Mobile Computing Course
6th Session

Bluetooth

Bluetooth overview

The Android platform includes support for the Bluetooth network stack, which allows a device to wirelessly exchange data with other Bluetooth devices. The application framework provides access to the Bluetooth functionality through the Android Bluetooth APIs. These APIs let applications wirelessly connect to other Bluetooth devices, enabling point-to-point and multipoint wireless features.

Using the Bluetooth APIs, an Android application can perform the following:

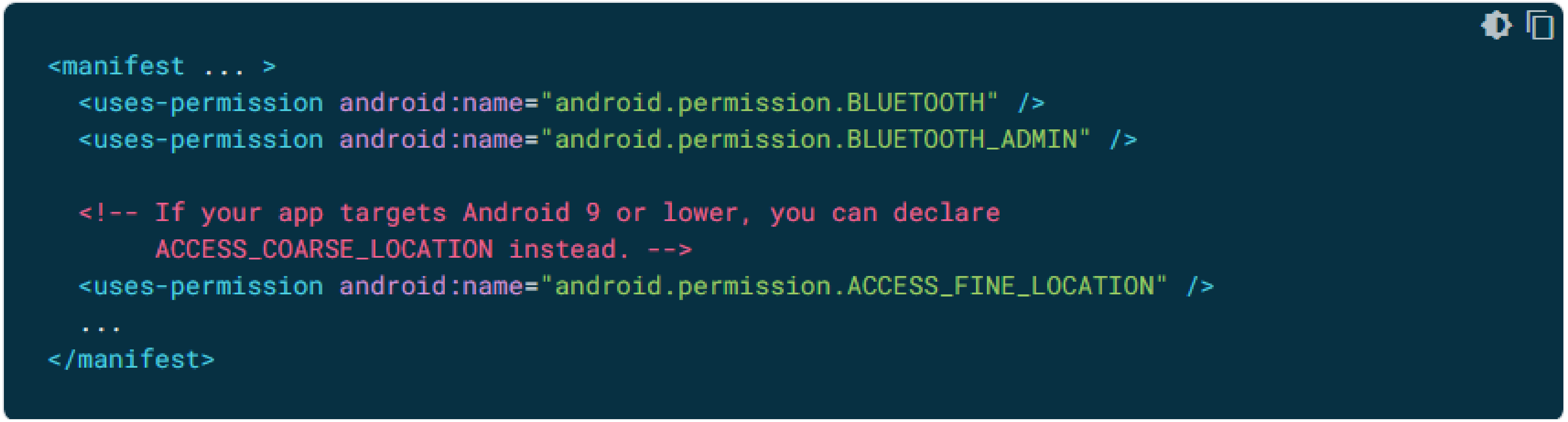
- Scan for other Bluetooth devices
- Query the local Bluetooth adapter for paired Bluetooth devices
- Establish RFCOMM channels
- Connect to other devices through service discovery
- Transfer data to and from other devices
- Manage multiple connections

The basics

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

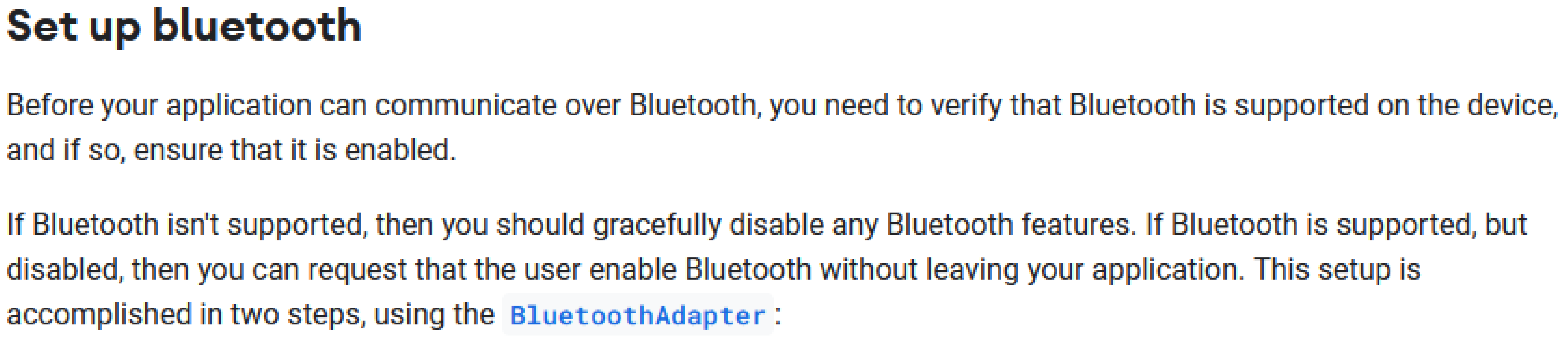
Bluetooth permissions

Declare the Bluetooth permission(s) in your application manifest file. For example:



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

  <!-- If your app targets Android 9 or lower, you can declare
       ACCESS_COARSE_LOCATION instead. -->
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  ...
</manifest>
```

If you want your app to initiate device discovery or manipulate Bluetooth settings, you must declare the BLUETOOTH_ADMIN permission in addition to the BLUETOOTH permission.



Set up bluetooth

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

If Bluetooth isn't supported, then you should gracefully disable any Bluetooth features. If Bluetooth is supported, but disabled, then you can request that the user enable Bluetooth without leaving your application. This setup is accomplished in two steps, using the `BluetoothAdapter`:

1. Get the BluetoothAdapter.
2. Enable Bluetooth.

1. Get the `BluetoothAdapter`.

   The `BluetoothAdapter` is required for any and all Bluetooth activity. To get the `BluetoothAdapter`, call the static `getDefaultAdapter()` method. This returns a `BluetoothAdapter` that represents the device's own Bluetooth adapter (the Bluetooth radio). There's one Bluetooth adapter for the entire system, and your application can interact with it using this object. If `getDefaultAdapter()` returns `null`, then the device doesn't support Bluetooth. For example:

Kotlin Java

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

2. Enable Bluetooth.
   Next, you need to ensure that Bluetooth is enabled. Call `isEnabled()` to check whether Bluetooth is currently enabled. If this method returns false, then Bluetooth is disabled. To request that Bluetooth be enabled, call `startActivityForResult()`, passing in an `ACTION_REQUEST_ENABLE` intent action. This call issues a request to enable Bluetooth through the system settings (without stopping your application). For example:

   Kotlin Java

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

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

   The `REQUEST_ENABLE_BT` constant passed to `startActivityForResult()` is a locally defined integer that must be greater than 0. The system passes this constant back to you in your `onActivityResult()` implementation as the `requestCode` parameter.

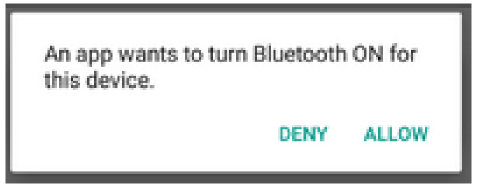
   
   An app wants to turn Bluetooth ON for this device.

   Figure 1: The enabling Bluetooth dialog.

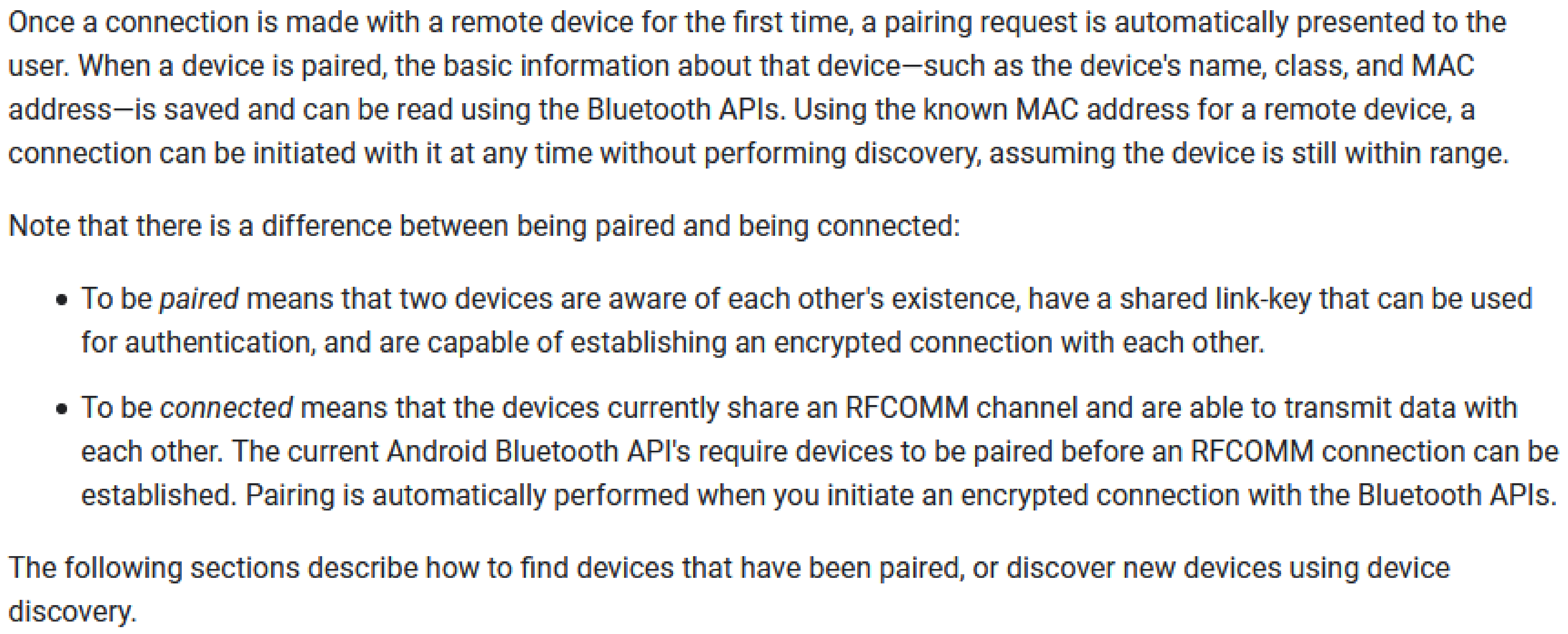
   If enabling Bluetooth succeeds, your activity receives the `RESULT_OK` result code in the `onActivityResult()` callback. If Bluetooth was not enabled due to an error (or the user responded "No") then the result code is `RESULT_CANCELED`.

Find devices

- Using the BluetoothAdapter, you can find remote Bluetooth devices either through
  - ❖ device discovery or
  - ❖ by querying the list of paired devices.

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

Find devices



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

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

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

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

Query paired devices

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

Kotlin Java

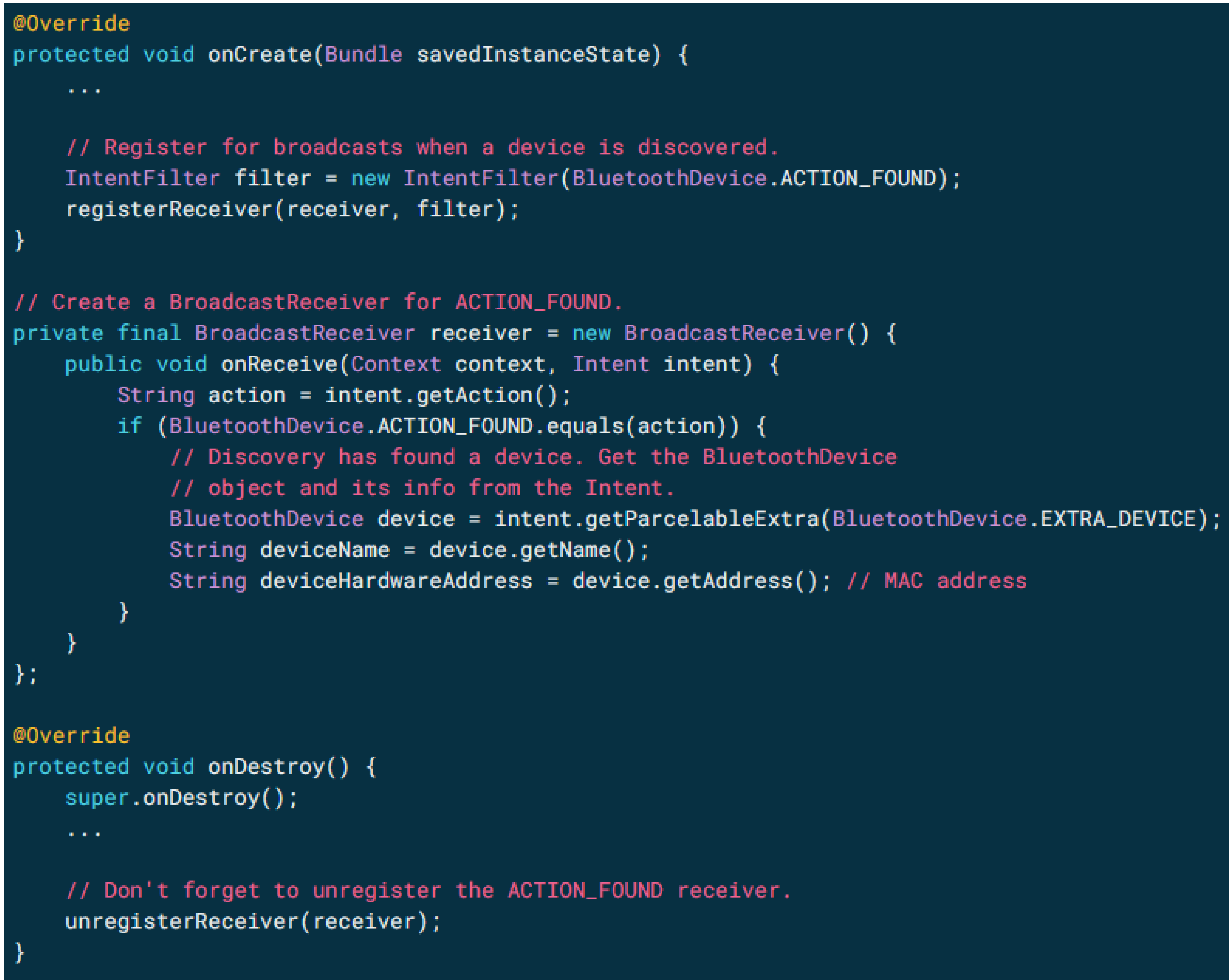
```java
Set<BluetoothDevice> pairedDevices = bluetoothAdapter.getBondedDevices();

