

OpenECHO for Processing Tutorial

神奈川工科大学 スマートハウス研究センター
エネルギーマネジメントシステム標準化における
接続・制御技術研究事業 成果物

株式会社ソニーコンピュータサイエンス研究所

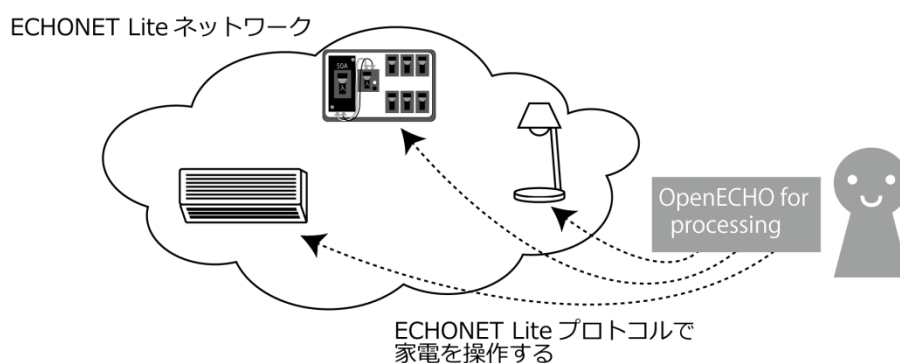
目次

| | |
|---------------------------------------|----|
| 序章 | 3 |
| 背景 | 3 |
| ECHONET Lite とは | 4 |
| OpenECHO とは | 4 |
| Processing とは | 4 |
| OpenECHO for Processing とは | 5 |
| チュートリアルを読むための下準備 | 6 |
| ECHONET Lite 公式ドキュメントの読み方 | 6 |
| 第一章 はじめてのノード作成 | 8 |
| ノードとは | 8 |
| はじめてのノード作成 | 9 |
| 他の家電機器を探す | 10 |
| 第二章 他のノードの情報を読みとる | 14 |
| EventListener で機器オブジェクトごとに処理を行う | 14 |
| Receiver を使って機器情報を受けとる | 15 |
| もう一つの例：センサー情報の取得 | 19 |
| 第三章 他のノードを制御する | 21 |
| 複数機器の一括制御 | 21 |
| 機器の個別制御 | 23 |
| 第四章 機器オブジェクトの実装 | 25 |
| 機器エミュレータ | 25 |
| 第五章 赤外線を利用した実機器オブジェクト作成 | 32 |
| iRemocon：ネットワークから制御可能な学習リモコン | 32 |
| 第六章 ノードを正しく作成する | 39 |
| ノードプロファイル | 39 |
| 必須でないメソッドのオーバーロードについて | 40 |
| プロパティ | 40 |
| プロパティマップ | 41 |
| 第七章 すべての機能を使う | 45 |
| 行うべき処理の概要 | 45 |
| ソースコード | 46 |
| おわりに | 48 |

序章

本チュートリアルは、オープンソースプロジェクトである「OpenECHO」を Processing 用にライブラリ化したものである「OpenECHO for Processing」の使い方について述べたものです。このチュートリアルを読むことで、以下のことができるようになります。

- ECHONET Lite 対応の市販の家電を操作する
- ECHONET Lite 非対応家電や自作機器を赤外線などを用いて ECHONET Lite に対応させ、ネットワークから操作する
- 市販のサービスと自作のサービスを融合させ、協調制御する



なお、本チュートリアルを理解するためには、基本的な Java のプログラミング、および Processing に関する知識が必要です。ECHONET Lite についての前提知識は不要です。また、ECHONET Lite は物理層を規定していませんが、OpenECHO は IPv4 上に構築される実装となっているため、OpenECHO for Processing も物理層に IPv4 を用いた ECHONET Lite ネットワークのみを対象としています。

背景

OpenECHO for Processing は、2011 年 12 月に公開された、「ECHONET Lite」という、ネットワーク家電やセンサーが相互に情報交換するための規約(プロトコル)に基づいたプログラムを、Processing から作りやすくするためのライブラリです。この OpenECHO for Processing ライブラリ、サンプルプログラム、および本チュートリアルは、神奈川工科大学 スマートハウス研究センター エネルギーマネジメントシステム標

