**Lab 8: Bluetooth Low Energy**

In case you found something to improve, please tell us!
https://forms.gle/Nf27cXFf7AwaBL55A

In this lab we are going to learn how to interact with Bluetooth Low Energy (BLE) peripherals. This includes setting up a BLE connection and working with a chest belt for heart rate measurement.

# 1 Android Studio Tricks

Here are very useful **Android Studio** tricks you should always use (check *Section 5* of **Lab1b** for more detailed explanation on how to use **Android Studio** tools):

1. Use **Alt+Enter** (**Option+Enter** for Mac users) when you have an error in your code: put the cursor on the error and click **Alt+Enter**. You can also use it to update the **`gradle`** dependencies to the latest version.
2. Use **Ctrl+Space** to check the documentation of a **`View`**, method or attribute: put the cursor on the object and do **Ctrl+Space**. You can also use it to complete the typing of these objects. Otherwise **Android Studio** always gives a list of suggestions where you can choose the object you need.
3. **Check for compilation errors:** They are usually quite self-explanatory.
4. **Check errors in logcat, and use the debugger:** errors are highlighted in logcat, click on the underlined blue line to go in the position of the code where the error is.

For more **useful keyboard shortcuts**, please check this LINK[1]

# 2 Introduction

Bluetooth Low Energy (BLE), also called Bluetooth Smart, is designed to consume less energy. It was included in version 4.0 of the Bluetooth Specifications which was adopted in 2010.

It is based on a central-peripheral architecture meaning that the central device scans, looking for advertisement, and the peripheral one advertises itself. They can use the Generic Attribute Profile (GATT) specification to communicate. In this lesson, the tablet will be the GATT client. It will scan and discover all services provided by the server as well as the different characteristics of a given service.

[1] <https://developer.android.com/studio/intro/keyboard-shortcuts>

We will use a cardiac chest band (such as the Polar H7 body sensor) as the server. We will integrate it in our current Sports Tracker app as an alternative source of data for the heart rate acquisition.

# 3 Android samples

Google provides code samples for Android, already available through *Android Studio*. To ease the Bluetooth setup, we will first use one of these sample projects. Then, we will extract parts of it to only select the components we are interested in.

## 3.1 Import the sample

In this first section, we will import the Google Sample related to Bluetooth Low Energy and test it to be sure it works.

To import the desired Google sample:

1. Launch Android Studio and click on **Import an Android code sample**
2. In the search bar of the window which pops up, type **Bluetooth**
3. In the results, select **Bluetooth Le Gatt** project within **Connectivity** module

*Android Studio* will then download the sample and create a new project containing four java files in your workspace. The project is composed of four java files:

- **`DeviceScanActivity`**`.java`: The first Activity, in charge of scanning to display a list of the current Bluetooth devices available nearby. When the user clicks on a device, an **`Intent`** is triggered which start the **`DeviceControlActivity`**.
- **`DeviceControlActivity`**`.java`: The second **`Activity`**, in charge of displaying the services provided by the selected device as well as the characteristics of the services. Once a characteristic is selected, its current value is displayed in the Data field. The two other fields are the device MAC address and the state of the connection.
- **`BluetoothLeService`**`.java`: The service used to manage a connection and data connection with a GATT server.
- **`SampleGattAttribute`**`.java`: A class containing a few standards GATT attributes. You can see that it already contains the unique numeric ID (UUID) of the Heart Rate Service and the Heart Rate Measurement characteristic.

### 3.2 Fix the sample

Before running the sample we should add a code in order to update the sample according to Android permissions. Let's add the required code:

In **AndroidManifest.xml**:

```
<!-- Application -> manifests -> AndroidManifest.xml -> <manifest ...> -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

In the **DeviceScanActivity**, add runtime permission in the **onCreate(...)** as we did for the ***wear*** app for the *Sensors* lab:

