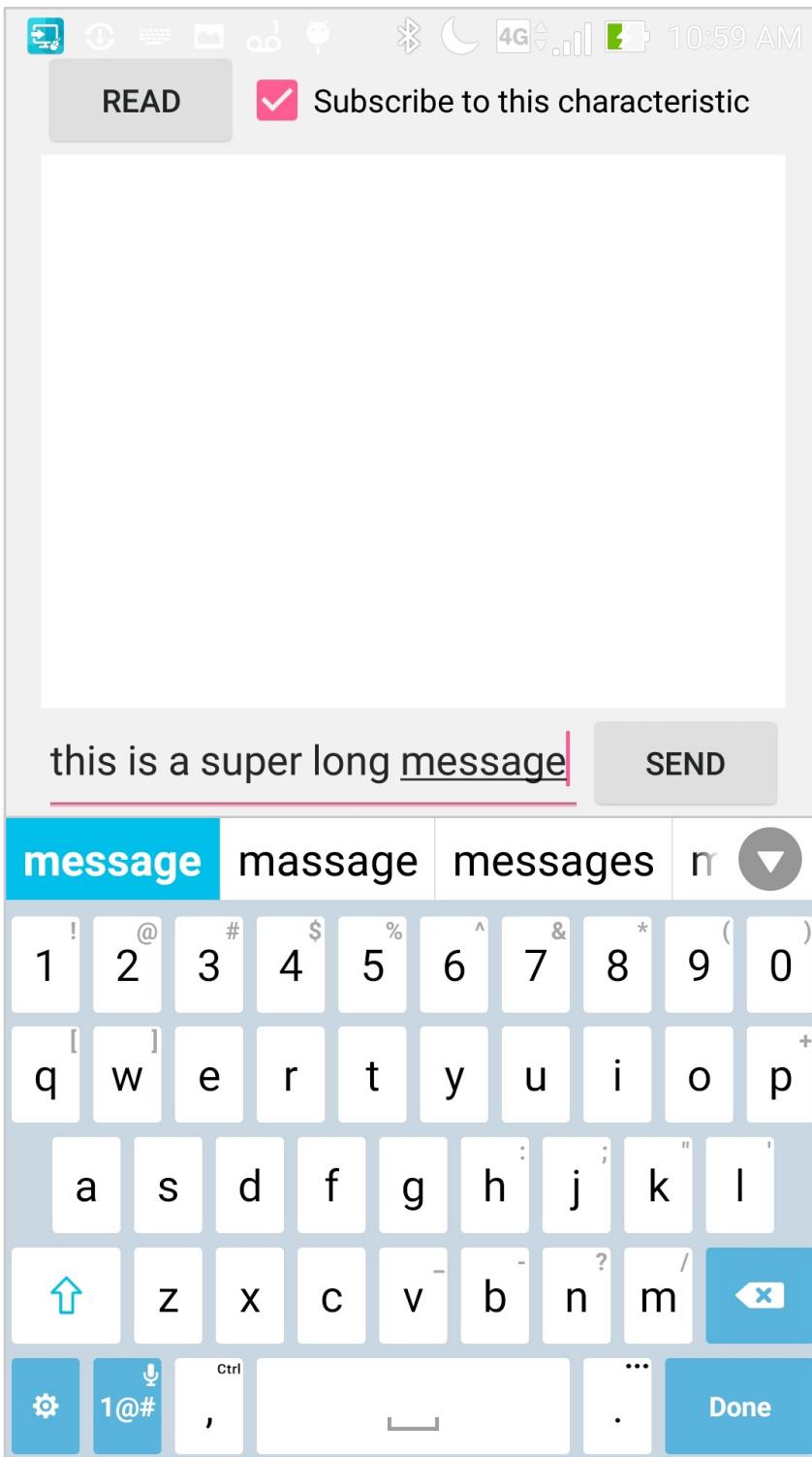
```

```
        updateResponseText(message);
    }
}

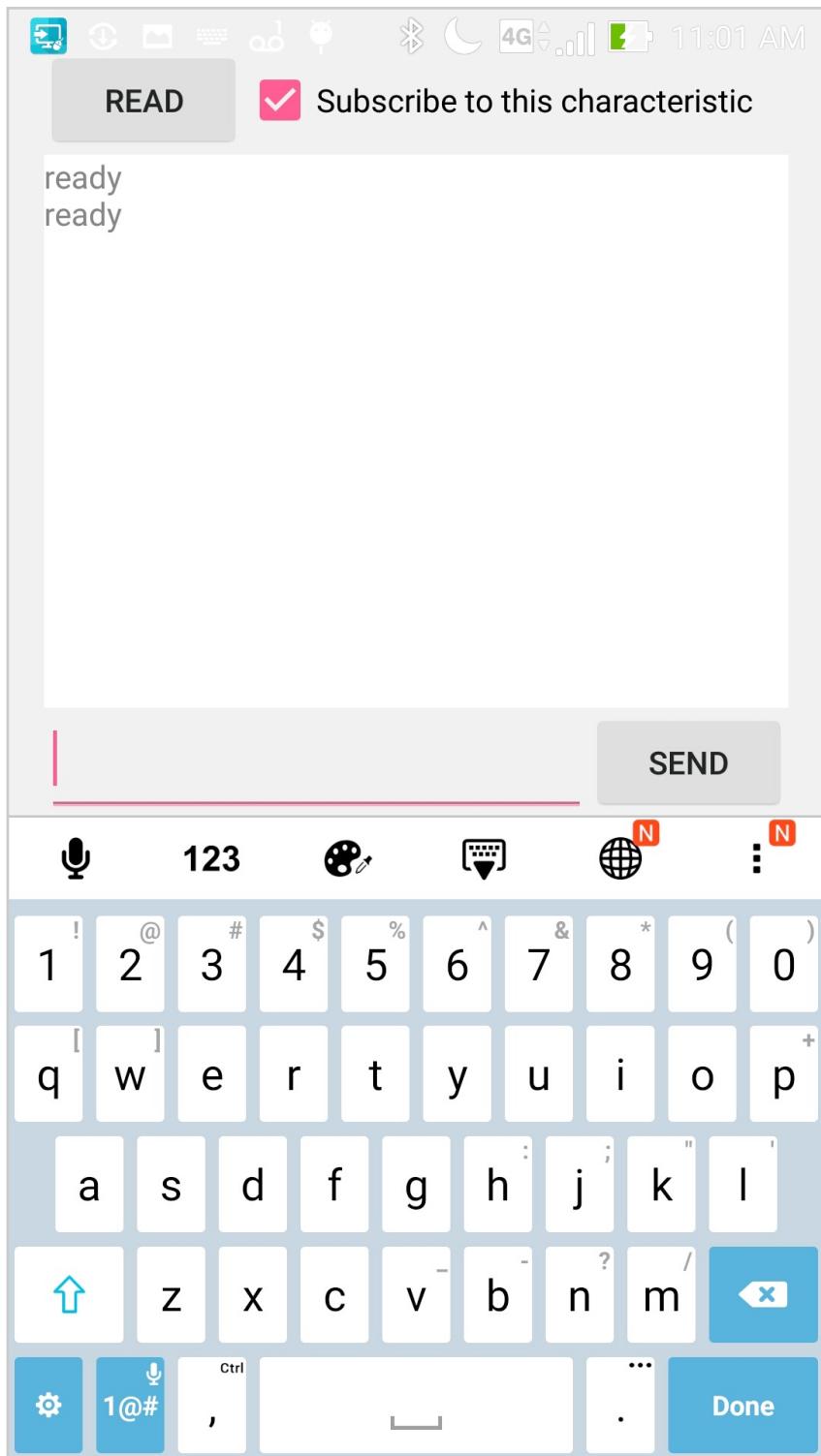
...
};

}
```

The resulting app can send larger amounts of data to a connected Peripheral by queueing ([Figure 10-2](#)) and transmitting packets one at a time ([Figure 10-3](#)).



**Figure 10-2. App screen showing multipart value queued to be sent to a Characteristic**



**Figure 10-3. App screen showing status updates from notifying Characteristic**

## Programming the Peripheral

The Peripheral in this example processes a value written to a Characteristic, then sets the Characteristic's value to the "ready" flow control message, and sends a notification to the Central. In this way, the Central is triggered to read the Characteristic.

When the Central reads the flow control message, it knows when to send the next packet of data.

When the Peripheral's Characteristic is written to, the `onCharacteristicWriteRequest` method is triggered in the `BluetoothGattServerCallback` object. When the Central subscribes or unsubscribes to notifications, the `onDescriptorWriteRequest` method is triggered.

**Table 10-2. BluetoothGattServerCallback**

Event	Description
<code>onConnectionStateChange</code>	Central connects or disconnects
<code>onCharacteristicReadRequest</code>	Characteristic supports notifications
<code>onCharacteristicWriteRequest</code>	Central attempted to read a value from a Characteristic
<code>onDescriptorWriteRequest</code>	Central attempted to change a Descriptor in a Characteristic
<code>onNotificationSent</code>	Central was notified of a change to a Characteristic

The implementation looks like this:

Set up the parameters of the flow control:

```
public static final String FLOW_CONTROL_VALUE = "ready";
public static final String CHARSET = "ASCII";
```

The `FLOW_CONTROL_VALUE` is written to the Characteristic and a notification is sent after the Central initiates a write request:

```
private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onCharacteristicWriteRequest(
        BluetoothDevice device,
        int requestId,
```

```

BluetoothGattCharacteristic characteristic,
boolean preparedwrite,
boolean responseNeeded,
int offset,
byte[] value
) {
    // bubble event up to parent object
    super.onCharacteristicWriteRequest(
        device,
        requestId,
        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
    // send write confirmation if supported by characteristic
    if (isCharacteristicWritableWithResponse(characteristic)) {
        characteristic.setValue(value);
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            0,
            null
        );
    }
    // set characteristic to flow control message
    try {
        characteristic.setValue(FLOW_CONTROL_VALUE.getBytes(CHARSET));
        // send a notification
        if (isCharacteristicNotifiable(characteristic)) {
            mGattServer.notifyCharacteristicChanged(
                device,
                characteristic,
                false

```

```

    );
}

} catch (Exception e) {
    Log.e(TAG, "problem converting string to bytes");
}

}

// onConnectionStateChange(),
// onDescriptorWriteRequest(),
// and onCharacteristicReadRequest() remain unchanged
};

}

```

## Putting It All Together

Copy the previous chapter's project into a new project.

### Objects

Add functionality to BlePeripheral to:

- describe the flow control value
- create a Characteristic that supports read, write, and notify Properties, as well as read and write Permissions.
- modify the onCharacteristicWriteRequest() method of BluetoothGattServerCallback to write the flow control value to the Characteristic and send a notification upon a write

### Example 10-3. java/example.com.exampleble/ble/BlePeripheral

```

...
/** Flow Control */
public static final String FLOW_CONTROL_VALUE = "ready";
public static final String CHARSET = "ASCII";
...

```

```

/**
 * Set up the Advertising name and GATT profile
 */
private void setupDevice() {
    // set the device name
    mBluetoothAdapter.setName(ADVERTISING_NAME);

    mService = new BluetoothGattService(
        SERVICE_UUID,
        BluetoothGattService.SERVICE_TYPE_PRIMARY
    );
    // provide WRITE permissions so Descriptors can be written to
    mCharacteristic = new BluetoothGattCharacteristic(
        CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ | \
        BluetoothGattCharacteristic.PROPERTY_WRITE | \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY,
        BluetoothGattCharacteristic.PERMISSION_READ | \
        BluetoothGattCharacteristic.PERMISSION_WRITE);

    // add a Notification descriptor
    BluetoothGattDescriptor notifyDescriptor = \
        new BluetoothGattDescriptor(
            NOTIFY_DESCRIPTOR_UUID,
            BluetoothGattDescriptor.PERMISSION_WRITE | \
            BluetoothGattDescriptor.PERMISSION_READ);
    mCharacteristic.addDescriptor(notifyDescriptor);
    mService.addCharacteristic(mCharacteristic);
    mGattServer.addService(mService);
}

...
private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onCharacteristicWriteRequest(

```

```
        BluetoothDevice device,
        int requestId,
        BluetoothGattCharacteristic characteristic,
        boolean preparedwrite,
        boolean responseNeeded,
        int offset,
        byte[] value
    ) {
    super.onCharacteristicWriteRequest(
        device,
        requestId,
        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
    Log.v(
        TAG,
        "Characteristic write request: " + Arrays.toString(value)
    );
    // send a notification to the BlePeripheralCallback
    // that a write has occurred
    mBlePeripheralCallback.onCharacteristicWritten(
        characteristic,
        value
    );
    // send a write confirmation if necessary
    if (isCharacteristicWritableWithResponse(characteristic)) {
        characteristic.setValue(value);
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            0,
            null
        );
    }
}
```

```
    );
}

// set characteristic to flow control value
try {
    characteristic.setValue(FLOW_CONTROL_VALUE.getBytes(CHARSET));
    // send a notification if necessary
    if (isCharacteristicNotifiable(characteristic)) {
        mGattServer.notifyCharacteristicChanged(
            device,
            characteristic,
            false
        );
    }
} catch (Exception e) {
    Log.e(TAG, "problem converting string to bytes");
}
}

...

```

## Activities

Modify the Main Activity to display incoming values into mCharacteristicLogTV without any formatting.

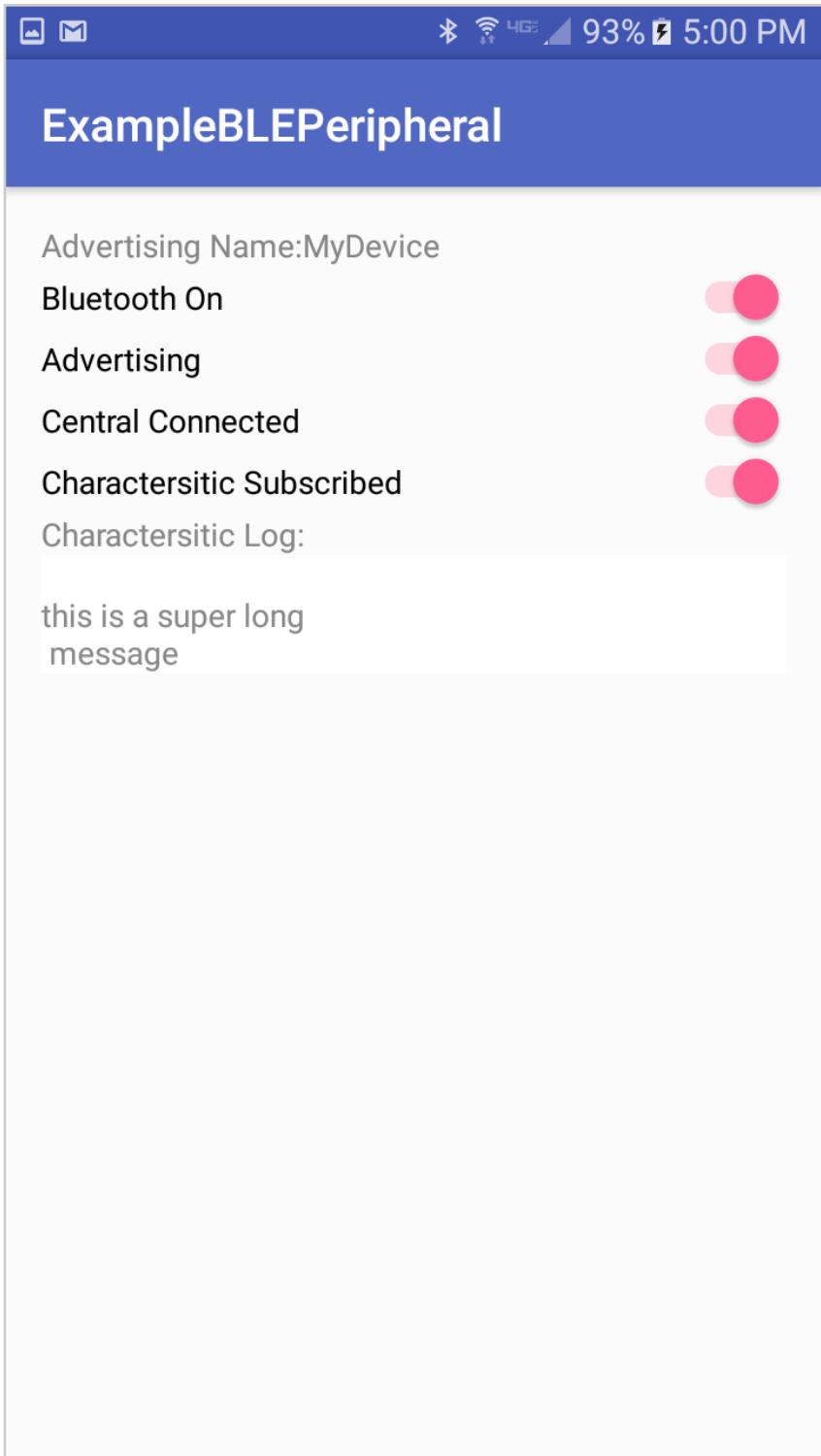
### Example 10-4. java/example.com.example>MainActivity

```
...
/***
 * Event trigger when Characteristic has been written to
 *
 * @param characteristic the Characteristic being written to
 * @param value the byte value being written
 */
public void onBleCharacteristicWritten(
    final BluetoothGattCharacteristic characteristic,
    final byte[] value

```

```
) {  
    mCharacteristicLogTV.append(characteristic.getUuid().toString());  
    // scroll to bottom of TextView  
    final int scrollAmount = mCharacteristicLogTV.getLayout().getLineTop(  
        mCharacteristicLogTV.getLineCount()  
    ) - mCharacteristicLogTV.getHeight();  
    if (scrollAmount > 0) {  
        mCharacteristicLogTV.scrollTo(0, scrollAmount);  
    } else {  
        mCharacteristicLogTV.scrollTo(0, 0);  
    }  
}  
...  
}
```

The resulting app can receive a queued stream of data from a Central by issuing a receive/response flow control on the Characteristic ([Figure 10-4](#)).



**Figure 10-4. App screen showing multipart value queued to be sent to a Characteristic**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter10>

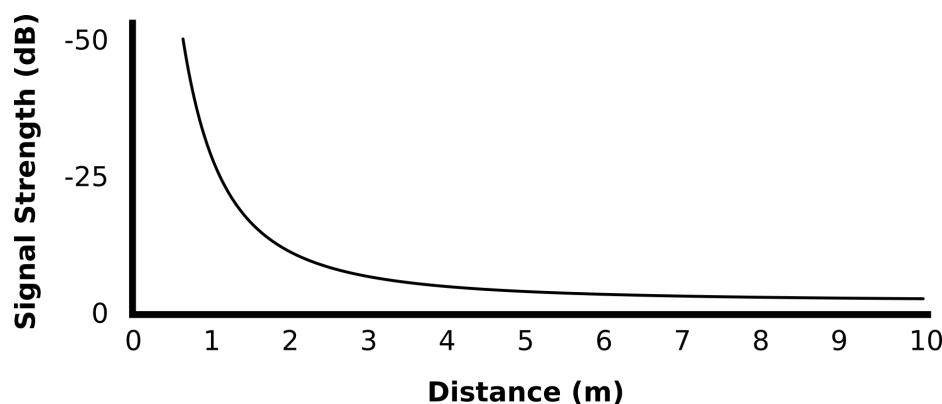
# Project: iBeacon

Beacons can be used for range finding or spacial awareness. iBeacons are a special type of Beacon that is widely supported by the industry. It supports certain data that identifies the iBeacons to makes range finding and spacial awareness easier across platforms.

Due to the nature of how radio signals diminish in intensity with distance, Bluetooth Peripherals can be used both for range finding and spacial awareness.

## Range Finding

Bluetooth signals can be used to approximate the distance between a Peripheral and a Central because the radio signal quality drops off in a predictable way with distance. The diagram below shows how the signal might drop as distance increases ([Figure 11-1](#)).



**Figure 11-1. Distance versus Bluetooth Signal Strength**

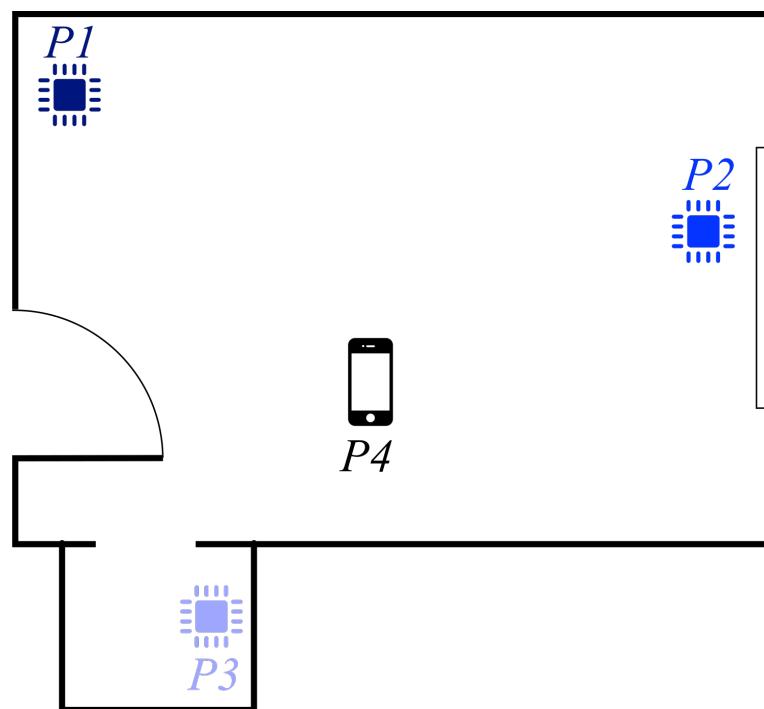
This drop-off rate, known as the Inverse-Square Law, is universal with electromagnetic radiation.

Due to radio interference and absorption from surrounding items, the radio signal propagation varies a lot from environment to environment, and even step to step. This makes it very difficult to know the precise distance between a Central and an iBeacon.

One or more Centrals can approximate their distance from a single iBeacon without connecting.

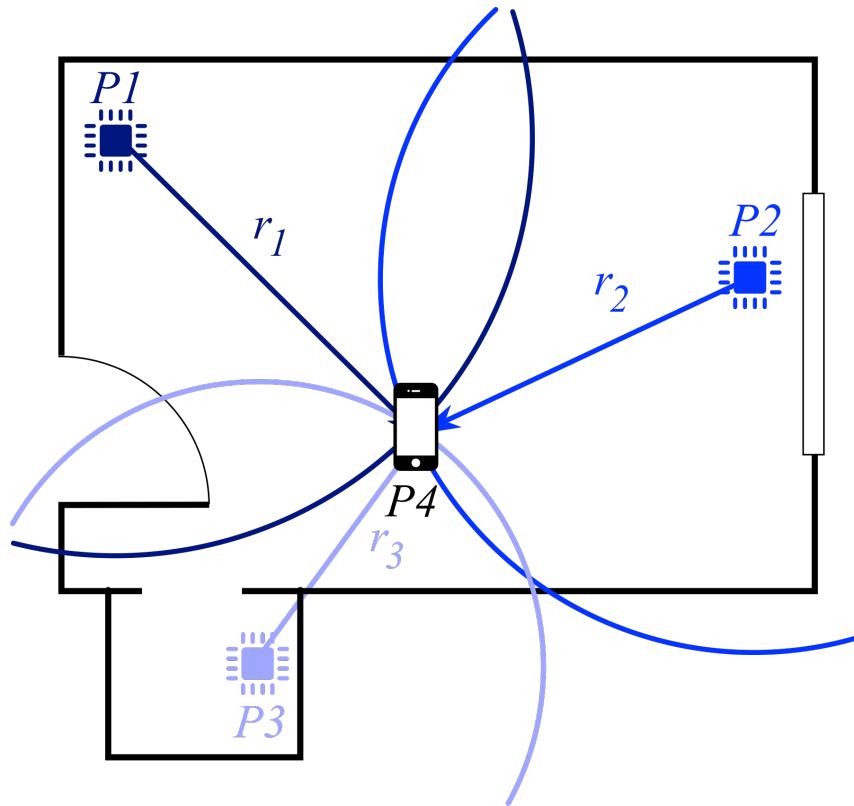
## Spacial Awareness

A Central can approximate its position in space using a process called trilateration. Trilateration works by computing series of equations when both the distance from and location of nearby iBeacons are known ([Figure 11-2](#)).



**Figure 11-2. Example Central and iBeacon positions in a room**

This is a pretty math-intensive process, but it's all based on the Pythagoras Theorem. By calculating the shape of the triangles made from the relative positions of all the iBeacons and the Central, one can determine the location of the Central ([Figure 11-3](#)).



**Figure 11-3. Distances from iBeacons to Central**

## iBeacons

The Scan Result allows a Central to read information from a Peripheral without connecting to it, in much the same way that the advertising name is read.

Although Android supports iBeacon scanning, it does not support iBeacon advertising. Therefore it is possible to build an app that discovers iBeacons but not possible to create an iBeacon in Android.

iBeacons are beacons that advertise information about their location and advertise intensity using the Scan Result feature of Bluetooth Low Energy.

The Scan Result allows a Central to read information from a Peripheral without connecting to it, in much the same way that the Device name is advertised.

There are only two tricks to creating an iBeacon client in Android:

1. iBeacons are identified by their network (MAC) address instead of by name, because all iBeacons for the same location service have the same name.

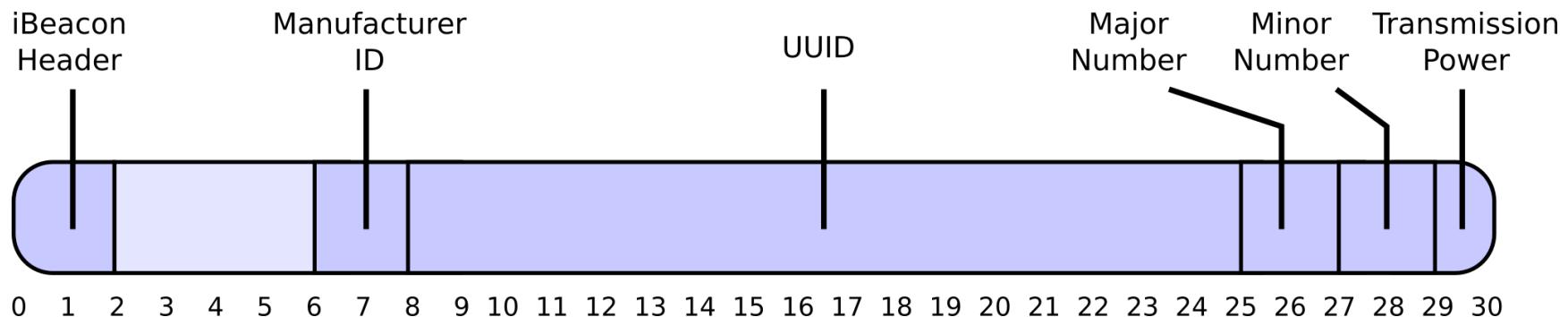
2. The distance to the iBeacon is calculated based on the RSSI.

iBeacons do this by advertising certain data that can be referenced when looking up where the iBeacons are located.

**Table 11-1. iBeacon Advertised data**

Data	Position	Length	Description
iBeacon Header	0	2	Identifies the Peripheral as an iBeacon
Manufacturer ID	6	2	A numeric value that represents the manufacturer
UUID	9	16	All iBeacons in a network have the same UUID
Major	25	2	A top level numeric identifier
Minor	27	2	A numeric identifier under the Major number
Transmission Power	29	1	Tells what the transmission power is in mDb

This data is stored as binary in the Peripheral's advertising data. Unlike other data in Bluetooth Low Energy, numeric values in this packet are stored in big endian format ([Figure 11-4](#)).



**Figure 11-4. iBeacon Advertising Packet**

An example implementation is a museum that has iBeacons at each exhibit in the museum. All iBeacons in the exhibit share the same UUID. The museum uses Major numbers to identify floors and Minor numbers to identify rooms of the exhibit.

The museum's smartphone app has an internal data set relating Major and Minor values to the floors and rooms of the exhibit. It scans for all iBeacons with a specific UUID. The nearby iBeacons are discovered and read. The discovered iBeacons' Major and Minor numbers are looked up to learn that the user is in a specific room on a specific floor in the museum. Relevant content is accessed from the museum's API and loaded in the smartphone app.

## Programming the Central

When a Peripheral is discovered during the scanning process, its Scan Records can be inspected to determine if it is an iBeacon.

```
byte[] IBEACON_HEADER = { 0x02, 0x01 };
byte[] scanRecordHeader = new byte[IBEACON_HEADER.length];
System.arraycopy(scanRecord, 0, scanRecordHeader, 0, 2);

boolean isIBeacon = (Arrays.equals(beaconHeader, scanRecordHeader))
The UUID can be extracted to isolate an iBeacon with a specific UUID.
byte[] uuidBytes = new byte[16];
System.arraycopy(scanRecord, 9, uuidBytes, 0, 16);

ByteBuffer buffer = ByteBuffer.wrap(bytes);
buffer.order(ByteOrder.BIG_ENDIAN);
```

```

UUID uuid = new UUID(buffer.getLong(), buffer.getLong());
Manufacturer ID, Major, and Minor numbers can be isolated also:
// convert big endian bytes to unsigned int
public int bytesToUnsignedInt(byte[] bytes) {
    ByteBuffer buffer = ByteBuffer.wrap(bytes);
    buffer.order(ByteOrder.BIG_ENDIAN);
    int intvalue = (int) buffer.getChar();
    if (intvalue < 0) intvalue = intvalue & 0xffffffff;
    return intvalue;
}

byte[] manufacturerBytes = new byte[2];
System.arraycopy(scanRecord, 6, manufacturerBytes, 0, 2);
int manufacturerId = bytesToUnsignedInt(manufacturerBytes);

byte[] majorBytes = new byte[2];
System.arraycopy(scanRecord, 25, majorBytes, 0, 2);
int majorNumber = bytesToUnsignedInt(majorBytes);

byte[] minorBytes = new byte[2];
System.arraycopy(scanRecord, 27, minorBytes, 0, 2);
int minorNumber = bytesToUnsignedInt(minorBytes);

```

The Transmission Power can also be isolated, useful for determining how far away an iBeacon is.

```

byte[] txPowerBytes = new byte[1];
System.arraycopy(scanRecord, 29, txPowerBytes, 0, 1);
int txPower = (int) txPowerBytes[0];

```

## Calculate Distance From iBeacons

Approximate the distance from a Central to an iBeacon using the RSSI reported at that distance, a reference RSSI at 1 meter, and an approximate propagation constant, using this equation:

$$d \approx 10^{\frac{A-R}{10n}}$$

where d = distance between iBeacon peripheral and central A = RSSI when central and peripheral are 1 meter apart

R = RSSI at distance d

n = The radio propagation constant; typically between 2.7 and 4.3

This is expressed in code as follows:

```
private double getDistanceFromRSSI(
    int rssi, int referenceRssi, float propagationConstant) {
    float exponent = (referenceRSSI - rssi)/(10*propagationConstant);
    double distance = Math.pow(10, exponent);
    return distance;
}
```

Due to radio interference and absorption from surrounding items, the radio propagation constant, n, varies a lot from environment to environment and even step to step. This makes it very difficult to know the exact distance between a Central and an iBeacon. The radio propagation constant can be approximated by testing the iBeacon's RSSI at 1 meter in conditions similar to what is expected in the field.

## Spacial Awareness

Following a few equations derived from Pythagorus' Theorem, it is possible to determine the location of the Central from 3 known iBeacon locations.

## List Known Variables

First of all, it's convenient to store position values as a PrecisePoint object:

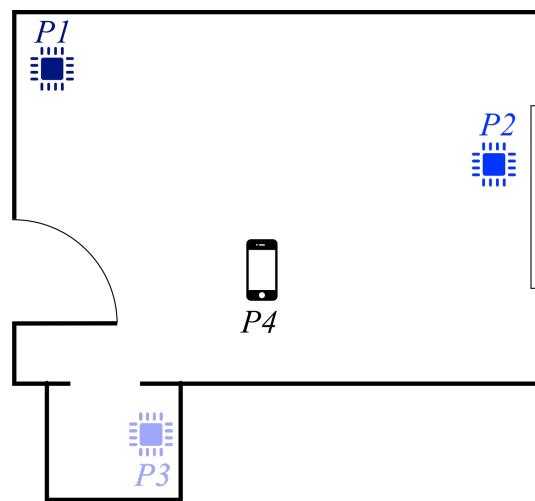
```
public class PrecisePoint {
```

```

double x,y;
public PrecisionPoint(double x, double y) {
    this.x = x;
    this.y = y;
}

```

Imagine a room with known iBeacon's locations, P1, P2, and P3 as x,y coordinates ([Figure 11-5](#)).



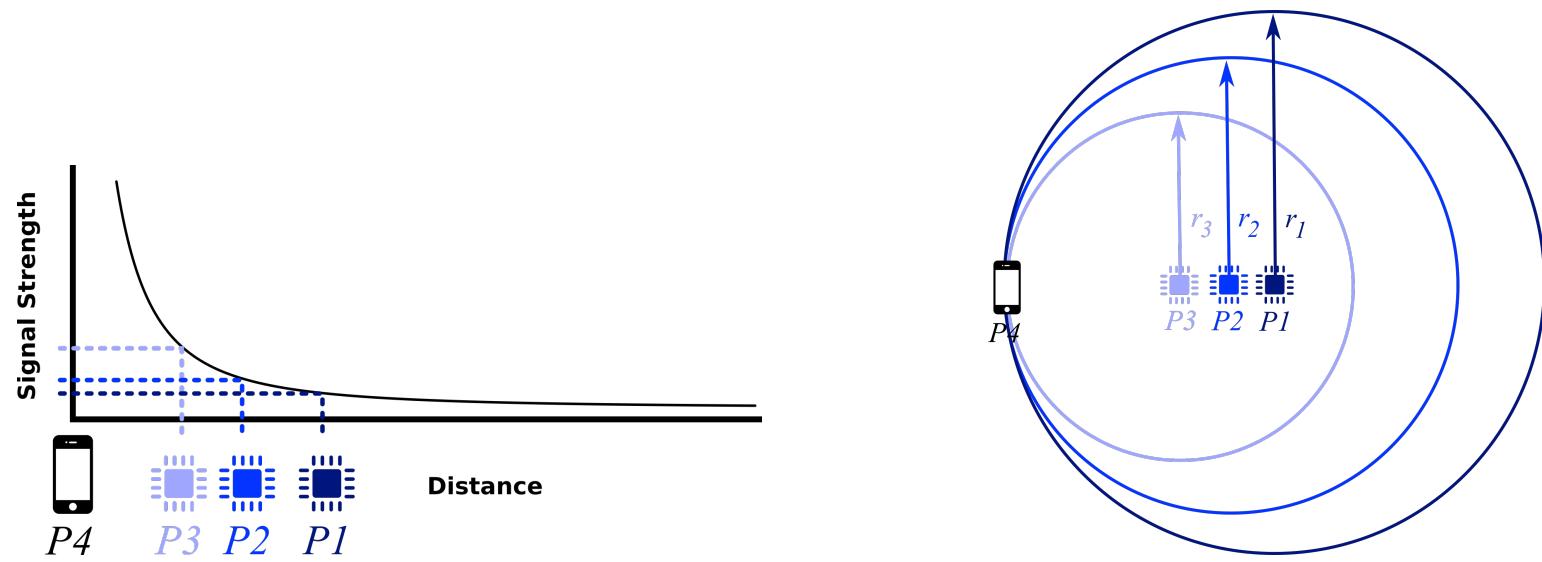
**Figure 11-5. Example iBeacon positions in a room**

```

// three iBeacons with known distances from a fixed spot
PrecisePoint P1 = new PrecisePoint(10, 10);
PrecisePoint P2 = new PrecisePoint(50, 30);
PrecisePoint P3 = new PrecisePoint(35, 50);

```

Using the `getDistanceFromRSSI` method, the distance between each iBeacon and the Central ([Figure 11-6](#)) has been derived as r1, r2, r3 ([Figure 11-7](#)).



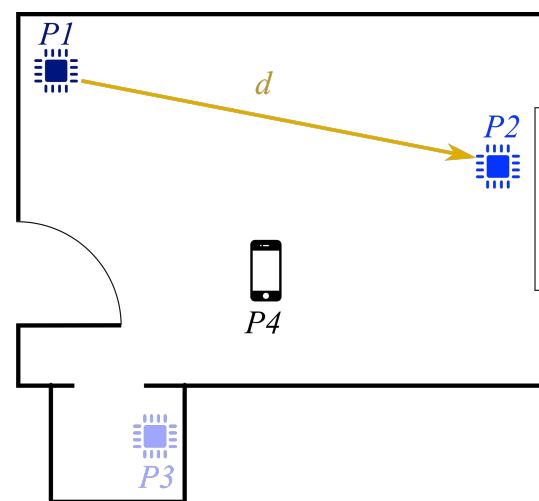
**Figure 11-6. Beacon distances from Central derived from their singal strengths**

**Figure 11-7. Beacon distances from Central**

```
// distance to each of the iBeacons has already been calculated
double r1 = 13.2,
double r2 = 10.8,
double r3 = 7.7;
```

## Calculate Distance Between iBeacons

Calculate the distance between iBeacons P1 and P2 using Pythagorean Theorem ([Figure 11-8](#)).



**Figure 12-8. Derived distance from iBeacon 1 to iBeacon 2**

$$d = \| P2 - P1 \|$$

Which is short-hand for this:

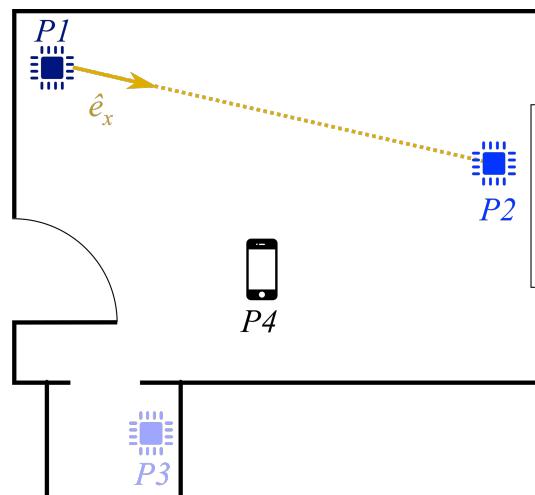
$$d = \sqrt{(P2_x - P1_x)^2 + (P2_y - P1_y)^2}$$

This is expressed in code as follows:

```
double adjascent = p2.getX() - p1.getX();
double opposite = p2.getY() - p1.getY();
double d = Math.sqrt(adjascent * adjascent + opposite * opposite);
```

## Calculate the Unit Vector between P1 and P2

A unit vector represents a direction, but not a distance. Here we calculate the unit vector between P1 and P2, which shows which direction P2 is relative to P1 ([Figure 11-9](#)).



**Figure 11-9. Derived direction from iBeacon 1 to iBeacon 2**

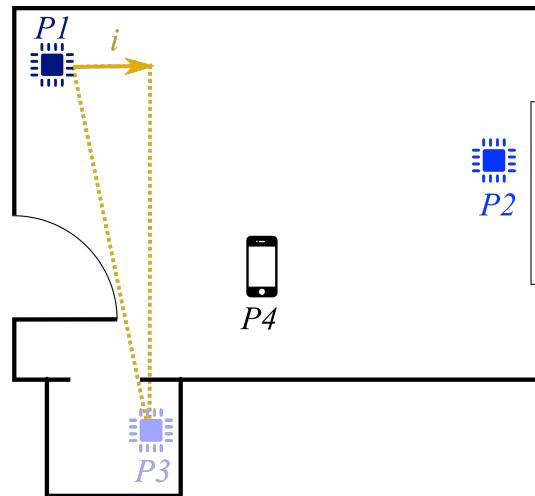
$$\hat{e}_x = \frac{P2 - P1}{\| P2 - P1 \|}$$

$$\hat{e}_x = \left( \frac{P2_x - P1_x}{d}, \frac{P2_y - P1_y}{d} \right)$$

```
double exx = (p2.getX() - p1.getX()) / d;
double exy = (p2.getY() - p1.getY()) / d;
```

## Calculate Magnitude of Distance from P1 to P3

Find the magnitude  $i$  of the distance between P1 and P3 in the x direction ([Figure 11-10](#)).



**Figure 11-10. Derived Horizontal component of distance from iBeacon 1 to iBeacon 2**

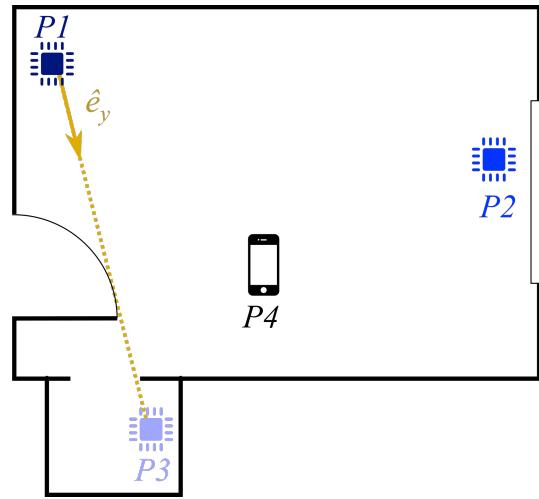
$$i = \hat{e}_x \cdot (P3 - P1)$$

$$i = \hat{e}_{x_x} (P3_x - P1_x) + \hat{e}_{x_y} (P3_y - P1_y)$$