準化における接続・制御技術研究事業において株式会社ソニーコンピュータサイエンス研究所で開発されたものです。

神奈川工科大学 HEMS(ECHONET Lite)認証センターのホームページ <http://sh-center.org/>

ECHONET Lite とは

ECHONET Lite とは、エコネットコンソーシアムが策定した通信プロトコルで、ISO/IEC により国際標準化されています。シンプルな制御体系や豊富な機器オブジェクト定義、物理層から独立したデザイン(つまり、既存の物理層の上に構築可能)などが特徴となっており、今後の広範な普及が見込まれます。とりわけ、経済産業省によって宅内ネットワーク用(特に、ホーム・エネルギー・マネージメント・システム：HEMS 用)の通信プロトコルとして認定され、HEMS 事業者はこのプロトコルをサポートすることにより補助金が見られるなど、積極的に普及施策がとられているところも魅力です。このプロトコルの背景や詳細情報については、ECHONET コンソーシアムの HP を参照してください。

エコネットコンソーシアム ホームページ <http://www.echonet.gr.jp/>

OpenECHO とは

ECHONET Lite は仕様が完全に公開されていますので、誰でも実装することが可能です。株式会社ソニーコンピュータサイエンス研究所(以下 Sony CSL)では、ECHONET Lite を Java で実装したクラスライブラリを開発し、オープンソース配布しています。ECHONET Lite は物理層を定義していませんので、その部分には IPv4,IPv6,ZigBee など様々なプロトコルを用いることができますが、IP での実装に関しては指針が与えられています。これに基づいて、OpenECHO は IPv4 で ECHONET Lite を実装したものとなっています。ECHONET Lite において詳細規定が存在する機器(つまり、実際に仕様が決められている機器)は 2013 年 1 月時点で 87 機器あり、OpenECHO はそれら全てをサポートしています(さらに詳細規定の存在しない「コントローラ」という機器もサポートしています)。

OpenECHO 配布元 <https://github.com/SonyCSL/OpenECHO/>

Processing とは

Processing とは、マサチューセッツ工科大学発のオープンソースのプログラミング環境で、「ソフトウェアのスケッチブック」というコンセプトで、主にプログラミング

の教育用途を念頭に開発されたものです。開発ツールのシンプルさと、豊富なライブラリとサンプルプログラムの充実から幅広い人気を獲得し、現在では教育用に限らず様々な目的、例えばメディアアートや簡単なハードウェア制御、自作システムのプロトタイピングや Android 向け開発まで広く利用されています。実際には特別な言語を用いているわけではなく、Java を使ってプログラミングをすることになります。Processing でのプログラミングについては、本家のページ等を参照してください。

Processing のホームページ <http://processing.org/>

OpenECHO for Processing とは

OpenECHO を Processing 用にコンパイルしたものが OpenECHO for Processing となります。Processing は通常の Java ライブラリをインポートできますので、このライブラリを作成するために OpenECHO のソースコードを変更する必要はほとんどなく、コンパイル方法の変更のみで済みます。ただし、コード記述量を最小化するために、ノードプロファイルや機器オブジェクトのデフォルト実装をしたクラスの追加などを行い、また、サンプルプログラムも同時に開発しています。本チュートリアルも、このサンプルプログラムをベースに記述されています。

なお、これらサンプルプログラムはもちろん Processing でそのまま動作するように作られていますが、OpenECHO を通常の Java から利用する場合も、ライブラリ使用部分のソースコードは全く同じになります。従って、本ドキュメントは OpenECHO の導入編としても利用して頂くことができます。Processing は非常に敷居の低い優れた開発環境ですが、標準のソースコードエディタにはピリオドを打った時にメンバー一覧を表示するコード補完機能や、抽象クラスを実装する場合に abstract メソッドを自動的に生成する機能などがありません。一方、Eclipse のように強力な開発支援機能のある環境を用いると、OpenECHO の各クラスのメソッド一覧をインタラクティブに確認しながら開発を進めることができるのでむしろ効率的な面もあります。OpenECHO の開発にひと通り慣れると、一番手間がかかるのは機器ごとに異なるメソッドの確認作業になります。OpenECHO for Processing の基本的な使い方に慣れて頂いたのちは、目的に応じてそういった開発環境を使って OpenECHO ご利用戴ければ開発の手間を劇的に削減することができますのでご検討ください。