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

To initiate a connection with a Bluetooth device, all that's needed from the associated `BluetoothDevice` object is the MAC address, which you retrieve by calling `getAddress()`. You can learn more about creating a connection in the section about Connecting Devices.

Discover devices

- To start discovering devices, simply call startDiscovery(). The process is asynchronous and returns a boolean value indicating whether discovery has successfully started. The discovery process usually involves an inquiry scan of about 12 seconds, followed by a page scan of each device found to retrieve its Bluetooth name.
- In order to receive information about each device discovered, your application must register a BroadcastReceiver for the ACTION_FOUND intent. The system broadcasts this intent for each device. The intent contains the extra fields EXTRA_DEVICE and EXTRA_CLASS, which in turn contain a BluetoothDevice and a BluetoothClass, respectively.

The following code snippet shows how you can register to handle the broadcast when devices are discovered:



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

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

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

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

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

Enable discoverability

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

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

Kotlin   Java

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

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

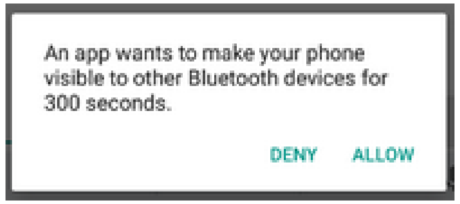

An app wants to make your phone visible to other Bluetooth devices for 300 seconds.

Figure 2: The enabling discoverability dialog.

Connect devices

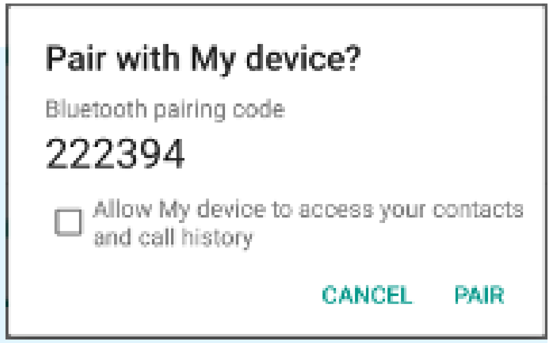
In order to create a connection between two devices, you must implement both the server-side and client-side mechanisms because one device must open a server socket, and the other one must initiate the connection using the server device's MAC address. The server device and the client device each obtain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

The server and client are considered connected to each other when they each have a connected `BluetoothSocket` on the same RFCOMM channel. At this point, each device can obtain input and output streams, and data transfer can begin, which is discussed in the section about Manage a connection. This section describes how to initiate the connection between two devices.

Connection techniques

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

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


Pair with My device?

Bluetooth pairing code

222394

☐ Allow My device to access your contacts and call history

Figure 3: The Bluetooth pairing dialog.

Connect as a server

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

Connect as a server

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

1. Get a `BluetoothServerSocket` by calling `listenUsingRfcommWithServiceRecord()`.

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

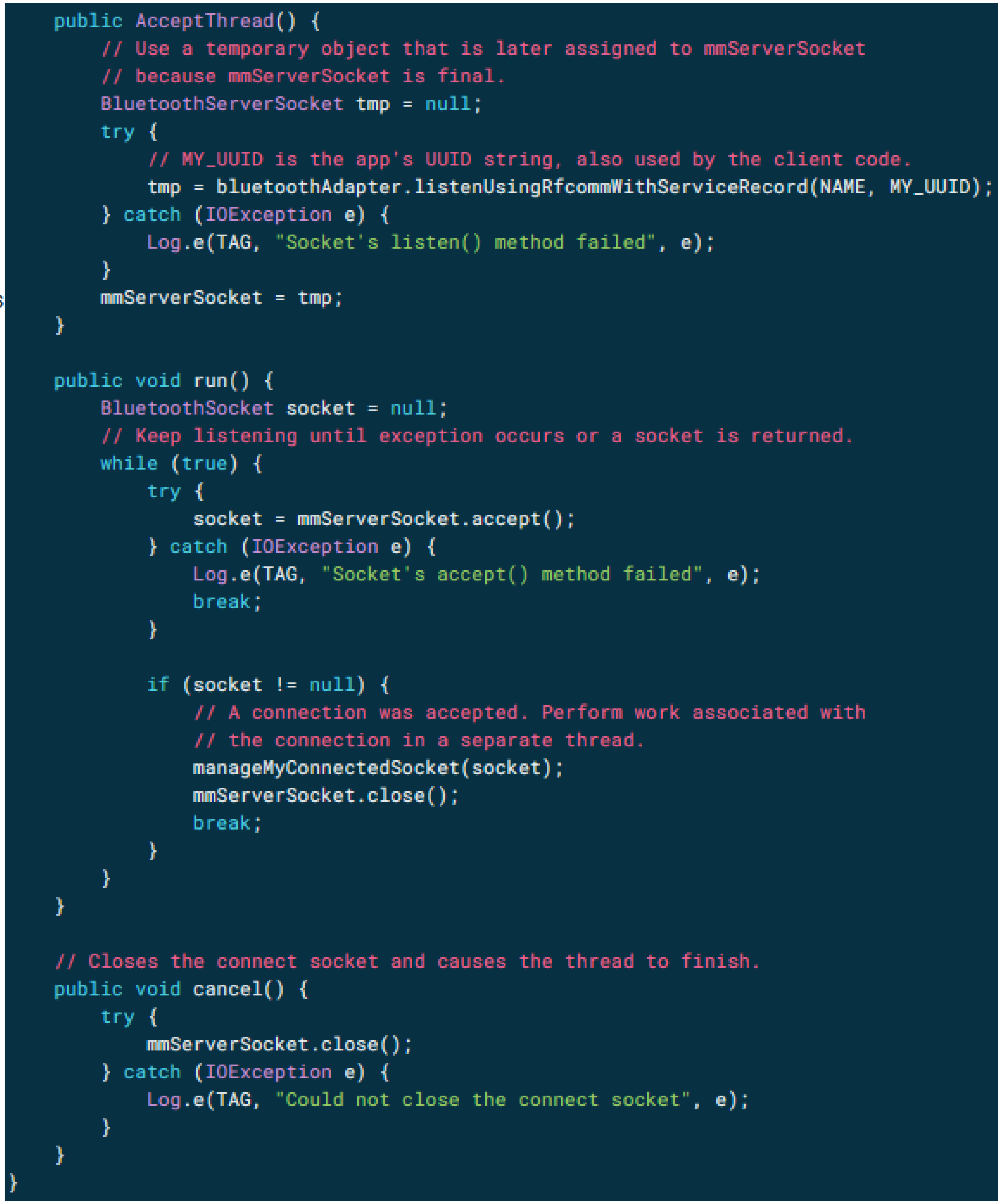
   A UUID is a standardized 128-bit format for a string ID used to uniquely identify information. The point of a UUID is that it's big enough that you can select any random ID and it doesn't clash with any other ID. In this case, it's used to uniquely identify your application's Bluetooth service. To get a UUID to use with your application, you can use one of the many random UUID generators on the web, then initialize a `UUID` with `fromString(String)`.

