

Présentation du Bluetooth Low Energy

Les objets connectés commencent à envahir notre quotidien : du simple capteur de température au bracelet connecté capable de mesurer l'activité physique de son utilisateur, en passant par les enceintes nomades, la plupart de ces nouveaux périphériques nécessitent un smartphone (ou une tablette) pour fonctionner. Dans la grande majorité des cas, le *Bluetooth Low Energy* (BLE, dans la suite de ce chapitre) est utilisé pour ces communications à courte distance. Cette norme, établie par la société Nokia en 2006 à partir de la norme Bluetooth, possède en effet beaucoup d'atouts : consommation d'énergie mesurée, distance de communication relativement importante (pratiquement 100 m dans un espace ouvert) et implémentation (relativement) aisée.

Ce chapitre abandonne temporairement l'application LocDVD au profit d'une petite application qui va détecter un objet BLE, s'y connecter et échanger des informations.

Comme souvent en informatique, plusieurs constructeurs ont décidé de mettre au point des protocoles spécifiques pour la connexion des objets BLE : que ce soit le format iBeacons d'Apple ou EddyStone pour Google, ces protocoles sont basés sur la norme Bluetooth Low Energy.

Au lieu de se cantonner à l'un de ces formats (EddyStone est le format de prédilection pour la plateforme Android), ce chapitre va présenter un processus de connexion qui fonctionne avec les deux formats, sans utiliser donc leurs spécificités. Si cela augmente quelque peu la complexité de mise en œuvre, toute latitude est ainsi laissée au lecteur quant au choix de l'objet connecté à utiliser pour les développements.

Détecter un périphérique BLE

Afin d'échanger des informations avec un objet BLE, plusieurs étapes sont nécessaires :

- La découverte de l'objet par le terminal Android.
- L'établissement de la connexion entre l'objet BLE et le terminal.
- Enfin, la lecture et l'écriture d'informations, ces étapes étant elles-mêmes composées de plusieurs sous-étapes que nous détaillerons plus en avant.

Cette section présente la première étape, qui va lancer une analyse Bluetooth pour détecter un objet BLE.

1. Préparation du projet

La première chose à faire est de créer une nouvelle application avec Android Studio ; cette application, très simple, ne comportera qu'un seul écran et devra cibler les terminaux équipés d'Android 4.3 ou supérieur : la norme BLE n'est pas compatible avec les versions antérieures d'Android.

- Créez un nouveau projet Android Studio, ayant pour nom BLE, et pour domaine, `exemple.com`.
- Ce projet doit être compatible smartphones et tablettes, à partir de la version Android 4.3 (Jelly bean, API 18)
- Le modèle de base choisi est « **Empty Activity** » et utilise, comme LocDVD, la bibliothèque de support V7 : l'activité principale, `MainActivity` hérite de `AppCompatActivity`.

L'assistant de création de projet terminé, Android Studio présente une classe `MainActivity`, et son fichier de layout associé, `activity_main.xml`.

L'interface de l'application est très simple : dans un premier temps, une zone de texte présente le nom et l'adresse de l'objet BLE détecté. Cette zone de texte doit permettre d'afficher plusieurs lignes.

Le layout correspondant est, par exemple, le suivant :

```
<?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"
    android:layout_margin="8dp">
    <TextView
        android:id="@+id/main_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Dans la méthode `onCreate` de l'activité `MainActivity`, il faut déclarer le composant `TextView`. Le code correspondant est, a minima, le suivant :

```
package com.exemple.ble;
```

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    TextView text;

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

        text =(TextView)findViewById(R.id.main_text);

    }
}
```

2. Gestion des permissions

La manipulation du Bluetooth Low Energy nécessite d'inscrire dans le Manifest les permissions correspondantes : BLUETOOTH et BLUETOOTH_ADMIN :

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

Plus étonnant, si l'application cible une version du sdk supérieure à la version 22 (ce qui est le cas ici, comme prévu par défaut par l'assistant de création de projet), il faut également demander la permission ACCESS_FINE_LOCATION, permission correspondant à la géolocalisation fine !

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

Par ailleurs, la permission ACCESS_FINE_LOCATION étant définie comme une permission sensible à partir d'Android 6 (Marshmallow), il faut également mettre en place une demande de permission à l'utilisateur, comme vu dans le chapitre Exploiter le téléphone. Le déroulé est succinctement rappelé ci-dessous :

→ Il faut commencer par vérifier que la permission n'a pas déjà été accordée :

```
if (ContextCompat.checkSelfPermission(this,
    Manifest.permission.ACCESS_FINE_LOCATION) !=
    PackageManager.PERMISSION_GRANTED) {
    // Permission non accordée
}
```

→ Ensuite, vérifier s'il faut donner une explication à l'utilisateur (dans le cas où il a déjà refusé cette permission) :

```

if (shouldShowRequestPermissionRationale(
    Manifest.permission.ACCESS_FINE_LOCATION)) {
    // Présenter une boîte de dialogue pour expliquer la demande...
}

```

- La demande de permission se fait en invoquant la méthode `requestPermissions`. La réponse est traitée dans la méthode surchargée `onRequestPermissionsResult` :

```

private void askPermission() {
    requestPermissions(
        new String[] {Manifest.permission.ACCESS_FINE_LOCATION},
        REQUEST_PERMISSION);
}

@Override
public void onRequestPermissionsResult(int requestCode, String[]
permissions, int grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if(requestCode==REQUEST_PERMISSION) {
        if(permissions[0].equals(Manifest.permission.ACCESS_FINE_LOCATION) &&
            grantResults[0] == PackageManager.PERMISSION_GRANTED) {

            // La permission est accordée!
        }
    }
}

```

- À l'issue de la demande de permission, c'est une nouvelle méthode, `startBLEScan` qui sera invoquée. Le code complet de gestion de la permission `ACCESS_FINE_LOCATION` est, par exemple, le suivant :

```

@Override
protected void onResume() {
    super.onResume();
    ensurePermission();
}

private void ensurePermission() {
    if(PackageManager.PERMISSION_GRANTED !=
ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)) {
        if (shouldShowRequestPermissionRationale(
            Manifest.permission.ACCESS_FINE_LOCATION)) {
            AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setTitle(R.string.demande_permission_titre);
            builder.setMessage(R.string.explication_permission);
            builder.setNegativeButton("Non", new
DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {

```

```

        Toast.makeText(MainActivity.this,
            R.string.permission_obligatoire,
            Toast.LENGTH_LONG).show();
        finish(); // On quitte l'application
    }
});
builder.setPositiveButton("Oui", new
DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        askPermission();
    }
});
builder.show();
} else {
    askPermission();
}
} else {
    startBLEScan();
}
}

private void askPermission() {
    requestPermissions(new String[]
{Manifest.permission.ACCESS_FINE_LOCATION}, REQUEST_PERMISSION);
}

@Override
public void onRequestPermissionsResult(int requestCode, String[]
permissions, int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions,
grantResults);
    if(requestCode==REQUEST_PERMISSION) {
        if(permissions[0].equals(Manifest.permission.ACCESS_FINE_LOCATION) &&
            grantResults[0] ==
PackageManager.PERMISSION_GRANTED)
        {
            startBLEScan();
        }
    }
}

private void startBLEScan() {
    // A faire...
}

```

3. Initialisation de BluetoothManager

La méthode `startBLEScan` doit initialiser les objets nécessaires au lancement du scan BLE, et lancer la recherche de périphérique Bluetooth Low Energy.