チュートリアルを読むための下準備

このチュートリアルに含まれるサンプルプログラムを実行するための下準備をしておきましょう。それには OpenECHO for Processing と、ControlP5 というライブラリを Processing にインストールします。これには、Processing のスケッチを保存するフォルダ内に libraries というフォルダを作り、その下に本ライブラリに含まれる二つのフォルダ

- controlP5 フォルダ
- OpenECHO フォルダ

を展開します。

再度 Processing を起動すると、Sketch->Import Library メニューに controlP5, OpenECHO の選択肢が現れていることが確認できると思います。もし、メニューに現れていない場合は現在アクティブになっているスケッチフォルダがどこになっているか、File->Preferences にて確認の上変更し、再起動してください。

展開されるフォルダには、ライブラリ・サンプルの他に、OpenECHO のソースから Doxygen によって自動生成された html のライブラリリファレンスも含まれています。このリファレンスは、各クラスに含まれるプロパティ名や機能を調べるのに役立ちます。(このライブラリリファレンスには、後述する ECHONET Lite 公式ドキュメント(英語版)からコピーしてきた簡単な説明がつけられています。ただ、この説明は完全なものではありません)

ControlP5 は、Processing 用の GUI ライブラリで、ボタンやテキストボックスといった部品を簡単に使うことができます。このチュートリアルでは、リモコンのボタンを作るのに使用しています。

ControlP5 のホームページ <http://www.sojamo.de/libraries/controlP5/>

ECHONET Lite 公式ドキュメントの読み方

ECHONET Lite に関連する技術情報はこちらの Web サイトから得られます。

エコーネット規格(一般公開) <http://www.echonet.gr.jp/spec/index.htm>

ただし、OpenECHO for Processing を利用するにあたってそれら全てに目を通す必要はありません。特に必要なものは、以下の 2 つだけです。

- ECHONET Lite 規格書の「第 2 部 ECHONET Lite 通信ミドルウェア仕様」の中にある「プロファイルオブジェクトスーパークラス規定概要」および「ノードプロファイルクラス詳細規定」

このドキュメントには、後述する *NodeProfile* オブジェクトに含まれるプロパティに関する情報が含まれています。2013 年 1 月時点で、ECHONET Lite 規格書のバージョンは 1.01 であり、上記の情報はその日本語版ドキュメントでは P.6-4 の 6.10.1 節、および P.6-6 の 6.11.1 節に書かれています。2013 年 1 月時点での最新版はここからダウンロードできます：

http://www.echonet.gr.jp/spec/pdf_v101_lite/ECHONET-Lite_Ver.1.01_o2_.pdf

- APPENDIX ECHONET 機器オブジェクト詳細規定

これには、個々の機器オブジェクトに含まれるプロパティの情報が書かれています。

これはリファレンスなので、自分が利用する機器やセンサーの部分だけ目を通せば十分です。2013 年 1 月現在で最新版は Release B(日本語版)というものですが、OpenECHO は ReleaseA(英語版)をベースに作られています。プロパティ名も英語版のドキュメントに基づいています。A と B の違いは、Release B のドキュメントの冒頭部分「改訂履歴」に書いてあります。いくつかの規定が追加されています。2013 年 1 月時点での最新版はここからダウンロードできます：

http://www.echonet.gr.jp/spec/pdf_spec_app_b/SpecAppendixB.pdf (日本語版)

http://www.echonet.gr.jp/english/spec/pdf_spec_app_a_e/SpecAppendixA.pdf (英語版)

第一章 はじめてのノード作成

本章では、ECHONET Lite ネットワークに参加して、他の機器情報にアクセスするためのプログラムの書き方を学びます。

対応するサンプルプログラム：Tutorial1_HowToMakeANode

各章のはじめには「対応するサンプルプログラム」という記述があります。これは、OpenECHO for Processing を起動した時に、File -> Examples -> Contributed Libraries -> OpenECHO の下に現れるサンプルプログラムです。これを Processing のスケッチに読みこむことで動作確認できます。