2. Start listening for connection requests by calling `accept()`.

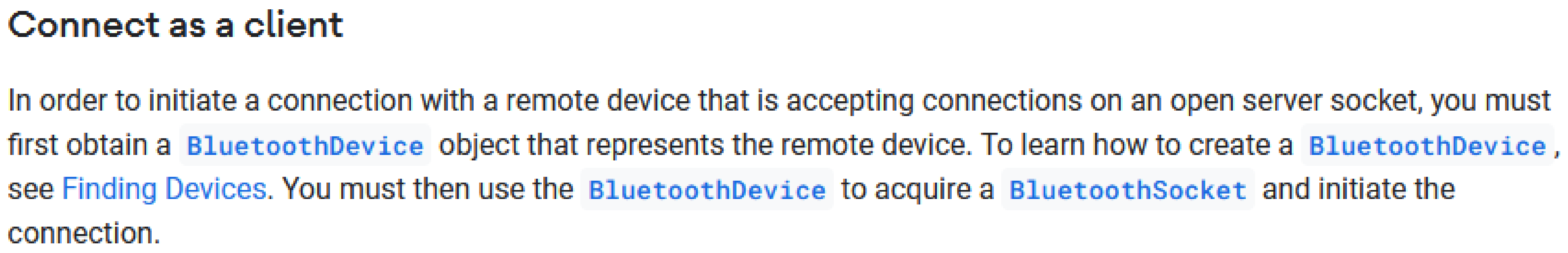
   This is a blocking call. It returns when either a connection has been accepted or an exception has occurred. A connection is accepted only when a remote device has sent a connection request containing a UUID that matches the one registered with this listening server socket. When successful, `accept()`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

3. Unless you want to accept additional connections, call `close()`.

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



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

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

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

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



Connect as a client

In order to initiate a connection with a remote device that is accepting connections on an open server socket, you must first obtain a `BluetoothDevice` object that represents the remote device. To learn how to create a `BluetoothDevice`, see Finding Devices. You must then use the `BluetoothDevice` to acquire a `BluetoothSocket` and initiate the connection.

Connect as a client

The basic procedure is as follows:

1. Using the `BluetoothDevice`, get a `BluetoothSocket` by calling `createRfcommSocketToServiceRecord(UUID)`.

   This method initializes a `BluetoothSocket` object that allows the client to connect to a `BluetoothDevice`. The UUID passed here must match the UUID used by the server device when it called `listenUsingRfcommWithServiceRecord(String, UUID)` to open its `BluetoothServerSocket`. To use a matching UUID, hard-code the UUID string into your application, and then reference it from both the server and client code.