```
double i = exx * (p3.getX() - p1.getX()) + exy *  
    (p3.getY() - p1.getY());
```

## Calculate Unit Vector between P1 and P3

Calculate the unit vector  $\hat{e}_y$  between P1 and P3, showing the direction between P1 and P3 ([Figure 11-11](#)).



**Figure 11-11. Derived direction from iBeacon 1 and iBeacon 3**

$$\hat{e}_y = \frac{P3 - P1 - i\hat{e}_x}{\| P3 - P1 - i\hat{e}_x \|}$$

$$\hat{e}_y = \left( \frac{P3_x - P1_x - i\hat{e}_{x_x}}{\sqrt{(P3_x - P1_x - i\hat{e}_{x_x})^2 + (P3_y - P1_y - i\hat{e}_{x_y})^2}}, \frac{P3_y - P1_y - i\hat{e}_{x_y}}{\sqrt{(P3_x - P1_x - i\hat{e}_{x_x})^2 + (P3_y - P1_y - i\hat{e}_{x_y})^2}} \right)$$

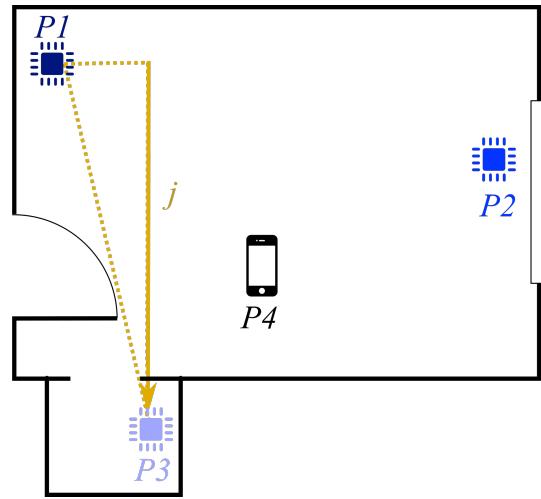
```

double eyx = (p3.getX() - p1.getX() - i * exx) /
    Math.sqrt(
        Math.pow(p3.getX() - p1.getX() - i * exx, 2) +
        Math.pow(p3.getY() - p1.getY() - i * exy, 2)
    );
double eyy = (p3.getY() - p1.getY() - i * exy) /
    Math.sqrt(
        Math.pow(p3.getX() - p1.getX() - i * exx, 2) +
        Math.pow(p3.getY() - p1.getY() - i * exy, 2)
    );

```

## Calculate Magnitude of Distance between P1 and P3

Calculate magnitude  $j$  of the distance between P1 and P3 in the y direction, showing how far apart they are (Figure 11-12).



**Figure 11-12. Derived Vertical component of distance between iBeacon 1 and iBeacon 3**

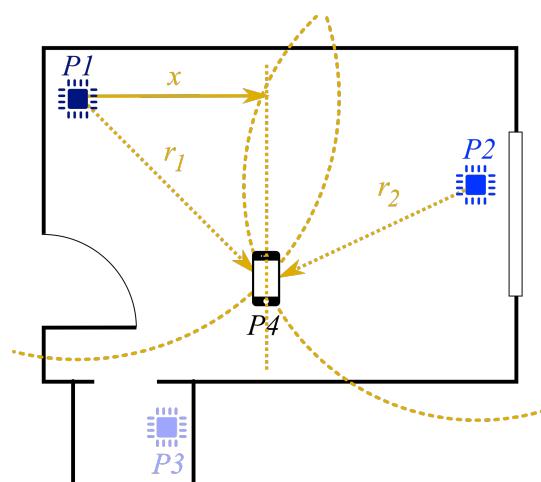
$$j = \hat{e}_y \cdot (P3 - P1)$$

$$j = \hat{e}_{y_x} (P3_x - P1_x) + \hat{e}_{y_y} (P3_y - P1_y)$$

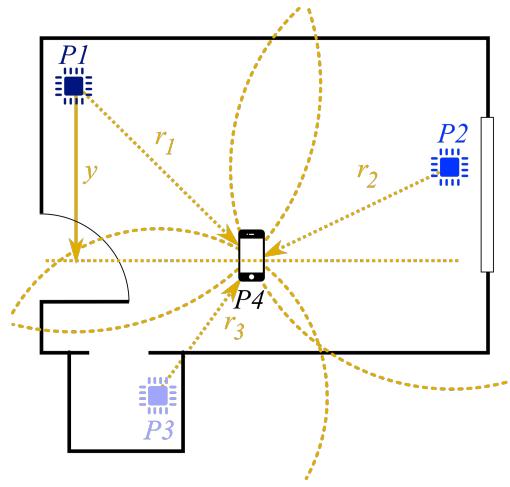
```
double j = eyx * (p2.getX() - p3.getY()) + eyy *
          (p3.getY() - p1.getY());
```

## Find Relative Distances from Central to iBeacons

Use the relative distances from the Central to iBeacons P1, P2, and P3 to find relative distances x and y from each iBeacon (Figure 11-13) (Figure 11-14).



**Figure 11-13. Derived positions of each iBeacon with horizontal position of Central**



**Figure 11-14. Derived position of each iBeacon with vertical position of Central**

This is done by solving for x and y in the following equations:

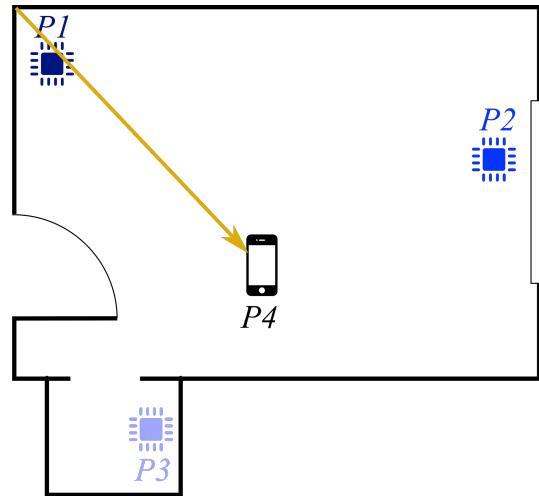
$$x = \frac{r_1^2 - r_2^2 + d^2}{2d}$$

$$y = \frac{r_1^2 - r_3^2 + i^2 + j^2}{2j} - \frac{i}{j}x$$

```
double x = (Math.pow(r1, 2) - Math.pow(r2, 2) + Math.pow(d, 2)) /
    (2 * p2p1Distance);
double y = (Math.pow(r1, 2) - Math.pow(r3, 2) + Math.pow(i, 2) + Math.pow(j, 2)) /
    (2 * j) - i * x / j;
```

## Calculate Absolute Position of Central at P4

Finally, find the position of the Central at P4 by adding the distance between P4 and P1 to the relative position of P1 ([Figure 11-15](#)).



**Figure 11-15. Derived Central position**

$$P4 = P1 + x\hat{e}_x + y\hat{e}_y$$

$$P4 = \left( P1_x + x\hat{e}_{x_x} + y\hat{e}_{y_x}, P1_y + x\hat{e}_{x_y} + y\hat{e}_{y_y} \right)$$

```
double p4x = p1.getX() + x * exx + y*eyx;
double p4y = p1.getY() + x * exy + y*eyy;
```

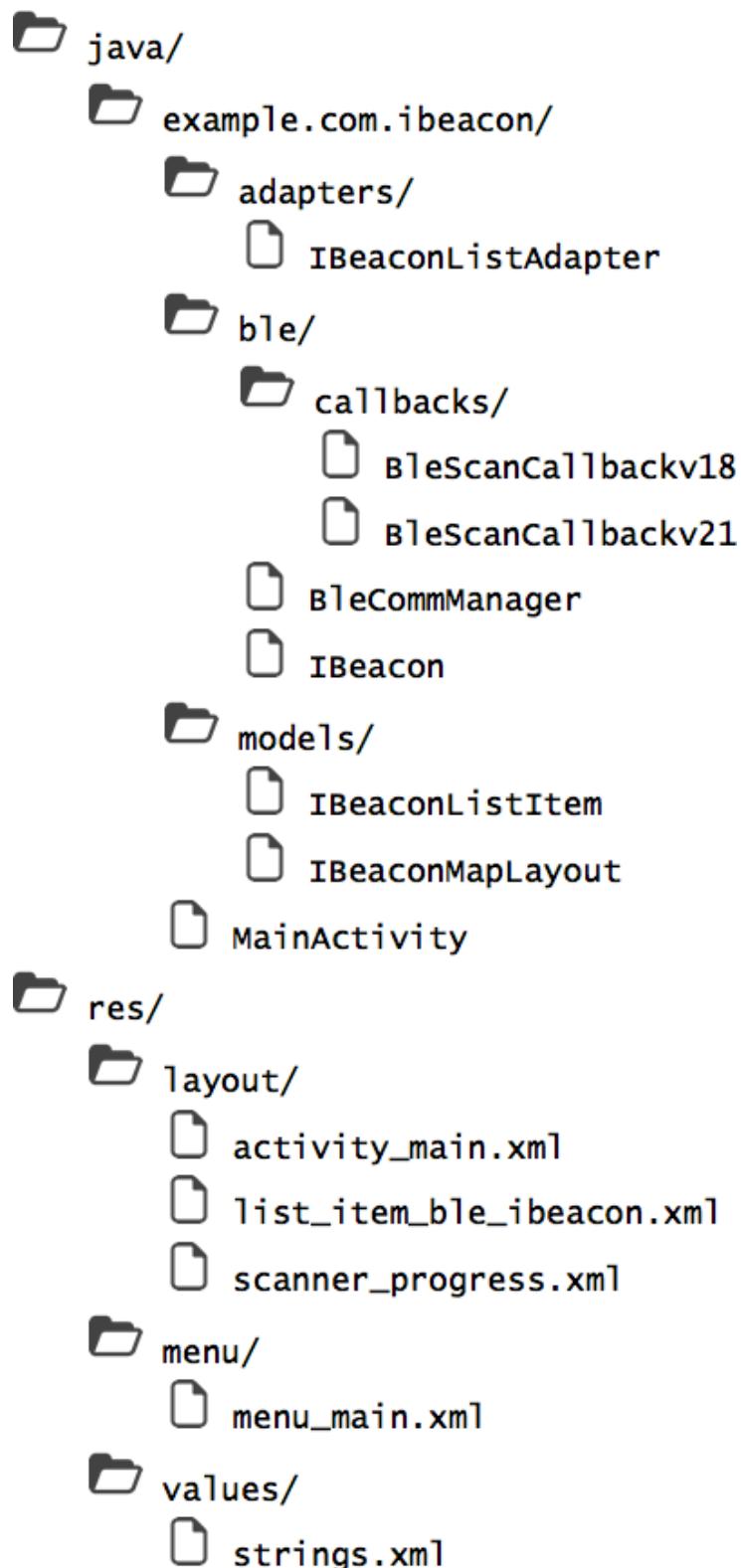
That is how to trilaterate the position of a Central from three known iBeacon positions.

## Putting It All Together

The following code will create a single Activity App that graphs the position of nearby iBeacons, lists their properties, and attempts to find its own location relative to the iBeacons.

Create a new app called ExampleBeaconLocator. Create the following package structure and copy files from the previous examples where they exist.

The final file structure should resemble this ([Figure 11-16](#)).



**Figure 11-16. Project Structure**

## Resources

Create the String resources for buttons and labels.

### Example 11-1. res/values/strings.xml

```
<resources>
  <string name="app_name">Beacon Search</string>
  <string name="action_start_scan">Scan</string>
```

```

<string name="action_stop_scan">Stop</string>
<string name="scanning">Scanning...</string>
<string name="beacon_list_empty">No iBeacons Found</string>
<string name="rssi">RSSI: %1$d</string>
<string name="major_number">Major: %d</string>
<string name="minor_number">Minor: %d</string>
<string name="distance">Distance: %1$s meters</string>
<string name="location">X: %1$s m, Y: %2$s m</string>
<string name="transmission_power">TX Power: %1$d</string>
<string name="central_position">Central at (%1$s m, %2$s m)</string>
</resources>

```

## Objects

The BleCommManager turns the Bluetooth radio on and scans for nearby Peripherals.

### Example 11-2. java/com.example.echoclient/ble/BleCommManager

```

package com.example.echoclient.ble
public class BleCommManager {
    private static final String TAG = BleCommManager.class.getSimpleName();
    // 5 seconds of scanning time
    private static final long SCAN_PERIOD = 5000;
    // Andrdoid's Bluetooth Adapter
    private BluetoothAdapter mBluetoothAdapter;
    // Ble scanner - API >= 21
    private BluetoothLeScanner mBluetoothLeScanner;
    // scan timer
    private Timer mTimer = new Timer();

    /**
     * Initialize the BleCommManager
     *
     * @param context the Activity context
     * @throws Exception Bluetooth Low Energy is

```

```

*      not supported on this Android device
*/
public BleCommManager(final Context context) throws Exception {
    // make sure Android device supports Bluetooth Low Energy
    if (!context.getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_BLUETOOTH_LE)
    ) {
        throw new Exception("Bluetooth Not Supported");
    }
    // get a reference to the Bluetooth Manager class,
    // which allows us to talk to talk to the BLE radio
    final BluetoothManager bluetoothManager = (BluetoothManager) \
        context.getSystemService(Context.BLUETOOTH_SERVICE);
    mBluetoothAdapter = bluetoothManager.getAdapter();
}

/**
 * Get the Android Bluetooth Adapter
 *
 * @return BluetoothAdapter Android Bluetooth Adapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}

/**
 * Scan for Peripherals
 *
 * @param bleScanCallbackv18 APIv18 compatible ScanCallback
 * @param bleScanCallbackv21 APIv21 compatible ScanCallback
 * @throws Exception
 */
public void scanForPeripherals(
    final BleScanCallbackv18 bleScanCallbackv18,
    final BleScanCallbackv21 bleScanCallbackv21) throws Exception
{

```

```

// Don't proceed if there is already a scan in progress
mTimer.cancel();

// Use BluetoothAdapter.startLeScan() for Android API 18, 19, and 20
if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
    // Scan for SCAN_PERIOD milliseconds.
    // at the end of that time, stop the scan.
    new Thread() {
        @Override
        public void run() {
            mBluetoothAdapter.startLeScan(bleScanCallbackv18);
            try {
                Thread.sleep(SCAN_PERIOD);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            mBluetoothAdapter.stopLeScan(bleScanCallbackv18);
        }
    }.start();
    // alert the system that BLE scanning
    // has stopped after SCAN_PERIOD milliseconds
    mTimer = new Timer();
    mTimer.schedule(new TimerTask() {
        @Override
        public void run() {
            stopScanning(bleScanCallbackv18, bleScanCallbackv21);
        }
    }, SCAN_PERIOD);
} else { // use BluetoothLeScanner.startScan() for API >= 21 (Lollipop)
    final ScanSettings settings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
        .build();
    final List<ScanFilter> filters = new ArrayList<ScanFilter>();
    mBluetoothLeScanner = mBluetoothAdapter.getBluetoothLeScanner();
    new Thread() {
        @Override
        public void run() {

```

```

        mBluetoothLeScanner.startScan(
            filters,
            settings,
            bleScanCallbackv21
        );
        try {
            Thread.sleep(SCAN_PERIOD);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        mBluetoothLeScanner.stopScan(bleScanCallbackv21);
    }
}.start();
// alert the system that BLE scanning
// has stopped after SCAN_PERIOD milliseconds
mTimer = new Timer();
mTimer.schedule(new TimerTask() {
    @Override
    public void run() {
        stopScanning(bleScanCallbackv18, bleScanCallbackv21);
    }
}, SCAN_PERIOD);
}

}

/***
 * Stop Scanning
 *
 * @param bleScanCallbackv18 APIv18 compatible ScanCallback
 * @param bleScanCallbackv21 APIv21 compatible ScanCallback
 */
public void stopScanning(
    final BleScanCallbackv18 bleScanCallbackv18,
    final BleScanCallbackv21 bleScanCallbackv21
) {
    mTimer.cancel();
}

```

```

    // propagate the onScanComplete through the system
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
        mBluetoothAdapter.stopLeScan(bleScanCallbackv18);
        bleScanCallbackv18.onScanComplete();
    } else {
        mBluetoothLeScanner.stopScan(bleScanCallbackv21);
        bleScanCallbackv21.onScanComplete();
    }
}
}

```

BleScanCallbackv18 is one of two callback handlers for BleCommManager to process Peripheral discovered during as scan. Which callback class used depends on the API version of the phone the App gets installed on.

### **Example 11-3. java/com.example.echoclient/ble/callbacks/BleScanCallbackv18**

```

package com.example.echoclient.ble.callbacks

public class BleScanCallbackv18 implements BluetoothAdapter.LeScanCallback {

    /**
     * New Perpheral found.
     *
     * @param bluetoothDevice The Peripheral Device
     * @param rssi The Peripheral's RSSI indicating
     *             how strong the radio signal is
     * @param scanRecord Other information about the scan result
     */
    //@Override
    public abstract void onLeScan(
        final BluetoothDevice bluetoothDevice,
        int rssi,
        byte[] scanRecord
    );

    /**
     * BLE Scan complete
     */
}

```

```
 */
public abstract void onScanComplete();
}
```

BleScanCallbackv21 handles Peripheral discovery and Bluetooth radio state changes in newer Android devices.

#### Example 11-4. java/com.example.echoclient/ble/callbacks/BleScanCallbackv21

```
package com.example.echoclient.ble.callbacks

public class BleScanCallbackv21 extends ScanCallback {

    /**
     * New Perpheral found.
     *
     * @param callbackType int: Determines how this callback was triggered.
     *                     Could be one of CALLBACK_TYPE_ALL_MATCHES,
     *                     CALLBACK_TYPE_FIRST_MATCH or CALLBACK_TYPE_MATCH_LOST
     * @param result a Bluetooth Low Energy Scan Result, containing
     *              the Bluetooth Device, RSSI, and other information
     */
    @Override
    public abstract void onScanResult(int callbackType, ScanResult result);

    /**
     * New Perpherals found.
     *
     * @param results List: List of scan results that are previously scanned.
     */
    @Override
    public abstract void onBatchScanResults(List<ScanResult> results);

    /**
     * Problem initializing the scan. See the error code for reason
     *
     * @param errorCode      int: Error code (one of SCAN_FAILED_*)
     *                      for scan failure.
     */
```

```

    */
    @Override
    public abstract void onScanFailed(int errorCode);

    /**
     * Scan has completed
     */
    public abstract void onScanComplete();
}

```

Create an object called IBeacon, responsible for extracting iBeacon information from ScanData

### **Example 11-5. java/example.com.beacon.ble/IBeacon.java**

```

package example.com.beacon.ble;

public class IBeacon {
    /** Constants */
    private static String TAG = IBeacon.class.getSimpleName();
    public static final double RADIO_PROPAGATION_CONSTANT = 3.5;

    // distances
    public static final int DISTANCE_UNKNOWN = 0;
    public static final int DISTANCE_IMMEDIATE = 1;
    public static final int DISTANCE_NEAR = 3;
    public static final int DISTANCE_FAR = 10;

    private static final int RANGE_UNKNOWN = 0;
    private static final int RANGE_IMMEDIATE = 1;
    private static final int RANGE_NEAR = 3;

    /** iBeacon GAP Header */
    private static final byte[] IBEACON_HEADER = { 0x02, 0x01 };
    private static final int IBEACON_HEADER_POSITION = 0;
    private static final int MANUFACTURER_POSITION = 6;
    private static final int MANUFACTURER_LENGTH = 2;
}

```

```

private static final int UUID_POSITION = 9;
private static final int UUID_LENGTH = 16;
private static final int MAJOR_NUMBER_POSITION = 25;
private static final int MAJOR_NUMBER_LENGTH = 2;
private static final int MINOR_NUMBER_POSITION = 27;
private static final int MINOR_NUMBER_LENGTH = 2;
private static final int TX_POWER_POSITION = 29;
private static final int TX_POWER_LENGTH = 1;

/** iBeacon Properties */
private UUID mUuid;
private int mManufacturerId;
private int mMajor;
private int mMinor;
private int mTransmissionPower;
private int mRssi;
private String mMacAddress;
private double mX = 0;
private double mY = 0;

/** Get and set iBeacon Properties */
public void setUuid(UUID uuid) { mUuid = uuid; }
public void setMajor(int major) { mMajor = major; }
public void setMinor(int minor) { mMinor = minor; }
public void setTransmissionPower(int transmissionPower) {
    mTransmissionPower = transmissionPower;
}
public void setRssi(int rssi) { mRssi = rssi; }
public void setMacAddress(String macAddress) { mMacAddress = macAddress; }
public void setManufacturerId(int manufacturerId) {
    mManufacturerId = manufacturerId;
}
public UUID getUuid() { return mUuid; }
public int getMajor() { return mMajor; }
public int getMinor() { return mMinor; }
public int getTransmissionPower() { return mTransmissionPower; }

```

```
public int getRSSI() { return mRSSI; }

public String getMacAddress() { return mMACAddress; }

public int getManufacturerID() { return mManufacturerID; }

/***
 * Set the X and Y location of the BleBeacon
 *
 * @param x x location
 * @param y y location
 */
public void setLocation(double x, double y) {
    setXLocation(x);
    setYLocation(y);
}

/***
 * Set the x location of the BleBeacon
 *
 * @param x x location
 */
public void setXLocation(double x) {
    mX = x;
}

/***
 * Get the x location
 *
 * @return x location
 */
public double getXLocation() {
    return mX;
}

/***
 * Set the Y location of the BleBeacon
 *
```

```

* @param y y location
*/
public void setYLocation(double y) {
    mY = y;
}

/**
 * Get the Y location
 *
 * @return y location
*/
public double getYLocation() {
    return mY;
}

/**
 * Test if two iBeacons are the same
 *
 * @param otherBeacon another iBeacon
 * @return <strong>true</strong> if two beacons are the same
*/
public boolean equals(IBeacon otherBeacon) {
    // iBeacons are the same if they have the same
    // UUID, Major, Minor, and Manufacturer ID
    boolean isSameBeacon = (
        (mUuid.equals(otherBeacon.getUuid())) && \
        (mMajor == otherBeacon.getMajor()) && \
        (mMinor == otherBeacon.mMinor) && \
        (mManufacturerId == otherBeacon.getManufacturerId())
    );
    Log.v(TAG, "beacons same?: "+isSameBeacon);
    return isSameBeacon;
}

/**
 * Determine if a discovered Peripheral's GAP belongs to an iBeacon,

```

```

* based on its GAP header fingerprint
*
* @param scanRecord the byte array of the Peripheral's GAP Scan Record
* @return <strong>true</strong> if Scan Record belongs to an iBeacon
*/
static public boolean isIBeacon(final byte[] scanRecord) {
    byte[] scanRecordHeader = new byte[IBEACON_HEADER.length];
    System.arraycopy(
        scanRecord,
        IBEACON_HEADER_POSITION,
        scanRecordHeader,
        0,
        IBEACON_HEADER.length
    );
    if (Arrays.equals(
        IBEACON_HEADER, scanRecordHeader)
    ) {
        return true;
    }
    return false;
}

/**
 * Create an iBeacon from a GAP Scan Record
*
* Scan record header looks like this:
*
* Header:          0-1 (Little Endian)
* MAC Address:    2-8
* iBeacon Prefix: 9-19
* Proximity UUID: 20-36
* Major:           38-39 (Big Endian)
* Minor:           40-41 (Big Endian)
* TX Power:        42-43 (Two's complement negative)
*
* example: 4C00 02 15 B9407F30F5F8466EAFF925556B57FE6D ED4E 8931 B6

```

```

/*
 * @param scanRecord an incoming Scan Record
 * @return IBeacon
 */
static public IBeacon fromScanRecord(byte[] scanRecord) throws Exception {
    if (!isIBeacon(scanRecord)) {
        throw new Exception("Scan Record does not represent an iBeacon");
    }
    IBeacon iBeacon = new IBeacon();
    iBeacon.setUuid(getUuidFromScanRecord(scanRecord));
    iBeacon.setManufacturerId(getManufacturerIdFromScanRecord(scanRecord));
    iBeacon.setMajor(getMajorNumberFromScanRecord(scanRecord));
    iBeacon.setMinor(getMinorNumberFromScanRecord(scanRecord));
    iBeacon.setTransmissionPower(
        getTransmissionPowerFromScanRecord(scanRecord)
    );
    Log.v(
        TAG,
        "txPower: " + iBeacon.getTransmissionPower() + \
        ", major: " + iBeacon.getMajor() + \
        ", minor: " + iBeacon.getMinor() + \
        ", uuid: " + iBeacon.getUuid().toString()
    );
    return iBeacon;
}

/**
 * Get iBeacon's UUID from an iBeacon's Scan Record
 *
 * @param scanRecord iBeacon Scan Record
 * @return UUID
 */
static public UUID getUuidFromScanRecord(final byte[] scanRecord) {
    byte[] uuidBytes = new byte[16];
    System.arraycopy(scanRecord, UUID_POSITION, uuidBytes, 0, UUID_LENGTH);
    return DataConverter.bytesToUuid(uuidBytes);
}

```

```

}

/***
 * Get iBeacon's UUID from an iBeacon's Scan Record
 *
 * @param scanRecord iBeacon Scan Record
 * @return Minor number
 */
public static int getMinorNumberFromScanRecord(final byte[] scanRecord) {
    byte[] minorBytes = new byte[2];
    System.arraycopy(
        scanRecord,
        MINOR_NUMBER_POSITION,
        minorBytes,
        0,
        MINOR_NUMBER_LENGTH
    );
    return DataConverter.bytesToUnsignedInt(minorBytes);
}

/***
 * Get iBeacon's Major number from an iBeacon's Scan Record
 *
 * @param scanRecord iBeacon Scan Record
 * @return Major number
 */
public static int getMajorNumberFromScanRecord(final byte[] scanRecord) {
    byte[] majorBytes = new byte[2];
    System.arraycopy(
        scanRecord,
        MAJOR_NUMBER_POSITION,
        majorBytes,
        0,
        MAJOR_NUMBER_LENGTH
    );
    return DataConverter.bytesToUnsignedInt(majorBytes);
}

```

```

}

/**
 * Get iBeacon's transmission power from an iBeacon's Scan Record
 *
 * @param scanRecord iBeacon Scan Record
 * @return transmission power in decibels
 */
public static int getTransmissionPowerFromScanRecord(
    final byte[] scanRecord
) {
    byte[] txPowerBytes = new byte[1];
    System.arraycopy(
        scanRecord,
        TX_POWER_POSITION,
        txPowerBytes,
        0,
        TX_POWER_LENGTH
    );
    return DataConverter.bytesToSignedInt(txPowerBytes);
}

/**
 * Get iBeacon's Manufacturer ID from an iBeacon's Scan Record
 *
 * @param scanRecord iBeacon Scan Record
 * @return Manufacturer ID number
 */
public static int getManufacturerIdFromScanRecord(final byte[] scanRecord)
{
    byte[] manufacturerBytes = new byte[MANUFACTURER_LENGTH];
    System.arraycopy(
        scanRecord,
        MANUFACTURER_POSITION,
        manufacturerBytes,
        0,

```

```

        MANUFACTURER_LENGTH
    );
    return DataConverter.bytesToUnsignedInt(manufacturerBytes);
}

/***
 * Get iBeacon's distance from central based on an RSSI
 *
 * @return Minor number
 */
public double getDistance() {
    if (mRssi == 0) {
        return -1.0; // if we cannot determine accuracy, return -1.
    }
    double ratio = (mTransmissionPower - mRssi) / \
        (10 * RADIO_PROPAGATION_CONSTANT);
    double distance = Math.pow(10, ratio);
    return distance;
}

/***
 * Get iBeacon's proximity range from a Central
 *
 * @return One of DISTANCE_UNKNOWN, DISTANCE_IMMEDIATE,
 *         DISTANCE_NEAR, or DISTANCE_FAR
 */
public int getProximity() {
    double distance = getDistance();
    if (distance <= RANGE_UNKNOWN) {
        return DISTANCE_UNKNOWN;
    }
    if (distance < RANGE_IMMEDIATE) {
        return DISTANCE_IMMEDIATE;
    }
    if (distance < RANGE_NEAR) {

```

```

        return DISTANCE_NEAR;
    }

    return DISTANCE_FAR;
}

}

```

Create an iBeaconListAdapter that describes how to display the iBeacons in a ListView

### **Example 11-6. java/example.com.exampleble/adapters/iBeaconListAdapter.java**

```

package example.com.beacon.adapters;

public class IBeaconsListAdapter extends BaseAdapter {
    private static String TAG = IBeaconsListAdapter.class.getSimpleName();
    // list of Peripherals
    private ArrayList<IBeaconListItem> mBeaconListItems = \
        new ArrayList<IBeaconListItem>();

    /**
     * How many items are in the ListView
     * @return the number of items in this ListView
     */
    @Override
    public int getCount() {
        return mBeaconListItems.size();
    }

    /**
     * Add a new Peripheral to the ListView
     *
     * @param iBeacon iBeacon device information
     * @param rssi Periheral's RSSI, indicating its radio signal quality
     */
    public void addIBeacon(IBeacon iBeacon) {
        // update UI stuff
        int listItemIndex = mBeaconListItems.size();

```

```

    IBeaconListItem listItem = new IBeaconListItem(iBeacon);
    listItem.setItemId(listItemId);
    listItem.setRssi(iBeacon.getRssi());
    // add to list
    mBeaconListItems.add(listItem);
}

/**
 * Get current state of ListView
 * @return ArrayList of BlePeripheralListItems
 */
public ArrayList<IBeaconListItem> getItems() {
    return mBeaconListItems;
}

/**
 * Clear all items from the ListView
 */
public void clear() {
    mBeaconListItems.clear();
}

/**
 * Get the IBeaconListItem held at some position in the ListView
 *
 * @param position the position of a desired item in the list
 * @return the IBeaconListItem at some position
 */
@Override
public IBeaconListItem getItem(int position) {
    return mBeaconListItems.get(position);
}

@Override
public long getItemId(int position) {
    return mBeaconListItems.get(position).getItemId();
}

```

```

}

/**
 * This viewHolder represents
 * what UI components are in each List Item in the ListView
 */
public static class viewHolder{
    public TextView mUuidTV;
    public TextView mRssiTV;
    public TextView mTransmissionPowerTV;
    public TextView mLocationTV;
    public TextView mDistanceTV;
    public TextView mMajorTV;
    public TextView mMinorTV;
}

/**
 * Generate a new ListItem for some known position in the ListView
 *
 * @param position the position of the ListItem
 * @param convertView An existing List Item
 * @param parent The Parent ViewGroup
 * @return The List Item
 */
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    View v = convertView;
    viewHolder peripheralListItemView;
    Context context = parent.getContext();
    Resources resources = context.getResources();
    // if this ListItem does not exist yet, generate it
    // otherwise, use it
    if(convertview == null) {
        // convert list_item_ibeacon.xml to view
        LayoutInflater inflater = LayoutInflater.from(context);
        v = inflater.inflate(R.layout.list_item_ibeacon, null);
    }
}

```

```

// match the UI stuff in the list item to what's in the xml file
peripheralListItemView = new ViewHolder();
peripheralListItemView.mUuidTV = (TextView) \
    v.findViewById(R.id.uuid);
peripheralListItemView.mRssiTV = (TextView) \
    v.findViewById(R.id.rssi);
peripheralListItemView.mTransmissionPowerTV = (TextView) \
    v.findViewById(R.id.transmission_power);
peripheralListItemView.mMajorTV = (TextView) \
    v.findViewById(R.id.major_number);
peripheralListItemView.mMinorTV = (TextView) \
    v.findViewById(R.id.minor_number);
peripheralListItemView.mDistanceTV = (TextView) \
    v.findViewById(R.id.distance);
peripheralListItemView.mLocationTV = (TextView) \
    v.findViewById(R.id.location);
v.setTag( peripheralListItemView );
} else {
    peripheralListItemView = (ViewHolder) v.getTag();
}
Log.v(TAG, "ListItem size: " + mBeaconListItems.size());
// if there are known Peripherals, create a ListItem that says so
// otherwise, display a ListItem with Bluetooth Peripheral information
if (mBeaconListItems.size() <= 0) {
    peripheralListItemView.mUuidTV.setText(R.string.beacon_list_empty);
} else {
    IBeaconListItem item = mBeaconListItems.get(position);
    peripheralListItemView.mUuidTV.setText(item.getUuid().toString());
    peripheralListItemView.mMajorTV.setText(
        String.format(
            resources.getString(R.string.major_number),
            item.getMajor()
        )
    );
    peripheralListItemView.mMinorTV.setText(
        String.format(

```

```
        resources.getString(R.string.minor_number),
        item.getMinor()
    )
);

peripheralListItemView.mRssiTV.setText(
    String.format(
        resources.getString(R.string.rssi),
        item.getRssi()
    )
);

String distance_m = "";
try {
    distance_m = String.format(
        resources.getString(R.string.distance),
        String.format("%.1f", item.getDistance())
    );
} catch (Exception e) {
    Log.d(TAG, "Could not convert distance to string");
}

peripheralListItemView.mDistanceTV.setText(distance_m);

String location = "";
try {
    String xLocation = String.format("%.1f", item.getXLocation());
    String yLocation = String.format("%.1f", item.getYLocation());
    location = String.format(
        resources.getString(R.string.location),
        xLocation,
        yLocation
    );
} catch (Exception e) {
    Log.d(TAG, "Could not convert location to string");
}

peripheralListItemView.mLocationTV.setText(location);

String transmissionPower = "";
```

```

        try {
            transmissionPower = String.format(
                resources.getString(R.string.transmission_power),
                item.getTransmissionPower()
            );
        } catch (Exception e) {
            Log.d(TAG, "Could not convert reference rssi to string");
        }
        peripheralListItemView.mTransmissionPowerTV.setText(
            transmissionPower
        );
    }
    return v;
}
}

```

Create an iBeaconLocator, which trilaterates a position from three known positions:

#### **Example 11-7. java/example.com.exampleble/views/BleBeaconListAdapter.java**

```

package example.com.beacon.utilities
public class IBeaconLocator {

    public static double[] trilaterate(
        ArrayList<IBeacon> beaconList) throws Exception
    {
        if (beaconList.size() < 3) {
            throw new Exception("Not enough points to perform a triangulation");
        }
        IBeacon p1 = beaconList.get(0);
        IBeacon p2 = beaconList.get(1);
        IBeacon p3 = beaconList.get(2);
        double r1 = beaconList.get(0).getDistance();
        double r2 = beaconList.get(1).getDistance();
        double r3 = beaconList.get(2).getDistance();
        //unit vector in a direction from point1 to point 2
    }
}

```

```

double p2p1Distance = Math.sqrt(
    Math.pow(p2.getXLocation() - p1.getXLocation(), 2) +
    Math.pow(p2.getYLocation() - p1.getYLocation(), 2)
);

double exx = (p2.getXLocation() - p1.getXLocation()) / p2p1Distance;
double exy = (p2.getYLocation() - p1.getYLocation()) / p2p1Distance;
//signed magnitude of the x component
double i = exx * (
    p3.getXLocation() - p1.getXLocation() + exy * \
    (p3.getYLocation() - p1.getYLocation())
);
//the unit vector in the y direction.
double eyx = (p3.getXLocation() - p1.getXLocation() - i * exx) / \
Math.sqrt(
    Math.pow(p3.getXLocation() - p1.getXLocation() - i * exx, 2) +
    Math.pow(p3.getYLocation() - p1.getYLocation() - i * exy, 2)
);
double eyy = (p3.getYLocation() - p1.getYLocation() - i * exy) / \
Math.sqrt(
    Math.pow(p3.getXLocation() - p1.getXLocation() - i * exx, 2) +
    Math.pow(p3.getYLocation() - p1.getYLocation() - i * exy, 2)
);
//the signed magnitude of the y component
double j = eyx * (p2.getXLocation() - p3.getYLocation()) + \
eyy * (p3.getYLocation() - p1.getYLocation());
//coordinates
double x = (Math.pow(r1, 2) - Math.pow(r2, 2) + \
Math.pow(p2p1Distance, 2)) / (2 * p2p1Distance);
double y = (Math.pow(r1, 2) - Math.pow(r3, 2) + Math.pow(i, 2) + \
Math.pow(j, 2)) / (2 * j) - i * x / j;
//result coordinates
double finalX = p1.getXLocation() + x * exx + y * eyx;
double finalY = p1.getYLocation() + x * exy + y * eyy;
return new double[] {finalX, finalY};
}
}

```

Create an DataConverter, which converts between data formats:

**Example 11-8. java/example.com.exampleble/utilities/DataConverter.java**

```
package example.com.beacon.utilities

public class DataConverter {

    private static final String TAG = DataConverter.class.getSimpleName();

    /**
     * Convert bytes to a hexadecimal String
     *
     * @param bytes a byte array
     * @return hexadecimal string
     */
    final protected static char[] hexArray = "0123456789ABCDEF".toCharArray();

    public static String bytesToHex(byte[] bytes) {
        char[] hexChars = new char[bytes.length * 3];
        for (int j = 0; j < bytes.length; j++) {
            int v = bytes[j] & 0xFF;
            hexChars[j * 3] = hexArray[v >>> 4];
            hexChars[j * 3 + 1] = hexArray[v & 0x0F];
            hexChars[j * 3 + 2] = 0x20; // space
        }
        return new String(hexChars);
    }

    /**
     * Convert hex String to a byte array
     *
     * @param hexString a String representation of hexadecimal
     * @return byte array
     */
    public static byte[] hexToBytes(String hexString) {
        int len = hexString.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
```

```

        data[i / 2] = (byte) ((Character.digit(
            hexString.charAt(i),
            16
        ) << 4) + Character.digit(hexString.charAt(i+1), 16));
    }

    return data;
}

/***
 * Convert uint16_t byte array into double
 *
 * @param bytes a 2-byte byte array
 * @return double value of bytes
 */
public static double bytesToDouble(byte[] bytes) {
    if (bytes.length < 2) {
        return 0;
    }

    ByteBuffer buffer = ByteBuffer.wrap(bytes);
    buffer.order(ByteOrder.BIG_ENDIAN);
    return (double) buffer.getChar();
}

/***
 * Convert uint16_t byte array into a signed integer
 *
 * @param bytes a 1-byte byte array
 * @return signed integer value of bytes
 */
public static int bytesToSignedInt(byte[] bytes) {
    if (bytes.length < 1) {
        return 0;
    }

    return (int) bytes[0];
}

```

```
/**  
 * Convert uint16_t byte array into an unsigned integer  
 *  
 * @param bytes a 2-byte byte array  
 * @return unsigned int value of bytes  
 */  
  
public static int bytesToUnsignedInt(byte[] bytes) {  
    if (bytes.length < 2) {  
        return 0;  
    }  
    ByteBuffer buffer = ByteBuffer.wrap(bytes);  
    buffer.order(ByteOrder.BIG_ENDIAN);  
  
    int intvalue = (int) buffer.getChar();  
    if (intvalue < 0) intvalue = intvalue & 0xffffffff;  
    return intvalue;  
}  
  
/**  
 * Convert uint16_t byte array into UUID  
 *  
 * @param bytes a 2-byte byte array  
 * @return a UUID  
 */  
  
public static UUID bytesToUuid(byte[] bytes) {  
    ByteBuffer buffer = ByteBuffer.wrap(bytes);  
    buffer.order(ByteOrder.BIG_ENDIAN);  
    UUID uuid = new UUID(buffer.getLong(), buffer.getLong());  
    return uuid;  
}  
}
```

## Models

Create an iBeaconMapLayout class that displays iBeacon and Central positions on the screen.

### Example 11-9. `java/example.com.exampleble/models/BeaconMapLayout.java`

```
package example.com.beacon.models;

public class IBeaconMapLayout extends LinearLayout {
    private static final String TAG = IBeaconMapLayout.class.getSimpleName();

    /** Graphic properties */
    private static final int BITMAP_WIDTH = 1000;
    private static final int BITMAP_HEIGHT = 800;
    private static final String PAINT_COLOR = "#CD5C5C";
    private static final int STROKE_COLOR = 5;
    private static final int X_OFFSET = 50;
    private static final int Y_OFFSET = 50;
    private static final int M_PX_MULTIPLIER = 400;
    // icons
    private Bitmap mIBeaconIcon, mCentralIcon;
    private Canvas mCanvas = new Canvas();
    // paint properties
    private Paint mPaint = new Paint();
    // rendered map
    private Bitmap mMapBitmap;
    // list of iBeacons
    private ArrayList<IBeacon> mIBeaconList = new ArrayList<IBeacon>();
    private boolean mIsCentralPositionSet = false;
    // central position
    private double[] mCentralPosition;

    /**
     * Create a new BeaconMapLayout
     *
     * @param context the Activity context
     */
}
```

```
public IBeaconMapLayout(Context context) {  
    super(context);  
    initialize();  
}  
/**  
 * Create a new BeaconMapLayout  
 *  
 * @param context the Activity context  
 * @param attrs  
 */  
public IBeaconMapLayout(Context context, AttributeSet attrs) {  
    super(context, attrs);  
    initialize();  
}  
  
/**  
 * Create a new BeaconMapLayout  
 *  
 * @param context the Activity context  
 * @param attrs  
 * @param defStyle  
 */  
public IBeaconMapLayout(  
    Context context,  
    AttributeSet attrs,  
    int defStyle  
) {  
    super(context, attrs, defStyle);  
    initialize();  
}  
  
/**  
 * Initialize the map  
 */  
public void initialize() {  
    mIBeaconIcon = BitmapFactory.decodeResource(getResources(),
```

```

        R.drawable.peripheral);

    mCentralIcon = BitmapFactory.decodeResource(getResources(),
        R.drawable.central);

    mPaint.setColor(Color.parseColor(PAINT_COLOR));
    mPaint.setStyle(Paint.Style.STROKE);
    mPaint.setStrokeWidth(STROKE_COLOR);
    mMapBitmap = Bitmap.createBitmap(
        BITMAP_WIDTH,
        BITMAP_HEIGHT,
        Bitmap.Config.ARGB_8888
    );
    mCanvas = new Canvas(mMapBitmap);
}

/**
 * Add a new Beacon
 *
 * @param iBeacon
 */
public void addBeacon(IBeacon iBeacon) {
    mIBeaconList.add(iBeacon);
}

/**
 * Position the central
 *
 * @param x x location
 * @param y y location
 */
public void setCentralPosition(double x, double y) {
    mIsCentralPositionSet = true;
    mCentralPosition = new double[]{x, y};
}

/**
 * Draw the Beacon Position

```

```

/*
 * @param beacon
 */
private void drawBeaconPosition(IBeacon iBeacon) {
    Log.d(TAG, "Drawing point: " + iBeacon);
    Rect sourceRect = new Rect(
        0,
        0,
        mIBeaconIcon.getWidth(),
        mIBeaconIcon.getHeight()
    );
    Rect destRect = new Rect(
        (int) (iBeacon.getXLocation() * M_PX_MULTIPLIER - 50 + X_OFFSET),
        (int) (iBeacon.getYLocation() * M_PX_MULTIPLIER - 50 + Y_OFFSET),
        (int) (iBeacon.getXLocation() * M_PX_MULTIPLIER + 50 + X_OFFSET),
        (int) (iBeacon.getYLocation() * M_PX_MULTIPLIER + 50 + Y_OFFSET)
    );
    mCanvas.drawBitmap(mIBeaconIcon, sourceRect, destRect, null);
    mCanvas.drawCircle(
        (float) iBeacon.getXLocation() * M_PX_MULTIPLIER + X_OFFSET,
        (float) iBeacon.getYLocation() * M_PX_MULTIPLIER + Y_OFFSET,
        (float) iBeacon.getDistance() * M_PX_MULTIPLIER, mPaint
    );
}

/***
 * Draw the Central onscreen
 *
 * @param location
 */
public void drawCentralPosition(double[] location) {
    Rect sourceRect = new Rect(
        0,
        0,
        mCentralIcon.getWidth(),
        mCentralIcon.getHeight()

```

```

);
Rect destRect = new Rect(
    (int) (location[0] * M_PX_MULTIPLIER - 36 + X_OFFSET),
    (int) (location[1] * M_PX_MULTIPLIER - 71 + Y_OFFSET),
    (int) (location[0] * M_PX_MULTIPLIER + 37 + X_OFFSET),
    (int) (location[1] * M_PX_MULTIPLIER + 72 + Y_OFFSET)
);
mCanvas.drawBitmap(mCentralIcon, sourceRect, destRect, null);
}

/**
 * Draw the frame
 */
public void draw() {
    // clear canvas
    mCanvas.drawColor(Color.WHITE);
    // draw each beacon
    for (IBeacon iBeacon : mIBeaconList) {
        drawBeaconPosition(iBeacon);
    }
    // draw central position
    if (mIsCentralPositionSet) {
        drawCentralPosition(mCentralPosition);
    }
    this.setBackgroundDrawable(new BitmapDrawable(mMapBitmap));
}
}

```

Create an `IBeaconListItem` class that displays `iBeacon` properties as items in a list.

#### **Example 11-10. `java/example.com.exampleble/models/IBeaconListItem.java`**

```

package example.com.beacon.models;
public class IBeaconListItem {
    private int mItemId;
    private int mRssi;

```

```

private IBeacon mIBeacon;

public IBeaconListItem(IBeacon iBeacon) {
    mIBeacon = iBeacon;
}

public void setItemId(int id) { mItemId = id; }
public void setRssi(int rssi) {
    mRssi = rssi;
}

public int getItemId() { return mItemId; }
public UUID getUuid() { return mIBeacon.getUuid(); }
public int getMajor() { return mIBeacon.getMajor(); }
public int getMinor() {
    return mIBeacon.getMinor();
}
public int getRssi() { return mRssi; }
public int getTransmissionPower() {
    return mIBeacon.getTransmissionPower();
}
public double getDistance() { return mIBeacon.getDistance(); }
public double getXLocation() { return mIBeacon.getXLocation(); }
public double getYLocation() { return mIBeacon.getYLocation(); }
public IBeacon getIBeacon() { return mIBeacon; }
}

```

Create a new layout file called `list_item_beacon.xml`. It will be responsible for displaying iBeacon information in a list.

### **Example 11-12. res/layout/list\_item\_beacon.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

```

```
    android:orientation="vertical" >
    <Textview
        android:id="@+id/uuid"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="10sp"
        android:paddingTop="@dimen/text_padding"/>
    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="2">
        <TextView
            android:id="@+id/major_number"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="10sp"
            android:paddingTop="@dimen/text_padding"/>
        <TextView
            android:id="@+id/minor_number"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="10sp"
            android:paddingTop="@dimen/text_padding"/>
        <TextView
            android:id="@+id/transmission_power"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="10sp"
            android:paddingTop="@dimen/text_padding"/>
    </LinearLayout>
    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="2">
```

```
<TextView
    android:id="@+id/rssi"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="10sp"
    android:paddingTop="@dimen/text_padding"/>

<TextView
    android:id="@+id/location"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="10sp"
    android:paddingTop="@dimen/text_padding"/>

<TextView
    android:id="@+id/distance"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="10sp"
    android:paddingTop="@dimen/text_padding"/>

</LinearLayout>
</LinearLayout>
```

## Activity

Create the MainActivity. It will be the main app.