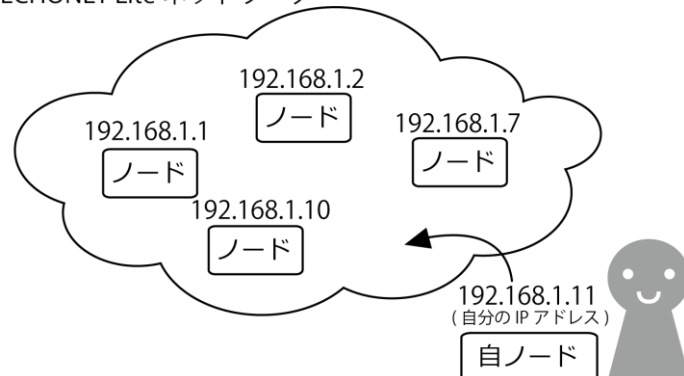
ノードとは

ECHONET Lite では、機器(家電やセンサーなど)同士の通信を「ノード」単位で行います。ノードとは、ECHONET Lite ネットワークの構成要素で、ECHONET Lite のネットワークにノード以外のものは存在していません。そのため、ネットワークに参加するには、まず「自分自身」を表現するノードを作成する必要があります。

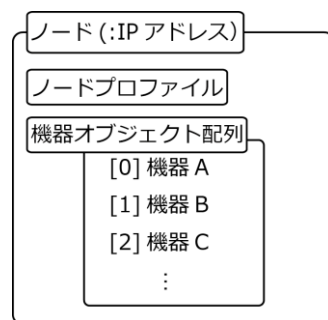
また、ノードには、ユニークな ID が必要です。IPv4 で実装されている OpenECHO では、そのノードが存在する IP アドレスがそのまま ID になります。全てのノードは違う ID を持たなければなら
ないので、1つの IP アドレスにつき、ノードは1つしか存在できないことになります。

なお、DHCP 環境下でプログラムを走らせる場合、IP アドレス(=ID)が変わることがあるので、注意が必要です。

ECHONET Lite ネットワーク



ノードは「ノードプロファイル」と1個以上の機器オブジェクト(センサーや家電機器一つ一つに対応するオブジェクト)を登録する「機器オブジェクト配列」から構成されています。ノードプロファイルは、所属するノードの情報や関連情報、現在の状態などを保持しています。一部の情報は外部から変更することもできます。一つのノードで複数の機器を表現できるので、ノード = 家電機器というわけではありません。



はじめてのノード作成

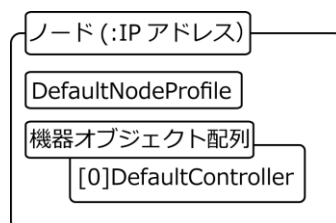
では早速ノードを作成してみましょう。プログラムを走らせるマシンに割り振られた IP アドレスが自動的にノードの ID となるので、これを敢えて明示的に指定する必要はありません。自分で作成しなければならないのは、

- ノードプロファイル(NodeProfile クラス)
- 機器オブジェクト配列(DeviceObject クラスの配列)

です。これらを指定してノードを作成するには、Echo.start メソッドを使用します。

```
try {
    Echo.start( new DefaultNodeProfile()
                ,new DeviceObject[]{new DefaultController()} );
} catch( IOException e){
    e.printStackTrace();
}
```

ここでは、ノードプロファイルとして、DefaultNodeProfile を、機器オブジェクト配列として、DefaultController のみを持つ配列を使用しています。それらを Echo.start メソッドに渡せば、ノードが作成されます。機器の配列は空ではいけないので、コントローラと呼ばれる特殊な機器を一つ作成しています。



なお、ここでは簡単のために、ノードプロファイルもコントローラもデフォルトのクラスを使用していますが、このクラスは自分向けに改変して実装しなければならない

メソッドをデフォルトの実装にしまっているため、ネットワーク内での使われ方によっては誤動作することがあります。それを避けて正しく実装する方法については、第五章を参照してください。