Il faut commencer par obtenir une instance de `BluetoothManager`, en invoquant la méthode

getSystemService. Le service concerné est Context.BLUETOOTH_SERVICE. Tous les objets doivent être accessibles dans le code de la classe MainActivity, ils sont donc déclarés en variables de classe :

```
BluetoothManager bluetoothManager;  
[...]  
bluetoothManager=  
    (BluetoothManager)getSystemService(Context.BLUETOOTH_SERVICE);
```

L'instance de BluetoothManager expose la méthode getAdapter qui permet d'obtenir un objet de type BluetoothAdapter. Cet objet se chargera, entre autres, de lancer la recherche de périphérique BLE.

```
BluetoothAdapter bluetoothAdapter;  
[...]  
bluetoothAdapter = bluetoothManager.getAdapter();
```

Avant de lancer l'analyse BLE, il est nécessaire de s'assurer que le Bluetooth est activé sur le terminal de l'utilisateur. Pour ce faire, BluetoothAdapter présente la méthode isEnabled, qui renvoie false si le Bluetooth n'est pas actif.

Idéalement, si le Bluetooth n'est pas actif, il faut présenter à l'utilisateur l'écran permettant d'activer cette fonctionnalité. Cela se fait en invoquant une intention implicite, avec l'action BluetoothAdapter.ACTION_REQUEST_ENABLE.

```
if(!bluetoothAdapter.isEnabled()) {  
    Intent askBLE = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);  
    startActivityForResult(askBLE, REQUEST_ENABLE_BLE);  
}
```

La réponse de l'utilisateur est traitée, comme pour tout appel à startActivityForResult, dans la méthode surchargée onActivityResult.

Une proposition de code, intégrant tous les points vus jusqu'à présent, est donnée ci-dessous :

```
BluetoothManager bluetoothManager=null;  
BluetoothAdapter bluetoothAdapter=null;  
  
final static int REQUEST_PERMISSION = 102;  
final static int REQUEST_ENABLE_BLE = 201;  
  
[...]  
private void startBLEScan() {  
    if(bluetoothManager==null)  
        bluetoothManager=  
            (BluetoothManager)getSystemService(Context.BLUETOOTH_SERVICE);  
    if(bluetoothAdapter==null)  
        bluetoothAdapter = bluetoothManager.getAdapter();  
  
    if(!bluetoothAdapter.isEnabled()) {  
        Intent askBLE = new
```

```

Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(askBLE, REQUEST_ENABLE_BLE);
}
}

@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if(requestCode==REQUEST_ENABLE_BLE && resultCode==RESULT_OK)
        startBLEScan();
}

```

4. Recherche d'objets Bluetooth Low Energy

La phase d'initialisation étant terminée, il est maintenant possible de lancer l'analyse et la recherche de périphérique (nommé communément scan) Bluetooth Low Energy.

Selon la version d'Android, le lancement du scan se fait différemment.

a. Lancer le scan avant Android 21 (Lollipop)

Le lancement du scan, sous Android JellyBean ou Kitkat, se fait à l'aide de l'objet `BluetoothAdapter` obtenu précédemment. Il faut pour cela invoquer la méthode `startLeScan`.

Cette méthode prend en unique paramètre un objet de type `BluetoothAdapter.LeScanCallback`, objet qui sera invoqué pour chaque objet BLE détecté : `startLeScan` est, comme la plupart des méthodes d'interaction avec le BLE, asynchrone.

- ➔ Déclarez, dans la classe `MainActivity`, un objet de type `BluetoothAdapter.LeScanCallback`. La méthode à implémenter est `onLeScan`.

```

BluetoothAdapter.LeScanCallback leScanCallback =
    new BluetoothAdapter.LeScanCallback() {
        @Override
        public void onLeScan(BluetoothDevice device, int rssi,
byte[] scanRecord) {

        }
    };

```

- ➔ Dans la méthode `startBLEScan`, il faut tester si la version d'Android exécutée est inférieure à la version 21, et lancer le scan :

```

if(android.os.Build.VERSION.SDK_INT<21) {
    bluetoothAdapter.startLeScan(leScanCallback);
} else {
    [...]
}

```

Nous verrons ci-après comment traiter les périphériques BLE détectés.

b. Lancer le scan à partir d'Android 21

Avec Lollipop, une nouvelle classe est chargée de lancer l'analyse BLE : la classe `BluetoothLeScanner`, qui expose la méthode `startScan`.

Pour obtenir une instance de `BluetoothLeScanner`, il faut invoquer la méthode `getBluetoothLeScanner` de l'objet `BluetoothAdapter`.

```
BluetoothLeScanner bluetoothLeScanner;  
[...]  
bluetoothLeScanner = bluetoothAdapter.getBluetoothLeScanner();
```

La méthode `startScan`, de l'objet `bluetoothLeScanner`, prend en paramètre un objet de type `ScanCallback`. C'est la méthode à implémenter `onScanResult` qui sera invoquée par le système lorsqu'un objet BLE sera détecté.

Déclarez, dans la classe `MainActivity`, une instance de `ScanCallback`. Le traitement sera vu ci-après.

```
ScanCallback scanCallback = new ScanCallback() {  
    @Override  
    public void onScanResult(int callbackType, ScanResult result) {  
        super.onScanResult(callbackType, result);  
    }  
};
```

L'objet `ScanCallback` étant défini, il est maintenant possible de lancer le scan pour les terminaux équipés de Lollipop.

```
if (android.os.Build.VERSION.SDK_INT < 21) {  
    bluetoothAdapter.startLeScan(leScanCallback);  
} else { // version >= Lollipop  
    bluetoothLeScanner = bluetoothAdapter.getBluetoothLeScanner();  
    bluetoothLeScanner.startScan(scanCallback);  
}
```

5. Arrêter l'analyse

La phase de recherche de périphérique Bluetooth Low Energy étant relativement énergivore, il est important de stopper la recherche soit lorsque l'objet recherché est trouvé, soit au bout d'un certain laps de temps. Par ailleurs, certains constructeurs de terminaux Android ne permettent pas de se connecter à un objet BLE si le scan est toujours actif.