### Example 11-13. java/example.com.exampleble/MainActivity.java

```
package example.com.beacon;

public class MainActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = MainActivity.class.getSimpleName();
    private static final int REQUEST_ENABLE_BT = 1;

    // Number of iBeacons required to find Central
    private static final int MIN_IBEACONS_FOR_TRILATERATION = 3;
```

```
/** Bluetooth Stuff */
private BleCommManager mBleCommManager;
private ArrayList<IBeacon> mFoundIBeacons = new ArrayList<IBeacon>();

/** UI Stuff */
private MenuItem mProgressSpinner;
private MenuItem mStartScanItem, mStopScanItem;
private TextView mCentralPosition;
private ListView mIBeaconsList;
private IBeaconsListAdapter mIBeaconsListAdapter;
private IBaconMapLayout mIBaconMap;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    // notify when bluetooth is turned on or off
    IntentFilter filter = new IntentFilter(
        BluetoothAdapter.ACTION_STATE_CHANGED
    );
    registerReceiver(mBleAdvertiseReceiver, filter);
    loadUI();
}

@Override
public void onResume() {
    super.onResume();
}

@Override
public void onPause() {
    super.onPause();
}
```

```

        stopScan();
    }

@Override
public void onDestroy() {
    super.onDestroy();
    unregisterReceiver(mBleAdvertiseReceiver);
}

public void loadUI() {
    mCentralPosition = (TextView) findViewById(R.id.central_position);
    mIBeaconsListAdapter = new IBeaconsListAdapter();
    mIBeaconsList = (ListView) findViewById(R.id.beacons_list);
    mIBeaconsList.setAdapter(mIBeaconsListAdapter);
    mIBeaconMap = (IBeaconMapLayout) findViewById(R.id.beacon_map);
}

/**
 * Create a menu
 * @param menu The menu
 * @return <b>true</b> if processed successfully
 */
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items
    // to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    mStartScanItem = menu.findItem(R.id.action_start_scan);
    mStopScanItem = menu.findItem(R.id.action_stop_scan);
    mProgressSpinner = menu.findItem(R.id.scan_progress_item);
    mStartScanItem.setVisible(true);
    mStopScanItem.setVisible(false);
    mProgressSpinner.setVisible(false);
    initializeBluetooth();
    return true;
}

```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_start_scan:
            // User chose the "Scan" item
            startScan();
            return true;
        case R.id.action_stop_scan:
            // User chose the "Stop" item
            stopScan();
            return true;
        default:
            // If we got here, the user's action was not recognized.
            // Invoke the superclass to handle it.
            return super.onOptionsItemSelected(item);
    }
}

/**
 * Turn on Bluetooth radio
 */
public void initializeBluetooth() {
    try {
        mBleCommManager = new BleCommManager(this);
    } catch (Exception e) {
        Log.d(TAG, "Could not initialize bluetooth");
        Log.d(TAG, e.getMessage());
        finish();
    }
    // should prompt user to open settings if Bluetooth is not enabled.
    if (!mBleCommManager.getBluetoothAdapter().isEnabled()) {
        Intent enableBtIntent = new Intent(
                BluetoothAdapter.ACTION_REQUEST_ENABLE
        );
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    }
}
```

```
    }

}

/***
 * Start scanning for iBeacons
 */
private void startScan() {
    mStartScanItem.setVisible(false);
    mStopScanItem.setVisible(true);
    mProgressSpinner.setVisible(true);
    mIBeaconsListAdapter.clear();
    mFoundIBeacons.clear();
    try {
        mBleCommManager.scanForPeripherals(
            mScanCallbackv18,
            mScanCallbackv21
        );
    } catch (Exception e) {
        Log.d(TAG, "Can't create Ble Device Scanner");
    }
}

/***
 * Stop scan
 */
public void stopScan() {
    mBleCommManager.stopScanning(mScanCallbackv18, mScanCallbackv21);
}

public void onBleScanStopped() {
    Log.v(TAG, "Scan complete");
    mStartScanItem.setVisible(true);
    mStopScanItem.setVisible(false);
    mProgressSpinner.setVisible(false);
}
```

```

/**
 * For adding fake iBeacons: calculate RSSI from a known distance
 *
 * @param txPower iBeacon Transmission Power
 * @param distance Distance between iBeacon and Central
 * @return
 */
private int getRssi(double txPower, double distance) {
    double ratio = Math.log10(distance);
    double difference = ratio * (10 * IBeacon.RADIO_PROPAGATION_CONSTANT);
    double rssi = txPower - difference;
    return (int) rssi;
}

/**
 * Event trigger when new Peripheral is discovered
 */
public void onIBeaconDiscovered(byte[] scanRecord, int rssi) {
    Log.v(
        TAG,
        "iBeacon discovered, GAP: " + \
        DataConverter.bytesToHex(scanRecord)
    );

    if (IBeacon.isIBeacon(scanRecord)) {
        try {
            final IBeacon iBeacon = IBeacon.fromScanRecord(scanRecord);
            // check if iBeacon is already in list
            boolean addBeacon = true;
            for (IBeaconListItem iBeaconListItem : \
                mIBeaconsListAdapter.getItems()
            ) {
                if (iBeaconListItem.getIBeacon().equals(iBeacon)) {
                    addBeacon = false;
                    break;
                }
            }
        }
    }
}

```

```

    }

    if (addBeacon) {
        // there is where to look up
        // the beacon position
        iBeacon.setRssi(rssi);
        mFoundIBeacons.add(iBeacon);
        mIBeaconsListAdapter.addIBeacon(iBeacon);
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                mIBeaconsListAdapter.notifyDataSetChanged();
            }
        });
        mIBeaconMap.addBeacon(iBeacon);
        mIBeaconMap.draw();
        if (mIBeaconsList.getCount() >= \
            MIN_IBEACONS_FOR_TRILATERATION
        ) {
            triangulateCentral();
        }
    }
} catch (Exception e) {
    Log.d(TAG, "Could not convert scanRecord into iBeacon");
}
} else {
    Log.d(TAG, "Not a beacon");
}
}

/***
 * when the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mBleAdvertiseReceiver = \
    new AdvertiseReceiver()
{
    @Override

```

```

public void onReceive(Context context, Intent intent) {
    final String action = intent.getAction();
    if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
        final int state = intent.getIntExtra(
            BluetoothAdapter.EXTRA_STATE,
            BluetoothAdapter.ERROR
        );
        switch (state) {
            case BluetoothAdapter.STATE_OFF:
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_OFF:
                break;
            case BluetoothAdapter.STATE_ON:
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_ON:
                break;
        }
    }
};

/***
 * Use this callback for Android API 21 (Lollipop) or greater
 */
private final BleScanCallbackV21 mScanCallbackV21 =
    new BleScanCallbackV21()
{
    /**
     * New Peripheral discovered
     *
     * @param callbackType int: Determines how this callback
     * was triggered. Could be one of CALLBACK_TYPE_ALL_MATCHES,
     * CALLBACK_TYPE_FIRST_MATCH or CALLBACK_TYPE_MATCH_LOST
     * @param result a Bluetooth Low Energy Scan Result, containing

```

```

        *      the Bluetooth Device, RSSI, and other information
        */
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        int rssi = result.getRssi();
        // Get Scan Record byte array (Be warned, this can be null)
        if (result.getScanRecord() != null) {
            byte[] scanRecord = result.getScanRecord().getBytes();
            onIBeaconDiscovered(scanRecord, rssi);
        }
    }

    /**
     * Several peripherals discovered when scanning in low power mode
     *
     * @param results List: List of previously scanned results
     */
    @Override
    public void onBatchScanResults(List<ScanResult> results) {
        for (ScanResult result : results) {
            int rssi = result.getRssi();
            // Get Scan Record byte array (Be warned, this can be null)
            if (result.getScanRecord() != null) {
                byte[] scanRecord = result.getScanRecord().getBytes();
                onIBeaconDiscovered(scanRecord, rssi);
            }
        }
    }

    /**
     * Scan failed to initialize
     *
     * @param errorCode int: Error code (one of SCAN_FAILED_*)
     *                   for scan failure.
     */
    @Override

```

```
public void onScanFailed(int errorCode) {
    switch (errorCode) {
        case SCAN_FAILED_ALREADY_STARTED:
            Log.e(
                TAG,
                "Fails to start scan as BLE scan with " + \
                "the same settings is already started by the app."
            );
            break;
        case SCAN_FAILED_APPLICATION_REGISTRATION_FAILED:
            Log.e(
                TAG,
                "Fails to start scan as app cannot be registered."
            );
            break;
        case SCAN_FAILED_FEATURE_UNSUPPORTED:
            Log.e(
                TAG,
                "Fails to start power optimized scan as this" + \
                "feature is not supported.");
            break;
        default: // SCAN_FAILED_INTERNAL_ERROR
            Log.e(TAG, "Fails to start scan due an internal error");
    }
}

runOnUiThread(new Runnable() {
    @Override
    public void run() {
        onBleScanStopped();
    }
});

/***
 * Scan completed
 */

```

```

public void onScanComplete() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

private BleScanCallbackV18 mScanCallbackV18 = new BleScanCallbackV18() {
    /**
     * Bluetooth LE Scan complete - timer expired out
     * while searching for bluetooth devices
     */
    @Override
    public void onLeScan(
        final BluetoothDevice bluetoothDevice,
        int rssi,
        byte[] scanRecord
    ) {
        onIBeaconDiscovered(scanRecord, rssi);
    }
    @Override
    public void onScanComplete() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleScanStopped();
            }
        });
    }
};

/**
 * plot the location of the Central on the map

```

```

*/
public void triangulateCentral() {
    try {
        double[] centralPosition = \
            IBeaconLocator.trilaterate(mFoundIBeacons);
        Log.d(
            TAG,
            "Central at " + centralPosition[0] + ", " + centralPosition[1]
        );
        String centralPositionString = "";
        try {
            String xPosition = String.format("%.1f", centralPosition[0]);
            String yPosition = String.format("%.1f", centralPosition[1]);
            centralPositionString = String.format(
                getResources().getString(R.string.central_position),
                xPosition,
                yPosition
            );
        } catch (Exception e) {
            Log.d(TAG, "Could not convert central location to string");
        }
        mCentralPosition.setText(centralPositionString);

        mIBeaconMap.setCentralPosition(
            centralPosition[0],
            centralPosition[1]
        );
        mIBeaconMap.draw();
    } catch (Exception e) {
        Log.d(
            TAG,
            "Not enough Beacons to perform a triangulation. Found " + \
            mFoundIBeacons.size()
        );
    }
}

```

}

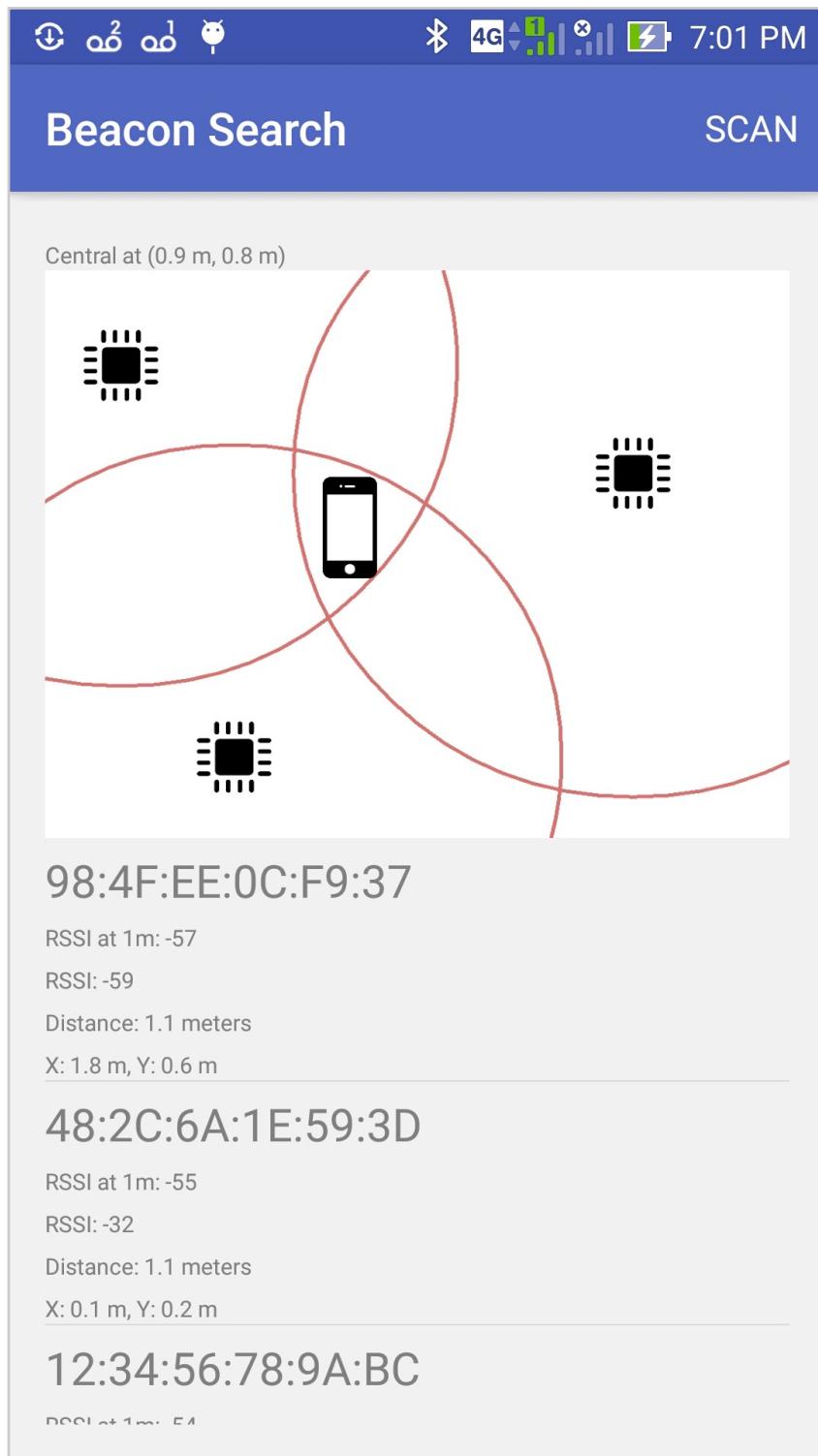
Edit the layout file for the main activity.

### Example 11149. res/layout/activity\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" android:fitsSystemWindows="true"
    tools:context="example.com.beacon.MainActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:orientation="vertical"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"
        tools:showIn="@layout/activity_main"
```

```
tools:context="example.com.beacon.MainActivity">
    <TextView
        android:id="@+id/central_position"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="10sp"
        android:paddingTop="@dimen/text_padding"/>
    <example.com.beacon.models.IBeaconMapLayout
        android:id="@+id/beacon_map"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="250dp"
        android:layout_alignParentBottom="false"
        android:layout_alignParentStart="true"
    />
    <ListView
        android:id="@+id/beacons_list"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
</android.support.design.widget.CoordinatorLayout>
```

The resulting app will be able to locate iBeacons and, if there are three or more iBeacons nearby, it can approximate its own location ([Figure 11-17](#)).



**Figure 11-17. App screen showing derived iBeacon and Central positions**

## Example code

The code for this chapter is available online

at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter11>

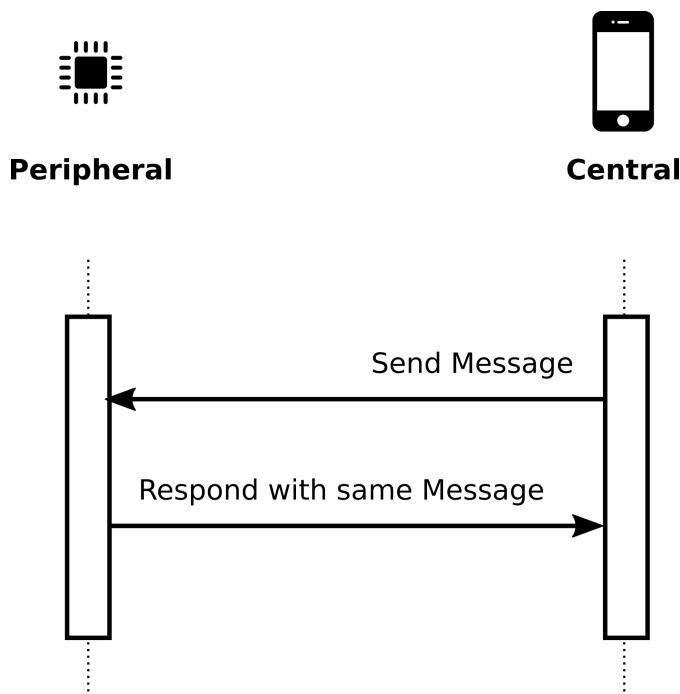
# Project: Echo Server

An Echo Server is the “Hello World” of network programming. It has the minimum features required to transmit, store, and respond to data on a network - the core features required for any network application.

And yet it must support all the features you’ve learned so far in this book - advertising, reads, writes, notifications, segmented data transfer, and encryption. It’s a sophisticated program!

The Echo Server works like this:

In this example, the Peripheral acts as a server, the “Echo Server” and the Central acts as a client ([Figure 12-1](#)).



**Figure 12-1. How an Echo Server works**

This project is based heavily on code seen up until Chapter 10, so there shouldn’t be any surprises.

# Programming the Central

The Echo Client will read and write messages on a Characteristic and will notify the Central when it is ready for new messages.

The Echo Server sends and receives text. Text is complicated because computers communicate using binary data, not text. As a result, there are a couple things that need to be done to protect the data from errors.

## Data Formatting

Bluetooth has a 20-byte maximum packet size. Longer messages must be divided and sent in parts. Both the client and server need a way to know if the data being transmitted belongs to part of a larger transmission.

One easy way to do this with text is to append a newline character to the end of the outbound transmission.

```
String sendText = "Hello world";
String message = sendText.getText().toString()+"\n";
```

When the message is broken apart for transmission, the parts are reassembled properly and the new line at the end separates new messages into new lines.

Everything else works the same way as any other Bluetooth app.

## Permissions

To work with Bluetooth Low Energy, in Android, first enable the manifest section of the Android Manifest:

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

```
<uses-feature android:name="android.hardware.bluetooth_le"  
    android:required="true" />
```

To turn on the Bluetooth radio programmatically, also enable Bluetooth admin permissions:

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

## Scanning and Connecting

The Central must scan for Peripherals.

```
bluetoothAdapter.startLeScan(scanCallback);
```

Whenever a device is discovered, the LeScanCallback.onLeScan callback is triggered. The Central can use this to search for a Bluetooth Peripheral with a known name.

```
if (device.getName() != null) {  
    if (device.getName().equals(blueoothDeviceName)) {  
        connectToDevice(device);  
    }  
}
```

## Connecting

Once a desired Bluetooth Peripheral is found, the Central can connect to it.

```
bluetoothGatt = device.connectGatt(context, false, gattCallback);
```

This process triggers a series of callbacks in BluetoothGattCallback. These callbacks activate the connection and discover the GATT profile of the Peripheral.

**Table 12-1. BluetoothGattCallback**

Event	Description
<b>onConnectionStateChange</b>	Triggered when a BLE Peripheral connects or disconnects
<b>onServicesDiscovered</b>	Triggered when GATT Services are discovered
<b>onCharacteristicRead</b>	Triggered when data has been downloaded from a GATT Characteristic
<b>onCharacteristicWrite</b>	Triggered when data has been uploaded to a GATT Characteristic
<b>onCharacteristicChanged</b>	Triggered when a GATT Characteristic's data has changed

## Discovering Characteristics

The configuration of Services, Characteristics on the Peripheral, and which ones can be written to or read from is called the GATT profile. Upon connection, the Client can learn about this in the `onServicesDiscovered` callback by asking for specific information about the discovered Services and Characteristics.

In this example, the IDs of the Characteristic used for communication are already known.

```
BluetoothGattService service = bluetoothGatt.getService(SERVICE_UUID);
BluetoothGattCharacteristic characteristic =
    service.getCharacteristic(CHARACTERISTIC_UUID);
int properties = characteristic.getProperties();
```

The Central can check if a Characteristic provides read permission:

```
int isReadable = properties & BluetoothGattCharacteristic.PROPERTY_READ;
```

It can check if a Characteristic has write permission:

```
int iswritable = properties &  
    (BluetoothGattCharacteristic.PROPERTY_WRITE |  
     BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE);
```

It can also check if the Characteristic can send notifications.

```
int isNotifiable = properties & BluetoothGattCharacteristic.PROPERTY_NOTIFY;
```

## Subscribe to Notifications

If the Characteristic can send notifications, the Central will subscribe to notifications by both setting notifications as enabled and writing to the descriptor of the Characteristic.

```
bluetoothGatt.setCharacteristicNotification(characteristic, enabled);  
descriptor.setValue(enabled ?  
    BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE : new byte[] { 0x00, 0x00 });  
bluetoothGatt.writeDescriptor(descriptor);
```

## Writing Messages

To write a message to the Characteristic, set the value of the Characteristic locally, then push the change to the Peripheral it to transmit the message.

```
characteristic.setValue(outgoingMessage);  
bluetoothGatt.writeCharacteristic(characteristic);
```

If the Central is sending text that is longer than the Characteristic's size, the message must be split into parts and sent one part at a time.

When working with text, it is important to make sure to append a terminator character, 0x00 to the end of the character array.

```
byte[] outgoingMessagePart = new byte[TRANSMISSION_LENGTH];  
System.arraycopy(  
    fullOutgoingMessage.getBytes(),  
    offset*TRANSMISSION_LENGTH,  
    outgoingMessagePart,  
    0,  
    TRANSMISSION_LENGTH);  
outgoingMessagePart[dataLength] = 0x00;
```

## Reading Responses

The Central will know that the Peripheral has echoed a message back when a notification is sent. This notification triggers the onCharacteristicChanged callback, where a read request can be issued:

```
bluetoothGatt.readCharacteristic(characteristic);
```

The onCharacteristicRead callback is triggered when the Central reads data from the Characteristic. Here the Central can convert the bytes of the message into a String:

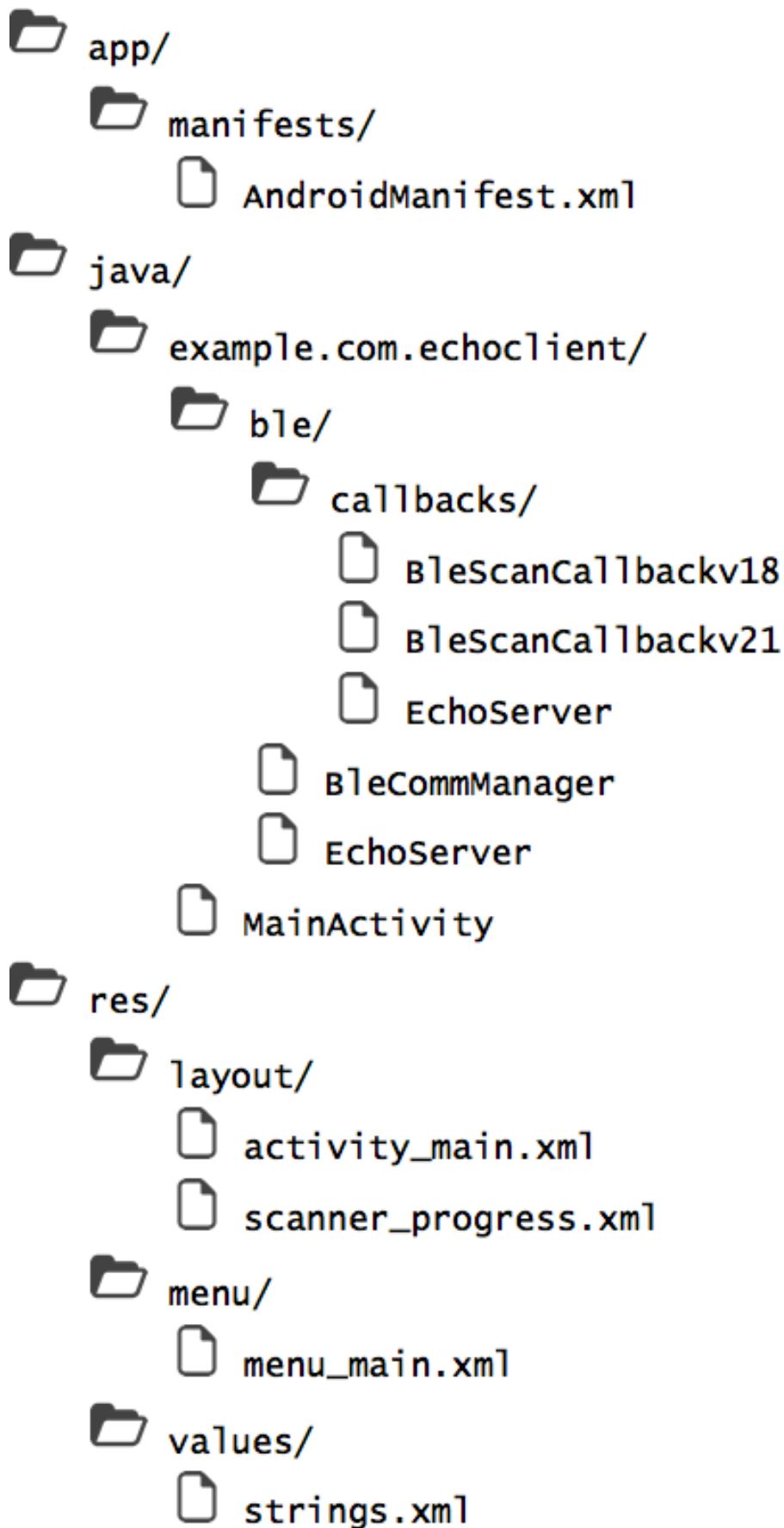
```
String incomingMessage = new String(data);
```

## Putting It All Together

The following code will create a single Activity App that connects to a Peripheral, allows a user to type a message, send that message to the Peripheral, and then prints the Peripheral's response.

Create a new project called EchoClient.

Create folders and classes, and XML files to reproduce the following structure ([Figure 12-2](#)).



**Figure 12-2. Project Structure**

Copy relevant classes from the previous chapters.

## Manifest

In the Manifest, add the following code to request permissions to use Bluetooth Low Energy.

### Example 12-1. app/manifests/AndroidManifest.xml

```
<manifest>
...
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
    <uses-feature android:name="android.hardware.bluetooth_le"
        android:required="true" />
...
</manifest>
```

## Resources

The App will indicate a connection status prior to connecting to the Peripheral. After connection, buttons and text fields are displayed that enable interaction with the Central. Each of these user interface elements will have some text labels so users understand their purpose. Let's add the text definitions to res/values/strings.xml.

### Example 12-2. res/values/strings.xml

```
<resources>
    <string name="app_name">BleEchoClient</string>
    <string name="scanning">Scanning...</string>
    <string name="connecting">Connecting...</string>
    <string name="loading">Loading...</string>
    <string name="send_button">Send</string>
    <string name="response_label">Response</string>
    <string name="no_central_found">Echo server not found</string>
</resources>
```

## Objects

The BleCommManager turns the Bluetooth radio on and scans for nearby Peripherals.

### Example 12-3. java/com.example.echoclient/ble/BleCommManager

```
package com.example.echoclient.ble
public class BleCommManager {
    private static final String TAG = BleCommManager.class.getSimpleName();
    // 5 seconds of scanning time
    private static final long SCAN_PERIOD = 5000;
    // Andrdoid's Bluetooth Adapter
    private BluetoothAdapter mBluetoothAdapter;
    // Ble scanner - API >= 21
    private BluetoothLeScanner mBluetoothLeScanner;
    // scan timer
    private Timer mTimer = new Timer();
    /**
     * Initialize the BleCommManager
     *
     * @param context the Activity context
     * @throws Exception Bluetooth Low Energy is
     *                  not supported on this Android device
     */
    public BleCommManager(final Context context) throws Exception {
        // make sure Android device supports Bluetooth Low Energy
        if (!context.getPackageManager().hasSystemFeature(
            PackageManager.FEATURE_BLUETOOTH_LE))
        {
            throw new Exception("Bluetooth Not Supported");
        }
        // get a reference to the bluetooth Manager class,
        // which allows us to talk to talk to the BLE radio
        final BluetoothManager bluetoothManager = (BluetoothManager) \
            context.getSystemService(Context.BLUETOOTH_SERVICE);
        mBluetoothAdapter = bluetoothManager.getAdapter();
```

```
}

/**
 * Get the Android Bluetooth Adapter
 *
 * @return BluetoothAdapter Android Bluetooth Adapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}

/**
 * Scan for Peripherals
 *
 * @param bleScanCallbackv18 APIv18 compatible scanCallback
 * @param bleScanCallbackv21 APIv21 compatible scanCallback
 * @throws Exception
 */
public void scanForPeripherals(
    final BleScanCallbackv18 bleScanCallbackv18, final BleScanCallbackv21 bleScanCallbackv21) throws Exception
{
    // Don't proceed if there is already a scan in progress
    mTimer.cancel();

    // Use BluetoothAdapter.startLeScan() for Android API 18, 19, and 20
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
        // Scan for SCAN_PERIOD milliseconds.
        // at the end of that time, stop the scan.
        new Thread() {
            @Override
            public void run() {
                mBluetoothAdapter.startLeScan(bleScanCallbackv18);
                try {
                    Thread.sleep(SCAN_PERIOD);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }
}
```

```

    }

    mBluetoothAdapter.stopLeScan(bleScanCallbackv18);

}

}.start();

// alert the system that BLE scanning has
// stopped after SCAN_PERIOD milliseconds
mTimer = new Timer();
mTimer.schedule(new TimerTask() {

    @Override
    public void run() {
        stopScanning(bleScanCallbackv18, bleScanCallbackv21);
    }
}, SCAN_PERIOD);

} else { // use BluetoothLeScanner.startScan() for API >= 21 (Lollipop)
    final ScanSettings settings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
        .build();

    final List<ScanFilter> filters = new ArrayList<ScanFilter>();
    mBluetoothLeScanner = mBluetoothAdapter.getBluetoothLeScanner();

    new Thread() {

        @Override
        public void run() {
            mBluetoothLeScanner.startScan(
                filters,
                settings,
                bleScanCallbackv21
            );
            try {
                Thread.sleep(SCAN_PERIOD);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            mBluetoothLeScanner.stopScan(bleScanCallbackv21);
        }
    }.start();
}

```

```

        // alert the system that BLE scanning has
        // stopped after SCAN_PERIOD milliseconds
        mTimer = new Timer();
        mTimer.schedule(new TimerTask() {
            @Override
            public void run() {
                stopScanning(bleScanCallbackv18, bleScanCallbackv21);
            }
        }, SCAN_PERIOD);
    }

}

/***
 * Stop Scanning
 *
 * @param bleScanCallbackv18 APIv18 compatible ScanCallback
 * @param bleScanCallbackv21 APIv21 compatible ScanCallback
 */
public void stopScanning(
    final BleScanCallbackv18 bleScanCallbackv18,
    final BleScanCallbackv21 bleScanCallbackv21
) {
    mTimer.cancel();
    // propagate the onScanComplete through the system
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
        mBluetoothAdapter.stopLeScan(bleScanCallbackv18);
        bleScanCallbackv18.onScanComplete();
    } else {
        mBluetoothLeScanner.stopScan(bleScanCallbackv21);
        bleScanCallbackv21.onScanComplete();
    }
}

}

```

BleScanCallbackv18 is one of two callback handlers for BleCommManager to process Peripheral discovered during as scan. Which callback class used depends on the API version of the phone the App gets installed on.

#### Example 12-4. java/com.example.echoclient/ble/callbacks/BleScanCallbackv18

```
package com.example.echoclient.ble.callbacks
public class BleScanCallbackv18 implements BluetoothAdapter.LeScanCallback {
    /**
     * New Peripheral found.
     *
     * @param bluetoothDevice The Peripheral Device
     * @param rssi The Peripheral's RSSI indicating
     *             how strong the radio signal is
     * @param scanRecord Other information about the scan result
     */
    // @Override
    public abstract void onLeScan(
        final BluetoothDevice bluetoothDevice,
        int rssi,
        byte[] scanRecord);

    /**
     * BLE Scan complete
     */
    public abstract void onScanComplete();
}
```

BleScanCallbackv21 handles Peripheral discovery and Bluetooth radio state changes in newer Android devices.

#### Example 12-5. java/com.example.echoclient/ble/callbacks/BleScanCallbackv21

```
package com.example.echoclient.ble.callbacks
public class BleScanCallbackv21 extends ScanCallback {
    /**

```

```

* New Perpheral found.
*
* @param callbackType int: Determines how this callback was triggered
* Could be one of CALLBACK_TYPE_ALL_MATCHES,
* CALLBACK_TYPE_FIRST_MATCH or CALLBACK_TYPE_MATCH_LOST
* @param result a Bluetooth Low Energy Scan Result, containing the
* Bluetooth Device, RSSI, and other information
*/
@Override
public abstract void onScanResult(int callbackType, ScanResult result);

/***
* New Perpherals found.
*
* @param results List: List of scan results that are previously scanned.
*/
@Override
public abstract void onBatchScanResults(List<ScanResult> results);

/***
* Problem initializing the scan. See the error code for reason
*
* @param errorCode      int: Error code (one of SCAN_FAILED_*) for
* scan failure.
*/
@Override
public abstract void onScanFailed(int errorCode);

/***
* Scan has completed
*/
public abstract void onScanComplete();
}

```

The EchoServer handles communication with the Bluetooth Peripheral. It processes incoming responses and sends out formatted commands.