他の家電機器を探す

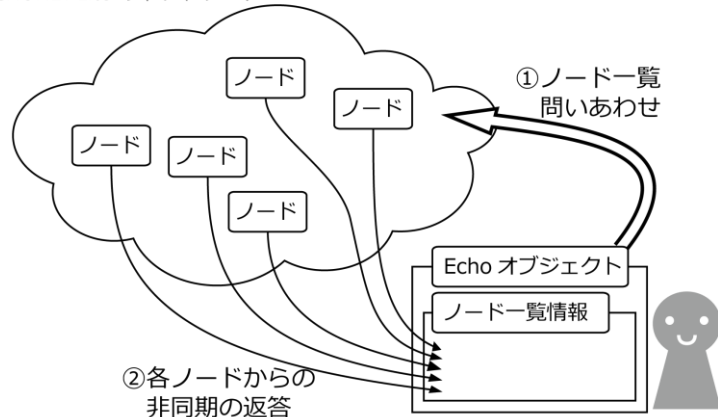
ノードを作成したことで、ひとまずネットワークに参加することができました。次に、ネットワークに参加している他のノードの情報を取得してみましょう。最初に述べたように、ノードの情報は各ノードプロファイルが保持しています。ノードに含まれている機器一覧もノードプロファイルに問い合わせることができるので、ネットワークにつながっている家電機器全てを取得するには、まず全ノードのノードプロファイルに対してそれぞれが持っている機器オブジェクトのリストをリクエストする(教えてもらう)必要があります。

```
NodeProfile.getG().reqGetSelfNodeInstanceListS().send();
```

このコードは、ネットワークに存在する `NodeProfile` すべてに対して機器オブジェクトの配列をリクエストします。様々なリクエストの書式や種類については、第二章以降で解説しますので、ここでは上記の一文を決まり文句だと思ってください。

OpenECHO では、上記の要求に対して返答を返したノードの情報を、Echo オブジェクト内部にその都度保存していきます。注意しなければならないのは、OpenECHO において、リクエストの送受信は非同期に行われているという点です。つまり、リクエストを送っても返答が返ってくるのを自動的に待ってくれるわけではありません。従って、他のノードから返答が返ってくるのを見計らって次の処理を開始する必要があります。サンプルプログラムでは 10 秒毎にリクエストを送信するようにしています。

ECHONET Lite のネットワーク



では、全てのノードから返答が返ってくるのに十分な時間を待ったとして、見つかったノードの情報を表示してみましょう。Echo オブジェクトの中にノードの情報が保存されているので、まず、それを取得します。

```
EchoNode[] nodes = Echo.getNodes();
```

getNodes の返答には自分自身のノードも含まれていることに注意してください。また、自分自身のノードのみを取得するには、getNode を使うことができます。

```
EchoNode localNode = Echo.getNode();
```

では、まず上記二つの行を実行したとして、得られた各ノードの ID(IP アドレス)を表示してみましょう。EchoNode.getAddress は、InetAddress (IP アドレスを表すオブジェクト)を返すので、それを getHostAddress で文字列に変換しています。

```
for(EchoNode en : nodes){
    if(en == localNode){
        println("Node id = " + en.getAddress().getHostAddress() + "(local)");
    }else{
        println("Node id = " + en.getAddress().getHostAddress());
    }
}
```

次に、各ノードのノードプロファイルにアクセスし、情報を読んでみましょう。

```
for(EchoNode en : nodes){
    println("NodeProfile=" + en.getNodeProfile());
}
```

すると、以下のような出力が得られると思います。

```
NodeProfile=groupCode:03,classCode:fo,instanceCode:01,address:192.168.0.3
```

※この末尾の IP アドレス(192.168.0.3)は環境により変動します。

ECHONET Lite において、NodeProfile や機器オブジェクトはその種類に応じて、「groupCode」と「classCode」を持っています。これらは ECHONET Lite の規格書において定められた値です。また、「instanceCode」というのは、ノードの中でその機器が何番目かを示す値で、機器の種類ごとに 1 から数えあげる番号です。これら 3 つの値を合