Le plus simple pour stopper le scan est d'utiliser un objet `Handler`, qui expose la méthode `postDelayed`. Cette méthode permet en effet de définir du code qui sera exécuté après un certain délai.


```
Handler handler = new Handler();
handler.postDelayed([...]);
```

La méthode `postDelayed` requiert deux paramètres : un objet de type `Runnable`, qui spécifie le code à exécuter, et un entier de type `long` qui précise le temps d'attente (en millisecondes) avant l'exécution du code.

`Runnable` est une interface, qui présente la méthode `run`.

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // code à définir
    }
};
```

À l'issue du délai précisé dans la méthode `postDelayed`, la méthode (à définir) `stopBLEScan` doit être invoquée.

Une première version du code est la suivante (le scan sera interrompu après 5 secondes) :

```
Handler handler = new Handler();
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        stopBLEScan();
    }
};
handler.postDelayed(runnable, 5000);
```

Comme souvent, les développeurs Android préfèrent une version plus compacte, version que l'on retrouve en général dans tous les exemples de code :

```
new Handler().postDelayed(new Runnable() {
    @Override
    public void run() {
        stopBLEScan();
    }
}, 5000);
```

Il faut maintenant définir la méthode `stopBLEScan`, qui doit arrêter la recherche BLE.

Comme pour le lancement du scan, le code diffère selon que le terminal utilisateur exécute Android Lollipop ou une version inférieure.

Pour les versions Android 18 à 20, c'est la méthode `stopLeScan` qui doit être invoquée. Cette méthode prend en paramètre l'objet de type `BluetoothAdapter.LeScanCallback` invoqué lors du lancement du scan.

Pour les versions égales ou supérieures à Android 21, c'est la méthode `stopScan` de l'objet

BluetoothLeScanner qui doit être utilisée. Cette méthode prend en paramètre l'objet de type ScanCallback utilisé lors de l'appel à startScan.

Le code de la méthode stopBLEScan est donc le suivant :

```
private void stopBLEScan() {
    if(android.os.Build.VERSION.SDK_INT<21)
        bluetoothAdapter.stopLeScan(leScanCallback);
    else
        bluetoothLeScanner.stopScan(scanCallback);
}
```

6. Exploiter le résultat du scan

Pour terminer la phase de recherche d'objet BLE, il reste à traiter le résultat du scan, soit la méthode onLeScan de l'objet BluetoothAdapter.LeScanCallback pour Android 18, soit la méthode onScanResult de l'objet ScanCallback pour les versions supérieures ou égales à Lollipop.

Les signatures de ces méthodes sont les suivantes :

Pour la méthode onLeScan (Android 18) :

```
@Override
public void onLeScan(BluetoothDevice device, int rssi, byte[] scanRecord)
```

Et pour onScanResult :

```
@Override
public void onScanResult(int callbackType, ScanResult result)
```

Si ces méthodes ont des signatures différentes, on retrouve en fait sensiblement les mêmes informations.

La méthode onLeScan présente les trois paramètres suivants :

- `BluetoothDevice device` : représente l'objet BLE détecté. C'est cet objet qui sera utilisé pour la connexion entre l'objet BLE physique et le terminal Android.
- `int rssi` : représente la force du signal Bluetooth Low Energy reçu par le terminal. RSSI est l'acronyme pour *Received Signal Strength Indicator*.
- `byte[] scanRecord` : tableau de byte contenant l'information éventuellement exposée par l'objet connecté.

La méthode onScanResult présente quant à elle les paramètres suivants :

- `int callbackType` : cet entier précise le résultat de l'analyse Bluetooth dans le cas où un ou plusieurs filtres auraient été appliqués à la recherche : nous n'aborderons pas la création de filtres de recherche dans cet ouvrage, mais sachez qu'il est possible d'appliquer plusieurs filtres pour faire une analyse BLE plus précise (il faut dans ce cas utiliser une classe d'assistance `ScanFilter.Builder`, et invoquer une version de `startScan` prenant en paramètre un tableau de `ScanFilter`). Le paramètre `callbackType` précise ainsi comment la méthode `onScanResult` a été appelée par rapport aux filtres définis.

- `ScanResult result` : cet objet représente le résultat du scan. Il expose notamment les méthodes `getDevice` qui renvoie un objet de type `BluetoothDevice` (comme le premier paramètre de `onLeScan`), et `getRssi`, qui retourne un entier représentant la force du signal : on retrouve ainsi les mêmes données qu'avec la méthode `onLeScan`.

Pour minimiser les traitements, il suffit donc de définir une méthode `onDeviceDetected`, qui sera invoquée par les méthodes `onLeScan` et `onScanResult` :

```
private void onDeviceDetected(BluetoothDevice device, int rssi) {

    // A faire !

}
```

Le traitement de la détection d'un objet BLE est donc, pour l'instant, le suivant :

```
// API 18
BluetoothAdapter.LeScanCallback leScanCallback =
    new BluetoothAdapter.LeScanCallback() {
        @Override
        public void onLeScan(BluetoothDevice device, int rssi, byte[]
scanRecord) {
            onDeviceDetected(device,rssi);
        }
    };

// API 21
ScanCallback scanCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        super.onScanResult(callbackType, result);
        onDeviceDetected(result.getDevice(), result.getRssi());
    }
};

private void onDeviceDetected(BluetoothDevice device, int rssi) {

    // A faire !