### Example 12-6. java/com.example.echoclient/ble/EchoServer

```
package com.example.echoclient.ble

public class EchoServer {
    private static final String TAG = EchoServer.class.getSimpleName();

    public static final String CHARACTER_ENCODING = "ASCII";

    private BluetoothDevice mBluetoothDevice;
    private BluetoothGatt mBluetoothGatt;
    private EchoServerCallback mEchoServerCallback;
    private BluetoothGattCharacteristic mReadCharacteristic;
    private BluetoothGattCharacteristic mWriteCharacteristic;

    /** Bluetooth Device stuff */
    public static final String ADVERTISE_NAME = "EchoServer";
    public static final UUID SERVICE_UUID = \
        UUID.fromString("0000180c-0000-1000-8000-00805f9b34fb");
    public static final UUID READ_CHARACTERISTIC_UUID = \
        UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
    public static final UUID WRITE_CHARACTERISTIC_UUID = \
        UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");
    public static final UUID NOTIFY_DESCRIPTOR_UUID = \
        UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");
    private Context mContext;

    /** Flow control stuff */
    private int mNumPacketsTotal;
    private int mNumPacketsSent;
    private int mCharacteristicLength = 20;
    private String mQueuedCharacteristicValue;

    /**
     * Create a new EchoServer

```

```

*
 * @param context the Activity context
 * @param echoServerCallback the EchoServerCallback
 */
public EchoServer(Context context, EchoServerCallback echoServerCallback) {
    mContext = context;
    mEchoServerCallback = echoServerCallback;
}

/**
 * Connect to a Peripheral
 *
 * @param bluetoothDevice the Bluetooth Device
 * @return a connection to the BluetoothGatt
 * @throws Exception if no device is given
 */
public BluetoothGatt connect(
    BluetoothDevice bluetoothDevice) throws Exception
{
    if (bluetoothDevice == null) {
        throw new Exception("No bluetooth device provided");
    }
    mBluetoothDevice = bluetoothDevice;
    mBluetoothGatt = bluetoothDevice.connectGatt(
        mContext,
        false,
        mGattCallback
    );
    //refreshDeviceCache();
    return mBluetoothGatt;
}

/**
 * Disconnect from a Peripheral
 */
public void disconnect() {

```

```

        if (mBluetoothGatt != null) {
            mBluetoothGatt.disconnect();
        }

    }

/***
 * A connection can only close after a successful disconnect.
 * Be sure to use the BluetoothGattCallback.onConnectionStateChanged event
 * to notify of a successful disconnect
 */
public void close() {
    if (mBluetoothGatt != null) {
        mBluetoothGatt.close(); // close connection to Peripheral
        mBluetoothGatt = null; // release from memory
    }
}

public BluetoothDevice getBluetoothDevice() {
    return mBluetoothDevice;
}

/***
 * write next packet in queue if necessary
 *
 * @param value
 * @return the value being written
 * @throws Exception
 */
public String processIncomingMessage(String value) throws Exception {

    if (morePacketsAvailableInQueue()) {
        try {
            writePartialValue(mQueuedCharacteristicValue, mNumPacketsSent);
        } catch (Exception e) {
            Log.d(TAG, "Unable to send next chunk of message");
        }
    }
}

```

```

        return value;
    }

/***
 * Clear the GATT Service cache.
 *
 * New in this chapter
 *
 * @return <b>true</b> if the device cache clears successfully
 * @throws Exception
 */
public boolean refreshDeviceCache() throws Exception {
    Method localMethod = mBluetoothGatt.getClass().getMethod(
        "refresh",
        new Class[0]
    );
    if (localMethod != null) {
        return ((Boolean) localMethod.invoke(
            mBluetoothGatt,
            new Object[0])
        ).booleanValue();
    }
    return false;
}

/***
 * Request a data/value read from a BLE Characteristic
 */
public void readvalue() {
    // Reading a characteristic requires both
    // requesting the read and handling the callback that is
    // sent when the read is successful
    mBluetoothGatt.readCharacteristic(mReadCharacteristic);
}

*/

```

```

* write a value to the characteristic
*
* @param value
* @throws Exception
*/
public void writevalue(String value) throws Exception {
    // reset the queue counters, prepare the message to be sent,
    // and send the value to the characteristic
    mQueuedCharacteristicValue = value;
    mNumPacketsSent = 0;
    byte[] bytevalue = value.getBytes();
    mNumPacketsTotal = (int) Math.ceil(
        (float) bytevalue.length / mCharacteristicLength
    );
    writePartialValue(value, mNumPacketsSent);
}

/**
 * Subscribe or unsubscribe from Characteristic Notifications
*
* @param characteristic
* @param enabled <b>true</b> for "subscribe"
*           <b>false</b> for "unsubscribe"
*/
public void setCharacteristicNotification(
    final BluetoothGattCharacteristic characteristic,
    final boolean enabled
) {
    // This is a 2-step process
    // Step 1: set the Characteristic Notification parameter locally
    mBluetoothGatt.setCharacteristicNotification(
        characteristic,
        enabled
    );
    // Step 2: write a descriptor to the Bluetooth GATT
    // enabling the subscription on the Peripheral
}

```

```

// turns out you need to implement a delay between
// setCharacteristicNotification and setValue.
// maybe it can be handled with a callback,
// but this is an easy way to implement
Log.v(TAG, "characteristic: "+characteristic);
final Handler handler = new Handler(Looper.getMainLooper());
handler.postDelayed(new Runnable() {
    @Override
    public void run() {
        BluetoothGattDescriptor descriptor = \
            characteristic.getDescriptor(NOTIFY_DESCRIPTOR_UUID);
        Log.v(TAG, "descriptor: "+descriptor);
        if (enabled) {
            descriptor.setValue(
                BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE
            );
        } else {
            descriptor.setValue(
                BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE
            );
        }
        mBluetoothGatt.writeDescriptor(descriptor);
    }
}, 10);
}

/**
 * Write a portion of a larger message to a characteristic
 *
 * @param message The message being written
 * @param offset The current packet index in queue to be written
 * @throws Exception
 */
public void writePartialValue(
    String message, int offset) throws Exception
{

```

```

byte[] temp = message.getBytes();
mNumPacketsTotal = (int) Math.ceil(
    (float) temp.length / mCharacteristicLength
);
int remainder = temp.length % mCharacteristicLength;
int dataLength = mCharacteristicLength;
if (offset >= mNumPacketsTotal) {
    dataLength = remainder;
}
byte[] packet = new byte[dataLength];
for (int localIndex = 0; localIndex < packet.length; localIndex++) {
    int index = (offset * dataLength) + localIndex;
    if (index < temp.length) {
        packet[localIndex] = temp[index];
    } else {
        packet[localIndex] = 0x00;
    }
}
Log.v(
    TAG,
    "Writing message: '" + new String(packet, "ASCII") + \
    "' to " + mWriteCharacteristic.getUuid().toString()
);
mWriteCharacteristic.setValue(packet);
mBluetoothGatt.writeCharacteristic(mWriteCharacteristic);
mNumPacketsSent++;
}

/**
 * Determine if a message has been completely
 * written to a characteristic or if more data is in queue
 *
 * @return <b>false</b> if all of a message is has been
 *         written to a characteristic, <b>true</b> otherwise
 */
public boolean morePacketsAvailableInQueue() {

```

```

        boolean morePacketsAvailable = mNumPacketsSent < mNumPacketsTotal;
        Log.v(
            TAG,
            mNumPacketsSent + " of " + mNumPacketsTotal + \
            " packets sent: "+morePacketsAvailable
        );
        return morePacketsAvailable;
    }

    /**
     * Determine how much of a message has been written to a characteristic
     *
     * @return integer representing how many packets have been
     *         written so far to characteristic
     */
    public int getCurrentOffset() {
        return mNumPacketsSent;
    }

    /**
     * Get the current message being written to a characteristic
     *
     * @return the message in queue for writing to a characteristic
     */
    public String getCurrentMessage() {
        return mQueuedCharacteristicValue;
    }

    /**
     * Check if a characteristic supports write permissions
     * @return Returns <b>true</b> if property is writable
     */
    public static boolean isCharacteristicWritable(
        BluetoothGattCharacteristic characteristic
    ) {
        return (characteristic.getProperties() & \

```

```

        (BluetoothGattCharacteristic.PROPERTY_WRITE | \
        BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE)) != 0;
    }

    /**
     * Check if a characteristic has read permissions
     *
     * @return Returns <b>true</b> if property is Readable
     */
    public static boolean isCharacteristicReadable(
        BluetoothGattCharacteristic characteristic
    ) {
        return ((characteristic.getProperties() & \
            BluetoothGattCharacteristic.PROPERTY_READ) != 0);
    }

    /**
     * Check if a characteristic supports Notifications
     *
     * @return Returns <b>true</b> if property is supports notification
     */
    public static boolean isCharacteristicNotifiable(
        BluetoothGattCharacteristic characteristic
    ) {
        return (characteristic.getProperties() & \
            BluetoothGattCharacteristic.PROPERTY_NOTIFY) != 0;
    }

    /**
     * Handle changes to connection and GATT profile
     */
    private final BluetoothGattCallback mGattCallback = \
        new BluetoothGattCallback()
    {
        @Override
        public void onCharacteristicRead(

```

```

        final BluetoothGatt gatt,
        final BluetoothGattCharacteristic characteristic,
        int status
    ) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            final byte[] data = characteristic.getValue();
            String message = "";
            try {
                message = new String(data, CHARACTER_ENCODING);
            } catch (Exception e) {
                Log.d(
                    TAG,
                    "Could not convert message byte array to String"
                );
            }
            Log.d(TAG, "received: "+message);
            final String messageText = message;
            mEchoServerCallback.messageReceived(messageText);
            try {
                processIncomingMessage(message);
            } catch (Exception e) {
                Log.d(TAG, "Could not send next message part");
            }
        }
    }

@Override
public void onCharacteristicWrite(
    BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic,
    int status
) {
    if (status == BluetoothGatt.GATT_SUCCESS) {
        Log.d(TAG, "characteristic written");
        mEchoServerCallback.messageSent();
    } else {

```

```
        Log.d(TAG, "problem writing characteristic");
    }

}

@Override
public void onCharacteristicChanged(
    BluetoothGatt gatt,
    final BluetoothGattCharacteristic characteristic
) {
    readValue();
}

@Override
public void onConnectionStateChange(
    final BluetoothGatt bluetoothGatt,
    int status, int newState
) {
    if (newState == BluetoothProfile.STATE_CONNECTED) {
        Log.d(TAG, "Connected to device");

        bluetoothGatt.discoverServices();
    } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        mEchoServerCallback.disconnected();
        Log.d(TAG, "Disconnected from device");

        disconnect();
    }
}

@Override
public void onServicesDiscovered(
    final BluetoothGatt bluetoothGatt,
    int status
) {
    Log.d(TAG, "SERVICE DISCOVERED!: ");
    // if services were discovered, then let's
    // iterate through them and display them on screen
}
```

```
if (status == BluetoothGatt.GATT_SUCCESS) {  
    // check if there are matching services and characteristics  
    BluetoothGattService service = \  
        bluetoothGatt.getService(EchoServer.SERVICE_UUID);  
    if (service != null) {  
        Log.d(TAG, "service found");  
        mReadCharacteristic = \  
            service.getCharacteristic(  
                EchoServer.READ_CHARACTERISTIC_UUID  
            );  
        mWriteCharacteristic = \  
            service.getCharacteristic(  
                EchoServer.WRITE_CHARACTERISTIC_UUID  
            );  
  
        Log.v(TAG, "read descriptors: ");  
        for (BluetoothGattDescriptor descriptor : \  
            mReadCharacteristic.getDescriptors()  
        ) {  
            Log.e(  
                TAG,  
                "BluetoothGattDescriptor: " + \  
                descriptor.getUuid().toString()  
            );  
        }  
        Log.v(  
            TAG,  
            "write descriptors: "  
        );  
        for (BluetoothGattDescriptor descriptor : \  
            mWriteCharacteristic.getDescriptors()  
        ) {  
            Log.e(  
                TAG,  
                "BluetoothGattDescriptor: " + \  
                descriptor.getUuid().toString()  
            );  
        }  
    }  
}
```

```

        );
    }

    if (isCharacteristicNotifiable(mReadCharacteristic)) {
        setCharacteristicNotification(
            mReadCharacteristic,
            true
        );
    }
}

} else {
    Log.d(
        TAG,
        "Something went wrong while discovering GATT " + \
        "services from this device"
    );
}
mEchoServerCallback.connected();
}
};

}

```

The EchoServerCallback alerts changes to the EchoServer state, including connections, disconnections, and message received notifications.

### **Example 12-7. java/com.example.echoclient/ble/callbacks/EchoServerCallback**

```

package com.example.echoclient.ble.callbacks

public abstract class EchoServerCallback {

    /**
     * Echo Server connected
     */
    public abstract void connected();

    /**
     * Echo Server disconnected
     */

```

```

public abstract void disconnected();

/**
 * Message sent to Echo Server
 */

public abstract void messageSent();

/**
 * Message received from Echo Server
 *
 * @param messageText the incoming text
 */
public abstract void messageReceived(final String messageText);
}

```

## Activities

Add new functions to the Talk activity to display user interface elements when a characteristic is notifiable, and a check box that lets us subscribe and unsubscribe.

### **Example 12-8. java/example.com.echoclient/MainActivity.java**

```

package example.com.echoclient
public class MainActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = MainActivity.class.getSimpleName();
    private final static int REQUEST_ENABLE_BT = 1;

    /** Bluetooth Stuff */
    private BleCommManager mBleCommManager;
    private EchoServer mEchoServer;
    private boolean mIsConnecting = false;

    /** UI Stuff */
    private MenuItem mProgressSpinner;

```

```
private TextView mResponseText;
private TextView mSendText;
private TextView mDeviceNameTV;
private TextView mDeviceAddressTV;
private Button mSendButton;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    // notify when bluetooth is turned on or off
    IntentFilter filter = new IntentFilter(
        BluetoothAdapter.ACTION_STATE_CHANGED
    );
    registerReceiver(mBleAdvertiseReceiver, filter);
    loadUI();
    mEchoServer = new EchoServer(this, mEchoServerCallback);
}

@Override
public void onResume() {
    super.onResume();
}

@Override
public void onPause() {
    super.onPause();
    stopScanning();
    disconnect();
}

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

```

        unregisterReceiver(mBleAdvertiseReceiver);
    }

    /**
     * Prepare the UI elements
     */
    public void loadUI() {
        mResponseText = (TextView) findViewById(R.id.response_text);
        mSendText = (TextView) findViewById(R.id.write_text);
        mDeviceNameTV = (TextView) findViewById(R.id.advertise_name);
        mDeviceAddressTV = (TextView) findViewById(R.id.mac_address);
        mSendButton = (Button) findViewById(R.id.write_button);
        mSendButton.setVisibility(View.GONE);
        mSendText.setVisibility(View.GONE);
        mResponseText.setVisibility(View.GONE);
    }

    /**
     * Create the menu
     *
     * @param menu
     * @return <b>true</b> if successful
     */
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu;
        // this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        mProgressSpinner = menu.findItem(R.id.scan_progress_item);
        mProgressSpinner.setVisible(true);
        initializeBluetooth();
        return true;
    }
}

```

```

/**
 * Turn on Bluetooth radio
 */
public void initializeBluetooth() {
    try {
        mBleCommManager = new BleCommManager(this);
    } catch (Exception e) {
        Log.d(TAG, "Could not initialize bluetooth");
        Log.d(TAG, e.getMessage());
        finish();
    }
    // should prompt user to open settings if Bluetooth is not enabled.
    if (!mBleCommManager.getBluetoothAdapter().isEnabled()) {
        Intent enableBtIntent = new Intent(
            BluetoothAdapter.ACTION_REQUEST_ENABLE
        );
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    } else {
        startScan();
    }
}

/**
 * Start scanning for Peripherals
 */
private void startScan() {
    mDeviceNameTV.setText(R.string.scanning);
    mProgressSpinner.setVisibility(true);
    try {
        mBleCommManager.scanForPeripherals(
            mScanCallbackv18,
            mScanCallbackv21
        );
    } catch (Exception e) {
        Log.d(TAG, "Can't create Ble Device Scanner");
    }
}

```

```

    }

}

/***
 * Stop scanning for Peripherals
 */
public void stopScanning() {
    mBleCommManager.stopScanning(mScanCallbackv18, mScanCallbackv21);
}

/***
 * Event trigger when BLE Scanning has stopped
 */
public void onBleScanStopped() {
}

/***
 * Connect to Peripheral
 */
public void connect(BluetoothDevice bluetoothDevice) {
    mDeviceNameTV.setText(R.string.connecting);
    mProgressSpinner.setVisibility(true);
    try {
        mIsConnecting = true;
        mEchoServer.connect(bluetoothDevice);
    } catch (Exception e) {
        mProgressSpinner.setVisibility(false);
        Log.d(TAG, "Error connecting to device");
    }
}

/***
 * Disconnect from Peripheral
 */
private void disconnect() {
    mIsConnecting = false;
}

```

```

    mEchoServer.disconnect();
    // remove callbacks
    mSendButton.removeCallbacks(null);
    finish();
}

/***
 * Update TextView when a new message is read from a Characteristic
 * Also scroll to the bottom so that new messages are always in view
 *
 * @param message the Characteristic value to display in the UI as text
 */
public void updateResponseText(String message) {
    mResponseText.append(message);
    final int scrollAmount = \
        mResponseText.getLayout().getLineTop(
            mResponseText.getLineCount()
        ) - mResponseText.getHeight();
    // if there is no need to scroll, scrollAmount will be <=0
    if (scrollAmount > 0) {
        mResponseText.scrollTo(0, scrollAmount);
    } else {
        mResponseText.scrollTo(0, 0);
    }
}

/***
 * Event trigger when new Peripheral is discovered
 */
public void onBlePeripheralDiscovered(
    final BluetoothDevice bluetoothDevice,
    int rssi
) {
    // only add the device if
    // - it has a name, or
    // - doesn't already exist in our list, or
}

```

```

// - is transmitting at a higher power (is closer)
// than an existing device

if (!mIsConnecting) {
    boolean addDevice = false;
    if (bluetoothDevice.getName() != null) {
        if (bluetoothDevice.getName().equals(
            EchoServer.ADVERTISE_NAME)
        ) {
            addDevice = true;
        }
    }
    if (addDevice) {
        stopScanning();
        connect(bluetoothDevice);
    }
}

/***
 * Bluetooth Peripheral connected. Update UI
 */
public void onBleConnected() {
    mProgressSpinner.setVisibility(false);
    mDeviceNameTV.setText(EchoServer.ADVERTISE_NAME);
    mDeviceAddressTV.setText(
        mEchoServer.getBluetoothDevice().getAddress()
    );
    mProgressSpinner.setVisibility(false);
    mResponseText.setVisibility(View.VISIBLE);
    // attach callbacks to the buttons and stuff
    mSendButton.setVisibility(View.VISIBLE);
    mSendText.setVisibility(View.VISIBLE);
    mSendButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Log.d(TAG, "Send button clicked");
        }
    });
}

```

```

        String value = mSendText.getText().toString()+"\n";
        try {
            mEchoServer.writeValue(value);
        } catch (Exception e) {
            Log.d(TAG, "problem sending message through bluetooth");
        }
    }

});

}

/***
 * Bluetooth Peripheral disconnected. Update UI
 */
public void onBleDisconnected() {
    mDeviceNameTV.setText("");
    mDeviceAddressTV.setText("");
    mProgressSpinner.setVisibility(false);
}

/***
 * Clear the input TextView when a characteristic is
 * successfully written to.
 */
public void onBleCharacteristicValueWritten() {
    mSendText.setText("");
}

*/

 * when the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mBleAdvertiseReceiver =
    new AdvertiseReceiver()
{
    @Override

```

```
public void onReceive(Context context, Intent intent) {
    Log.v(TAG, "onReceive");
    final String action = intent.getAction();

    if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
        final int state = intent.getIntExtra(
            BluetoothAdapter.EXTRA_STATE,
            BluetoothAdapter.ERROR
        );
        switch (state) {
            case BluetoothAdapter.STATE_OFF:
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_OFF:
                break;
            case BluetoothAdapter.STATE_ON:
                Log.v(TAG, "Bluetooth on");
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_ON:
                break;
        }
    }
}

/**
 * Use this callback for Android API 21 (Lollipop) or greater
 */
private final BleScanCallbackV21 mScanCallbackV21 = \
    new BleScanCallbackV21()
{
    /**
     * New Peripheral discovered
     *
     * @param callbackType int: Determines how this callback was

```

```

*      triggered. Could be one of CALLBACK_TYPE_ALL_MATCHES,
*      CALLBACK_TYPE_FIRST_MATCH or CALLBACK_TYPE_MATCH_LOST
* @param result a Bluetooth Low Energy Scan Result, containing
*      the Bluetooth Device, RSSI, and other information
*/
@Override
public void onScanResult(int callbackType, ScanResult result) {
    final BluetoothDevice bluetoothDevice = result.getDevice();
    int rssi = result.getRssi();
    onBlePeripheralDiscovered(bluetoothDevice, rssi);
}

/**
 * Several peripherals discovered when scanning in low power mode
 *
 * @param results List: List of scan results that are
 *      previously scanned.
*/
@Override
public void onBatchScanResults(List<ScanResult> results) {
    for (ScanResult result : results) {
        final BluetoothDevice bluetoothDevice = result.getDevice();
        int rssi = result.getRssi();
        onBlePeripheralDiscovered(bluetoothDevice, rssi);
    }
}

/**
 * Scan failed to initialize
 *
 * @param errorCode int: Error code (one of SCAN_FAILED_*)
 *      for scan failure.
*/
@Override
public void onScanFailed(int errorCode) {
    switch (errorCode) {

```

```
        case SCAN_FAILED_ALREADY_STARTED:
            Log.e(
                TAG,
                "Fails to start scan as BLE scan with " + \
                "the same settings is already started by the app."
            );
            break;

        case SCAN_FAILED_APPLICATION_REGISTRATION_FAILED:
            Log.e(
                TAG,
                "Fails to start scan as app cannot be registered."
            );
            break;

        case SCAN_FAILED_FEATURE_UNSUPPORTED:
            Log.e(
                TAG,
                "Fails to start power optimized scan as this " + \
                "feature is not supported."
            );
            break;

        default: // SCAN_FAILED_INTERNAL_ERROR
            Log.e(
                TAG,
                "Fails to start scan due an internal error"
            );
    }

    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

/**
 * Scan completed
```

```
 */
public void onScanComplete() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

/**
 * Use this callback for Android API 18, 19, and 20 (before Lollipop)
 */
private BleScanCallbackV18 mScanCallbackV18 = new BleScanCallbackV18() {

    /**
     * Bluetooth LE Scan complete - timer expired out while
     * searching for bluetooth devices
     */
    @Override
    public void onLeScan(
        final BluetoothDevice bluetoothDevice,
        int rssi,
        byte[] scanRecord
    ) {
        onBlePeripheralDiscovered(bluetoothDevice, rssi);
    }

    @Override
    public void onScanComplete() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleScanStopped();
            }
        });
    }
};
```

```
    }

};

private EchoServerCallback mEchoServerCallback = new EchoServerCallback() {

    @Override
    public void connected() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleConnected();
            }
        });
    }

    @Override
    public void disconnected() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleDisconnected();
            }
        });
    }

    @Override
    public void messageSent() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleCharacteristicValueWritten();
            }
        });
    }

    @Override
    public void messageReceived(final String messageText) {
```

```
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                updateResponseText(messageText);
            }
        });
    }
};
```

In the MainActivity layout, create a Button to send a message and two TextViews, one to type in a message and one to display the response from the Peripheral so that the user can type messages to the Echo server and see the result.

### Example 12-9. res/layout/activity\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".ConnectActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
```

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main"
    tools:context="example.com.bleechoclient.MainActivity"
    android:weightSum="1">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:text="@string/loading"
        android:id="@+id/advertise_name"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:text="@string/loading"
        android:id="@+id/mac_address"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="172dp"
        android:background="#ffffffff"
        android:id="@+id/response_text"
        android:layout_weight=".5" />

    <LinearLayout
        android:orientation="horizontal"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/write_text"
        android:inputType="textShortMessage"
        android:layout_weight="2" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/write_button"
        android:id="@+id/write_button" />
</LinearLayout>
</LinearLayout>
</android.support.design.widget.CoordinatorLayout>
```

The menu displays a progress scanner in the menu when attempting a connection:

### Example 12-10. res/menu/menu\_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="example.com.echoserver.MainActivity">
    <item
        android:id="@+id/scan_progress_item"
        android:title="@string/scanning"
        android:visible="true"
        android:orderInCategory="100"
        app:showAsAction="always"
        app:actionLayout="@layout/scanner_progress"
        android:layout_marginRight="@dimen/activity_horizontal_margin" />
</menu>
```

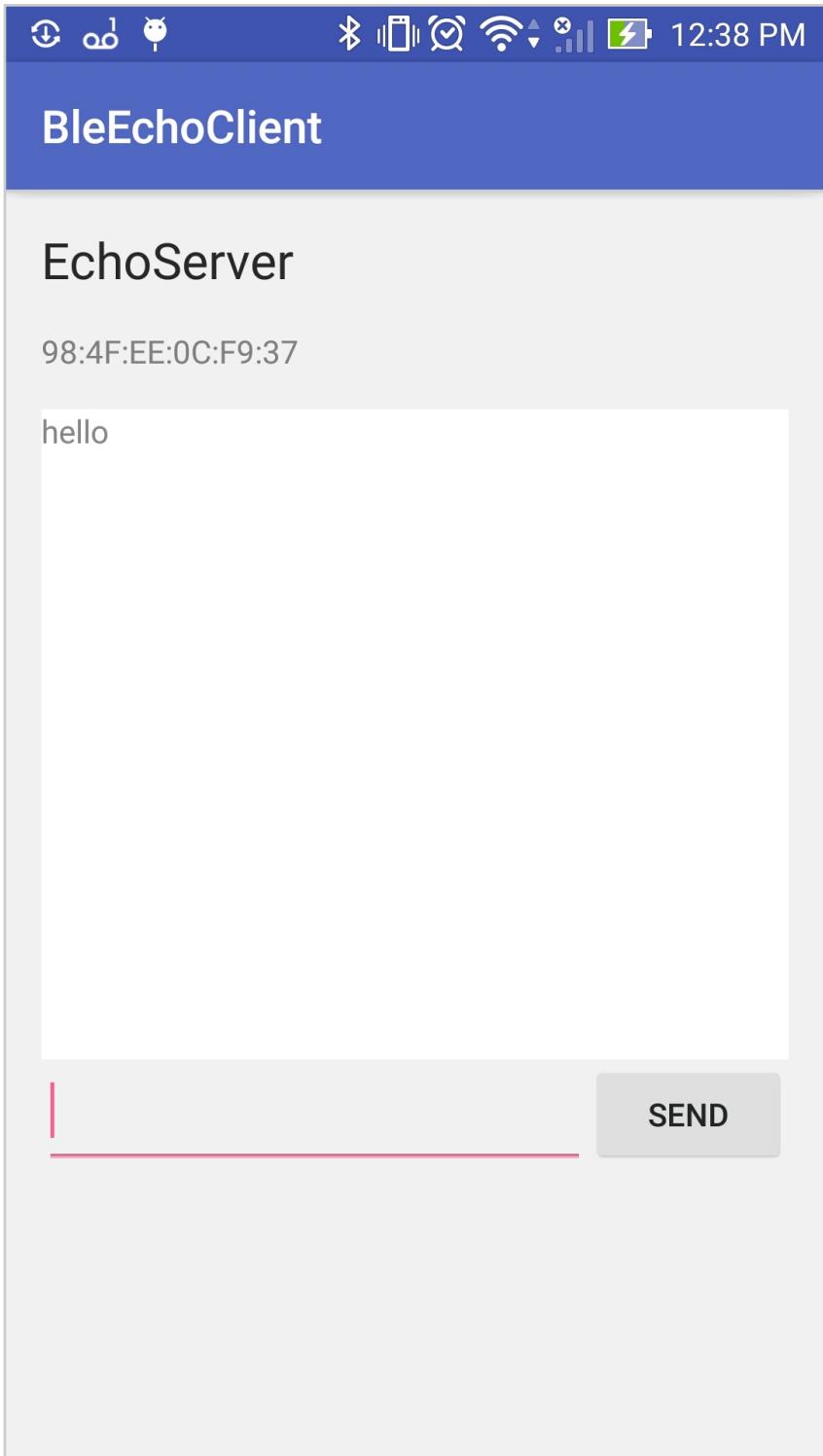
The progress scanner is as defined below:

**Example 12-13. res/layout/scanner\_progress.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<ProgressBar xmlns:android="http://schemas.android.com/apk/res/android"
    style="@android:style/widget.ProgressBar.Small"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/scan_progress" />
```

The resulting Central App can scan for and connect to a Bluetooth Low Energy Peripheral. Once connected, the Central can send and receive messages to the Peripheral.

If a user types “hello” into the bottom TextView and hit the Subscribe button, the word “hello” will appear above in the upper Text View ([Figure 12-3](#)).



**Figure 12-3. App screen showing interface to read from and write to the Echo Server**

## Programming the Peripheral

The Echo Server will handle incoming writes on one Characteristic (0x2a57) and echo back messages on another Characteristic (0x2a56).

It will also host a minimal GATT Profile that includes a battery percentage, device name, model number, and serial number.

# Permissions

To work with Bluetooth Low Energy, in Android, first enable the manifest section of the Android Manifest:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-feature android:name="android.hardware.bluetooth_le"
    android:required="true" />
```

To turn on the Bluetooth radio programmatically, also enable Bluetooth admin permissions:

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

## Advertising and GATT Profile

The Peripheral must advertise and host a GATT Profile, which will include a minimal GATT profile.

The Peripheral will host a read-only, notifiable Characteristic on UUID 0x2a56 and a write-only Characteristic on UUID 0x2a57:

```
public static final UUID SERVICE_UUID = \
    UUID.fromString("0000180c-0000-1000-8000-00805f9b34fb");
public static final UUID READ_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
public static final UUID WRITE_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");

mService = new BluetoothGattService(
    SERVICE_UUID,
    BluetoothGattService.SERVICE_TYPE_PRIMARY
);

mReadCharacteristic = new BluetoothGattCharacteristic(
```

```

READ_CHARACTERISTIC_UUID,
BluetoothGattCharacteristic.PROPERTY_READ | \
    BluetoothGattCharacteristic.PROPERTY_NOTIFY,
BluetoothGattCharacteristic.PERMISSION_READ);

mWriteCharacteristic = new BluetoothGattCharacteristic(
    WRITE_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_WRITE,
    BluetoothGattCharacteristic.PERMISSION_WRITE);

mService.addCharacteristic(mReadCharacteristic);
mService.addCharacteristic(mWriteCharacteristic);

mGattServer.addService(mService);

```

It will advertise as "EchoServer" to be discoverable by the corresponding Central:

```

public static final String ADVERTISING_NAME = "EchoServer";

// Build Advertise settings with transmission power and advertise speed
AdvertiseSettings advertiseSettings = new AdvertiseSettings.Builder()
    .setAdvertiseMode(mAdvertisingMode)
    .setTxPowerLevel(mTransmissionPower)
    .setConnectable(true)
    .build();

AdvertiseData.Builder advertiseBuilder = new AdvertiseData.Builder();

// set advertising name
advertiseBuilder.setIncludeDeviceName(true);
mBluetoothAdapter.setName(ADVERTISING_NAME);

// add Services to Advertising Data
List<BluetoothGattService> services = mGattServer.getServices();
for (BluetoothGattService service: services) {
    advertiseBuilder.addServiceUuid(new ParcelUuid(service.getUuid()));
}

```

```
}
```

## Handling Subscriptions and Writes

Once Central initiates a connection, it can subscribe to the read Characteristic, which will trigger the `onDescriptorWriteRequest()` method in `BluetoothGattServerCallback`, and it can send write requests to the write Characteristic:

**Table 12-2. BluetoothGattServerCallback**

Event	Description
<code>onConnectionStateChange</code>	Central connects or disconnects
<code>onCharacteristicReadRequest</code>	Characteristic supports notifications
<code>onCharacteristicWriteRequest</code>	Central attempted to read a value from a Characteristic
<code>onDescriptorWriteRequest</code>	Central attempted to change a Descriptor in a Characteristic
<code>onNotificationSent</code>	Central was notified of a change to a Characteristic

The `onDescriptorWriteRequest()` is triggered when the Central subscribes or unsubscribes from the read Characteristic:

```
private final BluetoothGattServerCallback mGattServerCallback = \  
    new BluetoothGattServerCallback() {  
        ...  
        @Override  
        public void onDescriptorWriteRequest(  
            BluetoothDevice device,  
            int requestId,  
            BluetoothGattDescriptor descriptor,  
            boolean preparedwrite,  
            boolean responseNeeded,  
            int offset,  
            byte[] value  
        ) {
```

```

Log.v(
    TAG,
    "Descriptor write Request " + descriptor.getUuid() + " " + \
        Arrays.toString(value)
);
super.onDescriptorWriteRequest(
    device,
    requestId,
    descriptor,
    preparedWrite,
    responseNeeded,
    offset,
    value
);
// determine which Characteristic is being requested
BluetoothGattCharacteristic characteristic = \
    descriptor.getCharacteristic();
// is the descriptor writeable?
if (isDescriptorWriteable(descriptor)) {
    descriptor.setValue(value);
    // was this a subscription or an unsubscription?
    if (descriptor.getUuid().equals(NOTIFY_DESCRIPTOR_UUID)) {
        if (value == \
            BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE)
        {
            mBlePeripheralCallback.onCharacteristicSubscribedTo(
                characteristic
            );
        } else if (value == \
            BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE)
        {
            mBlePeripheralCallback.onCharacteristicUnsubscribedFrom(
                characteristic
            );
        }
    }
    // send a confirmation if necessary
}

```

```

        if (isDescriptorReadable(descriptor)) {
            mGattServer.sendResponse(
                device,
                requestId,
                BluetoothGatt.GATT_SUCCESS,
                offset,
                value
            );
        }
    }

} else {
    // notify failure if necessary
    if (isDescriptorReadable(descriptor)) {
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_WRITE_NOT_PERMITTED,
            offset,
            value
        );
    }
}
};

}

```

When a Central writes data to the write Characteristic (0x2a57), the Echo Server will copy the value to the read Characteristic (0x2a56) and send a notification of the change.

```

...
@Override
public void onCharacteristicWriteRequest(
    BluetoothDevice device, int requestId,
    BluetoothGattCharacteristic characteristic,
    boolean preparedwrite,
    boolean responseNeeded,

```

```
    int offset, byte[] value
) {
    super.onCharacteristicWriteRequest(
        device,
        requestId,
        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
    Log.v(TAG, "Characteristic write request: " + Arrays.toString(value));
    // set the read characteristic value
    mReadCharacteristic.setValue(value);
    if (isCharacteristicWritableWithResponse(characteristic)) {
        characteristic.setValue(value);
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            0,
            null
        );
    }
    if (isCharacteristicNotifiable(characteristic)) {
        boolean isNotifiedofSend = false;
        mGattServer.notifyCharacteristicChanged(
            device,
            characteristic,
            isNotifiedofSend
        );
    }
}
...

```

## Putting It All Together

The following code will create a single Activity App that advertises a Peripheral, allows a Central to connect and write values to a writeable Characteristic, and will respond by copying that value to a readable, notifiable Characteristic.

Create a new project called EchoServer.

Create folders and classes, and XML files to reproduce the following structure ([Figure 12-4](#)).

```
app/
    manifests/
        AndroidManifest.xml
java/
    example.com.echoserver/
        ble/
            callbacks/
                BlePeripheralCallback
                EchoServerCallback
            BlePeripheral
            EchoServer
            utilities/
                DataConverter
        MainActivity
res/
    layout/
        activity_main.xml
    menu/
        menu_main.xml
    values/
        strings.xml
```

**Figure 12-4. Project Structure**

Copy relevant classes from the previous chapters.

## Manifest

In the Manifest, add the following code to request permissions to use Bluetooth Low Energy.

### **Example 12-14. app/manifests/AndroidManifest.xml**

```
<manifest>
    ...
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
    <uses-feature android:name="android.hardware.bluetooth_le"
        android:required="true" />
    ...
</manifest>
```

## **Resources**

The App will display the Peripheral name, Bluetooth radio state, connectivity with a Central, and a log of changes in the write Characteristic's value. Each of these states will need a text label so users understand their purpose. Add the text definitions to res/values/strings.xml.

### **Example 12-15. res/values/strings.xml**

```
<resources>
    <string name="app_name">BleEchoServer</string>
    <string name="advertising_name_label">Advertising Name: </string>
    <string name="bluetooth_on">Bluetooth On</string>
    <string name="central_connected">Echo Client Connected</string>
    <string name="characteristic_log">Characterstic Log:</string>
</resources>
```

## **Objects**

The BlePeripheral sets up a minimal GATT Profile and handles connections, disconnections, and subscriptions:

### **Example 12-16. java/com.example.echoserver/ble/BlePeripheral**

```
package com.example.echoserver.ble
public class BlePeripheral {
```

```

/** Constants */

private static final String TAG = BlePeripheral.class.getSimpleName();
// 5 minutes
private static final int BATTERY_STATUS_CHECK_TIME_MS = 5*60*1000;

/** Peripheral and GATT Profile */
private String mPeripheralAdvertisingName;
public static final UUID NOTIFY_DESCRIPTOR_UUID = \
    UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");
// Minimal GATT Profile UUIDs
public static final UUID DEVICE_INFORMATION_SERVICE_UUID = \
    UUID.fromString("0000180a-0000-1000-8000-00805f9b34fb");
public static final UUID BATTERY_LEVEL_SERVICE = \
    UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb");
public static final UUID DEVICE_NAME_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a00-0000-1000-8000-00805f9b34fb");
public static final UUID MODEL_NUMBER_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a24-0000-1000-8000-00805f9b34fb");
public static final UUID SERIAL_NUMBER_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a04-0000-1000-8000-00805f9b34fb");
public static final UUID BATTERY_LEVEL_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb");
public static final String CHARSET = "ASCII";
public static final int MAX_ADVERTISING_NAME_BYTE_LENGTH = 20;

/** Advertising settings */

// advertising mode can be one of:
// - ADVERTISE_MODE_BALANCED,
// - ADVERTISE_MODE_LOW_LATENCY,
// - ADVERTISE_MODE_LOW_POWER
int mAdvertisingMode = AdvertiseSettings.ADVERTISE_MODE_LOW_LATENCY;

// transmission power mode can be one of:
// - ADVERTISE_TX_POWER_HIGH

```

```

// - ADVERTISE_TX_POWER_MEDIUM
// - ADVERTISE_TX_POWER_LOW
// - ADVERTISE_TX_POWER_ULTRA_LOW
int mTransmissionPower = AdvertiseSettings.ADVERTISE_TX_POWER_HIGH;

/** Callback Handlers */
public BlePeripheralCallback mBlePeripheralCallback;

/** Bluetooth Stuff */
private BluetoothAdapter mBluetoothAdapter;
private BluetoothLeAdvertiser mBluetoothAdvertiser;

private BluetoothGattServer mGattServer;
private BluetoothGattService mDeviceInformationService;
private BluetoothGattService mBatteryLevelService;
private BluetoothGattCharacteristic mDeviceNameCharacteristic,
        mModelNumberCharacteristic,
        mSerialNumberCharacteristic,
        mBatteryLevelCharacteristic;
private Context mContext;
private String mModelNumber = "";
private String mSerialNumber = "";

/** 
 * Construct a new Peripheral
 *
 * @param context The Application Context
 * @param blePeripheralCallback The callback handler that
 *     interfaces with this Peripheral
 * @throws Exception Exception thrown if Bluetooth is not supported
 */
public BlePeripheral(
    final Context context,
    BlePeripheralCallback blePeripheralCallback) throws Exception
{
    mBlePeripheralCallback = blePeripheralCallback;
}

```

```

mContext = context;
// make sure Android device supports Bluetooth Low Energy
if (!context.getPackageManager().hasSystemFeature(
    PackageManager.FEATURE_BLUETOOTH_LE)
) {
    throw new Exception("Bluetooth Not Supported");
}
// get a reference to the Bluetooth Manager class,
// which allows us to talk to talk to the BLE radio
final BluetoothManager bluetoothManager = (BluetoothManager) \
    context.getSystemService(Context.BLUETOOTH_SERVICE);

mGattServer = bluetoothManager.openGattServer(
    context,
    mGattServerCallback
);
mBluetoothAdapter = bluetoothManager.getAdapter();
// Beware: this function doesn't work on some systems
if(!mBluetoothAdapter.isMultipleAdvertisementSupported()) {
    throw new Exception ("Peripheral mode not supported");
}
mBluetoothAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();
// Use this method instead for better support
if (mBluetoothAdvertiser == null) {
    throw new Exception ("Peripheral mode not supported");
}
}

/**
 * Get the system Bluetooth Adapter
 *
 * @return BluetoothAdapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}

```

```
/**  
 * Get the GATT Server  
 */  
public BluetoothGattServer getGattServer() {  
    return mGattServer;  
}  
  
/**  
 * Get the model number  
 */  
public String getModelNumber() {  
    return mModelNumber;  
}  
  
/** set the model number  
 *  
 */  
public void setModelNumber(String modelNumber) {  
    mModelNumber = modelNumber;  
}  
  
/**  
 * Get the serial number  
 */  
public String getSerialNumber() {  
    return mSerialNumber;  
}  
  
/**  
 * set the serial number  
 */  
public void setSerialNumber(String serialNumber) {  
    mSerialNumber = serialNumber;  
}
```

```
/**  
 * Get the actual battery level  
 */  
  
public int getBatteryLevel() {  
    BatteryManager batteryManager = \  
        (BatteryManager)mContext.getSystemService(BATTERY_SERVICE);  
    return batteryManager.getIntProperty(  
        BatteryManager.BATTERY_PROPERTY_CAPACITY);  
}  
  
/**  
 * Set up the GATT profile  
 */  
  
public void setupDevice() {  
    // build characteristics  
    mDeviceInformationService = \  
        new BluetoothGattService(  
            DEVICE_INFORMATION_SERVICE_UUID,  
            BluetoothGattService.SERVICE_TYPE_PRIMARY  
        );  
    mDeviceNameCharacteristic = new BluetoothGattCharacteristic(  
        DEVICE_NAME_CHARACTERISTIC_UUID,  
        BluetoothGattCharacteristic.PROPERTY_READ,  
        BluetoothGattCharacteristic.PERMISSION_READ);  
    mModelNumberCharacteristic = new BluetoothGattCharacteristic(  
        MODEL_NUMBER_CHARACTERISTIC_UUID,  
        BluetoothGattCharacteristic.PROPERTY_READ,  
        BluetoothGattCharacteristic.PERMISSION_READ);  
    mSerialNumberCharacteristic = new BluetoothGattCharacteristic(  
        SERIAL_NUMBER_CHARACTERISTIC_UUID,  
        BluetoothGattCharacteristic.PROPERTY_READ,  
        BluetoothGattCharacteristic.PERMISSION_READ);  
    mBatteryLevelService = new BluetoothGattService(  
        BATTERY_LEVEL_SERVICE,  
        BluetoothGattService.SERVICE_TYPE_PRIMARY  
    );
```

```

mBatteryLevelCharacteritic = new BluetoothGattCharacteristic(
    BATTERY_LEVEL_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_READ | \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY,
    BluetoothGattCharacteristic.PERMISSION_READ | \
        BluetoothGattCharacteristic.PERMISSION_WRITE);

// add Notification support to characteristic
BluetoothGattDescriptor notifyDescriptor = \
    new BluetoothGattDescriptor(
        BlePeripheral.NOTIFY_DESCRIPTOR_UUID,
        BluetoothGattDescriptor.PERMISSION_WRITE | \
            BluetoothGattDescriptor.PERMISSION_READ
    );

mBatteryLevelCharacteritic.addDescriptor(notifyDescriptor);
mDeviceInformationService.addCharacteristic(mDeviceNameCharacteristic);
mDeviceInformationService.addCharacteristic(
    mModelNumberCharacteristic
);
mDeviceInformationService.addCharacteristic(
    mSerialNumberCharacteristic
);
mBatteryLevelService.addCharacteristic(mBatteryLevelCharacteritic);

// put in fake values for the characteristic.
mModelNumberCharacteristic.setValue(mModelNumber);
mSerialNumberCharacteristic.setValue(mSerialNumber);

// add Services to Peripheral
mGattServer.addService(mDeviceInformationService);
mGattServer.addService(mBatteryLevelService);

// update the battery level every
// BATTERY_STATUS_CHECK_TIME_MS milliseconds
TimerTask updateBatteryTask = new TimerTask() {
    @Override
    public void run() {
        mBatteryLevelCharacteritic.setValue(
            getBatteryLevel(),
            BluetoothGattCharacteristic.FORMAT_UINT8,

```

```

        0
    );
    for (BluetoothDevice device : \
        mGattServer.getConnectedDevices())
    {
        mGattServer.notifyCharacteristicChanged(
            device,
            mBatteryLevelCharacterstic,
            true
        );
    }
}

Timer randomStringTimer = new Timer();
// schedule the battery update and run it once immediately
randomStringTimer.schedule(
    updateBatteryTask,
    0,
    BATTERY_STATUS_CHECK_TIME_MS
);
}

/**
 * Set the Advertising name of the Peripheral
 *
 * @param peripheralAdvertisingName
 */
public void setPeripheralAdvertisingName(
    String peripheralAdvertisingName) throws Exception
{
    mPeripheralAdvertisingName = peripheralAdvertisingName;
    mDeviceNameCharacteristic.setValue(mPeripheralAdvertisingName);
    int advertisingNameByteLength = \
        mPeripheralAdvertisingName.getBytes(CHARSET).length;
    if (advertisingNameByteLength > MAX_ADVERTISING_NAME_BYTE_LENGTH) {
        throw new Exception(

```

```

        "Advertising name too long. Must be less than " + \
        MAX_ADVERTISING_NAME_BYTE_LENGTH + " bytes"
    );
}

}

/***
 * Get the Advertising name of the Peripheral
 */
public String getPeripheralAdvertisingName() {
    return mPeripheralAdvertisingName;
}

/***
 * Set the Transmission Power mode
 */
public void setTransmissionPower(int transmissionPower) {
    mTransmissionPower = transmissionPower;
}

/***
 * Set the advertising mode
 */
public void setAdvertisingMode(int advertisingMode) {
    mAdvertisingMode = advertisingMode;
}

/***
 * Add a Service to the GATT Profile
 *
 * @param service the Service to add
 */
public void addService(BluetoothGattService service) {
    mGattServer.addService(service);
}