わせれば、ノード内での機器オブジェクトを一意に特定できることになります。これらをまとめて ECHONET Lite では[o3.f0][o1]のように表現して EOJ(ECHONET オブジェクト)と呼んでいます。本チュートリアル内では常に 16 進数で表現するので、以降接頭詞の ox などは使いません。ご注意ください。

次に、ノードの持つ機器オブジェクトをリストしてみましょう。機器オブジェクトを取得するには、`EchoNode.getDevices` を使用します。先程の全ノードを探索するループの内側に、一つ一つの機器にアクセスするループを追加します。

```
for(EchoNode en : nodes){
    println(" Devices:");
    DeviceObject[] dos = en.getDevices();
    for(DeviceObject d : dos){
        println("    " + d);
    }
}
```

ここで得られるのは、上記の `NodeProfile` と同じような出力になります。
`groupCode,classCode` からその機器がなんであるかを調べることができるでしょう。

ここまで説明したすべてのプログラムをまとめたものが以下になります。

```
import java.io.IOException;
import com.sonycs1.echo.Echo;
import com.sonycs1.echo.node.EchoNode;
import com.sonycs1.echo.eoj.profile.NodeProfile;
import com.sonycs1.echo.eoj.device.DeviceObject;

import com.sonycs1.echo.processing.defaults.DefaultNodeProfile;
import com.sonycs1.echo.processing.defaults.DefaultController;

void setup(){
    try {
        Echo.start( new DefaultNodeProfile(),new DeviceObject[] {new DefaultController()});
    } catch( IOException e){ e.printStackTrace(); }

    while(true) {
        try {
            // Query existing node profiles (amounts to finding existing nodes)
            NodeProfile.getG().reqGetSelfNodeInstanceListS().send();

            EchoNode[] nodes = Echo.getNodes();
            for( int i=0;i<nodes.length;++i ){
                EchoNode en = nodes[i];
                println( "node id = "+en.getAddress().getHostAddress());
            }
        }
    }
}
```

```

println( "node profile = "+en.getNodeProfile());

DeviceObject[] dos = en.getDevices();
println( "There are "+dos.length+" devices in this node");

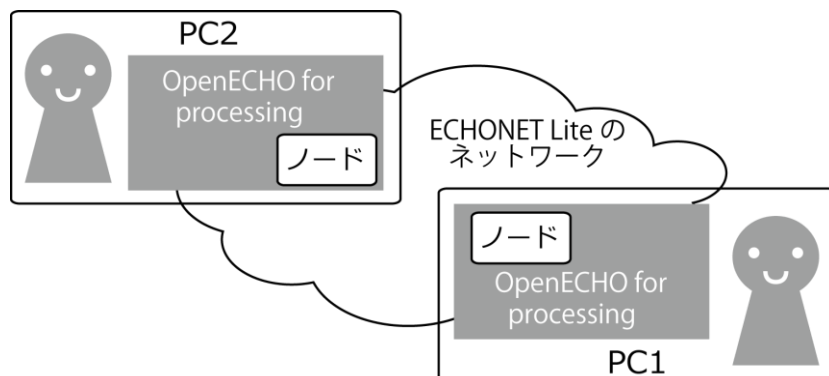
for( int j=0;j<dos.length;++j ){
    DeviceObject d = dos[j];
    println("device type = "+d.getClass().getSuperclass().getSimpleName());
}
println("----");
}
} catch( IOException e){ e.printStackTrace(); }

// 10 秒待つ。
try { Thread.sleep(10000); } catch (InterruptedException e) { e.printStackTrace(); }
}
}

```

このプログラムを実行させると、ネットワーク上に存在するノードが表示され、それらに含まれる機器オブジェクトの情報が表示されます。

ECHONET Lite 機器が家にはない場合は、同じローカルエリアネットワークに接続されている他のマシンでも同じプログラムを走らせてみましょう。そのノードがもう1つのノードとして発見されるはずです。



第二章 他のノードの情報を読みとる

前章では、とりあえずノードを作ってみて、他のノードの一覧と、その中に含まれる機器一覧を表示することを学びました。

本章では一歩進んで、特定の種類のノードから、そのノード固有の情報を取得してみましよう。ここでは、ECHONET Lite に対応した HEMS 機器(分電盤にセンサーをつけて電力量を取得したり、太陽電池の出力を蓄電池に溜めたりするための機器)を家庭に設置するとたいてい定義されている「分電盤オブジェクト」から、電力使用量を取得してみましよう。