}
```

Dans un premier temps, la méthode `onDeviceDetected` doit afficher, dans le composant `TextView`, quelques informations sur l'objet Bluetooth Low Energy détecté : son nom, son adresse ainsi que la force du signal.

Implémentez le contenu de `onDeviceDetected`. L'objet `BluetoothDevice` expose les méthodes `getName` et `getAddress` qui permettent d'obtenir le nom et l'adresse de l'objet.

Un exemple d'implémentation est donné ci-dessous :

```
private void onDeviceDetected(BluetoothDevice device, int rssi) {
```

```

String info = "Objet détecté :";
info+="\nNom : " + device.getName();
info+="\nAdresse : " + device.getAddress();
info+="\nRSSI : " + String.valueOf(rssi);
text.setText(info);
}

```

La détection de l'objet BLE étant effective, la connexion entre le BLE et le terminal Android peut être effectuée.

Le code complet de MainActivity est, pour l'instant, le suivant :

```

public class MainActivity extends AppCompatActivity {
    final static String TAG="MainActivity";

    TextView text;

    BluetoothManager bluetoothManager = null;
    BluetoothAdapter bluetoothAdapter=null;
    BluetoothLeScanner bluetoothLeScanner;

    final static int REQUEST_PERMISSION = 102;
    final static int REQUEST_ENABLE_BLE = 201;

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

        text =(TextView)findViewById(R.id.main_text);

        ensurePermission();
    }

    private void ensurePermission() {
        if(ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) !=
        PackageManager.PERMISSION_GRANTED) {
            if (shouldShowRequestPermissionRationale
(Manifest.permission.ACCESS_FINE_LOCATION)) {
                AlertDialog.Builder builder = new AlertDialog.Builder(this);
                builder.setTitle(R.string.demande_permission_titre);
                builder.setMessage(R.string.explication_permission);
                builder.setNegativeButton("Non", new
DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        Toast.makeText(MainActivity.this,
                            R.string.permission_obligatoire,
Toast.LENGTH_LONG).show();
                        finish();
                    }
                }
            }
        }
    }
}

```

```

    });
    builder.setPositiveButton("Oui", new
DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            askPermission();
        }
    });
    builder.show();
} else {
    askPermission();
}
} else {
    startBLEScan();
}
}

private void askPermission() {
    requestPermissions(new String[]
{Manifest.permission.ACCESS_FINE_LOCATION},
    REQUEST_PERMISSION);
}

@Override
public void onRequestPermissionsResult(int requestCode,
String[] permissions, int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions,
grantResults);
    if(requestCode==REQUEST_PERMISSION) {
        if(permissions[0].equals
(Manifest.permission.ACCESS_FINE_LOCATION) &&
            grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            startBLEScan();
        }
    }
}

private void startBLEScan() {
    if(bluetoothManager==null)
        bluetoothManager=
            (BluetoothManager)getSystemService(Context.BLUETOOTH_SERVICE);
    if(bluetoothAdapter==null)
        bluetoothAdapter = bluetoothManager.getAdapter();

    if(!bluetoothAdapter.isEnabled()) {
        Intent askBLE = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivityForResult(askBLE, REQUEST_ENABLE_BLE);
        return;
    }

    if(android.os.Build.VERSION.SDK_INT<21) {
        bluetoothAdapter.startLeScan(leScanCallback);
        Log.d(TAG, "Scan lancé en version 18");
    } else {
        bluetoothLeScanner = bluetoothAdapter.getBluetoothLeScanner();
    }
}

```

```

        bluetoothLeScanner.startScan(scanCallback);
        Log.d(TAG, "Scan lancé en version 21");
    }

    new Handler().postDelayed(new Runnable() {
        @Override
        public void run() {
            stopBLEScan();
        }
    }, 5000);
}

@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if(requestCode==REQUEST_ENABLE_BLE && requestCode==RESULT_OK)
        startBLEScan();
}

private void stopBLEScan() {
    if(android.os.Build.VERSION.SDK_INT<21)
        bluetoothAdapter.stopLeScan(leScanCallback);
    else
        bluetoothAdapter.getBluetoothLeScanner().stopScan(scanCallback);
}

// API 18
BluetoothAdapter.LeScanCallback leScanCallback =
    new BluetoothAdapter.LeScanCallback() {
        @Override
        public void onLeScan(BluetoothDevice device, int rssi,
byte[] scanRecord) {
            onDeviceDetected(device,rssi);
        }
    };

// API 21
ScanCallback scanCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        super.onScanResult(callbackType, result);
        onDeviceDetected(result.getDevice(), result.getRssi());
    }
};

private void onDeviceDetected(BluetoothDevice device, int rssi) {
    String info = "Objet détecté :";
    info+="\nNom : " + device.getName();
    info+="\nAdresse : " + device.getAddress();
    info+="\nRSSI : " + String.valueOf(rssi);
    text.setText(info);
}
}

```


Connecter un objet


Pour pouvoir interagir avec un objet BLE, le terminal Android et l'objet doivent être appairés, un objet BLE ne pouvant être appairé qu'à un seul terminal à la fois : cette phase est, à proprement parlé, la phase de connexion.

Dans le cadre de l'application, la connexion est lancée automatiquement lorsqu'un objet BLE est détecté.

La connexion se fait à l'aide de l'instance de `BluetoothDevice` obtenue dans la section précédente à l'issue de la phase de découverte. Pour cela, `BluetoothDevice` expose la méthode `connectGatt`, qui demande les paramètres suivants :

- `Context context` : le contexte d'exécution. Typiquement, l'activité courante.
- `boolean autoConnect` : indicateur permettant de spécifier si la connexion doit se faire immédiatement (`false`) ou automatiquement lorsque l'objet est à portée.
- `BluetoothGattCallback bluetoothGattCallback` : cet objet sera utilisé tout au long des échanges entre le terminal et l'objet BLE. Il expose plusieurs méthodes, chacune correspondant à une phase (ou un état) du processus de communication.

La méthode `connectGatt` retourne un objet `BluetoothGatt`, dont une référence doit être conservée par la classe `MainActivity`.

 L'acronyme Gatt est omniprésent dans les noms des classes permettant la communication Bluetooth Low Energy. Signifiant Generic Attribute Profile, ce terme représente l'ensemble des normes régissant la communication entre terminaux BLE.

- L'appel à la méthode `connectGatt` se fait suite à la détection de l'objet BLE : par exemple, dans la méthode `onDeviceDetected`. Auparavant, il faut définir un objet `BluetoothGattCallback` dans la classe `MainActivity`. Positionnez-vous après la méthode `onDeviceDetected`, et définissez une instance de `BluetoothGattCallback` : Android Studio doit, lorsque vous écrivez `new BluetoothGattCallback`, vous présenter une fenêtre pop-up vous permettant d'indiquer quelles sont les méthodes que vous souhaitez implémenter.
- Sélectionnez pour l'instant uniquement la première méthode listée, soit `onConnectionStateChange`. Si la popup n'apparaît pas, vous pouvez obtenir la liste des méthodes à surcharger en utilisant le raccourci-clavier [Ctrl] [Inser], et en sélectionnant l'entrée **Override Methods**.