```
/** Application -> DeviceScanActivity.java -> DeviceScanActivity -> onCreate(...) **/
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M
        && checkSelfPermission("android.permission.ACCESS_FINE_LOCATION")
                == PackageManager.PERMISSION_DENIED) {
    requestPermissions(new String[]{"android.permission.ACCESS_FINE_LOCATION"}, 0);
}
```

### 3.3 Run the sample

If you run the app, you should obtain a result similar to Figure 1. In the image on the right, to register for heart-rate updates and display the received values, we selected the **Heart Rate Service** and then its **Heart Rate Measurement** characteristic. The current heart-rate is displayed as well as the MAC address and the state of the connection.

***Note:*** The first 6 hexadecimal numbers are the manufacturer Organizationally Unique Identifier (OUI). If do a quick search, you'll find that the number **00:22:D0** corresponds to the Polar company.

## 4 Pick the right pieces

Now that we have a functioning core for the BLE app, we can use parts of it on our sport-tracker app.

### 4.1 Overview

A small diagram of the sample is available in Figure 2. We will need to copy all the required elements and dependencies into our own project.

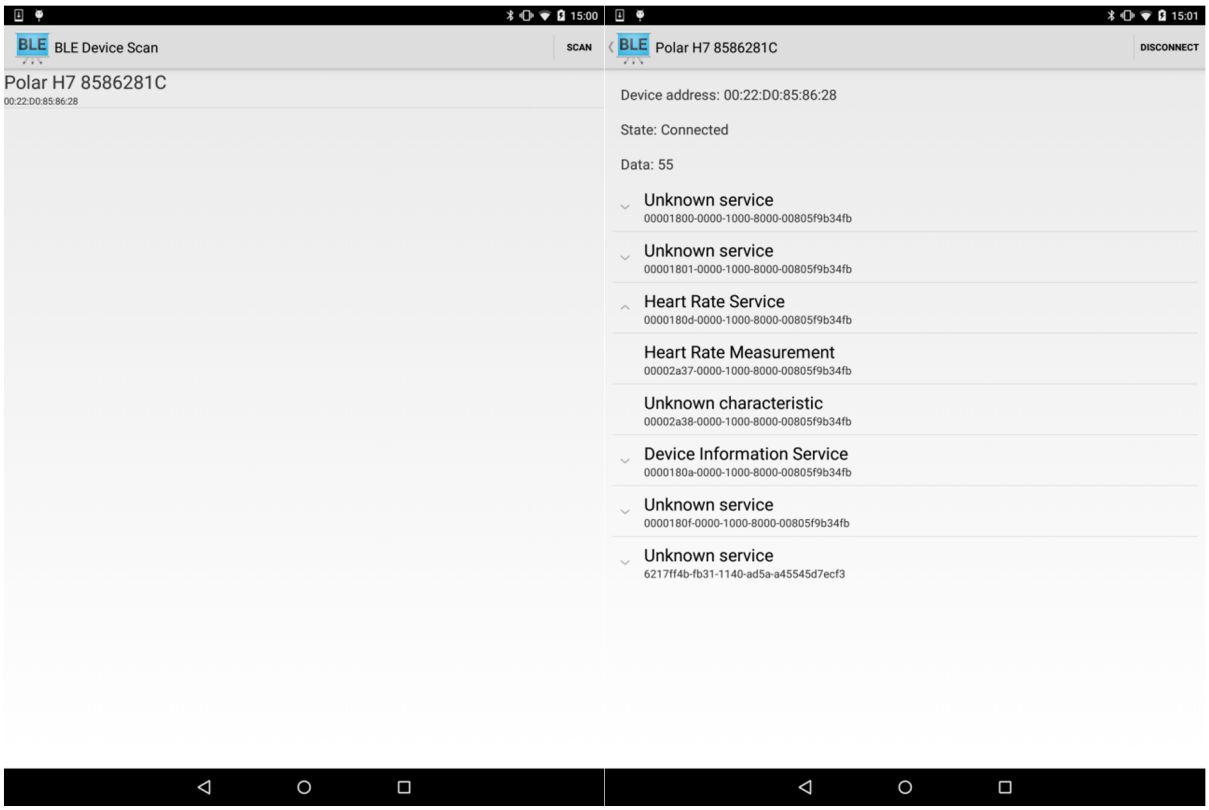


Figure 1: Result of running the Bluetooth sample

## 4.2 The files to copy to our sport-tracker app

The first activity in Figure 2, **DeviceScanActivity**, does a scan to search the nearby Bluetooth devices and displays them as list. We will copy it with no change. However, the second activity, **DeviceControlActivity**, needs to be merged with our **ExerciceLiveActivity**. First, copy the required elements:

- From **BluetoothLeGatt -> AndroidManifest.xml**:
  - the complementary permissions
  - the **DeviceScanActivity** (without **intent-filter**s)
  - the service
- the **BluetoothLeService** (read note bellow)
  - and its dependency **SampleGattAttribute**
- the **DeviceScanActivity**
- merge the **res -> values -> string** resources files
- the content of 'res -> menu' folder
- the layouts:
  - **actionbar_indeterminate_progress.xml**

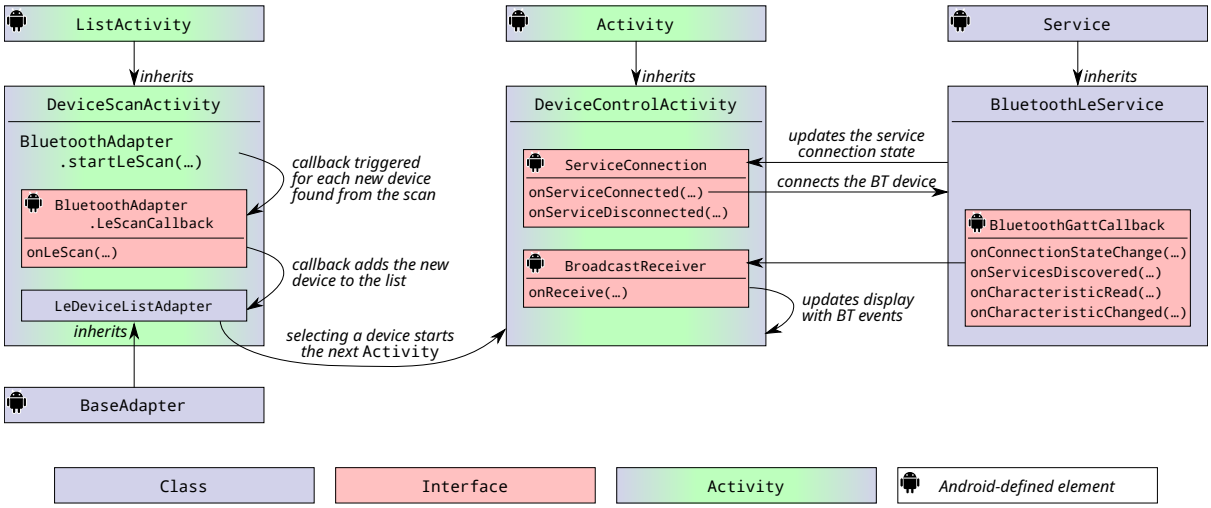


Figure 2: Architecture of the Bluetooth Low-Energy sample

- **listitem_device.xml**

***Note:*** for Java files, if you drag'n'drop or copy/paste in the target folder from *Android Studio*, it will update the package declaration accordingly. Otherwise, change the package name for each copied java file to match the name of the project.

## 4.3 The few code fixes

After copying the required files to our project, there are few things to correct before merging **DeviceControlActivity**:

1. Bluetooth Low Energy requires API level 18 at least. In the **...-> mobile -> build.gradle** from the *mobile* module of the project, change the **minSdkVersion x** to **18**.