対応するサンプルプログラム：Tutorial2a_PowerDistributionBoard

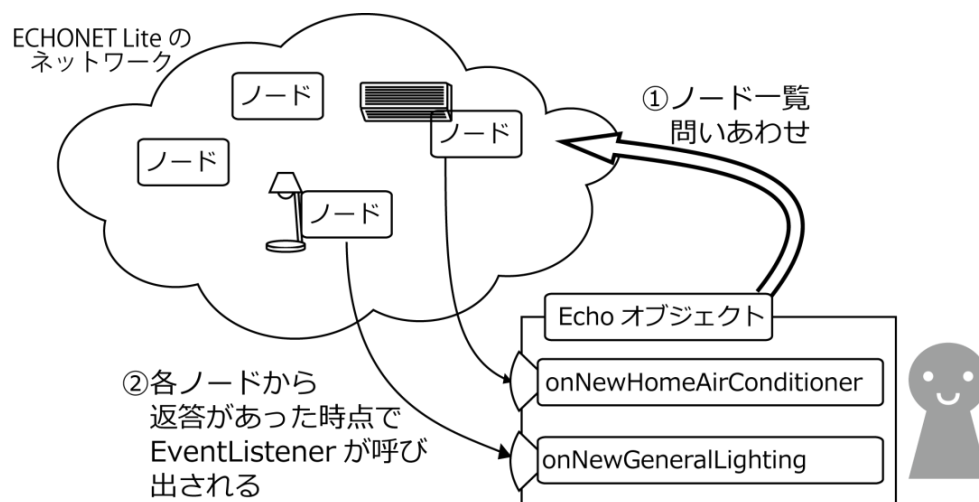
Tutorial2c_TemperatureHumiditySensor

EventListener で機器オブジェクトごとに処理を行う

ノードから情報を得るには、「EventListener」というものを設定します。

EventListener は、OpenECHO の様々な変化を受け取るために使うものです。ここではネットワーク上に新たな機器が見つかるごとに呼び出されるハンドラーを設定します。

前章では、すべてのノードから返答が返ってきた頃を見計らって、Echo.getNodes() でノード一覧を取得し、その中に含まれる機器に対して処理を行っていました。一方、EventListener を用いれば、自分が関心を持つ機器が見つかり次第、すぐに処理を行うことができます。



`EventListener` は一回だけ設定すれば十分なので、`Processing` の `setup` メソッドに書いていきます。`EventListener` クラスには、ECHONET Lite に定義された機器にそれぞれに対応したオーバーライド可能なハンドラーが定義されているので、ユーザーは、処理を行いたいノードに対応するハンドラーをオーバーライドし、自分の行いたい処理を実装します。`EventListener` に定義されているハンドラーはデフォルトで空(何もしない)ですが、オーバーライドした時に、親クラスのハンドラーも呼んでおいたほうがよいでしょう。それにより、関数名を間違えることがなくなります。

`EventListener` を登録するには `Echo.addEventListener` を呼び出します。試しに、空の `EventListener` を登録してみましょう(`EventListener` はデフォルトで空なので、次のコードは何の効果ももたらしません)。

```
Echo.addEventListener(new Echo.EventListener() {});
```

では、これに分電盤が見つかったときに実行されるコードを追加しましょう。それには、`Echo.EventListener` の持つハンドラーのうち、

```
void onNewPowerDistributionBoardMetering(PowerDistributionBoardMetering)
```

をオーバーライドします。その中で親クラスの同名のハンドラーも呼び出しておきます。

```
Echo.addEventListener(new Echo.EventListener() {  
    public void onNewPowerDistributionBoardMetering(  
                                                PowerDistributionBoardMetering device){  
        super.onNewPowerDistributionBoardMetering(device);  
    }  
});
```