```
BluetoothGattCallback bluetoothGattCallback = new BluetoothGattCallback() {  
    @Override  
    public void onConnectionStateChange(BluetoothGatt gatt, int status,  
    int newState) {  
        super.onConnectionStateChange(gatt, status, newState);  
    }  
};
```

- Cet objet défini, il est possible d'invoquer `connectGatt` dans la méthode `onDeviceDetected`. Le paramètre `autoConnect` est valorisé à `false`, pour que la connexion se fasse immédiatement :

```
BluetoothGatt bluetoothGatt;  
[...]  
private void onDeviceDetected(BluetoothDevice device, int rssi) {  
    String info = "Objet détecté :";  
    info+="\nNom : " + device.getName();  
}
```



```

        info+="\nAdresse :" + device.getAddress();
        info+="\nRSSI : " + String.valueOf(rssi);
        text.setText(info);

        bluetoothGatt = device.connectGatt(this, false, bluetoothGattCallback);
    }

```

Avant d'aller plus loin, il est essentiel de prévoir ce qui se passe lorsque l'application est fermée : si une connexion Bluetooth est établie, il est en effet indispensable de la fermer avant de quitter l'application, sans quoi l'objet BLE pourrait rester dans un état connecté.

C'est l'instance de `BluetoothGatt` qui se charge de fermer la connexion. Le plus simple, dans le cadre de l'application, est de fermer automatiquement la connexion lorsque l'activité est détruite :

- Dans le corps de la classe `MainActivity`, faites [Ctrl][Inser] et sélectionnez **Override Methods**, puis sélectionnez l'entrée **onDestroy**. Le code est ainsi généré automatiquement.
- Dans la méthode `onDestroy`, invoquez, si l'objet `bluetoothGatt` est non null, la méthode `close`, qui ferme la connexion Bluetooth, et valorisez `bluetoothGatt` à null.

```

@Override
protected void onDestroy() {
    super.onDestroy();
    if(bluetoothGatt!=null) {
        bluetoothGatt.close();
        bluetoothGatt = null;
    }
}

```

C'est, on l'a évoqué, l'objet de type `BluetoothGattCallback` qui porte l'essentiel des opérations de communication entre le terminal et l'objet BLE. Pour l'instant, seule la méthode `onConnectionStateChange` est implémentée (partiellement). Cette méthode est invoquée, comme son nom l'indique, dès qu'un changement se produit dans l'état de la connexion entre les deux périphériques BLE. Les paramètres de la méthode indiquent quelle est la nature du changement.

La signature de cette méthode est la suivante :

```

public void onConnectionStateChange(BluetoothGatt gatt, int status,
int newState)

```

`BluetoothGatt gatt` : c'est une référence à l'objet `BluetoothGatt` obtenu lors de l'appel à `connectGatt`.

`int status` : indicateur spécifiant si l'opération s'est déroulée avec succès, `status` prenant alors la valeur `BluetoothGatt.GATT_SUCCESS`. Si une erreur est survenue, `status` prend la valeur du code erreur retourné : la liste des codes erreurs est disponible à l'adresse https://developer.android.com/reference/android/bluetooth/BluetoothGatt.html#GATT_CONNECTION_CONGESTED.

`int newState` : indique si l'objet est maintenant connecté (valeur `BluetoothProfile.STATE_CONNECTED`) ou déconnecté (`BluetoothProfile.STATE_DISCONNECTED`).

- Pour l'instant, la méthode `onConnectionStateChange` indique juste si l'objet est connecté ou déconnecté. Cette information est simplement donnée via un message Toast. Cependant, la méthode

onConnectionStateChange n'étant pas invoquée par le thread principal de l'application, il n'est pas directement possible d'afficher un message Toast : il faut utiliser la méthode `runOnUiThread`, brièvement évoquée dans le chapitre Tâches asynchrones et services.

- `runOnUiThread`, méthode de la classe `Activity`, prend en paramètre un objet de type `Runnable` - comme la méthode `postDelayed` vue plus haut. Le plus simple ici est de définir dans `MainActivity` une méthode `showToastFromBackground`, qui prend en paramètre une chaîne de caractères, et qui force l'affichage du message Toast dans le thread principal :

```
private void showToastFromBackground(final String message) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(MainActivity.this, message,
                Toast.LENGTH_SHORT).show();
        }
    });
}
```

- Il est maintenant simple d'afficher, au changement d'état de connexion, un message pour informer l'utilisateur :

```
private void showToastFromBackground(final String message) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(MainActivity.this, message,
                Toast.LENGTH_SHORT).show();
        }
    });
}