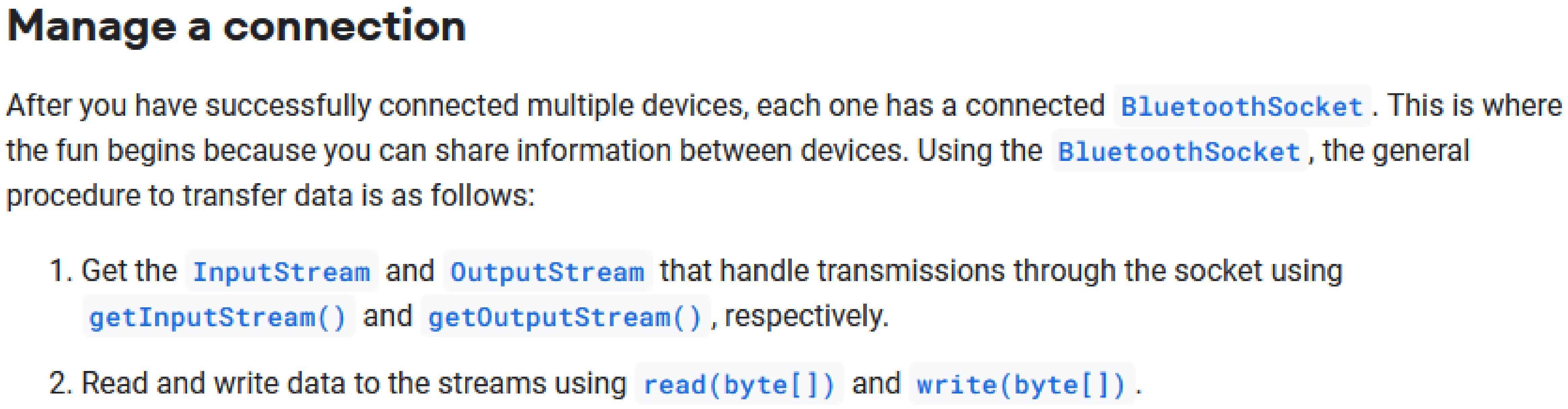
2. Initiate the connection by calling `connect()`. Note that this method is a blocking call.

   After a client calls this method, the system performs an SDP lookup to find the remote device with the matching UUID. If the lookup is successful and the remote device accepts the connection, it shares the RFCOMM channel to use during the connection, and the `connect()` method returns. If the connection fails, or if the `connect()` method times out (after about 12 seconds), then the method throws an `IOException`.

   Because `connect()` is a blocking call, you should always perform this connection procedure in a thread that is separate from the main activity (UI) thread.

Note: You should always call `cancelDiscovery()` to ensure that the device isn't performing device discovery before you call `connect()`. If discovery is in progress, then the connection attempt is significantly slowed, and it's more likely to fail.