```

```

/**
 * Build the Advertising Data, including the transmission power,
 * advertising name, and Services
 *
 * @return AdvertiseData
 */
private AdvertiseData buildAdvertisingData() {
    AdvertiseData.Builder advertiseBuilder = new AdvertiseData.Builder();
    // set advertising name
    if (mPeripheralAdvertisingName != null) {
        advertiseBuilder.setIncludeDeviceName(true);
        mBluetoothAdapter.setName(mPeripheralAdvertisingName);
    }
    // add Services to Advertising Data
    List<BluetoothGattService> services = mGattServer.getServices();
    for (BluetoothGattService service: services) {
        advertiseBuilder.addServiceUuid(
            new ParcelUuid(service.getUuid())
        );
    }
    return advertiseBuilder.build();
}

/**
 * Build Advertise settings with transmission power and advertise speed
 *
 * @return AdvertiseSettings for Bluetooth Advertising
 */
private AdvertiseSettings buildAdvertiseSettings() {
    AdvertiseSettings.Builder settingsBuilder = \
        new AdvertiseSettings.Builder();
    settingsBuilder.setAdvertiseMode(mAdvertisingMode);
    settingsBuilder.setTxPowerLevel(mTransmissionPower);
    // There's no need to connect to a Peripheral that
    // doesn't host a Gatt profile
    if (mGattServer != null) {

```

```

        settingsBuilder.setConnectable(true);
    } else {
        settingsBuilder.setConnectable(false);
    }
    return settingsBuilder.build();
}

/**
 * Start Advertising
 *
 * @throws Exception Exception thrown if Bluetooth Peripheral
 *                   mode is not supported
 */
public void startAdvertising() {
    AdvertiseSettings advertiseSettings = buildAdvertiseSettings();
    AdvertiseData advertiseData = buildAdvertisingData();
    // begin advertising
    mBluetoothAdvertiser.startAdvertising(
        advertiseSettings,
        advertiseData,
        mAdvertiseCallback
    );
}

/**
 * Stop advertising
 */
public void stopAdvertising() {
    if (mBluetoothAdvertiser != null) {
        mBluetoothAdvertiser.stopAdvertising(mAdvertiseCallback);
        mBlePeripheralCallback.onAdvertisingStopped();
    }
}

/**
 * Check if a characteristic supports write permissions

```

```

    * @return Returns <b>true</b> if property is writable
    */
public static boolean isCharacteristicwritable(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        (BluetoothGattCharacteristic.PROPERTY_WRITE | \
        BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE)) != 0;
}

/**
 * Check if a Characetricistic supports write wuthout response permissions
 * @return Returns <b>true</b> if property is writable
 */
public static boolean isCharacteristicwritablewithoutResponse(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE) != 0;
}

/**
 * Check if a Characetricistic supports write with permissions
 * @return Returns <b>true</b> if property is writable
 */
public static boolean isCharacteristicwritablewithResponse(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        BluetoothGattCharacteristic.PROPERTY_WRITE) != 0;
}

/**
 * Check if a Characteristic supports Notifications
 *
 * @return Returns <b>true</b> if property is supports notification

```

```

*/
public static boolean isCharacteristicNotifiable(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY) != 0;
}

/**
 * Check if a Descriptor can be read
 *
 * @param descriptor a descriptor to check
 * @return Returns <b>true</b> if descriptor is readable
 */
public static boolean isDescriptorReadable(
    BluetoothGattDescriptor descriptor
) {
    return (descriptor.getPermissions() & \
        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
}

/**
 * Check if a Descriptor can be written
 *
 * @param descriptor a descriptor to check
 * @return Returns <b>true</b> if descriptor is writeable
 */
public static boolean isDescriptorWriteable(
    BluetoothGattDescriptor descriptor
) {
    return (descriptor.getPermissions() & \
        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
}

private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()

```

```

{

    @Override
    public void onConnectionStateChange(
        BluetoothDevice device,
        final int status,
        int newState
    ) {
        super.onConnectionStateChange(device, status, newState);
        Log.v(TAG, "Connected");
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothGatt.STATE_CONNECTED) {
                mBlePeripheralCallback.onCentralConnected(device);
                stopAdvertising();
            } else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
                mBlePeripheralCallback.onCentralDisconnected(device);
                try {
                    startAdvertising();
                } catch (Exception e) {
                    Log.e(TAG, "error starting advertising");
                }
            }
        }
    }

    @Override
    public void onCharacteristicReadRequest(
        BluetoothDevice device,
        int requestId,
        int offset,
        BluetoothGattCharacteristic characteristic
    ) {
        super.onCharacteristicReadRequest(
            device,
            requestId,
            offset,
            characteristic
        );
    }
}

```

```
);

Log.d(
    TAG,
    "Device tried to read characteristic: " + \
    characteristic.getUuid()
);

Log.d(
    TAG,
    "Value: " + Arrays.toString(characteristic.getValue())
);

if (offset != 0) {
    mGattServer.sendResponse(
        device,
        requestId,
        BluetoothGatt.GATT_INVALID_OFFSET,
        offset,
        characteristic.getValue()
    );
    return;
}

mGattServer.sendResponse(
    device,
    requestId,
    BluetoothGatt.GATT_SUCCESS,
    offset, characteristic.getValue()
);

}

@Override
public void onNotificationSent(BluetoothDevice device, int status) {
    super.onNotificationSent(device, status);
    Log.v(TAG, "Notification sent. Status: " + status);
}

@Override
public void onCharacteristicWriteRequest(
```

```
        BluetoothDevice device,
        int requestId,
        BluetoothGattCharacteristic characteristic,
        boolean preparedwrite,
        boolean responseNeeded,
        int offset,
        byte[] value
    ) {
    super.onCharacteristicWriteRequest(
        device,
        requestId,
        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
    Log.v(
        TAG,
        "Characteristic write request: " + Arrays.toString(value)
    );
    mBlePeripheralcallback.onCharacteristicWritten(
        device,
        characteristic,
        value
    );
    if (isCharacteristicWritableWithResponse(characteristic)) {
        characteristic.setValue(value);
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            0,
            null
        );
    }
}
```

```
        if (isCharacteristicNotifiable(characteristic)) {
            boolean isNotifiedofSend = false;
            mGattServer.notifyCharacteristicChanged(
                device,
                characteristic,
                isNotifiedofSend
            );
        }
    }

@Override
public void onDescriptorWriteRequest(
    BluetoothDevice device,
    int requestId,
    BluetoothGattDescriptor descriptor,
    boolean preparedwrite,
    boolean responseNeeded,
    int offset,
    byte[] value
) {
    Log.v(
        TAG,
        "Descriptor write Request " + descriptor.getUuid() + \
        " " + Arrays.toString(value)
    );
    super.onDescriptorWriteRequest(
        device,
        requestId,
        descriptor,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
    // determine which Characteristic is being requested
    BluetoothGattCharacteristic characteristic = \
        descriptor.getCharacteristic();
}
```

```

// is the descriptor writeable?
if (isDescriptorWriteable(descriptor)) {
    descriptor.setvalue(value);
    // was this a subscription or an unsubscription?
    if (descriptor.getUuid().equals(NOTIFY_DESCRIPTOR_UUID)) {
        if (value == \
            BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE)
        {
            mBlePeripheralCallback.onCharacteristicSubscribedTo(
                characteristic
            );
        } else if (value == \
            BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE)
        {
            mBlePeripheralCallback
                .onCharacteristicUnsubscribedFrom(
                    characteristic
                );
        }
        // send a confirmation if necessary
        if (isDescriptorReadable(descriptor)) {
            mGattServer.sendResponse(
                device,
                requestId,
                BluetoothGatt.GATT_SUCCESS,
                offset,
                value
            );
        }
    }
} else {
    // notify failure if necessary
    if (isDescriptorReadable(descriptor)) {
        mGattServer.sendResponse(
            device,
            requestId,

```

```

        BluetoothGatt.GATT_WRITE_NOT_PERMITTED,
        offset,
        value
    );
}

}

};

public AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
    @Override
    public void onStartSuccess(AdvertiseSettings settingsInEffect) {
        super.onStartSuccess(settingsInEffect);
        mBlePeripheralCallback.onAdvertisingStarted();
    }

    @Override
    public void onStartFailure(int errorCode) {
        super.onStartFailure(errorCode);
        mBlePeripheralCallback.onAdvertisingFailed(errorCode);
    }
};
}

```

The BlePeripheralCallback relays BlePeripheral state changes and events to the EchoServer:

### **Example 12-17. java/com.example.echoserver/ble/callbacks/BlePeripheralCallback**

```

package com.example.echoserver.ble

public abstract class BlePeripheralCallback {

    /**
     * Advertising Started
     */
    public abstract void onAdvertisingStarted();
}

```

```
/**  
 * Advertising Could not Start  
 */  
public abstract void onAdvertisingFailed(int errorCode);  
  
/**  
 * Advertising Stopped  
 */  
public abstract void onAdvertisingStopped();  
  
/**  
 * Central Connected  
 *  
 * @param bluetoothDevice the BluetoothDevice representing the  
 * connected Central  
 */  
public abstract void onCentralConnected(  
    final BluetoothDevice bluetoothDevice  
);  
  
/**  
 * Central Disconnected  
 *  
 * @param bluetoothDevice the BluetoothDevice representing the  
 * disconnected Central  
 */  
public abstract void onCentralDisconnected(  
    final BluetoothDevice bluetoothDevice  
);  
  
/**  
 * characteristic written to  
 *  
 * @param characteristic The Characteristic that was written to  
 * @param value the byte value that was written  
 */
```

```

public abstract void onCharacteristicWritten(
    final BluetoothDevice connectedDevice,
    final BluetoothGattCharacteristic characteristic,
    final byte[] value
);

/**
 * Characteristic subscribed to
 *
 * @param characteristic The characteristic that was subscribed to
 */
public abstract void onCharacteristicSubscribedTo(
    final BluetoothGattCharacteristic characteristic
);

/**
 * Characteristic unsubscribed from
 *
 * @param characteristic The characteristic that was unsubscribed from
 */
public abstract void onCharacteristicUnsubscribedFrom(
    final BluetoothGattCharacteristic characteristic
);
}

```

The EchoServer handles communication with the Echo Server Peripheral:

#### **Example 12-18. java/com.example.echoserver/ble/EchoServer**

```

package com.example.echoserver.ble

public class EchoServer {
    /** Constants */
    private static final String TAG = EchoServer.class.getSimpleName();
    public static final String CHARSET = "ASCII";
    private static final String MODEL_NUMBER = "1AB2";
    private static final String SERIAL_NUMBER = "1234";
}

```

```

/** Peripheral and GATT Profile */
public static final String ADVERTISING_NAME = "EchoServer";
public static final UUID SERVICE_UUID = \
    UUID.fromString("0000180c-0000-1000-8000-00805f9b34fb");
public static final UUID READ_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
public static final UUID WRITE_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");
private static final int READ_CHARACTERISTIC_LENGTH = 20;
private static final int WRITE_CHARACTERISTIC_LENGTH = 20;

/** Callback Handlers */
public EchoServerCallback mEchoServerCallback;

/** Bluetooth Stuff */
private BlePeripheral mBlePeripheral;
private BluetoothGattService mService;
private BluetoothGattCharacteristic mReadCharacteristic,
    mWriteCharacteristic;

/**
 * Construct a new Peripheral
 *
 * @param context The Application Context
 * @param blePeripheralCallback The callback handler that
 *     interfaces with this Peripheral
 * @throws Exception Exception thrown if Bluetooth is not supported
 */
public EchoServer(
    final Context context,
    EchoServerCallback blePeripheralCallback) throws Exception
{
    mEchoServerCallback = blePeripheralCallback;
    mBlePeripheral = new BlePeripheral(context, mBlePeripheralCallback);
    setupDevice();
}

```

```

}

/**
 * Set up the Advertising name and GATT profile
 */
private void setupDevice() throws Exception {
    mBlePeripheral.setModelNumber(MODEL_NUMBER);
    mBlePeripheral.setSerialNumber(SERIAL_NUMBER);
    mBlePeripheral.setupDevice();
    mService = new BluetoothGattService(
        SERVICE_UUID,
        BluetoothGattService.SERVICE_TYPE_PRIMARY
    );
    mReadCharacteristic = new BluetoothGattCharacteristic(
        READ_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ | \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY,
        BluetoothGattCharacteristic.PERMISSION_READ
    );

    // add Notification support to Characteristic
    BluetoothGattDescriptor notifyDescriptor = \
        new BluetoothGattDescriptor(
            BlePeripheral.NOTIFY_DESCRIPTOR_UUID,
            BluetoothGattDescriptor.PERMISSION_WRITE | \
            BluetoothGattDescriptor.PERMISSION_READ
        );
    mReadCharacteristic.addDescriptor(notifyDescriptor);
    mWriteCharacteristic = new BluetoothGattCharacteristic(
        WRITE_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_WRITE,
        BluetoothGattCharacteristic.PERMISSION_WRITE);
    mService.addCharacteristic(mReadCharacteristic);
    mService.addCharacteristic(mWriteCharacteristic);
    mBlePeripheral.addService(mService);
}

```

```

/**
 * Start Advertising
 *
 * @throws Exception Exception thrown if Bluetooth Peripheral
 *                   mode is not supported
 */
public void startAdvertising() throws Exception {
    // set the device name
    mBlePeripheral.setPeripheralAdvertisingName(ADVERTISING_NAME);
    mBlePeripheral.startAdvertising();
}

/**
 * Stop advertising
 */
public void stopAdvertising() {
    mBlePeripheral.stopAdvertising();
}

/**
 * Get the BlePeripheral
 */
public BlePeripheral getBlePeripheral() {
    return mBlePeripheral;
}

private BlePeripheralCallback mBlePeripheralCallback = \
    new BlePeripheralCallback()
{
    @Override
    public void onAdvertisingStarted() {
    }

    @Override
    public void onAdvertisingFailed(int errorCode) {
}

```

```
}

@Override
public void onAdvertisingStopped() {
}

@Override
public void onCentralConnected(BluetoothDevice bluetoothDevice) {
    mEchoServerCallback.onCentralConnected(bluetoothDevice);
}

@Override
public void onCentralDisconnected(BluetoothDevice bluetoothDevice) {
    mEchoServerCallback.onCentralDisconnected(bluetoothDevice);
}

@Override
public void onCharacteristicWritten(
    BluetoothDevice connectedDevice,
    BluetoothGattCharacteristic characteristic,
    byte[] value)
{
    // copy value to the read characteristic
    mReadCharacteristic.setValue(value);
    Log.v(TAG, "setting readCharacteristic: "+ Arrays.toString(value));
    // send a notification
    mBlePeripheral.getGattServer().notifyCharacteristicChanged(
        connectedDevice,
        mReadCharacteristic,
        true
    );
    mEchoServerCallback.onMessageWritten(value);
}

@Override
public void onCharacteristicSubscribedTo(
```

```
        bluetoothGattCharacteristic characteristic)
    {
    }

    @Override
    public void onCharacteristicUnsubscribed(
        bluetoothGattCharacteristic characteristic)
    {
    }
};

}
```

The EchoServerCallback relays state changes and events from the EchoServer, such as writes and subscriptions:

#### Example 12-19. java/com.example.echoserver/ble/callbacks/EchoServerCallback

```
package com.example.echoserver.ble.callbacks
public abstract class EchoServerCallback {

    /**
     * Central Connected
     *
     * @param bluetoothDevice the BluetoothDevice representing the
     * connected Central
     */
    public abstract void onCentralConnected(
        final BluetoothDevice bluetoothDevice
    );

    /**
     * Central Disconnected
     *
     * @param bluetoothDevice the BluetoothDevice representing the
     * disconnected Central
     */
    public abstract void onCentralDisconnected(

```

```

        final BluetoothDevice bluetoothDevice
    );

    /**
     * Characteristic written to
     *
     * @param value the byte value that was written
     */
    public abstract void onMessageWritten(final byte[] value);
}

```

The DataConverter converts various data types, useful for debugging purposes:

#### **Example 12-20. java/com.example.echoserver/ble/utilities/DataConverter**

```

package com.example.echoserver.utilities

public class DataConverter {

    /**
     * convert bytes to hexadecimal for debugging purposes
     *
     * @param bytes
     * @return Hexadecimal string representation of the byte array
     */
    public static String bytesToHex(byte[] bytes) {
        if (bytes.length <=0) return "";
        char[] hexArray = "0123456789ABCDEF".toCharArray();
        char[] hexChars = new char[bytes.length * 3];
        for ( int j = 0; j < bytes.length; j++ ) {
            int v = bytes[j] & 0xFF;
            hexChars[j * 3] = hexArray[v >>> 4];
            hexChars[j * 3 + 1] = hexArray[v & 0x0F];
            hexChars[j * 3 + 2] = 0x20; // space
        }
        return new String(hexChars);
    }
}

```

## Activities

The Main Activity turns on the Bluetooth radio and creates an EchoServer. It responds to connectivity changes by toggling a Switch, and responds to EchoServer Characteristic writes by displaying the value in a TextView:

### Example 12-20. java/example.com.echoserver/MainActivity.java

```
package example.com.echoserver
public class MainActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = MainActivity.class.getSimpleName();
    private static final int REQUEST_ENABLE_BT = 1;

    /** Bluetooth Stuff */
    private EchoServer mEchoServer;

    /** UI Stuff */
    private TextView mAdvertisingNameTV, mCharacteristicLogTV;
    private Switch mBluetoothOnSwitch,
                  mCentralConnectedSwitch;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        // notify when bluetooth is turned on or off
        IntentFilter filter = new IntentFilter(
            BluetoothAdapter.ACTION_STATE_CHANGED
        );
        registerReceiver(mBleAdvertiseReceiver, filter);
        loadUI();
    }

    @Override
```

```
public void onPause() {
    super.onPause();
    // stop advertising when the activity pauses
    mEchoServer.stopAdvertising();
}

@Override
public void onResume() {
    super.onResume();
    initializeBluetooth();
}

@Override
public void onDestroy() {
    super.onDestroy();
    unregisterReceiver(mBleAdvertiseReceiver);
}

/**
 * Load UI components
 */
public void loadUI() {
    mAdvertisingNameTV = (TextView) findViewById(R.id.advertising_name);
    mCharacteristicLogTV = (TextView) findViewById(R.id.characteristic_log);
    mBluetoothOnSwitch = (Switch) findViewById(R.id.bluetooth_on);
    mCentralConnectedSwitch = (Switch) findViewById(R.id.central_connected);
    mAdvertisingNameTV.setText(EchoServer.ADVERTISING_NAME);
}

/**
 * Initialize the Bluetooth Radio
 */
public void initializeBluetooth() {
    // reset connection variables
    try {
```

```

        mEchoServer = new EchoServer(this, mBlePeripheralCallback);
    } catch (Exception e) {
        Toast.makeText(
            this,
            "Could not initialize bluetooth", Toast.LENGTH_SHORT
        ).show();
        Log.e(TAG, e.getMessage());
        finish();
    }
    mBluetoothOnSwitch.setChecked(
        mEchoServer.getBlePeripheral().getBluetoothAdapter().isEnabled()
    );
    // should prompt user to open settings if Bluetooth is not enabled.
    if (!mEchoServer.getBlePeripheral().getBluetoothAdapter().isEnabled())
    {
        Intent enableBtIntent = new Intent(
            BluetoothAdapter.ACTION_REQUEST_ENABLE
        );
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    } else {
        startAdvertising();
    }
}

/**
 * Start advertising Peripheral
 */
public void startAdvertising() {
    Log.v(TAG, "starting advertising...");
    try {
        mEchoServer.startAdvertising();
    } catch (Exception e) {
        Log.e(TAG, "problem starting advertising");
    }
}

```

```

/**
 * Event trigger when Central has connected
 *
 * @param bluetoothDevice
 */
public void onBleCentralConnected(final BluetoothDevice bluetoothDevice) {
    mCentralConnectedSwitch.setChecked(true);
}

/**
 * Event trigger when Central has disconnected
 * @param bluetoothDevice
 */
public void onBleCentralDisconnected(
    final BluetoothDevice bluetoothDevice
) {
    mCentralConnectedSwitch.setChecked(false);
}

/**
 * Event trigger when Characteristic has been written to
 *
 * @param value the byte value being written
 */
public void onBleMessageWritten(final byte[] value) {
    mCharacteristicLogTV.append("\n");
    try {
        mCharacteristicLogTV.append(
            new String(value, EchoServer.CHARSET)
        );
    } catch (Exception e) {
        Log.e(TAG, "error converting byte array to string");
    }
    // scroll to bottom of TextView
    final int scrollAmount =
        mCharacteristicLogTV.getLayout().getLineTop(

```

```

        mCharacteristicLogTV.getLineCount()
    ) - mCharacteristicLogTV.getHeight();
if (scrollAmount > 0) {
    mCharacteristicLogTV.scrollTo(0, scrollAmount);
} else {
    mCharacteristicLogTV.scrollTo(0, 0);
}

}

/***
 * When the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mBleAdvertiseReceiver = \
    new AdvertiseReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
            final int state = intent.getIntExtra(
                BluetoothAdapter.EXTRA_STATE,
                BluetoothAdapter.ERROR
            );
            switch (state) {
                case BluetoothAdapter.STATE_OFF:
                    Log.v(TAG, "Bluetooth turned off");
                    initializeBluetooth();
                    break;
                case BluetoothAdapter.STATE_TURNING_OFF:
                    break;
                case BluetoothAdapter.STATE_ON:
                    Log.v(TAG, "Bluetooth turned on");
                    startAdvertising();
                    break;
                case BluetoothAdapter.STATE_TURNING_ON:
                    break;
            }
        }
    }
}

```

```
        }

    }

};

/***
 * Respond to changes to the Bluetooth Peripheral state
 */
private final EchoServerCallback mBlePeripheralCallback = \
    new EchoServerCallback() {
    public void onCentralConnected(final BluetoothDevice bluetoothDevice) {
        Log.v(TAG, "Central connected");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleCentralConnected(bluetoothDevice);
            }
        });
    }

    public void onCentralDisconnected(
        final BluetoothDevice bluetoothDevice
    ) {
        Log.v(TAG, "Central disconnected");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleCentralDisconnected(bluetoothDevice);
            }
        });
    }

    public void onMessageWritten(final byte[] value) {
        Log.v(
            TAG,
            "Characteristic written: " + DataConverter.bytesToHex(value)
        );
        runOnUiThread(new Runnable() {
```

```

        @Override
        public void run() {
            onBleMessageWritten(value);
        }
    );
}
}

```

In the MainActivity layout, three TextViews and two Switches. One TextView displays the Advertised name of the Peripheral and the other displays a log of the write Characteristic's value. One switch shows the state of the Bluetooth radio and the other shows the connectivity state of the EchoServer.

### **Example 12-21. res/layout/activity\_main.xml**

```

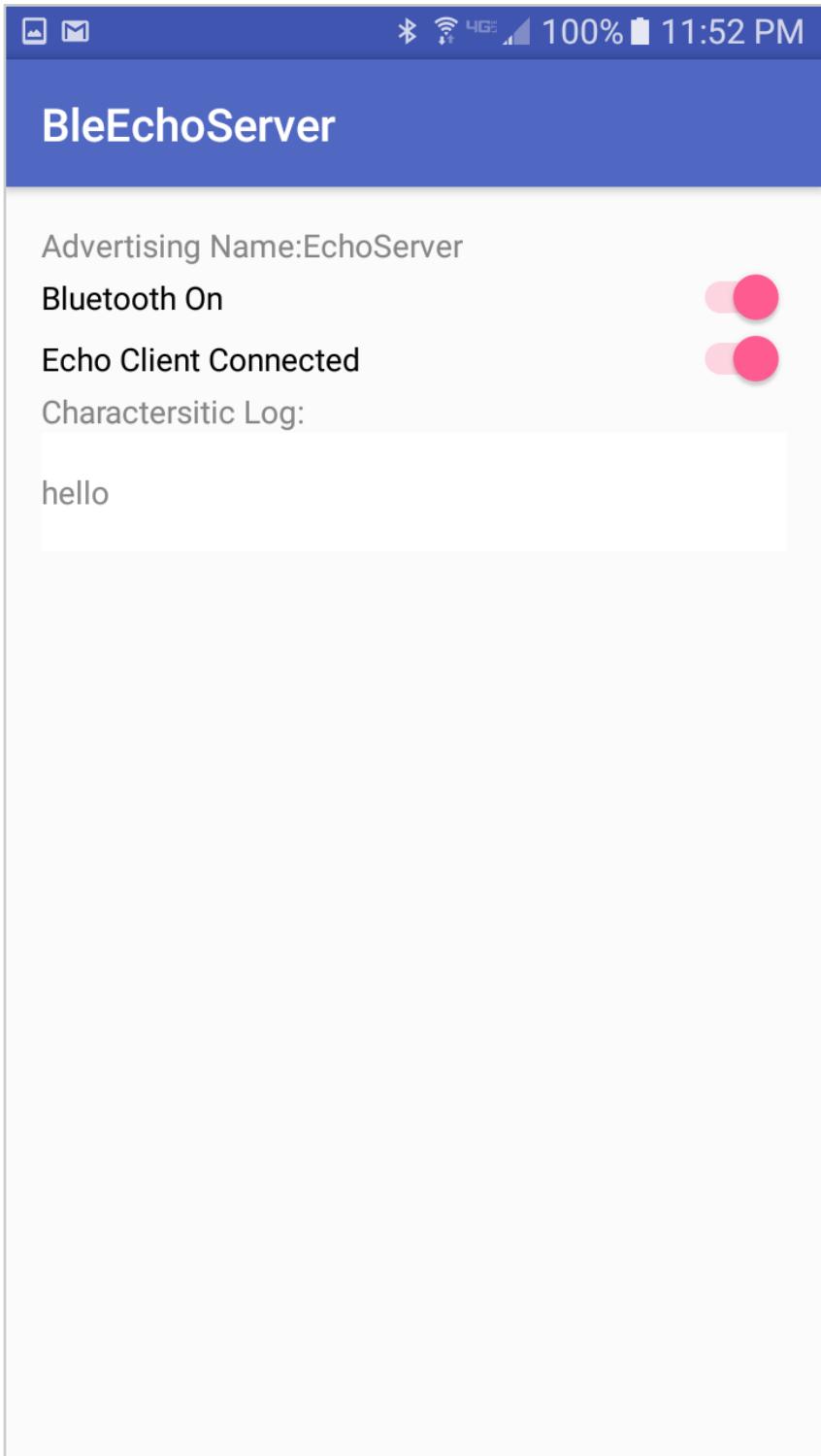
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
    <RelativeLayout

```

```
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main"
    tools:context=".MainActivity">
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/advertising_name_label" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/advertising_name" />
    </LinearLayout>
    <Switch
        android:clickable="false"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/bluetooth_on"
        android:id="@+id/bluetooth_on" />
    <Switch
        android:clickable="false"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:text="@string/central_connected"
        android:id="@+id/central_connected" />
<TextView
    android:text="@string/characteristic_log"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
<TextView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#ffffff"
    android:id="@+id/characteristic_log" />
</LinearLayout>
</RelativeLayout>
</android.support.design.widget.CoordinatorLayout>
```

The resulting Peripheral App can respond to incoming Characteristic write requests by echoing the value back through a second Characteristic, so that the connected Central can read it back ([Figure 12-5](#)).



**Figure 12-5. App screen showing interface to read from and write to the Echo Server**

## Example code

The code for this chapter is available online

at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter12>

# Project: Remote Control LED

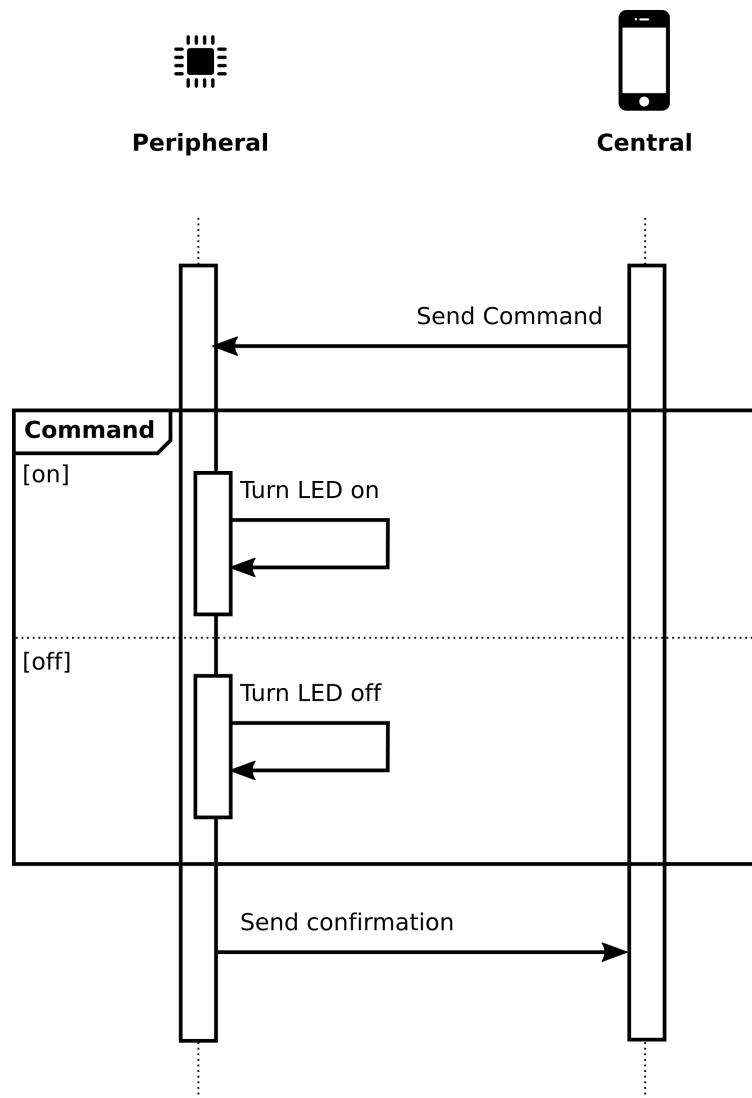
So far, this book has worked a lot with text data rather than binary data, because it's easy to text without using specialized tools such as oscilloscopes or logic analyzers.

Most real-world projects transmit binary instead of text. Binary is much more efficient in transmitting information.

Because binary data it is the language of computers, it is easier to work with than text. There is no need to worry about character sets, null characters, or cut-off words.

This project will show how to remotely control an LED on a Peripheral using software on a Central.

The LED Remote works like this ([Figure 13-1](#)).



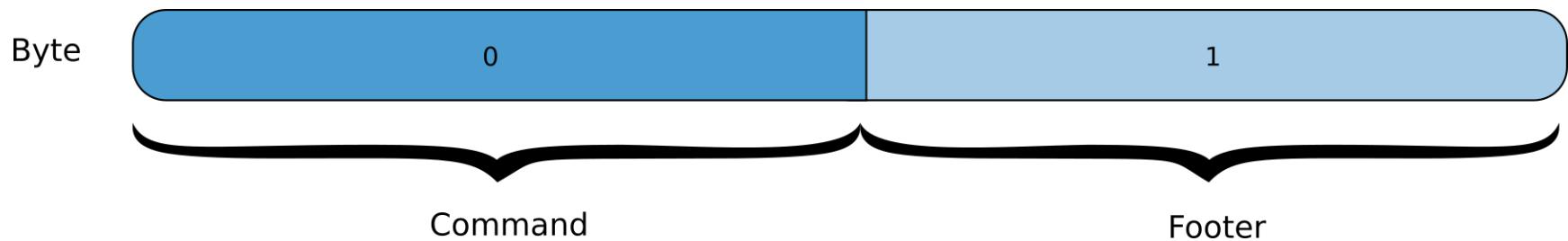
**Figure 13-1. How a Remote Control LED works**

In all the other examples, text was being sent between Central and Peripheral.

In order for the Central and Peripheral to understand each other, they need shared language between them. In this case, a data packet format.

## Sending Commands to Peripheral

When the Central sends a message, it should be able to specify if it is sending a command or an error. We can do this in two bytes, like this (Figure 13-2).



**Figure 13-2. Packet structure for commands**

The Peripheral reads the footer byte of the incoming message to determine the type of message, i.e., an error or a command. For example, define the message types as:

**Table 13-1. Footer Values**

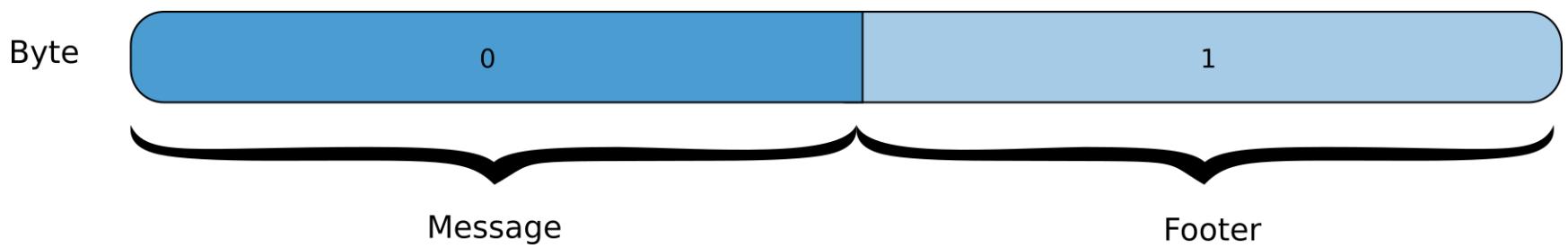
Name	Value	Description
<b>bleResponseError</b>	0	The Central is sending an error
<b>bleResponseConfirmation</b>	1	The Central is sending a confirmation
<b>bleResponseCommand</b>	2	The Central is sending a command

The Peripheral reads the first byte to determine the type of error or command. For example, define the commands as:

**Table 13-2. Command Values**

Name	Value	Description
<b>bleCommandLedOff</b>	1	Turn off the Peripheral's LED
<b>bleCommandLedOn</b>	2	Turn on the Peripheral's LED

The Peripheral then responds to the Central with a status message regarding the success or failure to execute the command. This can also be expressed as two bytes ([Figure 13-3](#)).



**Figure 13-3. Packet structure for responses**

If the Peripheral sends a confirmation that the LED state has changed, then the Central inspects the first byte of the message to determine what the current state of the Peripheral's LED is:

**Table 13-2. Confirmation Values**

Name	Value	Description
ledStateOff	1	The Peripheral's LED is off
ledStateOn	2	The Peripheral's LED is on

In this way, a common language is established between the Central and the Peripheral.

## Gatt Profile

The Bluetooth Low Energy specification provides a special Service, the Automation IO Service (0x1815), specifically for remote control devices such as this.

It is a best practice to use each Characteristic for a single purpose. For this reason, Characteristic 0x2a56 will be used for sending commands to the Peripheral and Characteristic 0x2a57 will be used for responses from the Peripheral:

**Table 13-4. Characteristic Usages**

UUID	Use
0x2a56	Send commands from Central to Peripheral
0x2a57	Send responses from Peripheral to Central

## Programming the Central

This project shows how to send commands to a Peripheral from a Central.

### GATT Profile

The GATT Profile will be set up as a digital input/output under the Automation IO (0x1815) Service, which commands to and responses from the Peripheral on separate Characteristics.

```
public static final UUID SERVICE_UUID = \
    UUID.fromString("00001815-0000-1000-8000-00805f9b34fb");
public static final UUID COMMAND_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
public static final UUID RESPONSE_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");
```

# Data Formatting

In order to read and write binary commands to the Peripheral, it and the Central must understand the same messages and formatting.

```
private static final int TRANSMISSION_LENGTH = 2;

/** Sending commands */
public static final int COMMAND_FOOTER_POSITION = 1;
public static final int COMMAND_DATA_POSITION = 0;

public static final byte COMMAND_FOOTER = 1;

public static final byte COMMAND_LED_OFF = 2;
public static final byte COMMAND_LED_ON = 1;

/** Receiving responses */
public static final int RESPONSE_FOOTER_POSITION = 1;
public static final int RESPONSE_DATA_POSITION = 0;

public static final byte RESPONSE_TYPE_ERROR = 0;
public static final byte RESPONSE_TYPE_CONFIRMATION = 1;

public static final int RESPONSE_LED_STATE_ERROR = 1;
public static final int LED_STATE_ON = 1;
public static final int LED_STATE_OFF = 2;
```

# Permissions

To work with Bluetooth Low Energy, in Android, first enable the manifest section of the Android Manifest:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-feature android:name="android.hardware.bluetooth_le"
    android:required="true" />
```

To turn on the Bluetooth radio programmatically, also enable Bluetooth admin permissions:

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

## Scanning and Connecting

The Central must scan for Peripherals.

```
bluetoothAdapter.startLeScan(scanCallback);
```

Whenever a device is discovered, the LeScanCallback.onLeScan callback is triggered. The Central can use this to search for a Bluetooth Peripheral with a known name.

```
if (device.getName() != null) {  
    if (device.getName().equals(blueoothDeviceName)) {  
        connectToDevice(device);  
    }  
}
```

## Connecting

Once a desired Bluetooth Peripheral is found, the Central can connect to it.

```
bluetoothGatt = device.connectGatt(context, false, gattCallback);
```

This process triggers a series of callbacks in BluetoothGattCallback. These callbacks activate the connection and discover the GATT profile of the Peripheral.

**Table 0. BluetoothGattCallback**

Event	Description
<b>onConnectionStateChange</b>	Triggered when a BLE Peripheral connects or disconnects
<b>onServicesDiscovered</b>	Triggered when GATT Services are discovered
<b>onCharacteristicRead</b>	Triggered when data has been downloaded from a GATT Characteristic
<b>onCharacteristicWrite</b>	Triggered when data has been uploaded to a GATT Characteristic
<b>onCharacteristicChanged</b>	Triggered when a GATT Characteristic's data has changed

## Discovering Characteristics

The configuration of Services, Characteristics on the Peripheral, and which ones can be written to or read from is called the GATT profile. Upon connection, the Client can learn about this in the `onServicesDiscovered` callback by asking for specific information about the discovered Services and Characteristics.

In this example, the IDs of the Characteristic used for communication are already known.

```
BluetoothGattService service = bluetoothGatt.getService(SERVICE_UUID);
BluetoothGattCharacteristic commandCharacteristic =
    service.getCharacteristic(COMMAND_CHARACTERISTIC_UUID);
BluetoothGattCharacteristic responseCharacteristic =
    service.getCharacteristic(RESPONSE_CHARACTERISTIC_UUID);
```

The Central can check if a Characteristic has write permission:

```
int commandP = commandCharacteristic.getProperties();
```

```
boolean isWritable = commandProperties &  
    (BluetoothGattCharacteristic.PROPERTY_WRITE |  
     BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE);
```

It can check if a Characteristic provides read permission:

```
int responseProperties = responseCharacteristic.getProperties();  
boolean isReadable = responseProperties &  
    BluetoothGattCharacteristic.PROPERTY_READ;
```

It can also check if the Characteristic can send notifications.

```
int isNotifiable = responseProperties &  
    BluetoothGattCharacteristic.PROPERTY_NOTIFY;
```

## Subscribe to Notifications

If the Characteristic can send notifications, the Central will subscribe to notifications by both setting notifications as enabled and by writing two bytes to the Client Characteristic Configuration (0x2902) descriptor of the Characteristic.

```
bluetoothGatt.setCharacteristicNotification(responseCharacteristic, enabled);  
BluetoothGattDescriptor descriptor = \  
    responseCharacteristic.getDescriptor(NOTIFY_DESCRIPTOR_UUID);  
if (enabled) {  
    descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);  
} else {  
    descriptor.setValue(BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE)  
}  
bluetoothGatt.writeDescriptor(descriptor);
```

# Writing Commands

A two-byte message can be created by assembling the command and footer.

```
byte[] message = new byte[TRANSMISSION_LENGTH];  
message[COMMAND_DATA_POSITION] = command;  
message[COMMAND_FOOTER_POSITION] = COMMAND_FOOTER;
```

To write a message to the Characteristic, set the value of the Characteristic locally, then write it to transmit the message.

```
commandCharacteristic.setValue(message);  
bluetoothGatt.writeCharacteristic(commandCharacteristic);
```

# Reading Responses

The Central will know that the Peripheral has echoed a message back when a notification is sent. This notification triggers the `onCharacteristicChanged` callback, where we can issue a read request:

```
bluetoothGatt.readCharacteristic(responseCharacteristic);
```

The `onCharacteristicRead` callback is triggered when the Central reads data from the Characteristic. Here the Central determines the meaning of the incoming message.

```
byte dataFooter = response[RESPONSE_FOOTER_POSITION];
```

```
int ledstate = LED_STATE_ERROR;  
if (dataFooter == RESPONSE_TYPE_CONFIRMATION) {  
    ledstate = response[RESPONSE_DATA_POSITION];  
}
```

## Putting It All Together

The following code will create a single Activity App with a toggle switch that connects to a Peripheral. Once connected the user can flip the toggle back and fourth, which issues a command to the Peripheral to turn an LED on or off. The switch changes state when the App receives confirmation that the

Create a new project called LedRemote.

Create folders and classes, and XML files to reproduce the following structure ([Figure 13-4](#)).



**Figure 13-4. Project structure**

## Manifest

Request permissions to use Bluetooth Low Energy in the Manifest.

### Example 13-1. app/manifests/AndroidManifest.xml

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

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-feature android:name="android.hardware.bluetooth_le"
    android:required="true" />
...
</manifest>
```

## Resources

The App will indicate a connection status prior to connecting to the Peripheral. After connection, a switch and descriptive label titled “LED On” is displayed that enable interaction with the Central.

### Example 13-2. res/values/strings.xml

```
<resources>
    <string name="app_name">BleLedRemote</string>
    <string name="scanning">Scanning...</string>
    <string name="connecting">Connecting...</string>
    <string name="loading">Loading...</string>
    <string name="send_button">Send</string>
    <string name="response_label">Response</string>
    <string name="no_peripheral_found">Remote light not found</string>
    <string name="led_switch">Led On</string>
</resources>
```

## Objects

The BleCommManager turns the Bluetooth radio on and scans for nearby Peripherals.