2. As we will start the **DeviceScanActivity** for **result**, we need to change a bit the logic required in the **DeviceScanActivity -> onListItemClick(...)**
   - The intent creation is now empty: **final Intent intent = new Intent();**
   - We finish the activity rather then starting a new one: **startActivity(intent)** becomes **finish();**
   - We set the result (just before **finish()**) by adding **setResult(Activity.RESULT_OK, intent);**

## 4.4 DeviceControlActivity

In our sports-tracker app, the **BluetoothLeGatt -> DeviceControlActivity** is our **ExerciseLiveActivity**. We need to integrate everything needed from it to our project:

1. Copy the following string constants:
   - **EXTRAS_DEVICE_NAME**
   - **EXTRAS_DEVICE_ADDRESS**
2. In **DeviceScanActivity**, update the references from **DeviceControlActivity** to **ExerciseLiveActivity**. (Now the project should compile, but we are not finished yet!)
3. Copy the following class members (and any depending fields):
   - **ServiceConnection mServiceConnection** and its implementation
   - **BroadcastReceiver mGattUpdateReceiver**, but however needs:
     - the removal of the **updateConnectionState(...)** calls
     - the removal of the **clearUI()** call
4. From the **BluetoothLeGatt -> DeviceControlActivity -> onCreate(...)**, copy everything not related to the display (excluding also the **getActionBar()** calls). Basically the **intents** and **bindService(...)**.
5. You will also need the **IntentFilter** method used to register the receiver **makeGattUpdateIntentFilter()**
6. Import all the code in relation to the receivers from the **onResume()**, **onPause()**, **onDestroy()**
7. We will copy and modify the **displayGattServices(...)** so:
   - its name becomes **registerHeartRateService(...)**
   - Remove the references to the **SimpleExpandableListAdapter** as it was used for display in the user interface and we will not need it
   - This method uses two nested **for** loops. The first loop is used to select the services one by one. Then, the second loop finds every characteristic of a given service. You can remove all the code not strictly required to run the loops (all **HashMap**s, **ArrayList**s and **String**s, with the exception of the **uuid** one), and simply add the following test in the inner loop to register to the *Heart Rate Measurement* from the *Heart Rate Service*

     ```
     // Find heart rate measurement (0x2A37)
     if (SampleGattAttributes.lookup(uuid, "unknown")
         .equals("Heart Rate Measurement")) {
     mBluetoothLeService.setCharacteristicNotification(
             gattCharacteristic, true);
     }
     ```

     The function should be in the range of 10-20 code statements after these modi-

fications, with the **BluetoothLeService** class listening for new heart-rate measurements, as follows:

```
private void registerHeartRateService(
        List<BluetoothGattService> gattServices) {
    if (gattServices == null) return;
    String uuid = null;
    // Loops through available GATT Services.
    for (BluetoothGattService gattService : gattServices) {
        List<BluetoothGattCharacteristic> gattCharacteristics =
                gattService.getCharacteristics();
        // Loops through available Characteristics.
        for (BluetoothGattCharacteristic
                gattCharacteristic : gattCharacteristics) {
            uuid = gattCharacteristic.getUuid().toString();
            // Find heart rate measurement (0x2A37)
            if (SampleGattAttributes.lookup(uuid, "unknown")
                    .equals("Heart Rate Measurement")) {
                Log.i(TAG, "Registering for HR measurement");
                mBluetoothLeService.setCharacteristicNotification(
                        gattCharacteristic, true);
            }
        }
    }
}
```

- Currently, the heart-rate value is an integer in the service, converted to a string, sent in the broadcast and received as a string. It's more elegant to keep it as an **int** from the beginning to the end:

```
//// From BluetoothLeService:
// Replace:
intent.putExtra(EXTRA_DATA, String.valueOf(heartRate));
// with:
intent.putExtra(EXTRA_DATA, heartRate);
//// From ExerciceLiveActivity
// Replace:
displayData(intent.getStringExtra(BluetoothLeService.EXTRA_DATA));
// with:
displayData(intent.getIntExtra(BluetoothLeService.EXTRA_DATA, 0));
```

- Finally, create the **displayData(...)** in the same way we implemented the **onReceive(...)** function from the **HeartRateBroadcastReceiver**. For this, you will

need to declare a new **XYplotSeriesList** class field, initialize it in the **onCrea**⌋**te(...)** and rename the existing one with a more sensible name to distinguish the two heart-rate series (from the watch and from the chest-band).

Everything required is now in place, we only need to actually connect to the device.

***Extra:*** you can change a little the **mGattUpdateReceiver** receiver to display the Bluetooth connection status, if you want! A nice addition would be to replace the generic string *HR belt* with the device's name.

# 5 Connecting to the Bluetooth belt

Make the **NewRecordingFragment** start the **DeviceScanActivity** for result, for example when the user taps on the corresponding **HR belt Switch**.

1. Create an **OnClickListener(...)**;
2. Launch an **intent** with **startActivityForResult(...)**;
3. Uses an **onActivityResult(...)** to retrieve the information obtained from **Device**⌋**ScanActivity**;

The resulting intent contains two things:

1. The selected device name, which can be used to update the **Switch**'s name accordingly,
2. The device address, which we will need to pass to the **ExerciceLiveActivity**.
3. Add this information to the **intent** on the **onCreateView(...)-> setOnClickListe**⌋**ner(...)-> runTransaction(...)-> onComplete(...)** of **NewRecordingFragment**.

To facilitate the **onClickListener** creation, we can wrap it up with the following class:

```
private class SwitchBeltOnCheckedChangeListener implements
                CompoundButton.OnCheckedChangeListener { @Override
    public void onCheckedChanged(CompoundButton compoundButton,
                                 boolean isChecked) {
        if (isChecked) {
            Intent intent = new Intent(getActivity(), DeviceScanActivity.class);
            startActivityForResult(intent, BLE_CONNECTION);
        } else {
            compoundButton.setText(R.string.hr_belt);
        }
    }
}
```

Add the following class members to **NewRecordingFragment**:

```
private Switch switchHRbelt;
private SwitchBeltOnCheckedChangeListener switchBeltOnCheckedChangeListener;
private static final int BLE_CONNECTION = 1;

public static final String EXTRAS_DEVICE_NAME = "DEVICE_NAME";
public static final String EXTRAS_DEVICE_ADDRESS = "DEVICE_ADDRESS";
private String mDeviceAddress;
```

Hence, before the **onCreateView -> setOnClickListener(...)**, place the following code:

```
switchBeltOnCheckedChangeListener = new SwitchBeltOnCheckedChangeListener();
switchHRbelt = fragmentView.findViewById(R.id.switchBelt);
```

The new **SwitchBeltOnCheckedChangeListener** will guarantee the exection of **DeviceScanActivity** and the information can be retrieved by **onActivityResult**:

```
@Override
public void onActivityResult(int requestCode, int resultCode,
                                    @Nullable Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if(requestCode== BLE_CONNECTION && resultCode== Activity.RESULT_OK){
        mDeviceAddress=data.getStringExtra(EXTRAS_DEVICE_ADDRESS);
    }
}
```

Moreover, the retrieved information can be then added to the **intent** on the **onComplete** method of **setOnClickListener**

```
intentStartLive.putExtra(EXTRAS_DEVICE_ADDRESS,mDeviceAddress);
```

Finally, we link the **switch** state with the **CompoundButton.OnCheckedChangeListener** based listener by the following:

```
@Override
public void onResume() {
    super.onResume();
    switchHRbelt.setOnCheckedChangeListener(switchBeltOnCheckedChangeListener);
}
```

```
@Override
public void onPause() {
    super.onPause();
    switchHRbelt.setOnCheckedChangeListener(null);
}
```

As the **ListView** used in the **DeviceScanActivity** comes from a very old Android version, it turns out it does not support well the newer Android themes (ie. Material design). This will show up as a runtime crash when calling the **getActionBar()** method. The best way to fix this is to apply from the **AndroidManifest** a legacy theme by substituting the **DeviceScanActivity** activity by the following:

```
<!-- mobile -> AndroidManifest.xml -> <manifest...> -> <application...> -->
    <activity
        android:name=".DeviceScanActivity"
        android:theme="@android:style/Theme.Holo.Light" />
```

As you should have extracted (from the sample) the part of the code which gets the Bluetooth address of the device to connect to, it should automatically connect if the address is valid.

# 6 The heart-rate plot

Now we have potentially two concurrent lines in the plot, we need to change a couple of things.

First, we will skip using the **XYplotSeriesList** helper class to control the series manually, and use a time in seconds for the x-axis position of the measurements. Therefore, we only need the following class fields:

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
private final SimpleXYSeries hrWatchSeries = new SimpleXYSeries(HR_PLOT_WATCH);
private final SimpleXYSeries hrBeltSeries = new SimpleXYSeries(HR_PLOT_BELT);
private long startTime = System.currentTimeMillis() / 1000;
```

The plot initialization in the **ExerciceLiveActivity -> onCreate(...)** now has the two series, one blue and the other one in red:

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity -> onCreate(...) **/
LineAndPointFormatter formatterRed =
```

```
        new LineAndPointFormatter(RED, TRANSPARENT, TRANSPARENT, null);
LineAndPointFormatter formatterBlue =
        new LineAndPointFormatter(BLUE, TRANSPARENT, TRANSPARENT, null);
formatterRed.getLinePaint().setStrokeWidth(8);
formatterBlue.getLinePaint().setStrokeWidth(8);
heartRatePlot.clear();
heartRatePlot.addSeries(hrWatchSeries, formatterRed);
heartRatePlot.addSeries(hrBeltSeries, formatterBlue);
heartRatePlot.redraw();
```

We now want to display the last 50 seconds of data rather than the last 50 points. This is because the HR belt and the smartwatch send samples at a different rate, and would therefore dis-synchronize the series and apparent lengths. Hence, you can rename the class field **NUMNER_OF_POINTS** to **NUMBER_OF_SECONDS**.

Finally, the last thing to do is to update the listeners for new values (from the smartwatch and from the belt). In case of the smartwatch, the listener becomes the following:

```
/** mobile -> ExerciceLiveActivity.java -> ExerciceLiveActivity
    -> HeartRateBroadcastReceiver **/
@Override
public void onReceive(Context context, Intent intent) {
    // Show HR in a TextView
    int heartRateWatch = intent.getIntExtra(HEART_RATE, -1);
    TextView hrTextView = findViewById(R.id.exerciseHRwatchLive);
    hrTextView.setText(String.valueOf(heartRateWatch));

    float time = System.currentTimeMillis() / 1000 - startTime;
    hrWatchSeries.addLast(time, heartRateWatch);

    while (hrWatchSeries.size() > 0 && (time - hrWatchSeries.getX(0).longValue()) >
            NUMBER_OF_SECONDS) {
        hrWatchSeries.removeFirst();
        heartRatePlot.setDomainBoundaries(0, 0, BoundaryMode.AUTO);
    }

    heartRatePlot.redraw();
}
```

# 7 Finishing the project

Now, you have everything in you hands to also add the values received from the HR belt to Firebase. You can use exactly the same implementation we had initially for the smartwatch. Indeed, the belt has no memory, so we do not need special process to retrieve the values if the user has been too far from the phone/tablet. Each received value is a value to save, the rest is lost.