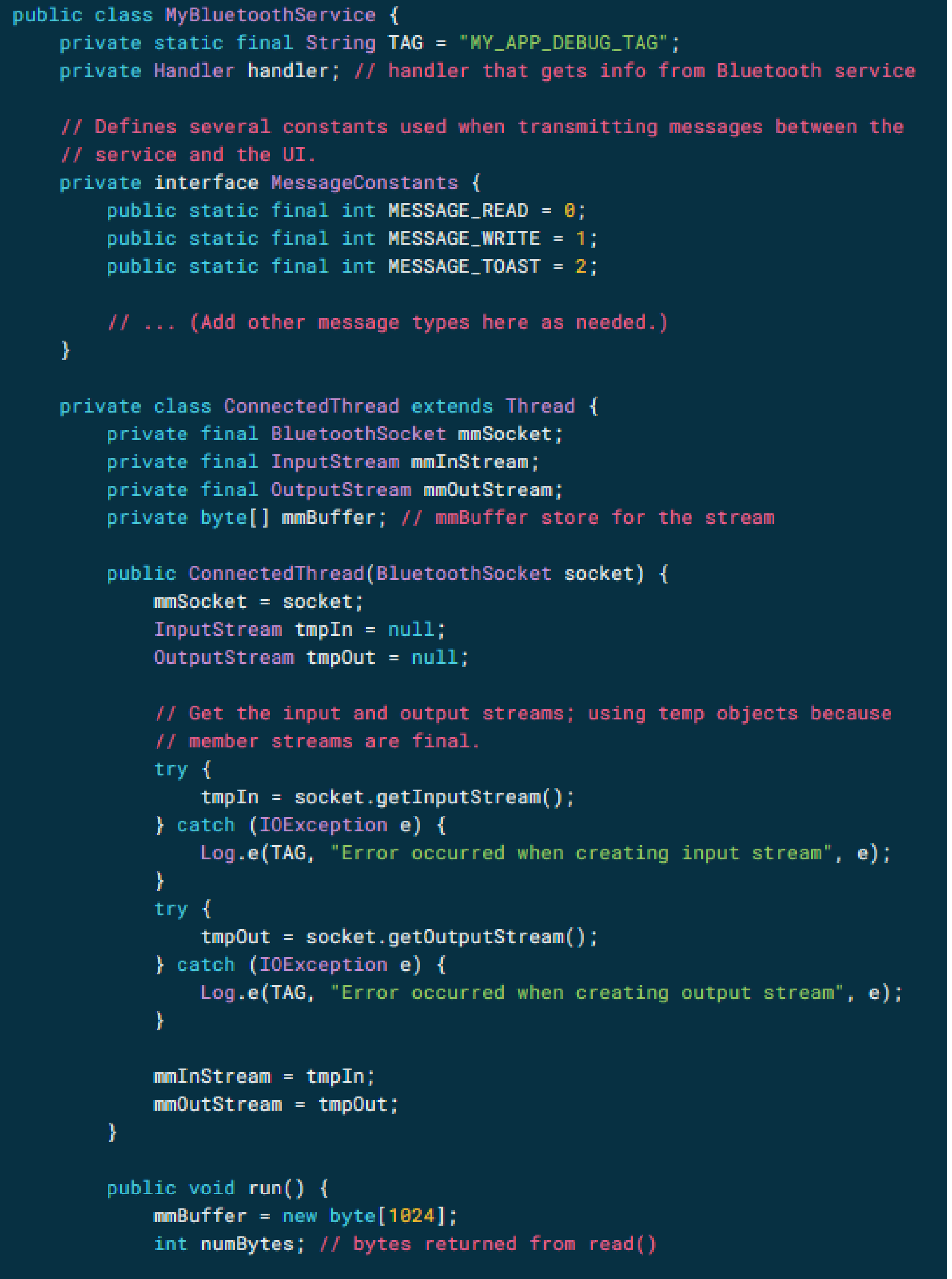
```java
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

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

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

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

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

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

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



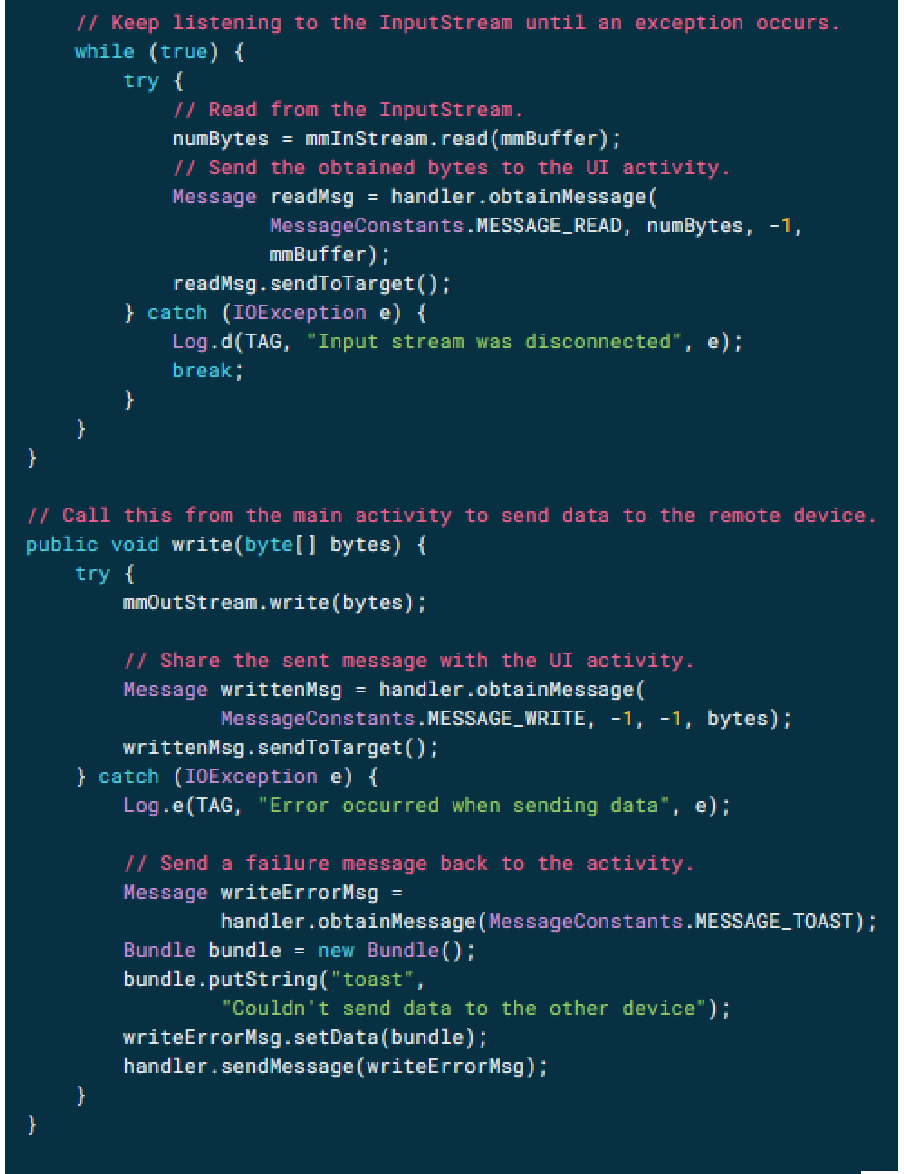
Manage a connection

After you have successfully connected multiple devices, each one has a connected `BluetoothSocket`. This is where the fun begins because you can share information between devices. Using the `BluetoothSocket`, the general procedure to transfer data is as follows:

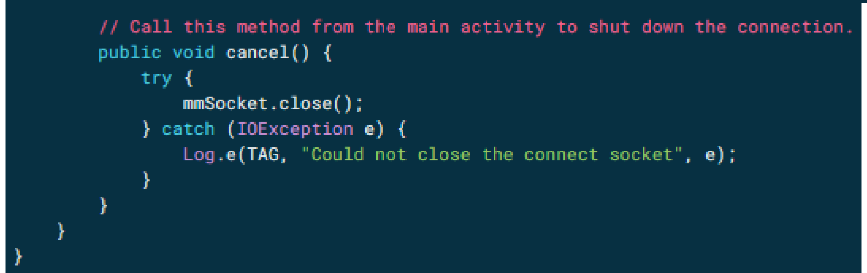
1. Get the `InputStream` and `OutputStream` that handle transmissions through the socket using `getInputStream()` and `getOutputStream()`, respectively.
2. Read and write data to the streams using `read(byte[])` and `write(byte[])`.



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

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

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

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

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

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

            mmInStream = tmpIn;
            mmOutStream = tmpOut;
        }

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



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

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

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

                // Send a failure message back to the activity.
                Message writeErrorMsg =
                        handler.obtainMessage(MessageConstants.MESSAGE_TOAST);
                Bundle bundle = new Bundle();
                bundle.putString("toast",
                        "Couldn't send data to the other device");
                writeErrorMsg.setData(bundle);
                handler.sendMessage(writeErrorMsg);
            }
        }
```



```java
        // Call this method from the main activity to shut down the connection.
        public void cancel() {
            try {
                mmSocket.close();
            } catch (IOException e) {
                Log.e(TAG, "Could not close the connect socket", e);
            }
        }
    }
}
```