### Example 13-3. java/com.example.ledremote/ble/BleCommManager

```
package com.example.ledremote.ble
public class BleCommManager {
    private static final String TAG = BleCommManager.class.getSimpleName();
    // 5 seconds of scanning time
```

```

private static final long SCAN_PERIOD = 5000;
// Andrdoid's Bluetooth Adapter
private BluetoothAdapter mBluetoothAdapter;
// Ble scanner - API >= 21
private BluetoothLeScanner mBluetoothLeScanner;
// scan timer
private Timer mTimer = new Timer();

/**
 * Initialize the BleCommManager
 *
 * @param context the Activity context
 * @throws Exception Bluetooth Low Energy is not supported
 *          on this Android device
 */
public BleCommManager(final Context context) throws Exception {
    // make sure Android device supports Bluetooth Low Energy
    if (!context.getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_BLUETOOTH_LE))
    {
        throw new Exception("Bluetooth Not Supported");
    }
    // get a reference to the Bluetooth Manager class,
    // which allows us to talk to talk to the BLE radio
    final BluetoothManager bluetoothManager = (BluetoothManager) \
        context.getSystemService(Context.BLUETOOTH_SERVICE);
    mBluetoothAdapter = bluetoothManager.getAdapter();
}

/**
 * Get the Android Bluetooth Adapter
 *
 * @return BluetoothAdapter Android Bluetooth Adapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}

```

```

}

/**
 * Scan for Peripherals
 *
 * @param bleScanCallbackv18 APIv18 compatible ScanCallback
 * @param bleScanCallbackv21 APIv21 compatible ScanCallback
 * @throws Exception
 */
public void scanForPeripherals(
    final BleScanCallbackv18 bleScanCallbackv18,
    final BleScanCallbackv21 bleScanCallbackv21) throws Exception
{
    // Don't proceed if there is already a scan in progress
    mTimer.cancel();

    // Use BluetoothAdapter.startLeScan() for Android API 18, 19, and 20
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
        // Scan for SCAN_PERIOD milliseconds.
        // at the end of that time, stop the scan.

        new Thread() {
            @Override
            public void run() {
                mBluetoothAdapter.startLeScan(bleScanCallbackv18);
                try {
                    Thread.sleep(SCAN_PERIOD);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                mBluetoothAdapter.stopLeScan(bleScanCallbackv18);
            }
        }.start();
        // alert the system that BLE scanning has
        // stopped after SCAN_PERIOD milliseconds
        mTimer = new Timer();
        mTimer.schedule(new TimerTask() {
            @Override

```

```
public void run() {
    stopScanning(bleScanCallbackv18, bleScanCallbackv21);
}
}, SCAN_PERIOD);
} else { // use BluetoothLeScanner.startScan() for API >= 21 (Lollipop)
final ScanSettings settings = new ScanSettings.Builder()
    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
    .build();
final List<ScanFilter> filters = new ArrayList<ScanFilter>();
mBluetoothLeScanner = mBluetoothAdapter.getBluetoothLeScanner();

new Thread() {
    @Override
    public void run() {
        mBluetoothLeScanner.startScan(
            filters,
            settings,
            bleScanCallbackv21
        );
        try {
            Thread.sleep(SCAN_PERIOD);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        mBluetoothLeScanner.stopScan(bleScanCallbackv21);
    }
}.start();
// alert the system that BLE scanning has
// stopped after SCAN_PERIOD milliseconds
mTimer = new Timer();
mTimer.schedule(new TimerTask() {
    @Override
    public void run() {
        stopScanning(bleScanCallbackv18, bleScanCallbackv21);
    }
}, SCAN_PERIOD);
```

```

        }

    }

    /**
     * Stop Scanning
     *
     * @param bleScanCallbackv18 APIv18 compatible ScanCallback
     * @param bleScanCallbackv21 APIv21 compatible ScanCallback
     */
    public void stopScanning(
        final BleScanCallbackv18 bleScanCallbackv18,
        final BleScanCallbackv21 bleScanCallbackv21
    ) {
        mTimer.cancel();
        // propagate the onScanComplete through the system
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
            mBluetoothAdapter.stopLeScan(bleScanCallbackv18);
            bleScanCallbackv18.onScanComplete();
        } else {
            mBluetoothLeScanner.stopScan(bleScanCallbackv21);
            bleScanCallbackv21.onScanComplete();
        }
    }
}

```

BleScanCallbackv18 is one of two callback handlers for BleCommManager to process Peripheral discovered during as scan. Which callback class used depends on the API version of the phone the App gets installed on.

#### **Example 13-4. java/com.example.ledremote/ble/callbacks/BleScanCallbackv18**

```

package com.example.ledremote.ble.callbacks
public class BleScanCallbackv18 implements BluetoothAdapter.LeScanCallback {
    /**
     * New Perpheral found.
     *

```

```

    * @param bluetoothDevice The Peripheral Device
    * @param rssi The Peripheral's RSSI indicating how strong the
    *             radio signal is
    * @param scanRecord Other information about the scan result
    */
    //@Override
    public abstract void onLeScan(
        final BluetoothDevice bluetoothDevice,
        int rssi,
        byte[] scanRecord
    );

    /**
     * BLE Scan complete
     */
    public abstract void onScanComplete();
}

```

BleScanCallbackv21 handles Peripheral discovery and Bluetooth radio state changes in newer Android devices.

### **Example 13-5. java/com.example.ledremote/ble/callbacks/BleScanCallbackv21**

```

package com.example.ledremote.ble.callbacks
public class BleScanCallbackv21 extends ScanCallback {
    /**
     * New Perpheral found.
     *
     * @param callbackType int: Determines how this callback was triggered.
     *                     Could be one of CALLBACK_TYPE_ALL_MATCHES,
     *                     CALLBACK_TYPE_FIRST_MATCH or CALLBACK_TYPE_MATCH_LOST
     * @param result a Bluetooth Low Energy Scan Result,
     *             containing the Bluetooth Device, RSSI, and other information
     */
    @Override
    public abstract void onScanResult(int callbackType, ScanResult result);
}

```

```

/**
 * New Peripherals found.
 *
 * @param results List: List of scan results that are previously scanned.
 */
@Override
public abstract void onBatchScanResults(List<ScanResult> results);

/**
 * Problem initializing the scan. See the error code for reason
 *
 * @param errorCode    int: Error code (one of SCAN_FAILED_*)
 *                      for scan failure.
 */
@Override
public abstract void onScanFailed(int errorCode);

/**
 * Scan has completed
 */
public abstract void onScanComplete();
}

```

The BleRemoteLed handles communication with the Bluetooth Peripheral. It processes incoming responses and sends out formatted commands.

### **Example 13-6. java/com.example.ledremote/ble/BleRemoteLed**

```

package com.example.ledremote.ble

public class BleRemoteLed {
    private static final String TAG = BleRemoteLed.class.getSimpleName();

    public static final String CHARACTER_ENCODING = "ASCII";

    private BluetoothDevice mBluetoothDevice;
}

```

```
private BluetoothGatt mBluetoothGatt;
private BleRemoteLedCallback mBleRemoteLedCallback;
private BluetoothGattCharacteristic mCommandCharacteristic,
    mResponseCharacteristic;
private Context mContext;

/** Bluetooth Device stuff */
public static final String ADVERTISED_NAME = "LedRemote";
public static final UUID SERVICE_UUID = \
    UUID.fromString("00001815-0000-1000-8000-00805f9b34fb");
public static final UUID COMMAND_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
public static final UUID RESPONSE_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");
public static final UUID NOTIFY_DESCRIPTOR_UUID = \
    UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");

/** Data packet */
private static final int TRANSMISSION_LENGTH = 2;

/** Sending commands */
public static final int COMMAND_FOOTER_POSITION = 1;
public static final int COMMAND_DATA_POSITION = 0;
public static final byte COMMAND_FOOTER = 1;
public static final byte COMMAND_LED_OFF = 2;
public static final byte COMMAND_LED_ON = 1;

/** Receiving responses */
public static final int RESPONSE_FOOTER_POSITION = 1;
public static final int RESPONSE_DATA_POSITION = 0;
public static final byte RESPONSE_TYPE_ERROR = 0;
public static final byte RESPONSE_TYPE_CONFIRMATION = 1;
public static final int RESPONSE_LED_STATE_ERROR = 1;
public static final int LED_STATE_ON = 1;
public static final int LED_STATE_OFF = 2;
```

```

public BleRemoteLed(
    Context context,
    BleRemoteLedCallback peripheralCallback
) {
    mContext = context;
    mBleRemoteLedCallback = peripheralCallback;
}

/**
 * Determine if the incoming value is a confirmation, or error
 *
 * @param value the incoming data value
 * @return integer message type. See @MESSAGE_TYPE_ERROR,
 *         @MESSAGE_TYPE_CONFIRMATION, and @MESSAGE_TYPE_COMMAND
 * @throws Exception
 */
public int getResponseType(byte[] value) throws Exception {
    byte dataFooter = value[RESPONSE_FOOTER_POSITION];
    int returnValue = RESPONSE_LED_STATE_ERROR;
    if (dataFooter == RESPONSE_TYPE_CONFIRMATION) {
        returnValue = value[RESPONSE_DATA_POSITION];
    }
    return returnValue;
}

/**
 * Connect to a Peripheral
 *
 * @param bluetoothDevice the Bluetooth Device
 * @return a connection to the BluetoothGatt
 * @throws Exception if no device is given
 */
public BluetoothGatt connect(
    BluetoothDevice bluetoothDevice) throws Exception
{
    if (bluetoothDevice == null) {

```

```

        throw new Exception("No bluetooth device provided");
    }

    mBluetoothDevice = bluetoothDevice;
    mBluetoothGatt = bluetoothDevice.connectGatt(
        mContext,
        false,
        mGattCallback
    );
    refreshDeviceCache();
    return mBluetoothGatt;
}

/***
 * Disconnect from a Peripheral
 */
public void disconnect() {
    if (mBluetoothGatt != null) {
        mBluetoothGatt.disconnect();
    }
}

/***
 * A connection can only close after a successful disconnect.
 * Be sure to use the BluetoothGattCallback.onConnectionStateChanged event
 * to notify of a successful disconnect
 */
public void close() {
    if (mBluetoothGatt != null) {
        mBluetoothGatt.close(); // close connection to Peripheral
        mBluetoothGatt = null; // release from memory
    }
}

public BluetoothDevice getBluetoothDevice() {
    return mBluetoothDevice;
}

```

```

/**
 * Clear the GATT Service cache.
 *
 * New in this chapter
 *
 * @return <b>true</b> if the device cache clears successfully
 * @throws Exception
 */
public boolean refreshDeviceCache() throws Exception {
    Method localMethod = mBluetoothGatt.getClass().getMethod(
        "refresh",
        new Class[0]
    );
    if (localMethod != null) {
        boolean bool = ((Boolean) localMethod.invoke(
            mBluetoothGatt,
            new Object[0]
        )).booleanValue();
        return bool;
    }
    return false;
}

/**
 * Request a data/value read from a Ble Characteristic
 *
 * @param characteristic
 */
public void readvalueFromCharacteristic(
    final BluetoothGattCharacteristic characteristic
) {
    // Reading a characteristic requires both requesting the
    // read and handling the callback that is
    // sent when the read is successful
    mBluetoothGatt.readCharacteristic(characteristic);
}

```

```

/**
 * Turn the remote LED on;
 */
public void turnLedOn() {
    writeCommand(COMMAND_LED_ON);
}

/**
 * Turn the remote LED off.
 */
public void turnLedOff() {
    writeCommand(COMMAND_LED_OFF);
}

/**
 * Convert bytes to a hexadecimal String
 *
 * @param bytes a byte array
 * @return hexadecimal string
 */
final protected static char[] hexArray = "0123456789ABCDEF".toCharArray();
public static String bytesToHex(byte[] bytes) {
    char[] hexChars = new char[bytes.length * 2];
    for ( int j = 0; j < bytes.length; j++ ) {
        int v = bytes[j] & 0xFF;
        hexChars[j * 2] = hexArray[v >>> 4];
        hexChars[j * 2 + 1] = hexArray[v & 0x0F];
    }
    return new String(hexChars);
}

/**
 * Write a value to a Characteristic
 *
 * @param command The command being written

```

```

* @throws Exception
*/
public void writeCommand(byte command) {
    // build data packet
    byte[] data = new byte[TRANSMISSION_LENGTH];
    data[COMMAND_DATA_POSITION] = command;
    data[COMMAND_FOOTER_POSITION] = COMMAND_FOOTER;
    Log.d(TAG, "Writing Message: "+bytesToHex(data));
    mCommandCharacteristic.setValue(data);
    mBluetoothGatt.writeCharacteristic(mCommandCharacteristic);
}

/**
 * Subscribe or unsubscribe from Characteristic Notifications
 *
 * New in this chapter
 *
 * @param characteristic
 * @param isEnabled <b>true</b> for "subscribe"
 *                  <b>false</b> for "unsubscribe"
 */
public void setCharacteristicNotification(
    final BluetoothGattCharacteristic characteristic,
    final boolean isEnabled
) {
    // This is a 2-step process
    // Step 1: set the Characteristic Notification parameter locally
    mBluetoothGatt.setCharacteristicNotification(
        characteristic,
        isEnabled
    );
    // Step 2: write a descriptor to the Bluetooth GATT
    // enabling the subscription on the Peripheral
    // turns out you need to implement a delay between
    // setCharacteristicNotification and setValue.
    // maybe it can be handled with a callback,
}

```

```

// but this is an easy way to implement

final Handler handler = new Handler(Looper.getMainLooper());
handler.postDelayed(new Runnable() {
    @Override
    public void run() {
        BluetoothGattDescriptor descriptor = \
            characteristic.getDescriptor(NOTIFY_DESCRIPTOR_UUID);
        Log.v(TAG, "descriptor: "+descriptor);
        if (isEnabled) {
            descriptor.setValue(
                BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE
            );
        } else {
            descriptor.setValue(
                BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE
            );
        }
        mBluetoothGatt.writeDescriptor(descriptor);
    }
}, 10);

}

/**
 * Check if a characteristic supports write permissions
 * @return Returns <b>true</b> if property is writable
 */
public static boolean isCharacteristicWritable(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        (BluetoothGattCharacteristic.PROPERTY_WRITE | \
        BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE)) != 0;
}

/**
 * Check if a characteristic has read permissions

```

```

*
 * @return Returns <b>true</b> if property is Readable
 */
public static boolean isCharacteristicReadable(
    BluetoothGattCharacteristic characteristic
) {
    return ((characteristic.getProperties() & \
        BluetoothGattCharacteristic.PROPERTY_READ) != 0);
}

/**
 * Check if a Characteristic supports Notifications
 *
 * @return Returns <b>true</b> if property is supports notification
 */
public static boolean isCharacteristicNotifiable(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY) != 0;
}

/**
 * BluetoothGattCallback handles connections, state changes, reads,
 * writes, and GATT profile listings to a Peripheral
 *
 */
private final BluetoothGattCallback mGattCallback = \
    new BluetoothGattCallback()
{
    /**
     * Characterstic successfully read
     *
     * @param gatt connection to GATT
     * @param characteristic The characterstic that was read
     * @param status the status of the operation

```

```

*/
@Override
public void onCharacteristicRead(
    final BluetoothGatt gatt,
    final BluetoothGattCharacteristic characteristic,
    int status
) {
    if (status == BluetoothGatt.GATT_SUCCESS) {
        final byte[] message = characteristic.getValue();
        Log.v(
            TAG,
            "Message received: " + BleRemoteLed.bytesToHex(message)
        );
        int ledState = RESPONSE_TYPE_ERROR;
        // we are looking to see if the remote command worked
        try {
            ledState = getResponseType(message);
        } catch (Exception e) {
            Log.e(
                TAG,
                "Could not discern message type from incoming message"
            );
        }
        switch (ledState) {
            case BleRemoteLed.LED_STATE_ON:
            case BleRemoteLed.LED_STATE_OFF:
            {
                mBleRemoteLedCallback.ledStateChanged(ledState);
            }
            break;
            default:
                mBleRemoteLedCallback.ledError();
        }
    }
}

```

```

/**
 * Characteristic was written successfully. update the UI
 *
 * @param gatt Connection to the GATT
 * @param characteristic The Characteristic that was written
 * @param status write status
 */
@Override
public void onCharacteristicWrite(
    BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic, int status
) {
    if (status == BluetoothGatt.GATT_SUCCESS) {
        Log.v(TAG, "command written");
        mBleRemoteLedCallback.commandWritten();
    } else {
        Log.e(TAG, "problem writing characteristic");
    }
}

/**
 * Charactersitic value changed. Read new value.
 * @param gatt Connection to the GATT
 * @param characteristic The Characterstic
 */
@Override
public void onCharacteristicChanged(
    BluetoothGatt gatt,
    final BluetoothGattCharacteristic characteristic
) {
    Log.v(TAG, "characteristic state changed");
    readValueFromCharacteristic(characteristic);
}

/**
 * Peripheral connected or disconnected. Update UI

```

```

    * @param bluetoothGatt Connection to GATT
    * @param status status of the operation
    * @param newState new connection state
    */
@Override
public void onConnectionStateChange(
    final BluetoothGatt bluetoothGatt,
    int status,
    int newState
) {
    if (newState == BluetoothProfile.STATE_CONNECTED) {
        Log.v(TAG, "connected");
        bluetoothGatt.discoverServices();
    } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        Log.e(TAG, "Disconnected from device");
        mBleRemoteLedCallback.disconnected();
        disconnect();
    }
}

/**
 * GATT Profile discovered. Update UI
 * @param bluetoothGatt connection to GATT
 * @param status status of operation
*/
@Override
public void onServicesDiscovered(
    final BluetoothGatt bluetoothGatt,
    int status
) {
    Log.v(TAG, "SERVICE DISCOVERED!: ");
    // if services were discovered, then let's
    // iterate through them and display them on screen
    if (status == BluetoothGatt.GATT_SUCCESS) {
        // check if there are matching services and characteristics
        List<BluetoothGattService> gattServices = \

```

```

        bluetoothGatt.getServices();

        for (BluetoothGattService gattService : gattServices) {
            Log.v(TAG, "service: "+gattService.getUuid().toString());
            // while we are here, let's ask for this
            // service's characteristics:
            List<BluetoothGattCharacteristic> characteristics = \
                gattService.getCharacteristics();
            for (BluetoothGattCharacteristic characteristic : \
                characteristics) {
                if (characteristic != null) {
                    Log.v(TAG, characteristic.getUuid().toString());
                }
            }
        }

        BluetoothGattService service = \
            bluetoothGatt.getService(BleRemoteLed.SERVICE_UUID);
        if (service != null) {
            Log.v(TAG, "service found");
            mCommandCharacteristic = \
                service.getCharacteristic(
                    BleRemoteLed.COMMAND_CHARACTERISTIC_UUID
                );
            mResponseCharacteristic = \
                service.getCharacteristic(
                    BleRemoteLed.RESPONSE_CHARACTERISTIC_UUID
                );
            if (isCharacteristicNotifiable(mResponseCharacteristic)) {
                setCharacteristicNotification(
                    mResponseCharacteristic,
                    true
                );
            }
        }
    } else {
        Log.v(

```

```

        TAG,
        "Something went wrong while discovering GATT " + \
            "services from this device"
    );
}

mBleRemoteLedCallback.connected();
}

};

}

```

The BleRemoteLedCallback alerts changes to the BleRemoteLed state, including connections, disconnections, and changes in the state of the LED.

### **Example 13-8. java/com.example.ledremote/ble/callbacks/BleRemoteLedCallback**

```

package com.example.ledremote.ble.callbacks
public abstract class BleRemoteLedCallback {

    /**
     * Led Remote connected
     */
    public abstract void connected();

    /**
     * Led Remote disconnected
     */
    public abstract void disconnected();

    /**
     * Led Remote received command
     */
    public abstract void commandwritten();

    /**
     * Led Remote successfully changed led state
     */

```

```

    * @param ledstate the LED state
    */
    public abstract void ledStateChanged(final int ledstate);

    /**
     * Led Remote experienced an error
     */
    public abstract void ledError();
}


```

## Activities

The Main activity will have a switch and a label describing what the switch does. When the user toggles the switch, the Main activity will issue a command to the Peripheral to turn its LED on or off.

### **Example 13-9. java/example.com.ledremote/MainActivity.java**

```

package example.com.ledremote

public class MainActivity extends AppCompatActivity {
    /** Constants */
    private static final String TAG = MainActivity.class.getSimpleName();
    private final static int REQUEST_ENABLE_BT = 1;

    /** Bluetooth Stuff */
    private BleCommManager mBleCommManager;
    private BleRemoteLed mBleRemoteLed;

    /** UI Stuff */
    private MenuItem mProgressSpinner;
    private TextView mDeviceNameTV, mDeviceAddressTV;
    private Switch mLedSwitch;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

```

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

Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

// notify when bluetooth is turned on or off
IntentFilter filter = new IntentFilter(
    BluetoothAdapter.ACTION_STATE_CHANGED
);
registerReceiver(mReceiver, filter);
loadUI();
mBleRemoteLed = new BleRemoteLed(this, mBleRemoteLedCallback);
}

@Override
public void onResume() {
    super.onResume();
}

/**
 * Unregister the bluetooth radio alerts on pause
 */
@Override
public void onPause() {
    super.onPause();
    stopScan();
    disconnect();
    unregisterReceiver(mReceiver);
}

/**
 * Load UI components
 */
public void loadUI() {
    mDeviceNameTV = (TextView) findViewById(R.id.advertise_name);
```

```

        mDeviceAddressTV = (TextView) findViewById(R.id.mac_address);
        mLedSwitch = (Switch) findViewById(R.id.led_switch);
        mLedSwitch.setVisibility(View.GONE);
    }

    /**
     * Create a menu
     * @param menu The menu
     * @return <b>true</b> if processed successfully
     */
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu;
        // this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        mProgressSpinner = menu.findItem(R.id.scan_progress_item);
        mProgressSpinner.setVisible(true);
        initializeBluetooth();
        return true;
    }

    /**
     * Initialize the Bluetooth Radio
     */
    public void initializeBluetooth() {
        try {
            mBleCommManager = new BleCommManager(this);
        } catch (Exception e) {
            Log.e(TAG, "Could not initialize bluetooth");
            Log.e(TAG, e.getMessage());
            finish();
        }
        // should prompt user to open settings if bluetooth is not enabled.
        if (!mBleCommManager.getBluetoothAdapter().isEnabled()) {
            Intent enableBtIntent = new Intent(
                BluetoothAdapter.ACTION_REQUEST_ENABLE

```

```

    );
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
} else {
    startScan();
}
}

/**
 * Start scanning for Peripherals
 */
private void startScan() {
    mDeviceNameTV.setText(R.string.scanning);
    mProgressSpinner.setVisibility(true);
    try {
        mBleCommManager.scanForPeripherals(
            mBleScanCallbackv18,
            mBleScanCallbackv21
        );
    } catch (Exception e) {
        Log.e(TAG, "Can't create Ble Device Scanner");
    }
}

/**
 * Event trigger when new Peripheral is discovered
 */
public void onBlePeripheralDiscovered(BluetoothDevice bluetoothDevice) {
    // only add the device if
    // - it has a name, on
    // - doesn't already exist in our list, or
    // - is transmitting at a higher power (is closer)
    // than an existing device
    Log.v(TAG, "discovered peripheral: " + bluetoothDevice.getName());
    boolean addDevice = false;
    if (bluetoothDevice.getName() != null) {
        if (bluetoothDevice.getName().equals(

```

```

        mBleRemoteLed.ADVERTISED_NAME)
    ) {
        Log.v(TAG, "found Remote Led!");
        addDevice = true;
    }
}

if (addDevice) {
    stopScan();
    connectToDevice(blueoothDevice);
}
}

/**
 * Stop scanning for Peripherals
 */
public void stopScan() {
    mBleCommManager.stopScanning(mBleScanCallbackv18, mBleScanCallbackv21);
}

/**
 * Event trigger when BLE Scanning has stopped
 */
public void onBleScanStopped() {
    mProgressSpinner.setVisibility(false);
}

/**
 * Hand the Peripheral Mac Address over to the Connect Activity
 *
 * @param blueoothDevice the MAC address of the selected Peripheral
 */
public void connectToDevice(BluetoothDevice blueoothDevice) {
    mDeviceNameTV.setText(R.string.connecting);
    mProgressSpinner.setVisibility(true);
    try {
        mBleRemoteLed.connect(blueoothDevice);
    }
}

```

```

        } catch (Exception e) {
            mProgressSpinner.setVisibility(false);
            Log.e(TAG, "Error connecting to device");
        }
    }

/***
 * Disconnect from Peripheral
 */
private void disconnect() {
    mBleRemoteLed.disconnect();
    // remove callbacks
    mLedSwitch.removeCallbacks(null);
    finish();
}

/***
 * Clear the input TextView when a characteristic is successfully written to.
 */
public void onBleCommandProcessed() {
    Log.v(TAG, "Remote reported success!");
    mLedSwitch.setEnabled(true);
}

/***
 * Problem sending the command to the Peripheral. Show error
 */
public void onBleCommandError() {
    Log.e(TAG, "Remote reported an error!");
    //mLedSwitch.setChecked(!mLedswitch.isChecked());
    mLedSwitch.setEnabled(true);
    Toast.makeText(this, "", Toast.LENGTH_LONG).show();
}

/***
 * Bluetooth Peripheral connected. Update UI
*/

```

```

*/
public void onBleConnected() {
    mProgressSpinner.setVisibility(false);
    mDeviceNameTV.setText(BleRemoteLed.ADVERTISED_NAME);
    mDeviceAddressTV.setText(
        mBleRemoteLed.getBluetoothDevice().getAddress()
    );
    mProgressSpinner.setVisibility(false);
    // attach callbacks to the buttons and stuff
    mLedSwitch.setVisibility(View.VISIBLE);
    mLedSwitch.setOnCheckedChangeListener(
        new CompoundButton.OnCheckedChangeListener()
    {
        @Override
        public void onCheckedChanged(
            CompoundButton buttonView, boolean isChecked)
        {
            if (isChecked) {
                mBleRemoteLed.turnLedOn();
            } else {
                mBleRemoteLed.turnLedOff();
            }
        }
    );
}

public void onBleDisconnected() {
    mDeviceNameTV.setText("");
    mDeviceAddressTV.setText("");
    mProgressSpinner.setVisibility(false);
}

/**
 * When the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mReceiver = new AdvertiseReceiver() {

```

```
@Override
public void onReceive(Context context, Intent intent) {
    final String action = intent.getAction();
    if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
        final int state = intent.getIntExtra(
            BluetoothAdapter.EXTRA_STATE,
            BluetoothAdapter.ERROR
        );
        switch (state) {
            case BluetoothAdapter.STATE_OFF:
                Log.v(TAG, "Bluetooth turned on");
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_OFF:
                break;
            case BluetoothAdapter.STATE_ON:
                Log.v(TAG, "Bluetooth turned on");
                initializeBluetooth();
                break;
            case BluetoothAdapter.STATE_TURNING_ON:
                break;
        }
    }
};

/**
 * Use this callback for Android API 21 (Lollipop) or greater
 */
private final BleScanCallbackV21 mBleScanCallbackV21 = \
    new BleScanCallbackV21()
{
    /**
     * New Peripheral discovered
     *
     * @param callbackType int: Determines how this callback was

```

```

* triggered. Could be one of CALLBACK_TYPE_ALL_MATCHES,
* CALLBACK_TYPE_FIRST_MATCH or CALLBACK_TYPE_MATCH_LOST
* @param result a Bluetooth Low Energy Scan Result,
* containing the Bluetooth Device, RSSI, and other information
*/
@Override
public void onScanResult(int callbackType, ScanResult result) {
    BluetoothDevice bluetoothDevice = result.getDevice();
    int rssi = result.getRssi();
    onBlePeripheralDiscovered(bluetoothDevice);
}

/**
 * Several peripherals discovered when scanning in low power mode
 *
 * @param results List<ScanResult>: List of scan results that are
 * previously scanned.
*/
@Override
public void onBatchScanResults(List<ScanResult> results) {
    for (ScanResult result : results) {
        BluetoothDevice bluetoothDevice = result.getDevice();
        int rssi = result.getRssi();
        onBlePeripheralDiscovered(bluetoothDevice);
    }
}

/**
 * Scan failed to initialize
 *
 * @param errorCode int: Error code (one of SCAN_FAILED_*)
 * for scan failure.
*/
@Override
public void onScanFailed(int errorCode) {
    switch (errorCode) {

```

```

        case SCAN_FAILED_ALREADY_STARTED:
            Log.e(
                TAG,
                "Fails to start scan as BLE scan with the " + \
                "same settings is already started by the app."
            );
            break;

        case SCAN_FAILED_APPLICATION_REGISTRATION_FAILED:
            Log.e(
                TAG,
                "Fails to start scan as app cannot be registered."
            );
            break;

        case SCAN_FAILED_FEATURE_UNSUPPORTED:
            Log.e(
                TAG,
                "Fails to start power optimized scan as this " + \
                "feature is not supported."
            );
            break;

        default: // SCAN_FAILED_INTERNAL_ERROR
            Log.e(TAG, "Fails to start scan due an internal error");
    }

    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleScanStopped();
        }
    });
}

/**
 * Scan completed
 */
public void onScanComplete() {
    runOnUiThread(new Runnable() {

```

```

        @Override
        public void run() {
            onBleScanStopped();
        }
    );
}

/**
 * Use this callback for Android API 18, 19, and 20 (before Lollipop)
 */
public final BleScanCallbackV18 mBleScanCallbackV18 = \
    new BleScanCallbackV18()
{
    /**
     * New Peripheral discovered
     * @param bluetoothDevice The Peripheral Device
     * @param rssi The Peripheral's RSSI indicating how
     *             strong the radio signal is
     * @param scanRecord Other information about the scan result
     */
    @Override
    public void onLeScan(
        BluetoothDevice bluetoothDevice,
        int rssi,
        byte[] scanRecord)
    {
        onBlePeripheralDiscovered(bluetoothDevice);
    }

    /**
     * Scan completed
     */
    @Override
    public void onScanComplete() {
        runOnUiThread(new Runnable() {

```

```
        @Override
        public void run() {
            onBleScanStopped();
        }
    );
}

};

public final BleRemoteLedCallback mBleRemoteLedCallback = \
    new BleRemoteLedCallback()
{
    @Override
    public void connected() {

        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleConnected();
            }
        });
    }

    @Override
    public void disconnected() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onBleDisconnected();
            }
        });
    }

    @Override
    public void commandWritten() {
    }
}
```

```

@Override
public void ledStateChanged(final int ledstate) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCommandProcessed();
        }
    });
}

@Override
public void ledError() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            onBleCommandError();
        }
    });
}

```

In the MainActivity layout, create a Switch representing the Peripheral's LED state and a TextView to describe what the switch does.

### **Example 13-10. res/layout/activity\_main.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

```

```
    android:fitsSystemWindows="true"
    tools:context=".ConnectActivity">
<android.support.design.widget.AppBarLayout
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:theme="@style/AppTheme.AppBarOverlay">
    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />
</android.support.design.widget.AppBarLayout>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main"
    tools:context="example.com.bleledremote.MainActivity"
    android:weightSum="1">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:text="@string/loading"
        android:id="@+id/advertise_name"/>
    <TextView
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:text="@string/loading"
        android:id="@+id/mac_address"/>
    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <Switch
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/led_switch"
            android:id="@+id/led_switch"
            android:layout_weight="2" />
    </LinearLayout>
</LinearLayout>
</android.support.design.widget.CoordinatorLayout>
```

The menu displays a progress scanner in the menu when attempting a connection:

### Example 13-11. res/menu/menu\_main.xml

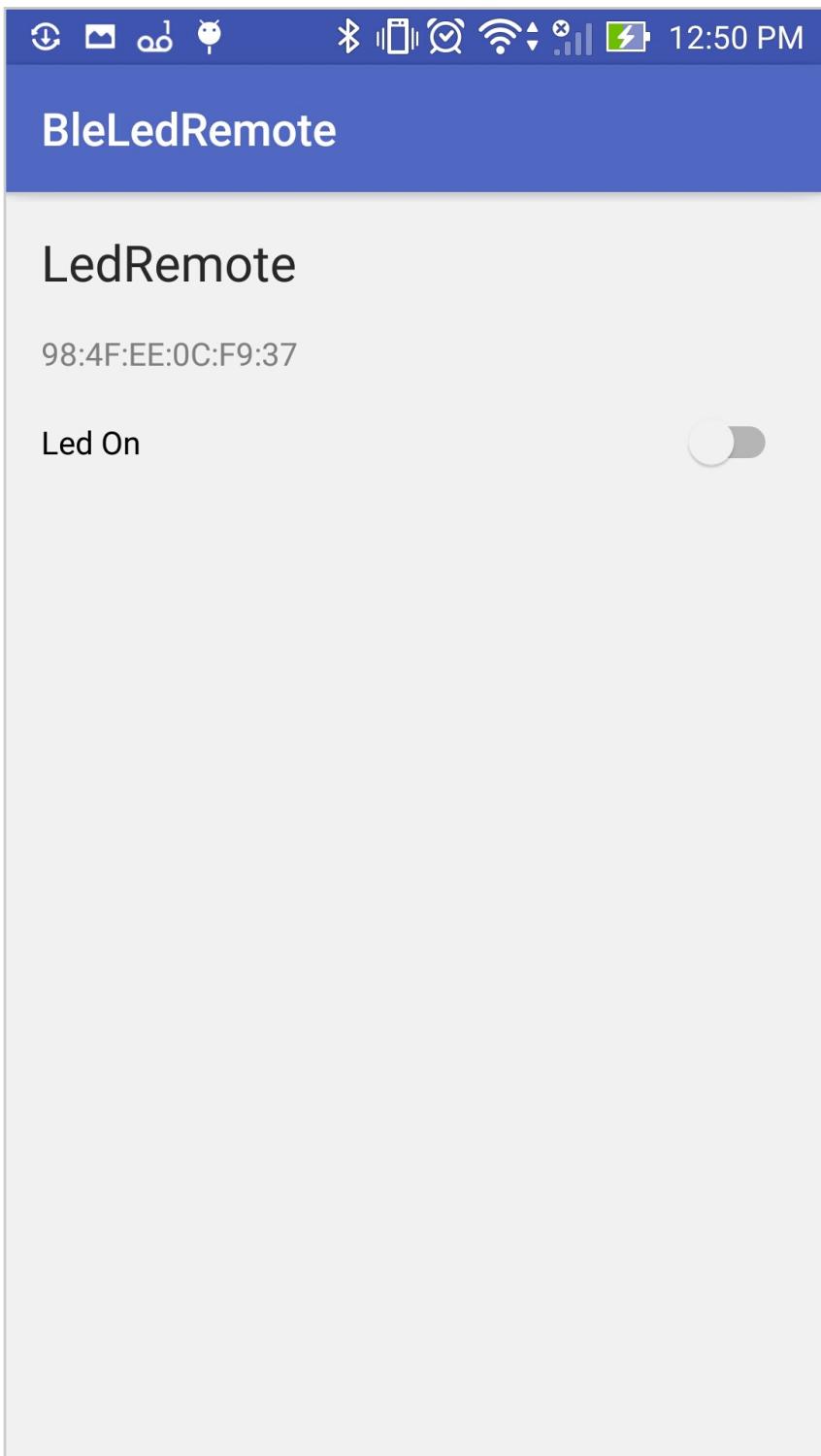
```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="example.com.bleledremote.MainActivity">
    <item
        android:id="@+id/scan_progress_item"
        android:title="@string/scanning"
        android:visible="true"
        android:orderInCategory="100"
        app:showAsAction="always"
        app:actionLayout="@layout/scanner_progress"
        android:layout_marginRight="@dimen/activity_horizontal_margin" />
</menu>
```

The progress scanner is as defined below:

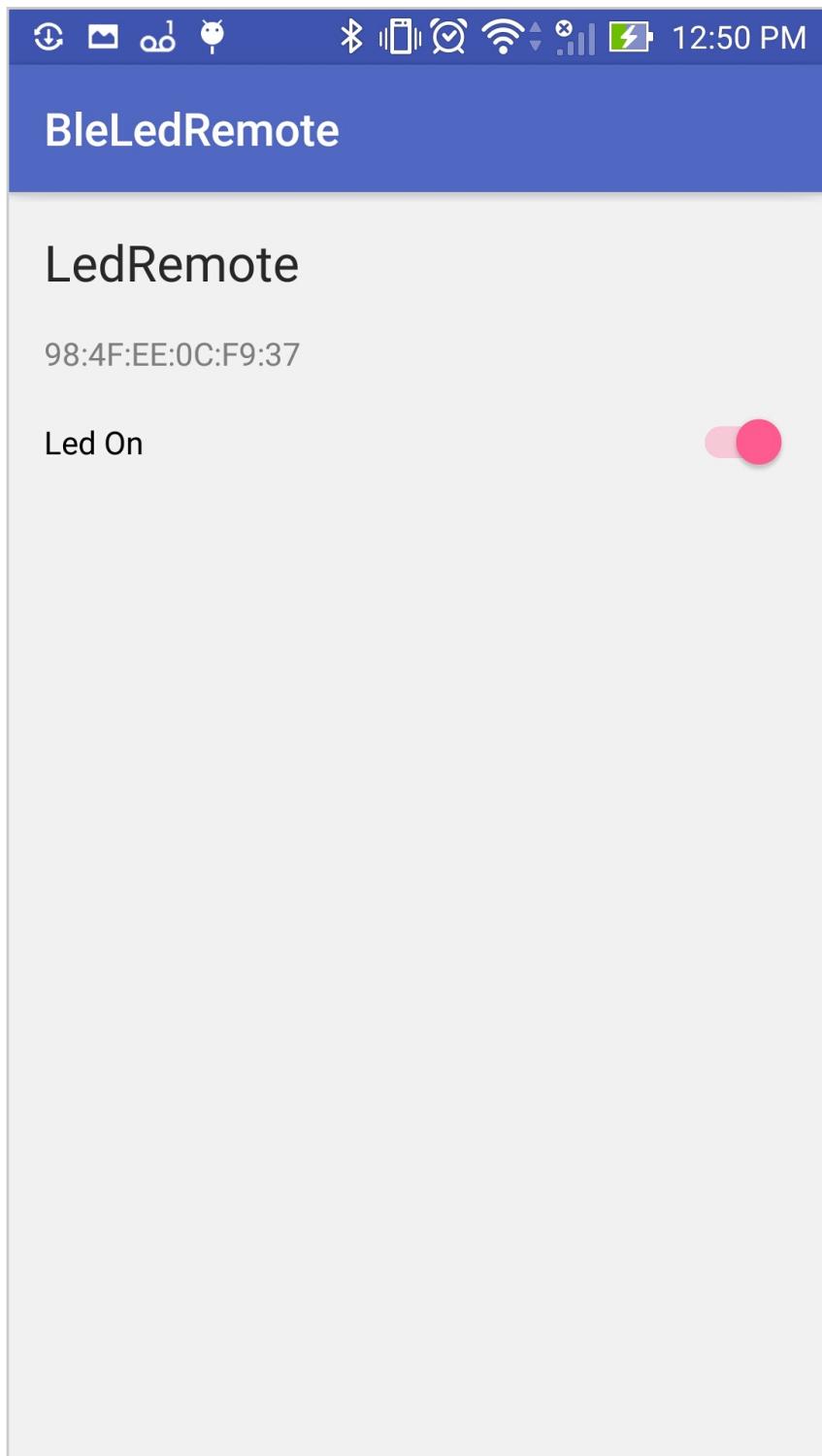
**Example 13-12. res/layout/scanner\_progress.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<ProgressBar xmlns:android="http://schemas.android.com/apk/res/android"
    style="@android:style/widget.ProgressBar.Small"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/scan_progress" />
```

The resulting App scan for and connect to a Bluetooth Low Energy Peripheral. Once connected, the user can turn the Peripheral's LED on or off using the switch. ([Figure 13-5](#)). When the LED turns on. The switch fully moves when the App receives confirmation from the Peripheral that the LED state has changed ([Figure 13-6](#)).



**Figure 13-5. App screen showing LED switch in off state**



**Figure 13-6. App screen showing LED switch in on state**

## Programming the Peripheral

This project shows how to process commands sent from a Central, and respond with status confirmations.

## GATT Profile

The GATT Profile will be set up as a digital input/output under the Automation IO (0x1815) Service, with commands to and responses from the Peripheral on separate Characteristics.

```
public static final UUID SERVICE_UUID = \
    UUID.fromString("00001815-0000-1000-8000-00805f9b34fb");
public static final UUID COMMAND_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
public static final UUID RESPONSE_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");
```

## Data Formatting

In order to read and write binary commands to the Peripheral, it and the Central must understand the same messages and formatting.

```
/** Commands */
public static final int COMMAND_FOOTER_POSITION = 1;
public static final int COMMAND_DATA_POSITION = 0;
public static final byte COMMAND_FOOTER = 1;
public static final byte COMMAND_LED_OFF = 2;
public static final byte COMMAND_LED_ON = 1;

/** Responses */
public static final int RESPONSE_FOOTER_POSITION = 1;
public static final int RESPONSE_DATA_POSITION = 0;
public static final byte RESPONSE_TYPE_ERROR = 0;
public static final byte RESPONSE_TYPE_CONFIRMATION = 1;
public static final byte RESPONSE_LED_STATE_ERROR = 1;
public static final byte LED_STATE_ON = 1;
public static final byte LED_STATE_OFF = 2;
```

## Permissions

To work with Bluetooth Low Energy, in Android, first enable the manifest section of the Android Manifest:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-feature android:name="android.hardware.bluetooth_le"
    android:required="true" />
```

To turn on the Bluetooth radio programmatically, also enable Bluetooth admin permissions:

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

Additionally, since this app controls the camera flash, permissions must be requested for those features as well:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.FLASHLIGHT" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.flash" />
```

## Advertising and GATT Profile

The Peripheral must advertise and host a GATT Profile, which will include a minimal GATT profile.

The Peripheral will host a writeable Characteristic on UUID 0x2a56 to receive commands and a read-only, notifiable Characteristic on UUID 0x2a57 to issue responses:

```
public static final UUID SERVICE_UUID = \
    UUID.fromString("0000180c-0000-1000-8000-00805f9b34fb");
```

```

public static final UUID COMMAND_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
public static final UUID RESPONSE_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");

mIOService = new BluetoothGattService(
    SERVICE_UUID,
    BluetoothGattService.SERVICE_TYPE_PRIMARY
);
mCommandCharacteristic = new BluetoothGattCharacteristic(
    COMMAND_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_WRITE,
    BluetoothGattCharacteristic.PERMISSION_WRITE
);
mResponseCharacteristic = new BluetoothGattCharacteristic(
    RESPONSE_CHARACTERISTIC_UUID,
    BluetoothGattCharacteristic.PROPERTY_READ | \
        BluetoothGattCharacteristic.PROPERTY_NOTIFY,
    BluetoothGattCharacteristic.PERMISSION_READ
);

mIOService.addCharacteristic(mCommandCharacteristic);
mIOService.addCharacteristic(mResponseCharacteristic);

// tell the BlePeripheral what the Gatt Profile is
mGattServer.addService(mIOService);

```

It will also advertise as "LedRemote" so as to be discoverable by the corresponding Central.

