Start advertising

When the Android Things program starts, it should start advertising, so that other devices can see which BLE services it exposes, and can connect to it.

```
// The BluetoothAdapter is required for any and all Bluetooth activity.
mBluetoothManager = (BluetoothManager) getSystemService(BLUETOOTH_SERVICE);
BluetoothAdapter bluetoothAdapter = mBluetoothManager.getAdapter();

// Some advertising settings. We don't set an advertising timeout
// since our device is always connected to AC power.
AdvertiseSettings settings = new AdvertiseSettings.Builder()
        .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
        .setConnectable(true)
        .setTimeout(0)
        .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
        .build();

// Defines which service to advertise.
AdvertiseData data = new AdvertiseData.Builder()
        .setIncludeDeviceName(true)
        .setIncludeTxPowerLevel(false)
        .addServiceUuid(new ParcelUuid(SERVICE_ID))
        .build();

// Starts advertising.
mBluetoothLeAdvertiser = bluetoothAdapter.getBluetoothLeAdvertiser();
mBluetoothLeAdvertiser.startAdvertising(settings, data, mAdvertiseCallback);
```

Advertising is battery-intensive. Here, our device is always connected to AC power so it will advertise continuously.
If it runs on battery, a good idea would be to add a timeout, and a physical button to start the advertising process. Also, you will need to stop the advertising once a client is connected.

The `startAdvertising` method needs an `AdvertiseCallback` instance, defined below:

```
private AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
    @Override
    public void onStartSuccess(AdvertiseSettings settingsInEffect) {
        Log.i(TAG, "LE Advertise Started.");
    }

    @Override
    public void onStartFailure(int errorCode) {
        Log.w(TAG, "LE Advertise Failed: " + errorCode);
    }
};
```

Creating the GATT service

We have to programmatically define our GATT service.
Remember, our service should contain 2 characteristics:

- A counter (read-only, supports subscriptions via a config descriptor)
- An interactor (write-only)

```
private BluetoothGattService createService() {
  BluetoothGattService service = new BluetoothGattService(SERVICE_UUID, SERVICE_TYPE_P

  // Counter characteristic (read-only, supports subscriptions)
  BluetoothGattCharacteristic counter = new BluetoothGattCharacteristic(CHARACTERISTIC_
  BluetoothGattDescriptor counterConfig = new BluetoothGattDescriptor(DESCRIPTOR_CONFI
  counter.addDescriptor(counterConfig);

  // Interactor characteristic
  BluetoothGattCharacteristic interactor = new BluetoothGattCharacteristic(CHARACTERIS

  service.addCharacteristic(counter);
  service.addCharacteristic(interactor);
  return service;
}
```

Starting the server

Then, we start the Bluetooth LE server with the `openGattServer` method.

```
mGattServer = mBluetoothManager.openGattServer(mContext, mGattServerCallback);
mGattServer.addService(createService());
```

This method takes a `BluetoothGattServerCallback` instance, which contains callbacks to implement when a characteristic / descriptor is read or written.

Returning the counter value

When a GATT client reads on the `CHARACTERISTIC_COUNTER_UUID`, we should return the value of the counter.
For that, we override the `onCharacteristicReadRequest` method of our `BluetoothGattServerCallback`, and return the `currentCounterValue` if there is a read request on the counter characteristic:

```
@Override
public void onCharacteristicReadRequest(BluetoothDevice device,
    int requestId, int offset, BluetoothGattCharacteristic characteristic) {
  if (CHARACTERISTIC_COUNTER_UUID.equals(characteristic.getUuid())) {
    byte[] value = Ints.toByteArray(currentCounterValue);
    mGattServer.sendResponse(device, requestId, GATT_SUCCESS, 0, value);
  }
}
```

Incrementing the counter

When a GATT client writes on the `CHARACTERISTIC_INTERACTOR_UUID`, we should increment the value of the counter.
For that, we can override the `onCharacteristicWriteRequest` method:

```
@Override
public void onCharacteristicWriteRequest(BluetoothDevice device,
    int requestId, BluetoothGattCharacteristic characteristic,
    boolean preparedWrite, boolean responseNeeded, int offset, byte[] value) {
  if (CHARACTERISTIC_INTERACTOR_UUID.equals(characteristic.getUuid())) {
    currentCounterValue++;
    notifyRegisteredDevices();
  }
}
```

Notice here the `notifyRegisteredDevices()` call.
Since the counter value has changed, we should notify devices. We will see the implementation later, but first, let’s handle the subscription.

Handling notifications

If a client wants to be notified of any changes in the counter characteristic value, it should write its intent on a config descriptor.
We override the `onDescriptorWriteRequest` and keep a reference of the Bluetooth device in a private list named `mRegisteredDevices`:

```
@Override
public void onDescriptorWriteRequest(BluetoothDevice device,
    int requestId, BluetoothGattDescriptor descriptor,
    boolean preparedWrite, boolean responseNeeded, int offset, byte[] value) {
  if (DESCRIPTOR_CONFIG_UUID.equals(descriptor.getUuid())) {
    if (Arrays.equals(ENABLE_NOTIFICATION_VALUE, value)) {
      mRegisteredDevices.add(device);
    } else if (Arrays.equals(DISABLE_NOTIFICATION_VALUE, value)) {
      mRegisteredDevices.remove(device);
    }

    if (responseNeeded) {
      mGattServer.sendResponse(device, requestId, GATT_SUCCESS, 0, null);
    }
  }
}
```

Now, we can create our `notifyRegisteredDevices` method that simply calls `notifyCharacteristicChanged` for each subscribed devices:

```
private void notifyRegisteredDevices() {
  BluetoothGattCharacteristic characteristic = mGattServer
```

```
        .getService(SERVICE_UUID)
        .getCharacteristic(CHARACTERISTIC_COUNTER_UUID);

    for (BluetoothDevice device : mRegisteredDevices) {
      byte[] value = Ints.toByteArray(currentCounterValue);
      counterCharacteristic.setValue(value);
      mGattServer.notifyCharacteristicChanged(device, characteristic, false);
    }
  }
```