BluetoothGattCallback bluetoothGattCallback = new BluetoothGattCallback() {
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt, int status,
        int newState) {
        super.onConnectionStateChange(gatt, status, newState);
        if(status==BluetoothGatt.GATT_SUCCESS) {
            String message = "";
            if(newState== BluetoothProfile.STATE_CONNECTED)
                message = "Objet connecté";
            else
                message = "Objet déconnecté";
            showToastFromBackground(message);
        }
    }
};
```

Lire une caractéristique

L'objet BLE connecté, il est désormais possible d'échanger des données avec le terminal Android.

Dans la norme Bluetooth Low Energy, les données d'un objet sont appelées caractéristiques (*characteristics*, en anglais). Les caractéristiques sont regroupées par service. Les caractéristiques et les services sont identifiés par un UUID (*Universally Unique Identifier*), l'ensemble étant défini par le consortium Bluetooth.

Les UUID des services et des caractéristiques sont construits selon le principe suivant :

- Le format général des UUID est fixe : 0000xxxx-0000-1000-8000-00805f9b34fb
- Le consortium Bluetooth définit, pour chaque service et chaque caractéristique normée, un numéro sur quatre octets, qui remplace les xxxx dans le format général.

Ainsi, par exemple, le service *Battery Level*, qui contient la caractéristique du même nom permettant de connaître le niveau de charge de la batterie de l'objet connecté, s'est vu attribuer le numéro 0x180F : son UUID est donc 0000**180f**-0000-1000-8000-00805f9b34fb.

Le consortium présente la liste des services et des caractéristiques normées aux adresses suivantes :

- Pour les services : <https://www.bluetooth.com/specifications/gatt/services>
- Pour les caractéristiques : <https://www.bluetooth.com/specifications/gatt/characteristics>

Le tableau ci-dessous est un extrait de la liste des services présentés par le consortium.

| Name | Uniform Type Identifier | Assigned Number | Specification Level |
|-------------------------------|---|-----------------|---------------------|
| Alert Notification Service | org.bluetooth.service.alert_notification | 0x1811 | Adopted |
| Automation IO | org.bluetooth.service.automation_io | 0x1815 | Adopted |
| Battery Service | org.bluetooth.service.battery_service | 0x180F | Adopted |
| Blood Pressure | org.bluetooth.service.blood_pressure | 0x1810 | Adopted |
| Body Composition | org.bluetooth.service.body_composition | 0x181B | Adopted |
| Bond Management | org.bluetooth.service.bond_management | 0x181E | Adopted |
| Continuous Glucose Monitoring | org.bluetooth.service.continuous_glucose_monitoring | 0x181F | Adopted |
| Current Time Service | org.bluetooth.service.current_time | 0x1805 | Adopted |

- Il faut bien veiller à n'utiliser que des minuscules pour les UUID : si les majuscules sont parfois tolérées, la plateforme Android applique les règles strictes concernant les UUIDs et n'accepte que des minuscules.

L'application doit, une fois connectée, afficher le niveau de charge de la batterie de l'objet. Il faut donc, selon la norme, lire la caractéristique de numéro 0x2A19 (l'UUID est donc 00002a19-0000-1000-8000-00805f9b34fb), caractéristique appartenant au service 0x180F.

Il n'est pas possible de lire une caractéristique directement, ni adresser un service : il faut passer par un processus de découverte des services, et pour le ou les services ciblés, parcourir la liste des caractéristiques exposées afin

d'obtenir une référence sur chaque caractéristique que l'on souhaite lire (ou écrire).

C'est l'objet `BluetoothGatt`, obtenue lors de l'appel à la méthode `connectGatt` qui permet de lancer la découverte des services, en exposant la méthode `discoverServices`.

- Dans la méthode `onConnectionStateChange`, si le nouvel état de l'objet est connecté, invoquez la méthode `discoverServices`.

```
if(newState== BluetoothProfile.STATE_CONNECTED) {  
    message = "Objet connecté";  
    bluetoothGatt.discoverServices();  
}  
else {  
    message = "Objet déconnecté";  
}  
showToastFromBackground(message);
```

La méthode `discoverServices` est asynchrone : c'est la méthode `onServicesDiscovered` de l'objet `BluetoothGattCallback` qui est invoquée lorsqu'un service est découvert.

- Dans le corps de la déclaration de l'objet `BluetoothGattCallback`, de la même manière que pour la méthode `onConnectionStateChange`, faites un raccourci-clavier [Ctrl] [Inser], et sélectionnez la méthode `onServicesDiscovered` : le code présenté ci-dessous est généré automatiquement.

```
BluetoothGattCallback bluetoothGattCallback = new  
BluetoothGattCallback() {  
    @Override  
    public void onConnectionStateChange(BluetoothGatt gatt, int status,  
int newState) {  
        [...]  
    }  
  
    @Override  
    public void onServicesDiscovered(BluetoothGatt gatt, int status)  
{  
        super.onServicesDiscovered(gatt, status);  
    }  
};
```

`onServicesDiscovered` présente deux paramètres : l'objet `BluetoothGatt`, déjà rencontré, et un entier représentant le statut de la découverte. La valeur `BluetoothGatt.GATT_SUCCESS` indique que le processus de découverte s'est correctement déroulé.

Une fois les services découverts, l'objet `BluetoothGatt` donne accès à la liste des services en exposant la méthode `getServices`. Il faut, pour obtenir une référence sur la caractéristique `Battery_Level`, itérer sur la liste des services, identifier le service correspondant à la caractéristique, et lister les caractéristiques de ce service.

La méthode `getServices` renvoie une liste d'instances de `BluetoothGattService`, objet représentant un service BLE.

- Dans la méthode `onServicesDiscovered`, si le statut est `GATT_SUCCESS`, faites une boucle `for each` pour itérer sur la liste des services :

```
for(BluetoothGattService service : gatt.getServices()) {
}
}
```

- Il faut identifier le service Battery Level à l'aide de son UUID. Pour obtenir un UUID à partir d'une chaîne de caractères, il faut invoquer le constructeur statique `UUID.fromString` de la classe `UUID`.

```
UUID batteryLevelServiceUUID =
    UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb");
```

- Dans la boucle `for`, testez si l'UUID de chaque service correspond à l'UUID définie. Pour obtenir l'UUID d'un objet `BluetoothGattService`, il faut invoquer la méthode `getUuid` :

```
UUID batteryLevelServiceUUID =
    UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb");

for(BluetoothGattService service : gatt.getServices()) {
    if(service.getUuid().equals(batteryLevelServiceUUID)) {

    }
}
```

- Lorsque le service recherché est identifié, il faut parcourir la liste des caractéristiques qu'il expose, et tester l'UUID de chaque service. La liste des caractéristiques d'un service est accessible via la méthode `getCharacteristics` de l'objet `BluetoothGattService`. Cette méthode renvoie une liste de `BluetoothGattCharacteristic`. Il faut, auparavant, définir l'UUID de la caractéristique recherchée :

```
UUID batteryLevelServiceUUID =
    UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb");
UUID batteryLevelCharacteristicUUID =
    UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb");

for(BluetoothGattService service : gatt.getServices()) {
    if(service.getUuid().equals(batteryLevelServiceUUID)) {
        for(BluetoothGattCharacteristic c :
service.getCharacteristics()) {

        }
    }
}
```

- Pour obtenir la référence sur la caractéristique voulue, il ne reste plus qu'à tester l'UUID de chaque caractéristique exposée. Là aussi, l'UUID d'une caractéristique s'obtient en invoquant la méthode `getUuid` de la classe `BluetoothGattCharacteristic` :

```
for(BluetoothGattCharacteristic c : service.getCharacteristics())
{
    if(c.getUuid().equals(batteryLevelCharacteristicUUID)) {

    }
}
```

-
- La référence sur la caractéristique Battery Level obtenue, il faut, pour lire la valeur de la caractéristique, invoquer la méthode `readCharacteristic` de l'objet `BluetoothGatt`. La méthode `readCharacteristic` prend en unique paramètre l'objet `BluetoothGattCharacteristic` sélectionné.