```

public static final String ADVERTISING_NAME = "LedRemote";

// Build Advertise settings with transmission power and advertise speed
AdvertiseSettings advertiseSettings = new AdvertiseSettings.Builder()
    .setAdvertiseMode(mAdvertisingMode)

```

```

.setTxPowerLevel(mTransmissionPower)
.setConnectable(true)
.build();

Advertiser.Builder advertiseBuilder = new Advertiser.Builder();

// set advertising name
advertiseBuilder.setIncludeDeviceName(true);
mBluetoothAdapter.setName(ADVERTISING_NAME);

// add Services to Advertising Data
List<BluetoothGattService> services = mGattServer.getServices();
for (BluetoothGattService service: services) {
    advertiseBuilder.addServiceUuid(new ParcelUuid(service.getUuid()));
}

```

## Handling Subscriptions and Writes

Once Central initiates a connection, it can subscribe to the response Characteristic, which will trigger the `onDescriptorWriteRequest()` method in `BluetoothGattServerCallback`, and it can send write requests to the command Characteristic

**Table 13-5. BluetoothGattServerCallback**

Event	Description
<code>onConnectionStateChange</code>	Central connects or disconnects
<code>onCharacteristicReadRequest</code>	Characteristic supports notifications
<code>onCharacteristicWriteRequest</code>	Central attempted to read a value from a Characteristic
<code>onDescriptorWriteRequest</code>	Central attempted to change a Descriptor in a Characteristic
<code>onNotificationSent</code>	Central was notified of a change to a Characteristic

The `onDescriptorWriteRequest()` is triggered when the Central subscribes or unsubscribes from the response Characteristic (0x2a57):

```
private final BluetoothGattServerCallback mGattServerCallback = \  
    new BluetoothGattServerCallback(){  
  
    ...  
  
    @Override  
    public void onDescriptorWriteRequest(  
        BluetoothDevice device,  
        int requestId,  
        BluetoothGattDescriptor descriptor,  
        boolean preparedwrite,  
        boolean responseNeeded,  
        int offset,  
        byte[] value  
    ) {  
        Log.v(  
            TAG,  
            "Descriptor write Request " + descriptor.getUuid() + " " + \  
            Arrays.toString(value)  
        );  
        super.onDescriptorWriteRequest(  
            device,  
            requestId,  
            descriptor,  
            preparedwrite,  
            responseNeeded,  
            offset,  
            value  
        );  
        // determine which Characteristic is being requested  
        BluetoothGattCharacteristic characteristic = \  
            descriptor.getCharacteristic();  
        // is the descriptor writeable?  
        if (isDescriptorWriteable(descriptor)) {
```

```
descriptor.setValue(value);

// was this a subscription or an unsubscription?
if (descriptor.getUuid().equals(NOTIFY_DESCRIPTOR_UUID)) {
    if (value == \
        BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE)
    {
        mBlePeripheralCallback.onCharacteristicSubscribedTo(
            characteristic
        );
    } else if (value == \
        BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE)
    {
        mBlePeripheralCallback.onCharacteristicUnsubscribedFrom(
            characteristic
        );
    }
    // send a confirmation if necessary
    if (isDescriptorReadable(descriptor)) {
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            offset,
            value
        );
    }
}
} else {
    // notify failure if necessary
    if (isDescriptorReadable(descriptor)) {
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_WRITE_NOT_PERMITTED,
            offset,
            value
        );
    }
}
```

```
        );
    }
}
};


```

When a Central writes data to the command Characteristic (0x2a56), the LedRemote will process the command and change the LED state. It will change the value of the response Characteristic (0x2a56) to match the new LED state and send a notification of that change.

```
...
@Override
public void onCharacteristicWriteRequest(
    BluetoothDevice device,
    int requestId,
    BluetoothGattCharacteristic characteristic,
    boolean preparedwrite,
    boolean responseNeeded,
    int offset,
    byte[] value
) {
    super.onCharacteristicWriteRequest(
        device,
        requestId,
        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
    Log.v(TAG, "Characteristic write request: " + Arrays.toString(value));
    // set the read characteristic value
    byte responseType = RESPONSE_TYPE_ERROR;
    byte responseState = RESPONSE_LED_STATE_ERROR;
    if (bleCommandValue[COMMAND_FOOTER_POSITION] == COMMAND_FOOTER) {
```

```

        Log.v(TAG, "Command found");

        switch (bleCommandValue[COMMAND_DATA_POSITION]) {
            case COMMAND_LED_ON:
                Log.v(TAG, "Command to turn LED on");
                turnLedOn();
                responseType = RESPONSE_TYPE_CONFIRMATION;
                responseState = LED_STATE_ON;
                break;

            case COMMAND_LED_OFF:
                Log.v(TAG, "Command to turn LED off");
                responseType = RESPONSE_TYPE_CONFIRMATION;
                responseState = LED_STATE_OFF;
                turnLedOff();
                break;

            default:
                Log.d(TAG, "Unknown incoming command");
        }
    }

    byte[] responseValue = new byte[TRANSMISSION_LENGTH];
    responseValue[RESPONSE_FOOTER_POSITION] = responseType;
    responseValue[RESPONSE_DATA_POSITION] = responseState;
    mResponseCharacteristic.setValue(responseValue);

    if (isCharacteristicWritableWithResponse(characteristic)) {
        characteristic.setValue(value);
        mGattServer.sendResponse(
            device,
            requestId,
            BluetoothGatt.GATT_SUCCESS,
            0,
            null
        );
    }

    if (isCharacteristicNotifiable(characteristic)) {
        boolean isNotifiedOfSend = false;
        mGattServer.notifyCharacteristicChanged(
            device,

```

```
        characteristic,  
        isNotifiedofSend  
    );  
}  
...  
}
```

## Putting It All Together

The following code will create a single Activity App that turns the camera flash on and off when a command is received over a writeable Characteristic. A readable Characteristic is used to respond and notify the Connected central of the updated LED state.

Create a new project called LedRemote.

Create folders and classes, and XML files to reproduce the following structure ([Figure 13-7](#)).

```
app/
  manifests/
    AndroidManifest.xml
java/
  example.com.remotedled/
    ble/
      callbacks/
        BlePeripheralCallback
        BleRemoteLedCallback
        BlePeripheral
        BleRemoteLed
      MainActivity
res/
  layout/
    activity_main.xml
  menu/
    menu_main.xml
  values/
    strings.xml
```

Figure 13-7. Project structure

## Manifest

Request permissions to use Bluetooth Low Energy in the Manifest.

### Example 13-13. app/manifests/AndroidManifest.xml

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

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-feature android:name="android.hardware.bluetooth_le"
    android:required="true" />
...
</manifest>
```

## Resources

The App will display the Peripheral name, Bluetooth radio state, connectivity with a Central, and the state of the camera flash. Each of these states will need a text label so users understand their purpose. Add the text definitions to res/values/strings.xml.

### Example 13-14. res/values/strings.xml

```
<resources>
    <string name="app_name">LedRemote</string>
    <string name="advertising_name_label">Advertising Name: </string>
    <string name="bluetooth_on">Bluetooth On</string>
    <string name="central_connected">Remote Control Connected</string>
    <string name="led_state">Led On</string>
</resources>
```

## Objects

The BlePeripheral sets up a minimal GATT Profile and handles connections, disconnections, and subscriptions:

### Example 13-15. java/com.example.remoteled/ble/BlePeripheral

```
package com.example.remoteled.ble
public class BlePeripheral {

    /** Constants */
    private static final String TAG = BlePeripheral.class.getSimpleName();
    private static final int BATTERY_STATUS_CHECK_TIME_MS = 5*60*1000; // 5 minutes
```

```

/** Peripheral and GATT Profile */
private String mPeripheralAdvertisingName;

public static final UUID NOTIFY_DESCRIPTOR_UUID = \
    UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");
// Minimal GATT Profile UUIDs
public static final UUID DEVICE_INFORMATION_SERVICE_UUID = \
    UUID.fromString("0000180a-0000-1000-8000-00805f9b34fb");
public static final UUID BATTERY_LEVEL_SERVICE = \
    UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb");
public static final UUID DEVICE_NAME_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a00-0000-1000-8000-00805f9b34fb");
public static final UUID MODEL_NUMBER_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a24-0000-1000-8000-00805f9b34fb");
public static final UUID SERIAL_NUMBER_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a04-0000-1000-8000-00805f9b34fb");
public static final UUID BATTERY_LEVEL_CHARACTERISTIC_UUID = \
    UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb");
public static final String CHARSET = "ASCII";
public static final int MAX_ADVERTISING_NAME_BYTE_LENGTH = 20;

/** Advertising settings */

// advertising mode
int mAdvertisingMode = AdvertiseSettings.ADVERTISE_MODE_LOW_LATENCY;

// transmission power mode
int mTransmissionPower = AdvertiseSettings.ADVERTISE_TX_POWER_HIGH;

/** Callback Handlers */
public BlePeripheralCallback mBlePeripheralCallback;

/** Bluetooth Stuff */
private BluetoothAdapter mBluetoothAdapter;
private BluetoothLeAdvertiser mBluetoothAdvertiser;
private BluetoothManager mBluetoothManager;

```

```

private BluetoothDevice mConnectedCentral;
private BluetoothGattServer mGattServer;
private BluetoothGattService mDeviceInformationService,
    mBatteryLevelService;
private BluetoothGattCharacteristic mDeviceNameCharacteristic,
    mModelNumberCharacteristic,
    mSerialNumberCharacteristic,
    mBatteryLevelCharacteristic;
private Context mContext;
private String mModelNumber = "";
private String mSerialNumber = "";

/**
 * Construct a new Peripheral
 *
 * @param context The Application Context
 * @param blePeripheralCallback The callback handler that
 *     interfaces with this Peripheral
 * @throws Exception Exception thrown if Bluetooth is not supported
 */
public BlePeripheral(
    final Context context,
    BlePeripheralCallback blePeripheralCallback) throws Exception
{
    mBlePeripheralCallback = blePeripheralCallback;
    mContext = context;
    // make sure Android device supports Bluetooth Low Energy
    if (!context.getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_BLUETOOTH_LE))
    {
        throw new Exception("Bluetooth Not Supported");
    }
    // get a reference to the Bluetooth Manager class, which allows us
    // to talk to talk to the BLE radio
    mBluetoothManager = (BluetoothManager) \
        context.getSystemService(Context.BLUETOOTH_SERVICE);
}

```

```

mGattServer = mBluetoothManager.openGattServer(
    context, mGattServerCallback
);

mBluetoothAdapter = mBluetoothManager.getAdapter();
// Beware: this function doesn't work on some systems
if(!mBluetoothAdapter.isMultipleAdvertisementSupported()) {
    throw new Exception ("Peripheral mode not supported");
}

mBluetoothAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();
// Use this method instead for better support
if (mBluetoothAdvertiser == null) {
    throw new Exception ("Peripheral mode not supported");
}

}

/***
 * Get the system Bluetooth Adapter
 *
 * @return BluetoothAdapter
 */
public BluetoothAdapter getBluetoothAdapter() {
    return mBluetoothAdapter;
}

/***
 * Get the GATT Server
 */
public BluetoothGattServer getGattServer() {
    return mGattServer;
}

/***
 * Get the model number
 */
public String getModelNumber() {
    return mModelNumber;
}

```

```
}

/** set the model number
 *
 */
public void setModelNumber(String modelNumber) {
    mModelNumber = modelNumber;
}

/**
 * Get the serial number
 */
public String getSerialNumber() {
    return mSerialNumber;
}

/**
 * set the serial number
 */
public void setSerialNumber(String serialNumber) {
    mSerialNumber = serialNumber;
}

/**
 * Get the actual battery level
 */
public int getBatteryLevel() {
    BatteryManager batteryManager = \
        (BatteryManager)mContext.getSystemService(BATTERY_SERVICE);
    return batteryManager.getIntProperty(
        BatteryManager.BATTERY_PROPERTY_CAPACITY
    );
}

/**
 * Set up the GATT profile

```

```

*/
public void setupDevice() {
    // build characteristics
    mDeviceInformationService = new \
        BluetoothGattService(
            DEVICE_INFORMATION_SERVICE_UUID,
            BluetoothGattService.SERVICE_TYPE_PRIMARY
        );
    mDeviceNameCharacteristic = new BluetoothGattCharacteristic(
        DEVICE_NAME_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ,
        BluetoothGattCharacteristic.PERMISSION_READ);
    mModelNumberCharacteristic = new BluetoothGattCharacteristic(
        MODEL_NUMBER_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ,
        BluetoothGattCharacteristic.PERMISSION_READ);
    mSerialNumberCharacteristic = new BluetoothGattCharacteristic(
        SERIAL_NUMBER_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ,
        BluetoothGattCharacteristic.PERMISSION_READ);
    mBatteryLevelService = new BluetoothGattService(
        BATTERY_LEVEL_SERVICE,
        BluetoothGattService.SERVICE_TYPE_PRIMARY
    );
    mBatteryLevelCharacterstic = new BluetoothGattCharacteristic(
        BATTERY_LEVEL_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ | \
            BluetoothGattCharacteristic.PROPERTY_NOTIFY,
        BluetoothGattCharacteristic.PERMISSION_READ | \
            BluetoothGattCharacteristic.PERMISSION_WRITE);
    // add Notification support to characteristic
    BluetoothGattDescriptor notifyDescriptor = new \
        BluetoothGattDescriptor(BlePeripheral.NOTIFY_DESCRIPTOR_UUID,
        BluetoothGattDescriptor.PERMISSION_WRITE | \
            BluetoothGattDescriptor.PERMISSION_READ);
    mBatteryLevelCharacterstic.addDescriptor(notifyDescriptor);
}

```

```

mDeviceInformationService.addCharacteristic(mDeviceNameCharacteristic);
mDeviceInformationService.addCharacteristic(
    mModelNumberCharacteristic
);
mDeviceInformationService.addCharacteristic(
    mSerialNumberCharacteristic
);
mBatteryLevelService.addCharacteristic(mBatteryLevelCharacterstic);
// put in fake values for the Characteristic.
mModelNumberCharacteristic.setValue(mModelNumber);
mSerialNumberCharacteristic.setValue(mSerialNumber);
// add Services to Peripheral
mGattServer.addService(mDeviceInformationService);
mGattServer.addService(mBatteryLevelService);
// update the battery level every
// BATTERY_STATUS_CHECK_TIME_MS milliseconds
TimerTask updateBatteryTask = new TimerTask() {
    @Override
    public void run() {
        mBatteryLevelCharacterstic.setValue(
            getBatteryLevel(),
            BluetoothGattCharacteristic.FORMAT_UINT8,
            0
        );
        if (mConnectedCentral != null) {
            mGattServer.notifyCharacteristicChanged(
                mConnectedCentral,
                mBatteryLevelCharacterstic,
                true
            );
        }
    }
};

Timer randomStringTimer = new Timer();
// schedule the battery update and run it once immediately
randomStringTimer.schedule(

```

```

        updateBatteryTask,
        0,
        BATTERY_STATUS_CHECK_TIME_MS
    );
}

/***
 * Set the Advertising name of the Peripheral
 *
 * @param peripheralAdvertisingName
 */
public void setPeripheralAdvertisingName(
    String peripheralAdvertisingName) throws Exception
{
    mPeripheralAdvertisingName = peripheralAdvertisingName;
    mDeviceNameCharacteristic.setValue(mPeripheralAdvertisingName);
    int advertisingNameByteLength = \
        mPeripheralAdvertisingName.getBytes(CHARSET).length;
    if (advertisingNameByteLength > MAX_ADVERTISING_NAME_BYTE_LENGTH) {
        throw new Exception(
            "Advertising name too long. Must be less than " + \
            MAX_ADVERTISING_NAME_BYTE_LENGTH + " bytes"
        );
    }
}

/***
 * Get the Advertising name of the Peripheral
 */
public String getPeripheralAdvertisingName() {
    return mPeripheralAdvertisingName;
}

/***
 * Set the Transmission Power mode
 */

```

```
public void setTransmissionPower(int transmissionPower) {
    mTransmissionPower = transmissionPower;
}

/***
 * Set the advertising mode
 */
public void setAdvertisingMode(int advertisingMode) {
    mAdvertisingMode = advertisingMode;
}

/***
 * Add a Service to the GATT Profile
 *
 * @param service the Service to add
 */
public void addService(BluetoothGattService service) {
    mGattServer.addService(service);
}

/***
 * Build the Advertising Data, including the transmission power,
 * advertising name, and Services
 *
 * @return AdvertiseData
 */
private AdvertiseData buildAdvertisingData() {
    AdvertiseData.Builder advertiseBuilder = new AdvertiseData.Builder();
    // set advertising name
    if (mPeripheralAdvertisingName != null) {
        advertiseBuilder.setIncludeDeviceName(true);
        mBluetoothAdapter.setName(mPeripheralAdvertisingName);
    }
    // add Services to Advertising Data
    List<BluetoothGattService> services = mGattServer.getServices();
    for (BluetoothGattService service: services) {
```

```

        advertiseBuilder.addServiceUuid(new ParcelUuid(service.getUuid()));
    }

    return advertiseBuilder.build();
}

/**
 * Build Advertise settings with transmission power and advertise speed
 *
 * @return AdvertiseSettings for Bluetooth Advertising
 */
private AdvertiseSettings buildAdvertiseSettings() {
    AdvertiseSettings.Builder settingsBuilder = new \
        AdvertiseSettings.Builder();
    settingsBuilder.setAdvertiseMode(mAdvertisingMode);
    settingsBuilder.setTxPowerLevel(mTransmissionPower);
    // There's no need to connect to a Peripheral that doesn't
    // host a Gatt profile
    if (mGattServer != null) {
        settingsBuilder.setConnectable(true);
    } else {
        settingsBuilder.setConnectable(false);
    }
    return settingsBuilder.build();
}

/**
 * Start Advertising
 *
 * @throws Exception Exception thrown if Bluetooth Peripheral
 *                   mode is not supported
 */
public void startAdvertising() {
    AdvertiseSettings advertiseSettings = buildAdvertiseSettings();
    AdvertiseData advertiseData = buildAdvertisingData();
    // begin advertising
    mBluetoothAdvertiser.startAdvertising(

```

```

        advertiseSettings,
        advertiseData,
        mAdvertiseCallback
    );
}

/***
 * Stop advertising
 */
public void stopAdvertising() {
    if (mBluetoothAdvertiser != null) {
        mBluetoothAdvertiser.stopAdvertising(mAdvertiseCallback);
        mBlePeripheralCallback.onAdvertisingStopped();
    }
}

/***
 * Check if a Characetricistic supports write permissions
 * @return Returns <b>true</b> if property is writable
 */
public static boolean isCharacteristicWritable(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \
        (BluetoothGattCharacteristic.PROPERTY_WRITE | \
        BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE)) != 0;
}

/***
 * Check if a Characetricistic supports write wuthout response permissions
 * @return Returns <b>true</b> if property is writable
 */
public static boolean isCharacteristicWritablewithoutResponse(
    BluetoothGattCharacteristic characteristic
) {
    return (characteristic.getProperties() & \

```

```

        BluetoothGattCharacteristic.PROPERTY_WRITE_NO_RESPONSE) != 0;
    }

    /**
     * Check if a characteristic supports write with permissions
     * @return Returns <b>true</b> if property is writable
     */
    public static boolean isCharacteristicWritableWithResponse(
        BluetoothGattCharacteristic characteristic
    ) {
        return (characteristic.getProperties() & \
            BluetoothGattCharacteristic.PROPERTY_WRITE) != 0;
    }

    /**
     * Check if a characteristic supports Notifications
     *
     * @return Returns <b>true</b> if property is supports notification
     */
    public static boolean isCharacteristicNotifiable(
        BluetoothGattCharacteristic characteristic
    ) {
        return (characteristic.getProperties() & \
            BluetoothGattCharacteristic.PROPERTY_NOTIFY) != 0;
    }

    /**
     * Check if a Descriptor can be read
     *
     * @param descriptor a descriptor to check
     * @return Returns <b>true</b> if descriptor is readable
     */
    public static boolean isDescriptorReadable(
        BluetoothGattDescriptor descriptor
    ) {
        return (descriptor.getPermissions() & \

```

```

        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
    }

}

/**
 * Check if a Descriptor can be written
 *
 * @param descriptor a descriptor to check
 * @return Returns <b>true</b> if descriptor is writeable
 */
public static boolean isDescriptorWriteable(
    BluetoothGattDescriptor descriptor
) {
    return (descriptor.getPermissions() & \
        BluetoothGattCharacteristic.PERMISSION_WRITE) != 0;
}

private final BluetoothGattServerCallback mGattServerCallback = \
    new BluetoothGattServerCallback()
{
    @Override
    public void onConnectionStateChange(
        BluetoothDevice device,
        final int status,
        int newState
    ) {
        super.onConnectionStateChange(device, status, newState);
        Log.v(TAG, "Connected");
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothGatt.STATE_CONNECTED) {
                mConnectedCentral = device;
                mBlePeripheralCallback.onCentralConnected(device);
                stopAdvertising();
            } else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
                mConnectedCentral = null;
                mBlePeripheralCallback.onCentralDisconnected(device);
            }
        }
    }
}

```

```
        try {
            startAdvertising();
        } catch (Exception e) {
            Log.e(TAG, "error starting advertising");
        }
    }

}

@Override
public void onCharacteristicReadRequest(
    BluetoothDevice device,
    int requestId,
    int offset,
    BluetoothGattCharacteristic characteristic
) {
    super.onCharacteristicReadRequest(
        device,
        requestId,
        offset,
        characteristic
);
    Log.d(
        TAG,
        "Device tried to read characteristic: " + \
        characteristic.getUuid()
);
    Log.d(TAG, "value: " + Arrays.toString(
        characteristic.getValue()
));
    if (offset != 0) {
        mGattServer.sendResponse(
            device, requestId,
            BluetoothGatt.GATT_INVALID_OFFSET,
            offset,
            characteristic.getValue()
        );
    }
}
```

```
        );
        return;
    }

    mGattServer.sendResponse(
        device,
        requestId,
        BluetoothGatt.GATT_SUCCESS,
        offset,
        characteristic.getValue()
    );
}

@Override
public void onNotificationSent(BluetoothDevice device, int status) {
    super.onNotificationSent(device, status);
    Log.v(TAG, "Notification sent. Status: " + status);
}

@Override
public void onCharacteristicWriteRequest(
    BluetoothDevice device,
    int requestId,
    BluetoothGattCharacteristic characteristic,
    boolean preparedwrite,
    boolean responseNeeded,
    int offset,
    byte[] value
) {
    super.onCharacteristicWriteRequest(
        device,
        requestId,
        characteristic,
        preparedwrite,
        responseNeeded,
        offset,
        value
    );
}
```

```
);

Log.v(
    TAG,
    "Characteristic write request: " + Arrays.toString(value)
);

mBlePeripheralCallback.onCharacteristicWritten(
    device,
    characteristic,
    value
);
if (isCharacteristicWritableWithResponse(characteristic)) {
    characteristic.setValue(value);
    mGattServer.sendResponse(
        device,
        requestId,
        BluetoothGatt.GATT_SUCCESS,
        0,
        null
    );
}
if (isCharacteristicNotifiable(characteristic)) {
    boolean isNotifiedToSend = false;
    mGattServer.notifyCharacteristicChanged(
        device,
        characteristic,
        isNotifiedToSend
    );
}
}

@Override
public void onDescriptorWriteRequest(
    BluetoothDevice device,
    int requestId,
    BluetoothGattDescriptor descriptor,
    boolean preparedWrite,
```

```

        boolean responseNeeded,
        int offset,
        byte[] value
    ) {
    Log.v(
        TAG,
        "Descriptor write Request " + descriptor.getUuid() + " " + \
        Arrays.toString(value)
    );
    super.onDescriptorwriteRequest(
        device,
        requestId,
        descriptor,
        preparedwrite,
        responseNeeded,
        offset, value);
    // determine which characteristic is being requested
    BluetoothGattCharacteristic characteristic = \
        descriptor.getCharacteristic();
    // is the descriptor writeable?
    if (isDescriptorWriteable(descriptor)) {
        descriptor.setValue(value);
        // was this a subscription or an unsubscription?
        if (descriptor.getUuid().equals(NOTIFY_DESCRIPTOR_UUID)) {
            if (value == \
                BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE)
            {
                mBlePeripheralCallback.onCharacteristicSubscribedTo(
                    characteristic
                );
            } else if (value == \
                BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE)
            {
                mBlePeripheralCallback.
                    onCharacteristicUnsubscribedFrom(characteristic);
            }
        }
    }
}

```

```

        // send a confirmation if necessary
        if (isDescriptorReadable(descriptor)) {
            mGattServer.sendResponse(
                device,
                requestId,
                BluetoothGatt.GATT_SUCCESS,
                offset,
                value
            );
        }
    } else {
        // notify failure if necessary
        if (isDescriptorReadable(descriptor)) {
            mGattServer.sendResponse(
                device,
                requestId,
                BluetoothGatt.GATT_WRITE_NOT_PERMITTED,
                offset,
                value
            );
        }
    }
};

public AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
    @Override
    public void onStartSuccess(AdvertiseSettings settingsInEffect) {
        super.onStartSuccess(settingsInEffect);
        mBlePeripheralCallback.onAdvertisingStarted();
    }

    @Override
    public void onStartFailure(int errorCode) {
        super.onStartFailure(errorCode);
    }
};

```

```
    mBlePeripheralCallback.onAdvertisingFailed(errorCode);
}
};

}
```

The BlePeripheralCallback relays BlePeripheral state changes and events to the RemoteLed:

#### Example 13-16. java/com.example.remoteled/ble/callbacks/BlePeripheralCallback

```
package com.example.remoteled.ble.callbacks
public abstract class BlePeripheralCallback {

    /**
     * Advertising Started
     */
    public abstract void onAdvertisingStarted();

    /**
     * Advertising Could not Start
     */
    public abstract void onAdvertisingFailed(int errorCode);

    /**
     * Advertising Stopped
     */
    public abstract void onAdvertisingStopped();

    /**
     * Central Connected
     *
     * @param bluetoothDevice the BluetoothDevice representing the
     * connected Central
     */
    public abstract void onCentralConnected(
        final BluetoothDevice bluetoothDevice
    );
}
```

```
);

/***
 * Central Disconnected
 *
 * @param bluetoothDevice the BluetoothDevice representing the
 * disconnected Central
 */
public abstract void onCentralDisconnected(
    final BluetoothDevice bluetoothDevice
);

/***
 * Characteristic written to
 *
 * @param characteristic The connected Central that wrote the value
 * @param characteristic The Characteristic that was written to
 * @param value the byte value that was written
 */
public abstract void onCharacteristicWritten(
    final BluetoothDevice connectedDevice,
    final BluetoothGattCharacteristic characteristic,
    final byte[] value
);

/***
 * Characteristic subscribed to
 *
 * @param characteristic The Characteristic that was subscribed to
 */
public abstract void onCharacteristicsSubscribedTo(
    final BluetoothGattCharacteristic characteristic
);

/***
 * Characteristic unsubscribed from
*/
```

```

*
 * @param characteristic The characteristic that was unsubscribed from
 */
public abstract void onCharacteristicUnsubscribedFrom(
    final BluetoothGattCharacteristic characteristic
);
}

```

The BleRemoteLed handles communication with the Peripheral, describes the command and response structure, and relays state changes through the response Characteristic:

### **Example 13-17. java/com.example.remoteled/ble/BleRemoteLed**

```

package com.example.remoteled.ble
public class BleRemoteLed {

    /** Constants */
    private static final String TAG = BleRemoteLed.class.getSimpleName();

    public static final String CHARSET = "ASCII";

    private static final String MODEL_NUMBER = "1AB2";
    private static final String SERIAL_NUMBER = "1234";

    /** Peripheral and GATT Profile */
    public static final String ADVERTISING_NAME = "LedRemote";
    // Automation IO Service
    public static final UUID AUTOMATION_IO_SERVICE_UUID = \
        UUID.fromString("00001815-0000-1000-8000-00805f9b34fb");
    public static final UUID COMMAND_CHARACTERISTIC_UUID = \
        UUID.fromString("00002a56-0000-1000-8000-00805f9b34fb");
    public static final UUID RESPONSE_CHARACTERISTIC_UUID = \
        UUID.fromString("00002a57-0000-1000-8000-00805f9b34fb");
    private static final int COMMAND_CHARACTERISTIC_LENGTH = 20;
    private static final int RESPONSE_CHARACTERISTIC_LENGTH = 20;
}

```

```

/** Data packet */
private static final int TRANSMISSION_LENGTH = 2;

/** Sending commands */
public static final int COMMAND_FOOTER_POSITION = 1;
public static final int COMMAND_DATA_POSITION = 0;
public static final byte COMMAND_FOOTER = 1;
public static final byte COMMAND_LED_OFF = 2;
public static final byte COMMAND_LED_ON = 1;

/** Receiving responses */
public static final int RESPONSE_FOOTER_POSITION = 1;
public static final int RESPONSE_DATA_POSITION = 0;
public static final byte RESPONSE_TYPE_ERROR = 0;
public static final byte RESPONSE_TYPE_CONFIRMATION = 1;
public static final byte RESPONSE_LED_STATE_ERROR = 1;
public static final byte LED_STATE_ON = 1;
public static final byte LED_STATE_OFF = 2;

/** callback Handlers */
public BleRemoteLedCallback mBleRemoteLedCallback;

/** Bluetooth Stuff */
private BlePeripheral mBlePeripheral;
private BluetoothGattService mAutomationIoService;
private BluetoothGattCharacteristic mCommandCharacteristic,
    mResponseCharacteristic;

/**
 * Construct a new Peripheral
 *
 * @param context The Application Context
 * @param bleRemoteLedCallback The callback handler that

```

```

*      interfaces with this Peripheral
* @throws Exception Exception thrown if Bluetooth is not supported
*/
public BleRemoteLed(
    final Context context,
    BleRemoteLedCallback bleRemoteLedCallback) throws Exception
{
    mBleRemoteLedCallback = bleRemoteLedCallback;
    mBlePeripheral = new BlePeripheral(context, mBlePeripheralCallback);
    setupDevice();
}

/**
 * Set up the Advertising name and GATT profile
 */
private void setupDevice() throws Exception {
    mBlePeripheral.setModelNumber(MODEL_NUMBER);
    mBlePeripheral.setSerialNumber(SERIAL_NUMBER);
    mBlePeripheral.setupDevice();
    mAutomationIoService = \
        new BluetoothGattService(AUTOMATION_IO_SERVICE_UUID,
            BluetoothGattService.SERVICE_TYPE_PRIMARY);
    mCommandCharacteristic = new BluetoothGattCharacteristic(
        COMMAND_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_WRITE,
        BluetoothGattCharacteristic.PERMISSION_WRITE);
    mResponseCharacteristic = new BluetoothGattCharacteristic(
        RESPONSE_CHARACTERISTIC_UUID,
        BluetoothGattCharacteristic.PROPERTY_READ | \
            BluetoothGattCharacteristic.PROPERTY_NOTIFY,
        BluetoothGattCharacteristic.PERMISSION_READ);
    // add Notification support to Characteristic
    BluetoothGattDescriptor notifyDescriptor = \
        new BluetoothGattDescriptor(
            BlePeripheral.NOTIFY_DESCRIPTOR_UUID,

```

```

        BluetoothGattDescriptor.PERMISSION_WRITE | \
        BluetoothGattDescriptor.PERMISSION_READ
    );
    mResponseCharacteristic.addDescriptor(notifyDescriptor);
    mAutomationIoService.addCharacteristic(mCommandCharacteristic);
    mAutomationIoService.addCharacteristic(mResponseCharacteristic);
    mBlePeripheral.addService(mAutomationIoService);
}

/**
 * Start Advertising
 *
 * @throws Exception Exception thrown if Bluetooth Peripheral
 *                   mode is not supported
 */
public void startAdvertising() throws Exception {
    // set the device name
    mBlePeripheral.setPeripheralAdvertisingName(ADVERTISING_NAME);
    mBlePeripheral.startAdvertising();
}

/**
 * Stop advertising
 */
public void stopAdvertising() {
    mBlePeripheral.stopAdvertising();
}

/**
 * Get the BlePeripheral
 */
public BlePeripheral getBlePeripheral() {
    return mBlePeripheral;
}

/**

```

```

* Make sense of the incoming byte array as a command
*
* @param bleCommandValue the incoming Bluetooth value
*/
private void processCommand(
    final BluetoothDevice connectedDevice,
    final byte[] bleCommandValue
) {
    if (bleCommandValue[COMMAND_FOOTER_POSITION] == COMMAND_FOOTER) {
        Log.v(TAG, "Command found");
        switch (bleCommandValue[COMMAND_DATA_POSITION]) {
            case COMMAND_LED_ON:
                Log.v(TAG, "Command to turn LED on");
                sendBleResponse(connectedDevice, LED_STATE_ON);
                mBleRemoteLedCallback.onLedTurnedOn();
                break;
            case COMMAND_LED_OFF:
                Log.v(TAG, "Command to turn LED off");
                sendBleResponse(connectedDevice, LED_STATE_OFF);
                mBleRemoteLedCallback.onLedTurnedOff();
                break;
            default:
                Log.d(TAG, "Unknown incoming command");
        }
    }
}

/**
 * Send a formatted response out via a Bluetooth characteristic
 *
 * @param ledState the new LED state
 */
private void sendBleResponse(
    final BluetoothDevice connectedDevice,
    byte ledState
)

```

```
byte[] responseValue = new byte[TRANSMISSION_LENGTH];
responseValue[RESPONSE_FOOTER_POSITION] = RESPONSE_TYPE_CONFIRMATION;
responseValue[RESPONSE_DATA_POSITION] = ledState;
Log.v(
    TAG,
    "sending response: " + Arrays.toString(
        mResponseCharacteristic.getValue()
    ) + " to characteristic: "+mResponseCharacteristic);
mResponseCharacteristic.setValue(responseValue);
mBlePeripheral.getGattServer().notifyCharacteristicChanged(
    connectedDevice,
    mResponseCharacteristic,
    true
);
}

private BlePeripheralCallback mBlePeripheralCallback = \
new BlePeripheralCallback()
{
    @Override
    public void onAdvertisingStarted() {

    }

    @Override
    public void onAdvertisingFailed(int errorCode) {

    }

    @Override
    public void onAdvertisingStopped() {

    }

    @Override
    public void onCentralConnected(BluetoothDevice bluetoothDevice) {
```

```

        mBleRemoteLedCallback.onCentralConnected(blueoothDevice);
    }

    @Override
    public void onCentralDisconnected(BluetoothDevice blueoothDevice) {
        mBleRemoteLedCallback.onCentralDisconnected(blueoothDevice);
    }

    @Override
    public void onCharacteristicWritten(
        BluetoothDevice connectedDevice,
        BluetoothGattCharacteristic characteristic,
        byte[] value
    ) {
        // copy value to the read characteristic
        processCommand(connectedDevice, value);
    }

    @Override
    public void onCharacteristicSubscribedTo(
        BluetoothGattCharacteristic characteristic
    ) {
    }

    @Override
    public void onCharacteristicUnsubscribedFrom(
        BluetoothGattCharacteristic characteristic
    ) {
    }
}

```

The BleRemoteLedCallback relays state changes and events from the BleRemoteLed, such as subscriptions and changes to the LED state:

## Example 13-18. java/com.example.remotedled/ble/BleRemoteLedCallback.java

```
package example.com.remotedled.ble.callbacks;
public abstract class BleRemoteLedCallback {
    /**
     * Central Connected
     *
     * @param bluetoothDevice the BluetoothDevice representing the
     * connected Central
     */
    public abstract void onCentralConnected(
        final BluetoothDevice bluetoothDevice
    );

    /**
     * Central Disconnected
     *
     * @param bluetoothDevice the BluetoothDevice representing the
     * disconnected Central
     */
    public abstract void onCentralDisconnected(
        final BluetoothDevice bluetoothDevice
    );

    /**
     * Led Turned On
     */
    public abstract void onLedTurnedon();

    /**
     * Led Turned Off
     */
    public abstract void onLedTurnedoff();
}
```

## Activities

The Main Activity turns on the Bluetooth radio and creates an BleRemoteLed. It responds to connectivity changes by toggling a Switch, and responds to commands by turning the Android camera flash on and off:

### Example 13-19. java/example.com.remotedled/MainActivity.java

```
package example.com.remotedled

public class MainActivity extends AppCompatActivity {

    /** Constants */
    private static final String TAG = MainActivity.class.getSimpleName();
    private static final int REQUEST_ENABLE_BT = 1;

    /** Bluetooth Stuff */
    private BleRemoteLed mBleRemoteLed;

    /** Camera Stuff */
    private boolean mIsFlashAvailable;
    private CameraManager mCameraManager;
    private String mCameraId;

    /** UI Stuff */
    private TextView mAdvertisingNameTV;
    private Switch mBluetoothOnSwitch,
                  mCentralConnectedSwitch,
                  mLedStateSwitch;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        // notify when bluetooth is turned on or off
        IntentFilter filter = new IntentFilter(
            BluetoothAdapter.ACTION_STATE_CHANGED
        );
    }
}
```

```
);

registerReceiver(mBleAdvertiseReceiver, filter);

mIsFlashAvailable = getApplicationContext().getPackageManager()
    .hasSystemFeature(PackageManager.FEATURE_CAMERA_FLASH);

if (mIsFlashAvailable) {
    mCameraManager = (CameraManager) getSystemService(
        Context.CAMERA_SERVICE
    );
    try {
        mCameraId = mCameraManager.getCameraIdList()[0];
    } catch (CameraAccessException e) {
        e.printStackTrace();
    }
}

loadUI();

}

@Override
public void onPause() {
    super.onPause();
    // stop advertising when the activity pauses
    mBleRemoteLed.stopAdvertising();
    onLedOffCommand();
}

@Override
public void onResume() {
    super.onResume();
    initializeBluetooth();
}

@Override
public void onDestroy() {
    super.onDestroy();
    unregisterReceiver(mBleAdvertiseReceiver);
}
```

```

/**
 * Load UI components
 */
public void loadUI() {
    mAdvertisingNameTV = (TextView) findViewById(R.id.advertising_name);
    mBluetoothOnSwitch = (Switch) findViewById(R.id.bluetooth_on);
    mCentralConnectedSwitch = (Switch) findViewById(R.id.central_connected);
    mLedStateSwitch = (Switch) findViewById(R.id.led_state);
    mAdvertisingNameTV.setText(BleRemoteLed.ADVERTISING_NAME);
}

/**
 * Initialize the Bluetooth Radio
 */
public void initializeBluetooth() {
    // reset connection variables
    try {
        mBleRemoteLed = new BleRemoteLed(this, mBlePeripheralCallback);
    } catch (Exception e) {
        Log.e(TAG, "Could not initialize bluetooth");
        Log.e(TAG, e.getMessage());
        e.printStackTrace();
        finish();
    }
    Log.v(TAG, "bluetooth switch: "+mBluetoothOnSwitch);
    Log.v(TAG, "mBleRemoteLed: "+mBleRemoteLed);
    Log.v(TAG, "peripheral: "+mBleRemoteLed.getBlePeripheral());
    mBluetoothOnSwitch.setChecked(
        mBleRemoteLed.getBlePeripheral().getBluetoothAdapter().isEnabled()
    );
}

// should prompt user to open settings if Bluetooth is not enabled.
if (!mBleRemoteLed.getBlePeripheral()
    .getBluetoothAdapter().isEnabled())
{

```

```

        Intent enableBtIntent = new Intent(
            BluetoothAdapter.ACTION_REQUEST_ENABLE
        );
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    } else {
        startAdvertising();
    }
}

/**
 * Start advertising Peripheral
 */
public void startAdvertising() {
    try {
        mBleRemoteLed.startAdvertising();
    } catch (Exception e) {
        Log.e(TAG, "error starting advertising: " + e.getMessage());
        e.printStackTrace();
    }
}

/**
 * Event trigger when Central has connected
 *
 * @param bluetoothDevice
 */
public void onBleCentralConnected(final BluetoothDevice bluetoothDevice) {
    mCentralConnectedSwitch.setChecked(true);
}

/**
 * Event trigger when Central has disconnected
 * @param bluetoothDevice
 */
public void onBleCentralDisconnected(

```

```
    final BluetoothDevice bluetoothDevice
) {
    mCentralConnectedSwitch.setChecked(false);
}

/**
 * Led turned on remotely
 */
public void onLedOnCommand() {
    mLedStateSwitch.setChecked(true);

    if (mIsFlashAvailable) {
        try {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                mCameraManager.setTorchMode(mCameraId, true);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

/**
 * Led turn off remotely
 */
public void onLedOffCommand() {
    mLedStateSwitch.setChecked(false);
    if (mIsFlashAvailable) {
        try {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                mCameraManager.setTorchMode(mCameraId, false);
            }
        } catch (Exception e) {
            Log.d(TAG, "could not open camera flash: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

```
    }

}

/***
 * When the Bluetooth radio turns on, initialize the Bluetooth connection
 */
private final AdvertiseReceiver mBleAdvertiseReceiver = \
    new AdvertiseReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();

        if (action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
            final int state = intent.getIntExtra(
                BluetoothAdapter.EXTRA_STATE,
                BluetoothAdapter.ERROR
            );
            switch (state) {
                case BluetoothAdapter.STATE_OFF:
                    Log.v(TAG, "Bluetooth turned off");
                    initializeBluetooth();
                    break;
                case BluetoothAdapter.STATE_TURNING_OFF:
                    break;
                case BluetoothAdapter.STATE_ON:
                    Log.v(TAG, "Bluetooth turned on");
                    startAdvertising();
                    break;
                case BluetoothAdapter.STATE_TURNING_ON:
                    break;
            }
        }
    }
};
```