Key classes and interfaces

All of the Bluetooth APIs are available in the `android.bluetooth` package. Here's a summary of the classes and interfaces you need to create Bluetooth connections:

`BluetoothAdapter`

Represents the local Bluetooth adapter (Bluetooth radio). The `BluetoothAdapter` is the entry-point for all Bluetooth interaction. Using this, you can discover other Bluetooth devices, query a list of bonded (paired) devices, instantiate a `BluetoothDevice` using a known MAC address, and create a `BluetoothServerSocket` to listen for communications from other devices.

`BluetoothDevice`

Represents a remote Bluetooth device. Use this to request a connection with a remote device through a `BluetoothSocket` or query information about the device such as its name, address, class, and bonding state.

`BluetoothSocket`

Represents the interface for a Bluetooth socket (similar to a TCP `Socket`). This is the connection point that allows an application to exchange data with another Bluetooth device using `InputStream` and `OutputStream`.

`BluetoothServerSocket`

Represents an open server socket that listens for incoming requests (similar to a TCP `ServerSocket`). In order to connect two Android devices, one device must open a server socket with this class. When a remote Bluetooth device makes a connection request to this device, the device accepts the connection, then returns a connected `BluetoothSocket`.

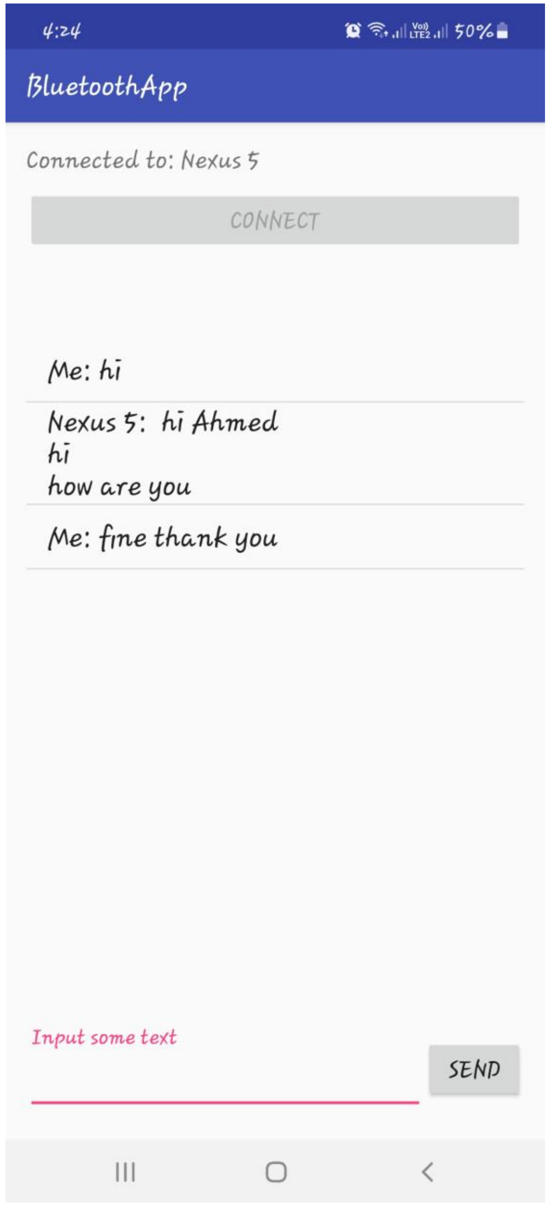
`BluetoothClass`

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

Example: BluetoothChatAppAndroid

Make a simple chat application through bluetooth in Android

- Project: https://github.com/DevExchanges/BluetoothChatAppAndroid
- Example Details: http://www.devexchanges.info/2016/10/simple-bluetooth-chat-application-in.htm



Connected to: Nexus 5

Me: hi

Nexus 5: hi Ahmed
hi
how are you

Me: fine thank you

Task !!!

Create a wireless networks logger android application: an application that collects the RSSI of available Access points , Bluetooth device name and its MAC address and Cell info.

See you in next session