```
for(BluetoothGattCharacteristic c : service.getCharacteristics()) {  
    if(c.getUuid().equals(batteryLevelCharacteristicUUID)) {  
        bluetoothGatt.readCharacteristic(c);  
    }  
}
```

Là aussi, la méthode `readCharacteristic` est asynchrone : c'est la méthode `onCharacteristicRead` de l'objet `BluetoothGattCallback` qui sera invoquée pour chaque lecture de caractéristique.

- Comme pour les méthodes `onConnectionStateChanged` et `onServicesDiscovered`, surchargez la méthode `onCharacteristicRead` de l'objet `BluetoothGattCallback`.

```
@Override  
public void onCharacteristicRead(BluetoothGatt gatt,  
    BluetoothGattCharacteristic characteristic, int status) {  
    super.onCharacteristicRead(gatt, characteristic, status);  
}
```

La méthode `onCharacteristicRead` expose, outre l'objet `BluetoothGatt`, une instance de `BluetoothGattCharacteristic` - la caractéristique lue, et un entier indiquant le statut de l'opération de lecture.

- Dans la méthode `onCharacteristicRead`, il faut tester si la caractéristique lue est bien celle recherchée (battery Level), en vérifiant son UUID. La valeur est ensuite obtenue en invoquant la méthode `getValue` de la classe `BluetoothGattCharacteristic`, ou l'une de ses déclinaisons selon le format de la donnée à lire.
- Pour connaître le format de la donnée, il faut se référer aux informations données par la norme Bluetooth pour chaque caractéristique normée (cf. adresses données en début de section, du consortium). Pour la caractéristique Battery Level, le consortium donne les informations suivantes (obtenues en cliquant sur la caractéristique dans la liste) :

Name: Battery Level

Type: [org.bluetooth.characteristic.battery_level](#) [Download / View](#)

Assigned Number: 0x2A19

Abstract:

The current charge level of a battery. 100% represents fully charged while 0% represents fully discharged.

Value Fields

| Names | Field Requirement | Format | Minimum Value | Maximum Value | Additional Information | |
|--|-------------------|--------|---------------|---------------|------------------------|----------|
| Level | Mandatory | uint8 | 0 | 100 | Enumerations | |
| Unit: org.bluetooth.unit.percentage | | | | | Key | Value |
| | | | | | 101 - 255 | Reserved |

Le niveau de batterie est un entier, au format uint8. La valeur peut donc être lue en utilisant la méthode `getIntValue` de la classe `BluetoothGattCharacteristic`. La méthode `getIntValue` prend en paramètre le format de donnée à utiliser, et un entier représentant l'offset de la donnée, s'il y en a. Ici, l'offset est 0, rien n'étant indiqué dans la norme. Le format est défini par l'une des constantes `FORMAT_xxxx` de la classe `BluetoothGattCharacteristic`, ici `FORMAT_UINT8` :

```
if (characteristic.equals(UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb"))) {
    Log.d(TAG, "Niveau de batterie :" + characteristic.getIntValue(BluetoothGattCharacteristic.FORMAT_UINT8, 0));
}
```

→ Il ne reste plus qu'à afficher le niveau de batterie lu. Dans le cadre de l'application, il suffit de modifier quelque peu le fichier de layout pour y ajouter un `TextView` en dessous du `TextView` déjà existant (en modifiant la taille de ce dernier) et en invoquant la méthode `setText` de ce nouveau `TextView`. Attention, là aussi, la méthode `onCharacteristicRead` étant invoquée depuis un thread d'arrière-plan, il faut faire appel à la méthode `runOnUiThread` :

```
private void setBatteryLevelFromBackground(final int value) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            batteryLevel.setText(String.format("Niveau batterie :%d %%",value));
        }
    });
}

[...]

@Override
public void onCharacteristicRead(BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic, int status) {
```

```

super.onCharacteristicRead(gatt, characteristic, status);
if(status==BluetoothGatt.GATT_SUCCESS) {
    if (characteristic.getUuid().equals(UUID.fromString("00002a19-0000-1000-
8000-00805f9b34fb"))) {
        int batteryLevelValue =
characteristic.getIntValue(BluetoothGattCharacteristic.FORMAT_UINT8, 0);
        setBatteryLevelFromBackground(batteryLevelValue);
    }
}
}
}

```

Ainsi, le code complet pour la lecture du niveau de batterie d'un objet connecté est le suivant :

```

public class MainActivity extends AppCompatActivity {
    final static String TAG="MainActivity";

    TextView text;
    TextView batteryLevel;

    BluetoothManager bluetoothManager = null;
    BluetoothAdapter bluetoothAdapter=null;
    BluetoothLeScanner bluetoothLeScanner;
    BluetoothGatt bluetoothGatt;

    final static int REQUEST_PERMISSION = 102;
    final static int REQUEST_ENABLE_BLE = 201;

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

        text =(TextView)findViewById(R.id.main_text);
        batteryLevel =(TextView)findViewById(R.id.main_batteryLevel);

        ensurePermission();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if(bluetoothGatt!=null) {
            bluetoothGatt.close();
            bluetoothGatt = null;
        }
    }

    private void ensurePermission() {
        if(ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) !=

        PackageManager.PERMISSION_GRANTED) {

```



```

        if (shouldShowRequestPermissionRationale
(Manifest.permission.ACCESS_FINE_LOCATION)) {
            AlertDialog.Builder builder = new
AlertDialog.Builder(this);
            builder.setTitle(R.string.demande_permission_titre);
            builder.setMessage(R.string.explication_permission);
            builder.setNegativeButton("Non", new
DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int
which) {
                    Toast.makeText(MainActivity.this,
R.string.permission_obligatoire, Toast.LENGTH_LONG).show();
                    finish();
                }
            });
            builder.setPositiveButton("Oui", new
DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int
which) {
                    askPermission();
                }
            });
            builder.show();
        } else {
            askPermission();
        }
    } else {
        startBLEScan();
    }
}

private void askPermission() {
    requestPermissions(new String[]
{Manifest.permission.ACCESS_FINE_LOCATION}, REQUEST_PERMISSION);
}

@Override
public void onRequestPermissionsResult(int requestCode,
String[] permissions, int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions,
grantResults);
    if(requestCode==REQUEST_PERMISSION) {
        if(permissions[0].equals(Manifest.permission.ACCESS_FINE_LOCATION) &&
grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
            startBLEScan();
        }
    }
}

private void startBLEScan() {
    if(bluetoothManager==null)

```

```

        bluetoothManager= (BluetoothManager) getSystemService
(Context.BLUETOOTH_SERVICE);
        if(bluetoothAdapter==null)
            bluetoothAdapter = bluetoothManager.getAdapter();

        if(!bluetoothAdapter.isEnabled()) {
            Intent askBLE = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(askBLE, REQUEST_ENABLE_BLE);
            return;
        }
        if(android.os.Build.VERSION.SDK_INT<21) {
            bluetoothAdapter.startLeScan(leScanCallback);
            Log.d(TAG,"Scan lancé en version 18");
        } else {
            bluetoothLeScanner = bluetoothAdapter.getBluetoothLeScanner();
            bluetoothLeScanner.startScan(scanCallback);
            Log.d(TAG,"Scan lancé en version 21");
        }