```
/**  
 * Respond to changes to the Bluetooth Peripheral state  
 */  
  
private final BleRemoteLedCallback mBlePeripheralCallback = \  
    new BleRemoteLedCallback()  
{  
  
    public void onCentralConnected(final BluetoothDevice bluetoothDevice) {  
        Log.v(TAG, "Central connected");  
        runOnUiThread(new Runnable() {  
            @Override  
            public void run() {  
                onBleCentralConnected(bluetoothDevice);  
            }  
        });  
    }  
  
    public void onCentralDisconnected(  
        final BluetoothDevice bluetoothDevice  
    ) {  
        Log.v(TAG, "Central disconnected");  
        runOnUiThread(new Runnable() {  
            @Override  
            public void run() {  
                onBleCentralDisconnected(bluetoothDevice);  
            }  
        });  
    }  
  
    @Override  
    public void onLedTurnedOn() {  
        runOnUiThread(new Runnable() {  
            @Override  
            public void run() {  
                onLedOnCommand();  
            }  
        });  
    }  
}
```

```
    @Override
    public void onLedTurnedoff() {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                onLedoffCommand();
            }
        });
    }
}
```

In the MainActivity layout, create Switches representing the Bluetooth radio state, the Central connectivity, and the LED state, plus a TextView showing the Peripheral name:

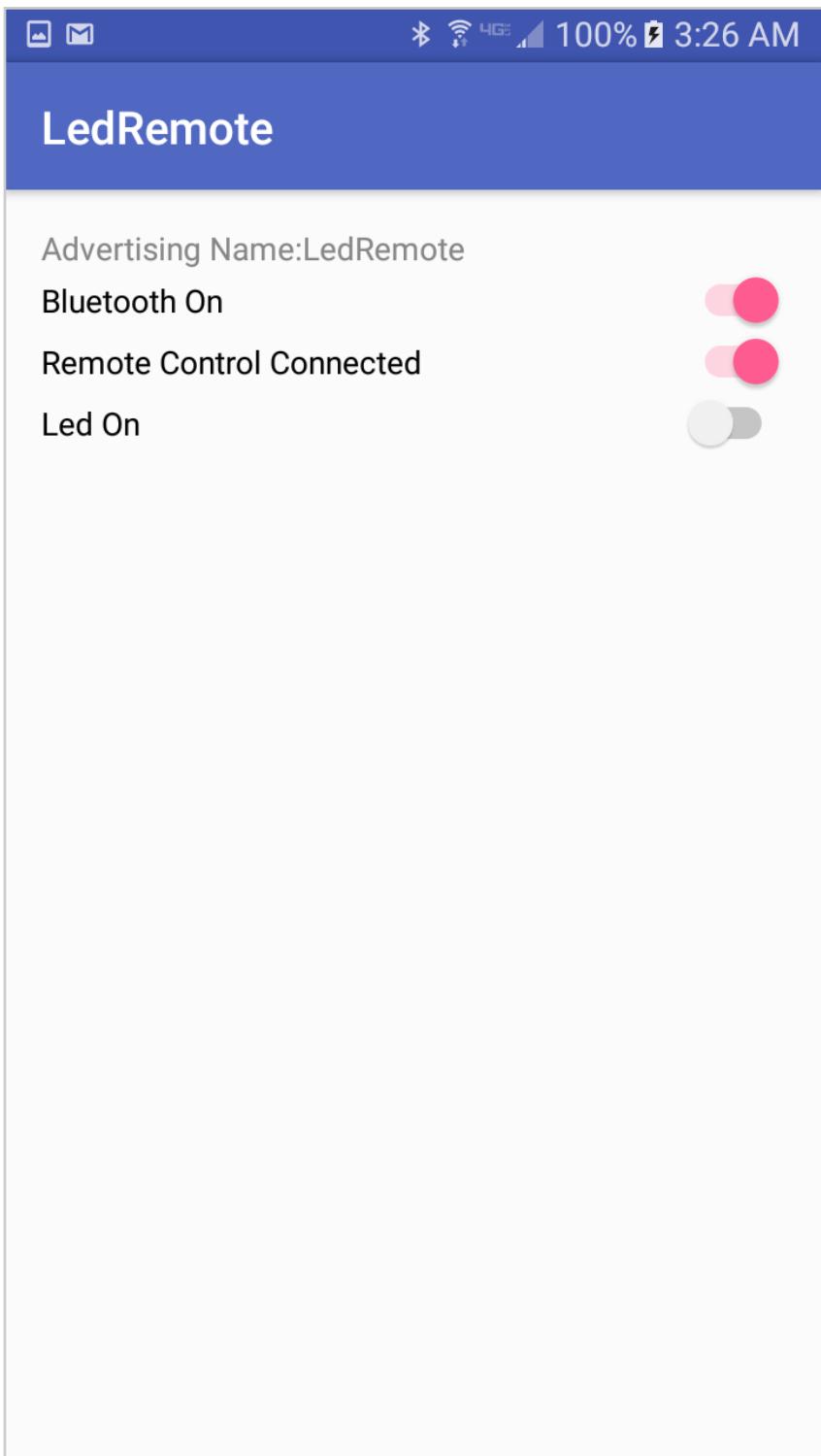
#### Example 13-20. res/layout/activity\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
```

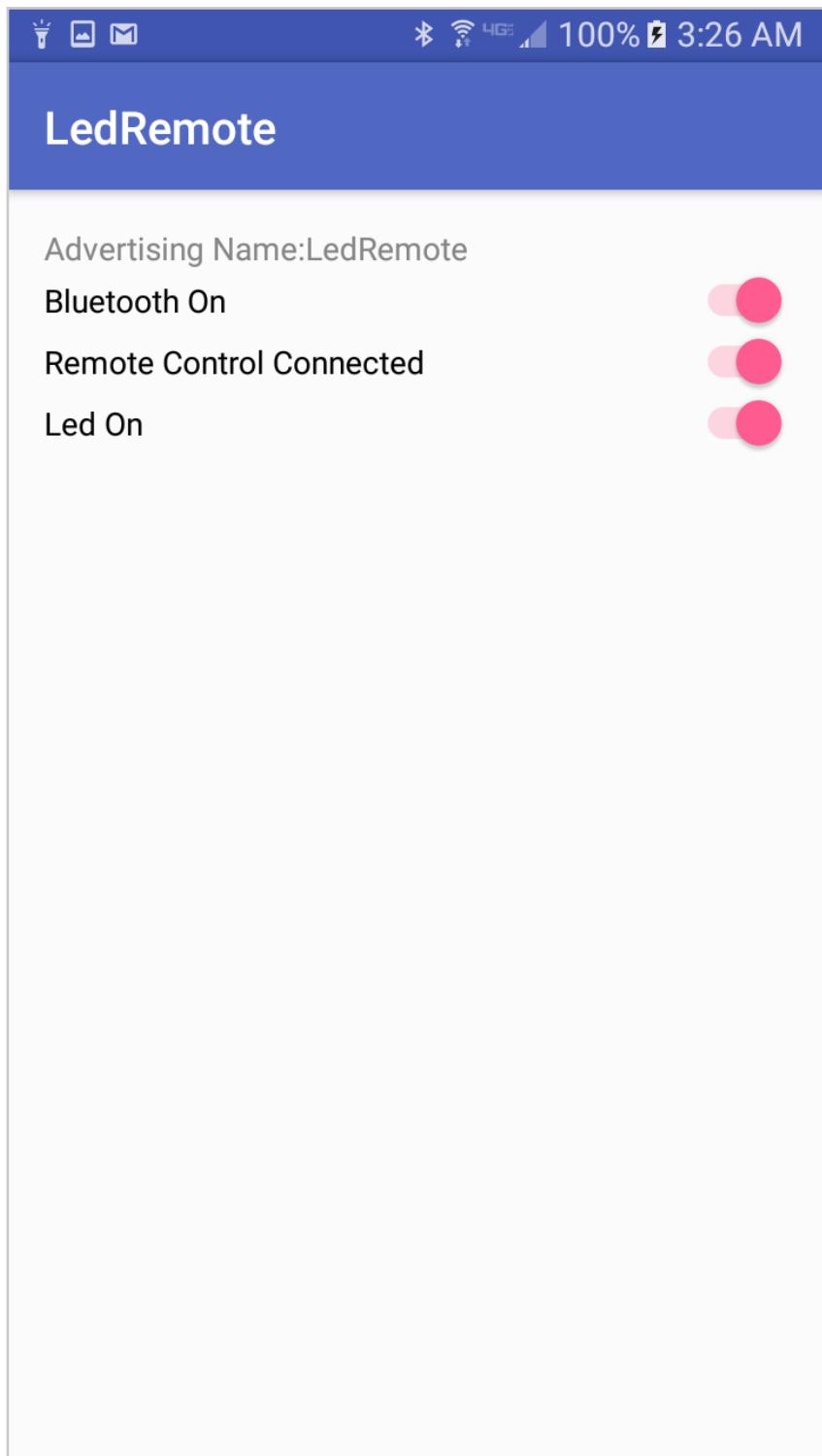
```
    app:popupTheme="@style/AppTheme.PopupOverlay" />
</android.support.design.widget.AppBarLayout>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main" tools:context=".MainActivity">
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/advertising_name_label" />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/advertising_name" />
</LinearLayout>
<Switch
    android:clickable="false"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bluetooth_on"
    android:id="@+id/bluetooth_on" />
<Switch
    android:clickable="false"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/central_connected"
        android:id="@+id/central_connected" />
    <Switch
        android:clickable="false"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/led_state"
        android:id="@+id/led_state" />
    </LinearLayout>
</RelativeLayout>
</android.support.design.widget.CoordinatorLayout>
```

The resulting App can turn the camera flash on and off in response to commands from a connected Central ([Figure 13-8](#)). When the command is processed, a response is sent through the response Characteristic ([Figure 13-9](#)).



**Figure 13-5. App screen showing LED switch in off state**



**Figure 13-6. App screen showing LED switch in on state**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyInAndroidJava/Chapter13>

# Appendix

For reference, the following are properties of the Bluetooth Low Energy network and hardware.

<b>Range</b>	100 m (330 ft)
<b>Data Rate</b>	1M bit/s
<b>Application Throughput</b>	0.27 Mbit/s
<b>Security</b>	128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)
<b>Robustness</b>	Adaptive Frequency Hopping, Lazy Acknowledgement, 24-bit CRC, 32-bit Message Integrity Check
<b>Range</b>	100 m (330 ft)
<b>Data Rate</b>	1M bit/s
<b>Application Throughput</b>	0.27 Mbit/s
<b>Security</b>	128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)
<b>Peak Current Consumption</b>	< 15 mA
<b>Byte-Order in Broadcast</b>	Big Endian (most significant bit at end)
<b>Range</b>	100 m (330 ft)
<b>Data Rate</b>	1M bit/s
<b>Application Throughput</b>	0.27 Mbit/s
<b>Security</b>	128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)

## Appendix II: UUID Format

Bluetooth Low Energy has tight space requirements. Therefore it is preferred to transmit 16-bit UUIDs instead of 32-bit UUIDs. UUIDs can be converted between 16-bit and 32-bit with the standard Bluetooth Low Energy UUID format:

**Table II-1. 16-bit to 32-bit UUID Conversion Standard**

UUID Format	uuid16	Resulting uuid32
00000000-0000-1000-8000-00805f9b34fb	0x2A56	00002A56-0000-1000-8000-00805f9b34fb

## Appendix III: Minimal Recommended GATT

As a best practice, it is good to host a standard set of Services and Characteristics in a Peripheral's GATT Profile. These Characteristics allow connected Centrals to get the make and model number of the device, and the battery level if the Peripheral is battery-powered:

**Table III-1. Minimal GATT Profile**

GATT Type	Name	Data Type	UUID
Service	Device Information Service		0x180a
Characteristic	Device Name	char array	0x2a00
Characteristic	Model Number	char array	0x2a24
Characteristic	Serial Number	char array	0x2a04
Service	Battery Level		0x180f
Characteristic	Battery Level	integer	0x2a19

## Appendix IV: Reserved GATT Services

Services act as a container for Characteristics or other Services, providing a tree-like structure for organizing Bluetooth I/O.

These Services UUIDs have been reserved for special contexts, such as Device Information (0x180A) Which may contain Characteristics that communicate information about the Peripheral's name, version number, or settings.

*Note: All Bluetooth Peripherals should have a Battery Service (0x180F) Service containing a Battery Level (0x2A19) Characteristic.*

**Table IV-1. Reserved GATT Services**

Specification Name	UUID	Specification Type
Alert Notification Service	0x1811	org.bluetooth.service.alert_notification
Automation IO	0x1815	org.bluetooth.service.automation_io
Battery Service	0x180F	org.bluetooth.service.battery_service
Blood Pressure	0x1810	org.bluetooth.service.blood_pressure
Body Composition	0x181B	org.bluetooth.service.body_composition
Bond Management	0x181E	org.bluetooth.service.bond_management
Continuous Glucose Monitoring	0x181F	org.bluetooth.service.continuous_glucose_monitoring

Specification Name	UUID	Specification Type
Current Time Service	0x1805	org.bluetooth.service.current_time
Cycling Power	0x1818	org.bluetooth.service.cycling_power
Cycling Speed and Cadence	0x1816	org.bluetooth.service.cycling_speed_and_cadence
Device Information	0x180A	org.bluetooth.service.device_information
Environmental Sensing	0x181A	org.bluetooth.service.environmental_sensing
Generic Access	0x1800	org.bluetooth.service.generic_access
Generic Attribute	0x1801	org.bluetooth.service.generic_attribute
Glucose	0x1808	org.bluetooth.service.glucose
Health Thermometer	0x1809	org.bluetooth.service.health_thermometer
Heart Rate	0x180D	org.bluetooth.service.heart_rate
HTTP Proxy	0x1823	org.bluetooth.service.http_proxy
Human Interface Device	0x1812	org.bluetooth.service.human_interface_device
Immediate Alert	0x1802	org.bluetooth.service.immediate_alert
Indoor Positioning	0x1821	org.bluetooth.service.indoor_positioning
Internet Protocol Support	0x1820	org.bluetooth.service.internet_protocol_support

Specification Name	UUID	Specification Type
Link Loss	0x1803	org.bluetooth.service.link_loss
Location and Navigation	0x1819	org.bluetooth.service.location_and_navigation
Next DST Change Service	0x1807	org.bluetooth.service.next_dst_change
Object Transfer	0x1825	org.bluetooth.service.object_transfer
Phone Alert Status Service	0x180E	org.bluetooth.service.phone_alert_status
Pulse Oximeter	0x1822	org.bluetooth.service.pulse_oximeter
Reference Time Update Service	0x1806	org.bluetooth.service.reference_time_update
Running Speed and Cadence	0x1814	org.bluetooth.service.running_speed_and_cadence
Scan Parameters	0x1813	org.bluetooth.service.scan_parameters
Transport Discovery	0x1824	org.bluetooth.service.transport_discovery
Tx Power	0x1804	org.bluetooth.service.tx_power
User Data	0x181C	org.bluetooth.service.user_data
Weight Scale	0x181D	org.bluetooth.service.weight_scale

**Source:** Bluetooth SIG: GATT Services

Retrieved from <https://www.bluetooth.com/specifications/gatt/services>

# Appendix V: Reserved GATT Characteristics

Characteristics act a data port that can be read from or written to.

These Characteristic UUIDs have been reserved for specific types of data, such as Device Name (0x2A00) which may read the Peripheral's current battery level.

*Note: All Bluetooth Peripherals should have a Battery Level (0x2A19) Characteristic, contained inside a Battery Service (0x180F) Service.*

**Table V-1. Reserved GATT Characteristics**

Specification Name	UUID	Specification Type
Aerobic Heart Rate Lower Limit	0x2A7E	org.bluetooth.characteristic.aerobic_heart_rate_lower_limit
Aerobic Heart Rate Upper Limit	0x2A84	org.bluetooth.characteristic.aerobic_heart_rate_upper_limit
Aerobic Threshold	0x2A7F	org.bluetooth.characteristic.aerobic_threshold
Age	0x2A80	org.bluetooth.characteristic.age
Aggregate	0x2A5A	org.bluetooth.characteristic.aggregate
Alert Category ID	0x2A43	org.bluetooth.characteristic.alert_category_id
Alert Category ID Bit Mask	0x2A42	org.bluetooth.characteristic.alert_category_id_bit_mask
Alert Level	0x2A06	org.bluetooth.characteristic.alert_level

Specification Name	UUID	Specification Type
Alert Status	0x2A3F	org.bluetooth.characteristic.alert_status
Altitude	0x2AB3	org.bluetooth.characteristic.altitude
Anaerobic Heart Rate Lower Limit	0x2A81	org.bluetooth.characteristic.anaerobic_heart_rate_lower_limit
Anaerobic Heart Rate Upper Limit	0x2A82	org.bluetooth.characteristic.anaerobic_heart_rate_upper_limit
Anaerobic Threshold	0x2A83	org.bluetooth.characteristic.anaerobic_threshold
Analog	0x2A58	org.bluetooth.characteristic.analog
Apparent Wind Direction	0x2A73	org.bluetooth.characteristic.apparent_wind_direction
Apparent Wind Speed	0x2A72	org.bluetooth.characteristic.apparent_wind_speed
Appearance	0x2A01	org.bluetooth.characteristic.gap.appearance
Barometric Pressure Trend	0x2AA3	org.bluetooth.characteristic.barometric_pressure_trend
Battery Level	0x2A19	org.bluetooth.characteristic.battery_level
Blood Pressure Feature	0x2A49	org.bluetooth.characteristic.blood_pressure_feature
Blood Pressure Measurement	0x2A35	org.bluetooth.characteristic.blood_pressure_measurement
Body Composition Feature	0x2A9B	org.bluetooth.characteristic.body_composition_feature
Body Composition Measurement	0x2A9C	org.bluetooth.characteristic.body_composition_measurement
Body Sensor Location	0x2A38	org.bluetooth.characteristic.body_sensor_location

Specification Name	UUID	Specification Type
Bond Management Control Point	0x2AA4	org.bluetooth.characteristic.bond_management_control_point
Bond Management Feature	0x2AA5	org.bluetooth.characteristic.bond_management_feature
Boot Keyboard Input Report	0x2A22	org.bluetooth.characteristic.boot_keyboard_input_report
Boot Keyboard Output Report	0x2A32	org.bluetooth.characteristic.boot_keyboard_output_report
Boot Mouse Input Report	0x2A33	org.bluetooth.characteristic.boot_mouse_input_report
Central Address Resolution	0x2AA6	org.bluetooth.characteristic.gap.central_address_resolution_support
CGM Feature	0x2AA8	org.bluetooth.characteristic.cgm_feature
CGM Measurement	0x2AA7	org.bluetooth.characteristic.cgm_measurement
CGM Session Run Time	0x2AAB	org.bluetooth.characteristic.cgm_session_run_time
CGM Session Start Time	0x2AAA	org.bluetooth.characteristic.cgm_session_start_time
CGM Specific Ops Control Point	0x2AAC	org.bluetooth.characteristic.cgm_specific_ops_control_point
CGM Status	0x2AA9	org.bluetooth.characteristic.cgm_status
CSC Feature	0x2A5C	org.bluetooth.characteristic.csc_feature
CSC Measurement	0x2A5B	org.bluetooth.characteristic.csc_measurement
Current Time	0x2A2B	org.bluetooth.characteristic.current_time

Specification Name	UUID	Specification Type
Cycling Power Feature	0x2A65	org.bluetooth.characteristic.cycling_power_feature
Cycling Power Measurement	0x2A63	org.bluetooth.characteristic.cycling_power_measurement
Cycling Power Vector	0x2A64	org.bluetooth.characteristic.cycling_power_vector
Database Change Increment	0x2A99	org.bluetooth.characteristic.database_change_increment
Date of Birth	0x2A85	org.bluetooth.characteristic.date_of_birth
Date of Threshold Assessment	0x2A86	org.bluetooth.characteristic.date_of_threshold_assessment
Date Time	0x2A08	org.bluetooth.characteristic.date_time
Day Date Time	0x2A0A	org.bluetooth.characteristic.day_date_time
Day of Week	0x2A09	org.bluetooth.characteristic.day_of_week
Descriptor Value Changed	0x2A7D	org.bluetooth.characteristic.descriptor_value_changed
Device Name	0x2A00	org.bluetooth.characteristic.gap.device_name
Dew Point	0x2A7B	org.bluetooth.characteristic.dew_point
Digital	0x2A56	org.bluetooth.characteristic.digital
DST Offset	0x2A0D	org.bluetooth.characteristic.dst_offset
Elevation	0x2A6C	org.bluetooth.characteristic.elevation

Specification Name	UUID	Specification Type
Exact Time 256	0x2A0C	org.bluetooth.characteristic.exact_time_256
Fat Burn Heart Rate Lower Limit	0x2A88	org.bluetooth.characteristic.fat_burn_heart_rate_lower_limit
Fat Burn Heart Rate Upper Limit	0x2A89	org.bluetooth.characteristic.fat_burn_heart_rate_upper_limit
Firmware Revision String	0x2A26	org.bluetooth.characteristic.firmware_revision_string
First Name	0x2A8A	org.bluetooth.characteristic.first_name
Five Zone Heart Rate Limits	0x2A8B	org.bluetooth.characteristic.five_zone_heart_rate_limits
Floor Number	0x2AB2	org.bluetooth.characteristic.floor_number
Gender	0x2A8C	org.bluetooth.characteristic.gender
Glucose Feature	0x2A51	org.bluetooth.characteristic.glucose_feature
Glucose Measurement	0x2A18	org.bluetooth.characteristic.glucose_measurement
Glucose Measurement Context	0x2A34	org.bluetooth.characteristic.glucose_measurement_context
Gust Factor	0x2A74	org.bluetooth.characteristic.gust_factor
Hardware Revision String	0x2A27	org.bluetooth.characteristic.hardware_revision_string
Heart Rate Control Point	0x2A39	org.bluetooth.characteristic.heart_rate_control_point
Heart Rate Max	0x2A8D	org.bluetooth.characteristic.heart_rate_max
Heart Rate Measurement	0x2A37	org.bluetooth.characteristic.heart_rate_measurement

Specification Name	UUID	Specification Type
Heat Index	0x2A7A	org.bluetooth.characteristic.heat_index
Height	0x2A8E	org.bluetooth.characteristic.height
HID Control Point	0x2A4C	org.bluetooth.characteristic.hid_control_point
HID Information	0x2A4A	org.bluetooth.characteristic.hid_information
Hip Circumference	0x2A8F	org.bluetooth.characteristic.hip_circumference
HTTP Control Point	0x2ABA	org.bluetooth.characteristic.http_control_point
HTTP Entity Body	0x2AB9	org.bluetooth.characteristic.http_entity_body
HTTP Headers	0x2AB7	org.bluetooth.characteristic.http_headers
HTTP Status Code	0x2AB8	org.bluetooth.characteristic.http_status_code
HTTPS Security	0x2ABB	org.bluetooth.characteristic.https_security
Humidity	0x2A6F	org.bluetooth.characteristic.humidity
IEEE 11073-20601 Regulatory Certification Data List	0x2A2A	org.bluetooth.characteristic.ieee_11073-20601_regulatory_certification_data_list
Indoor Positioning Configuration	0x2AAD	org.bluetooth.characteristic.indoor_positioning_configuration
Intermediate Cuff Pressure	0x2A36	org.bluetooth.characteristic.intermediate_cuff_pressure
Intermediate Temperature	0x2A1E	org.bluetooth.characteristic.intermediate_temperature

Specification Name	UUID	Specification Type
Language	0x2AA2	org.bluetooth.characteristic.language
Last Name	0x2A90	org.bluetooth.characteristic.last_name
Latitude	0x2AAE	org.bluetooth.characteristic.latitude
LN Control Point	0x2A6B	org.bluetooth.characteristic.ln_control_point
LN Feature	0x2A6A	org.bluetooth.characteristic.ln_feature
Local East Coordinate	0x2AB1	org.bluetooth.characteristic.local_east_coordinate
Local North Coordinate	0x2AB0	org.bluetooth.characteristic.local_north_coordinate
Local Time Information	0x2A0F	org.bluetooth.characteristic.local_time_information
Location and Speed	0x2A67	org.bluetooth.characteristic.location_and_speed
Location Name	0x2AB5	org.bluetooth.characteristic.location_name
Longitude	0x2AAF	org.bluetooth.characteristic.longitude
Magnetic Declination	0x2A2C	org.bluetooth.characteristic.magnetic_declination
Magnetic Flux Density - 2D	0x2AA0	org.bluetooth.characteristic.magnetic_flux_density_2D
Magnetic Flux Density - 3D	0x2AA1	org.bluetooth.characteristic.magnetic_flux_density_3D
Manufacturer Name String	0x2A29	org.bluetooth.characteristic.manufacturer_name_string
Maximum Recommended Heart Rate	0x2A91	org.bluetooth.characteristic.maximum_recommended_heart_rate

Specification Name	UUID	Specification Type
Measurement Interval	0x2A21	org.bluetooth.characteristic.measurement_interval
Model Number String	0x2A24	org.bluetooth.characteristic.model_number_string
Navigation	0x2A68	org.bluetooth.characteristic.navigation
New Alert	0x2A46	org.bluetooth.characteristic.new_alert
Object Action Control Point	0x2AC5	org.bluetooth.characteristic.object_action_control_point
Object Changed	0x2AC8	org.bluetooth.characteristic.object_changed
Object First-Created	0x2AC1	org.bluetooth.characteristic.object_first_created
Object ID	0x2AC3	org.bluetooth.characteristic.object_id
Object Last-Modified	0x2AC2	org.bluetooth.characteristic.object_last_modified
Object List Control Point	0x2AC6	org.bluetooth.characteristic.object_list_control_point
Object List Filter	0x2AC7	org.bluetooth.characteristic.object_list_filter
Object Name	0x2ABE	org.bluetooth.characteristic.object_name
Object Properties	0x2AC4	org.bluetooth.characteristic.object_properties
Object Size	0x2AC0	org.bluetooth.characteristic.object_size
Object Type	0x2ABF	org.bluetooth.characteristic.object_type
OTS Feature	0x2ABD	org.bluetooth.characteristic.ots_feature

Specification Name	UUID	Specification Type
Peripheral Preferred Connection Parameters	0x2A04	org.bluetooth.characteristic.gap.peripheral_preferred_connection_parameters
Peripheral Privacy Flag	0x2A02	org.bluetooth.characteristic.gap.peripheral_privacy_flag
PLX Continuous Measurement	0x2A5F	org.bluetooth.characteristic.plx_continuous_measurement
PLX Features	0x2A60	org.bluetooth.characteristic.plx_features
PLX Spot-Check Measurement	0x2A5E	org.bluetooth.characteristic.plx_spot_check_measurement
PnP ID	0x2A50	org.bluetooth.characteristic.pnp_id
Pollen Concentration	0x2A75	org.bluetooth.characteristic.pollen_concentration
Position Quality	0x2A69	org.bluetooth.characteristic.position_quality
Pressure	0x2A6D	org.bluetooth.characteristic.pressure
Protocol Mode	0x2A4E	org.bluetooth.characteristic.protocol_mode
Rainfall	0x2A78	org.bluetooth.characteristic.rainfall
Reconnection Address	0x2A03	org.bluetooth.characteristic.gap.reconnection_address
Record Access Control Point	0x2A52	org.bluetooth.characteristic.record_access_control_point
Reference Time Information	0x2A14	org.bluetooth.characteristic.reference_time_information
Report	0x2A4D	org.bluetooth.characteristic.report

Specification Name	UUID	Specification Type
Resolvable Private Address Only	0x2AC9	org.bluetooth.characteristic.resolvable_private_address_only
Resting Heart Rate	0x2A92	org.bluetooth.characteristic.resting_heart_rate
Ringer Control Point	0x2A40	org.bluetooth.characteristic.ringer_control_point
Ringer Setting	0x2A41	org.bluetooth.characteristic.ringer_setting
RSC Feature	0x2A54	org.bluetooth.characteristic.rsc_feature
RSC Measurement	0x2A53	org.bluetooth.characteristic.rsc_measurement
SC Control Point	0x2A55	org.bluetooth.characteristic.sc_control_point
Scan Interval Window	0x2A4F	org.bluetooth.characteristic.scan_interval_window
Scan Refresh	0x2A31	org.bluetooth.characteristic.scan_refresh
Sensor Location	0x2A5D	org.bluetooth.characteristic.sensor_location
Serial Number String	0x2A25	org.bluetooth.characteristic.serial_number_string
Service Changed	0x2A05	org.bluetooth.characteristic.gatt.service_changed
Software Revision String	0x2A28	org.bluetooth.characteristic.software_revision_string
Sport Type for Aerobic and Anaerobic Thresholds	0x2A93	org.bluetooth.characteristic.sport_type_for_aerobic_and_anaerobic_thresholds
Supported New Alert Category	0x2A47	org.bluetooth.characteristic.supported_new_alert_category

Specification Name	UUID	Specification Type
System ID	0x2A23	org.bluetooth.characteristic.system_id
TDS Control Point	0x2ABC	org.bluetooth.characteristic.tds_control_point
Temperature	0x2A6E	org.bluetooth.characteristic.temperature
Temperature Measurement	0x2A1C	org.bluetooth.characteristic.temperature_measurement
Temperature Type	0x2A1D	org.bluetooth.characteristic.temperature_type
Three Zone Heart Rate Limits	0x2A94	org.bluetooth.characteristic.three_zone_heart_rate_limits
Time Accuracy	0x2A12	org.bluetooth.characteristic.time_accuracy
Time Source	0x2A13	org.bluetooth.characteristic.time_source
Time Update Control Point	0x2A16	org.bluetooth.characteristic.time_update_control_point
Time Update State	0x2A17	org.bluetooth.characteristic.time_update_state
Time with DST	0x2A11	org.bluetooth.characteristic.time_with_dst
Time Zone	0x2A0E	org.bluetooth.characteristic.time_zone
True Wind Direction	0x2A71	org.bluetooth.characteristic.true_wind_direction
True Wind Speed	0x2A70	org.bluetooth.characteristic.true_wind_speed
Two Zone Heart Rate Limit	0x2A95	org.bluetooth.characteristic.two_zone_heart_rate_limit
Tx Power Level	0x2A07	org.bluetooth.characteristic.tx_power_level

Specification Name	UUID	Specification Type
Uncertainty	0x2AB4	org.bluetooth.characteristic.uncertainty
Unread Alert Status	0x2A45	org.bluetooth.characteristic.unread_alert_status
URI	0x2AB6	org.bluetooth.characteristic.uri
User Control Point	0x2A9F	org.bluetooth.characteristic.user_control_point
User Index	0x2A9A	org.bluetooth.characteristic.user_index
UV Index	0x2A76	org.bluetooth.characteristic.uv_index
VO2 Max	0x2A96	org.bluetooth.characteristic.vo2_max
Waist Circumference	0x2A97	org.bluetooth.characteristic.waist_circumference
Weight	0x2A98	org.bluetooth.characteristic.weight
Weight Measurement	0x2A9D	org.bluetooth.characteristic.weight_measurement
Weight Scale Feature	0x2A9E	org.bluetooth.characteristic.weight_scale_feature
Wind Chill	0x2A79	org.bluetooth.characteristic.wind_chill

**Source:** Bluetooth SIG: GATT Characteristics

Retrieved from <https://www.bluetooth.com/specifications/gatt/characteristics>

# Appendix VI: GATT Descriptors

The following GATT Descriptor UUIDs have been reserved for specific uses.

GATT Descriptors describe features within a Characteristic that can be altered, for instance, the Client Characteristic Configuration (0x2902) which can be flagged to allow a connected Central to subscribe to notifications on a Characteristic.

**Table VI-1. Reserved GATT Descriptors**

Specification Name	UUID	Specification Type
Characteristic Aggregate Format	0x2905	org.bluetooth.descriptor.gatt.characteristic_aggregate_format
Characteristic Extended Properties	0x2900	org.bluetooth.descriptor.gatt.characteristic_extended_properties
Characteristic Presentation Format	0x2904	org.bluetooth.descriptor.gatt.characteristic_presentation_format
Characteristic User Description	0x2901	org.bluetooth.descriptor.gatt.characteristic_user_description
Client Characteristic Configuration	0x2902	org.bluetooth.descriptor.gatt.client_characteristic_configuration
Environmental Sensing Configuration	0x290B	org.bluetooth.descriptor.es_configuration
Environmental Sensing Measurement	0x290C	org.bluetooth.descriptor.es_measurement

Specification Name	UUID	Specification Type
Number of Digits	0x2909	org.bluetooth.descriptor.number_of_digital
Report Reference	0x2908	org.bluetooth.descriptor.report_reference
Server Characteristic Configuration	0x2903	org.bluetooth.descriptor.gatt.server_characteristic_configuration
Time Trigger Setting	0x290E	org.bluetooth.descriptor.time_trigger_setting
Valid Range	0x2906	org.bluetooth.descriptor.valid_range
Value Trigger Setting	0x290A	org.bluetooth.descriptor.value_trigger_setting

**Source:** Bluetooth SIG: GATT Descriptors

Retrieved from <https://www.bluetooth.com/specifications/gatt/descriptors>

## Appendix VII: Company Identifiers

The following companies have specific Manufacturer Identifiers, which identify Bluetooth devices in the Generic Access Profile (GAP). Peripherals with no specific manufacturer use ID 65535 (0xffff). All other IDs are reserved, even if not yet assigned.

This is a non-exhaustive list of companies. A full list and updated can be found on the Bluetooth SIG website.

**Table VII-1. Company Identifiers**

Decimal	Hexadecimal	Company
0	0x0000	Ericsson Technology Licensing
1	0x0001	Nokia Mobile Phones
2	0x0002	Intel Corp.
3	0x0003	IBM Corp.
4	0x0004	Toshiba Corp.
5	0x0005	3Com
6	0x0006	Microsoft
7	0x0007	Lucent

Decimal	Hexadecimal	Company
8	0x0008	Motorola
13	0x000D	Texas Instruments Inc.
19	0x0013	Atmel Corporation
29	0x001D	Qualcomm
36	0x0024	Alcatel
37	0x0025	NXP Semiconductors (formerly Philips Semiconductors)
60	0x003C	BlackBerry Limited (formerly Research In Motion)
76	0x004C	Apple, Inc.
86	0x0056	Sony Ericsson Mobile Communications
89	0x0059	Nordic Semiconductor ASA
92	0x005C	Belkin International, Inc.
93	0x005D	Realtek Semiconductor Corporation
101	0x0065	Hewlett-Packard Company
104	0x0068	General Motors
117	0x0075	Samsung Electronics Co. Ltd.

Decimal	Hexadecimal	Company
120	0x0078	Nike, Inc.
135	0x0087	Garmin International, Inc.
138	0x008A	Jawbone
184	0x00B8	Qualcomm Innovation Center, Inc. (QuIC)
215	0x00D7	Qualcomm Technologies, Inc.
216	0x00D8	Qualcomm Connected Experiences, Inc.
220	0x00DC	Procter & Gamble
224	0x00E0	Google
359	0x0167	Bayer HealthCare
367	0x016F	Podo Labs, Inc
369	0x0171	Amazon Fulfillment Service
387	0x0183	Walt Disney
398	0x018E	Fitbit, Inc.
425	0x01A9	Canon Inc.
427	0x01AB	Facebook, Inc.

Decimal	Hexadecimal	Company
474	0x01DA	Logitech International SA
558	0x022E	Siemens AG
605	0x025D	Lexmark International Inc.
637	0x027D	HUAWEI Technologies Co., Ltd. ( )
720	0x02D0	3M
876	0x036C	Zipcar
897	0x0381	Sharp Corporation
921	0x0399	Nikon Corporation
1117	0x045D	Boston Scientific Corporation
65535	0xFFFF	No Device ID

**Source:** Bluetooth SIG: Company Identifiers

Retrieved from

<https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers>

# Glossary

The following is a list of Bluetooth Low Energy terms and their meanings.

**Android** - An open-source operating system used for smartphones and tablet computers

**Attribute** - An unit of a GATT Profile which can be accessed by a Central, such as a Service or a Characteristic.

**Beacon** - A Bluetooth Low Energy Peripheral which continually Broadcasts so that Centrals can discern their location from information gleaned from the properties of the broadcast.

**Bluetooth Low Energy (BLE)** - A low power, short range wireless protocol used on micro electronics.

**Broadcast** - A feature of Bluetooth Low Energy where a Peripheral outputs a name and other specific data about a itself

**Central** - A Bluetooth Low Energy device that can connect to several Peripherals.

**Channel** - A finely-tuned radio frequency used for Broadcasting or data transmission.

**Characteristic** - A port or data endpoint where data can be read or written.

**Descriptor** - A feature of a Characteristic that allows for some sort of data interaction, such as Read, Write, or Notify.

**E0** - The encryption algorithm built into Bluetooth Low Energy.

**Generic Attribute (GATT) Profile** - A list of Services and Characteristics which are unique to a Peripheral and describe how data is served from the Peripheral. GATT profiles are hosted by a Peripheral

**iBeacon** - An Apple compatible Beacon which allows a Central to download a specific packet of data to inform the Central of its absolute location and other properties.

**Notify** - An operation where a Peripheral alerts a Central of a change in data.

**nRFx** - A series of Bluetooth-enabled programmable microcontrollers produced by Nordic Semiconductors®.

**Peripheral** - A Bluetooth Low Energy device that can connect to a single Central. Peripherals host a Generic Attribute (GATT) profile.

**Read** - An operation where a Central downloads data from a Characteristic.

**Scan** - The process of a Central searching for Broadcasting Peripherals.

**Scan Response** - A feature of Bluetooth Low Energy which allows Centrals to download a small packet of data without connecting.

**Service** - A container structure used to organize data endpoints. Services are hosted by a Peripheral.

**Universally Unique Identifier (UUID)** - A long, randomly generated alphanumeric string that is unique regardless of where it's used. UUIDs are designed to avoid name collisions that may happen when countless programs are interacting with each other.

**Write** - An operation where a Central alters data on a Characteristic.

(This page intentionally left blank)



# About the Author



Tony's infinite curiosity compels him to want to open up and learn about everything he touches, and his excitement compels him to share what he learns with others.

He has two true passions: branding and inventing.

His passion for branding led him to start a company that did branding and marketing in 4 countries for firms such as Apple, Intel, and Sony BMG. He loves weaving the elements of design, writing, product, and strategy into an

essential truth that defines a company.

His passion for inventing led him to start a company that uses brain imaging to quantify meditation and to predict seizures, a company acquired \$1.5m in funding and was incubated in San Francisco where he currently resides.

Those same passions have led him on some adventures as well, including living in a Greek monastery with orthodox monks and to tagging along with a gypsy in Spain to learn to play flamenco guitar.

(This page intentionally left blank)

# About this Book

This book is a practical guide to programming Bluetooth Low Energy for Android phones and Tablets

In this book, you will learn the basics of how to program an Android device to communicate with any Central or Peripheral device over Bluetooth Low Energy. Each chapter of the book builds on the previous one, culminating in three projects:

- A Beacon and Scanner
- An Echo Server and Client
- A Remote Controlled Device

Through the course of the book you will learn important concepts that relate to:

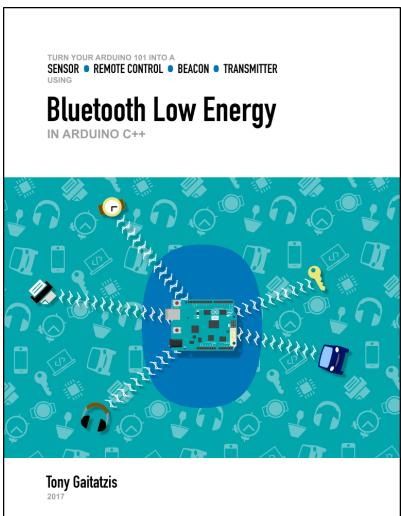
- How Bluetooth Low Energy works
- How data is sent and received
- Common paradigms for handling data

## Skill Level

This book is excellent for anyone who has basic or advanced knowledge of Java programming on Android.

# Other Books in this Series

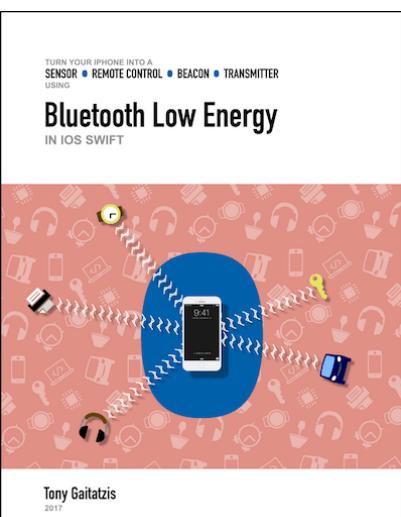
If you are interested in programming other Bluetooth Low Energy Devices, please check out the other books in this series or visit [bluetoothlowenergybooks.com](http://bluetoothlowenergybooks.com):



## Bluetooth Low Energy Programming in Arduino 101

Tony Gaitatzis, 2017

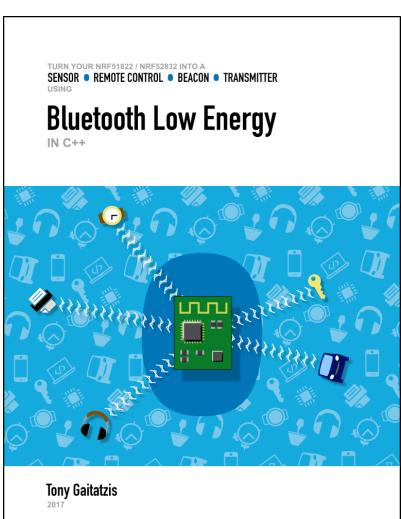
ISBN: 978-1-7751280-6-9



## Bluetooth Low Energy Programming in iOS Swift

Tony Gaitatzis, 2017

ISBN: 978-1-7751280-5-2



## Bluetooth Low Energy Programming in C++ for nRF Microcontrollers

Tony Gaitatzis, 2017

ISBN: 978-1-7751280-7-6

(This page intentionally left blank)