        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                stopBLEScan();
            }
        }, 5000);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if(requestCode==REQUEST_ENABLE_BLE && requestCode==RESULT_OK)
            startBLEScan();
    }

    private void stopBLEScan() {
        if(android.os.Build.VERSION.SDK_INT<21)
            bluetoothAdapter.stopLeScan(leScanCallback);
        else
            bluetoothAdapter.getBluetoothLeScanner().stopScan(scanCallback);
    }

    // API 18
    BluetoothAdapter.LeScanCallback leScanCallback = new
BluetoothAdapter.LeScanCallback() {
        @Override
        public void onLeScan(BluetoothDevice device, int rssi,
byte[] scanRecord) {
            onDeviceDetected(device,rssi);
        }
    };

    // API 21
    ScanCallback scanCallback = new ScanCallback() {

```

```

        @Override
        public void onScanResult(int callbackType, ScanResult result) {
            super.onScanResult(callbackType, result);
            onDeviceDetected(result.getDevice(), result.getRssi());
        }
    };

    private void onDeviceDetected(BluetoothDevice device, int rssi) {
        String info = "Objet détecté :";
        info+="\nNom :" + device.getName();
        info+="\nAdresse :" + device.getAddress();
        info+="\nRSSI : " + String.valueOf(rssi);
        text.setText(info);
        bluetoothGatt = device.connectGatt(this, false,
bluetoothGattCallback);
    }

    private void showToastFromBackground(final String message) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                Toast.makeText(MainActivity.this, message,
Toast.LENGTH_SHORT).show();
            }
        });
    }

    private void setBatteryLevelFromBackground(final int value) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                batteryLevel.setText(String.format("Niveau batterie
:%d %%",value));
            }
        });
    }

    BluetoothGattCallback bluetoothGattCallback = new
BluetoothGattCallback() {
        @Override
        public void onConnectionStateChange(BluetoothGatt gatt, int
status, int newState) {
            super.onConnectionStateChange(gatt, status, newState);
            if(status==BluetoothGatt.GATT_SUCCESS) {
                String message = "";
                if(newState== BluetoothProfile.STATE_CONNECTED) {
                    message = "Objet connecté";
                    bluetoothGatt.discoverServices();
                }
                else {
                    message = "Objet déconnecté";
                }
                showToastFromBackground(message);
            }
        }
    }
}

```

```

@Override
public void onServicesDiscovered(BluetoothGatt gatt, int status) {
    super.onServicesDiscovered(gatt, status);
    if(status==BluetoothGatt.GATT_SUCCESS) {
        UUID batteryLevelServiceUUID =
UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb");
        UUID batteryLevelCharacteristicUUID =
UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb");
        for(BluetoothGattService service : gatt.getServices()) {
            if(service.getUuid().equals(batteryLevelServiceUUID)) {
                for(BluetoothGattCharacteristic c :
service.getCharacteristics()) {
                    if(c.getUuid().equals(batteryLevelCharacteristicUUID)) {
                        bluetoothGatt.readCharacteristic(c);
                    }
                }
            }
        }
    }
}

@Override
public void onCharacteristicRead(BluetoothGatt gatt,
BluetoothGattCharacteristic characteristic, int status) {
    super.onCharacteristicRead(gatt, characteristic, status);
    Log.d(TAG,"Caractéristique lue :" + characteristic.getUuid());
    if(status==BluetoothGatt.GATT_SUCCESS) {
        if
(characteristic.getUuid().equals(UUID.fromString("00002a19-0000-1000-8000-
00805f9b34fb")))) {
            int batteryLevelValue =
characteristic.getIntValue(BluetoothGattCharacteristic.FORMAT_UINT8, 0);
            setBatteryLevelFromBackground(batteryLevelValue);
        }
    }
}
};
}

```

Écrire une caractéristique

De la même façon que l'on ne peut lire les caractéristiques d'un objet BLE que s'il est connecté, l'écriture de valeurs dans les caractéristiques se fait uniquement lorsque l'objet est connecté au terminal Android.

Le processus à mettre en place est exactement le même : il faut commencer par lancer une découverte des services, puis, le service ciblé, itérer sur les caractéristiques du service.

La seule différence se situe lorsque la caractéristique voulue a été trouvée : au lieu d'invoquer la méthode `readCharacteristic` de l'objet `BluetoothGatt`, il faut invoquer la méthode `writeCharacteristic`.

Cette méthode ne prend, là aussi, qu'un seul paramètre, à savoir la caractéristique en question (soit l'objet `BluetoothGattCharacteristic` vu précédemment).

Il faut, bien entendu, affecter la valeur voulue à la caractéristique. Cela se fait en invoquant l'une des variantes de la méthode `setValue`, variantes proposant différents types de données : tableau d'octets, entier, chaîne de caractères.

Lorsque la donnée a été écrite, c'est, une fois encore, l'objet `BluetoothGattCallback` qui portera le retour de l'écriture, via la méthode à surcharger `onCharacteristicWrite`, méthode dont la signature est en tout point identique à celle de la méthode `onCharacteristicRead`.

Avant de refermer ce chapitre de présentation du Bluetooth Low Energy, il reste à mentionner la notion de notification : certains objets BLE offrent la possibilité au terminal de définir des notifications pour les caractéristiques à lire. Le terminal ainsi abonné à une caractéristique se verra notifié, toujours au travers du `BluetoothGattCallback` de chaque changement de valeur de la caractéristique ciblée.

Cette possibilité est propre à chaque caractéristique, qui peut ainsi, selon les objets BLE, être disponible en lecture, écriture, notification.

Mentionnons enfin, pour aider le développeur dans la découverte de la programmation avec le Bluetooth Low Energy, que plusieurs applications, développées en général par les constructeurs de puces Bluetooth eux-mêmes, sont disponibles sur le Play Store. Ces applications permettent en général d'analyser les objets BLE détectables, de se connecter à un BLE, et de lire et écrire les caractéristiques exposées.

Pour ne citer qu'une application, nous recommandons l'application `nRF Connect`, fournie gratuitement par la société `Nordic Semiconductor`.