※このようにオーバーライド可能なハンドラーは、ECHONET Lite の機器オブジェクトの数だけ存在します。これらの一覧は、OpenECHO リファレンスの `com.sonycsl.echo.Echo.EventListener` の定義を参照してください。

Receiver を使って機器情報を受けとる

今回は分電盤の現在の電力消費量を取得したいので、オーバーライドした `onNewPowerDistributionBoardMetering` 内で以下の操作を行います。

- 分電盤オブジェクトに対し、Receiver と呼ばれるオブジェクトを登録する。
- 分電盤にリクエストを送る。

ネットワーク家電の常として、リクエストを送ってもその返答が返ってくるまでには時間がかかります。Receiver は、このように非同期で返ってくる返答を処理するために用いるクラスです。分電盤から電力量を取得するには、取得リクエストを送る前に Receiver クラスの中にある `onGetMeasuredInstantaneousCurrents` というハンドラーをオーバーライドしておく、返答を受けとることができるようになります。以下の例では、それに加えて `onGetOperationStatus` という、電源が入っているかどうかを調べるハンドラーも設定しています。

```
Echo.addListener(new Echo.EventListener() {
    // 新たな分電盤がネットワーク上に見付かったときに呼び出される
    public void onNewPowerDistributionBoardMetering(PowerDistributionBoardMetering device) {
        // 親クラスのメソッドを呼び出しておく
        super.onNewPowerDistributionBoardMetering(device);

        // 後で get した時に、その返答を受けとるための Receiver を設定しておく
        device.setReceiver(new PowerDistributionBoardMetering.Receiver() {
            // 電力量を取得するためのハンドラー
            protected void onGetMeasuredInstantaneousCurrents(EchoObject eoj, short tid, byte esv,
                                                                EchoProperty property, boolean success) {
                super.onGetMeasuredInstantaneousCurrents(eoj, tid, esv, property, success);
                System.out.println("GetMeasuredInstantaneousCurrents : "+toHexStr(property.edt));
            }
            // 電源が入っているかどうかを取得するためのハンドラー
            protected void onGetOperationStatus(EchoObject eoj, short tid, byte esv,
                                                EchoProperty property, boolean success) {
                super.onGetOperationStatus(eoj, tid, esv, property, success);
                System.out.println("PowerDistributionBoardMetering power : "+toHexStr(property.edt));
            }
        });
        // レシーバーの設定が完了

        // 分電盤に対して情報をリクエストする (後述)
        try {
            device.get().reqGetMeasuredInstantaneousCurrents().reqGetOperationStatus().send();
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 分電盤発見時の処理完了
    }
});

// EventListener 設定完了。
// 全ての準備が整ったので、ネットワークにノード一覧を問い合わせる
```



```
try {
    Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
    NodeProfile.getG().reqGetSelfNodeInstanceListS().send();
} catch( IOException e){ e.printStackTrace(); }
```

※toHexStr()は、byte 配列を 16 進数の文字列に変換するための関数で、サンプルの中に定義されています

このプログラムを実行すると、ネットワーク内にある分電盤を探し、そこから電力量を取得すると電源が入っているかどうか(入っている時は 30、入っていないければ 31 を返します)と、現在の電力量が取得できればそれを表示します。

Receiver の中にあるハンドラーは、その種類によらず引数の並びは同じです。機器の情報を読みとる場合最も重要なのは、`byte[] property.edt` です。例えば電力量の場合、ここに取得した電力量が入っています。byte の範囲に収まらないような大きな返答データを扱う場合は、複数の byte に分けられて格納されることになります。リクエストに応じてどのような結果が返ってくるかは、ECHONET Lite の規格書を参照してください。

それ以外の引数の意味は、以下のようにになります。eoj は、リクエストを受け取った機器、tid はフレーム番号、esv はリクエストの種類、property は、受け取ったデータ、success は成功したかどうかです。property は、epc,pdc,edt というメンバを持ちます。epc はプロパティ ID (機器の状態を表す変数の ID だと思ってください)、edt は先程の説明の通りで、受け取ったバイト列、pdc は edt の長さです。

Receiver の登録が終わったら、実際に機器に情報を送ってくれるようにリクエストを送りましょう。それには次のようにします。

```
device.get().reqGetMeasuredInstantaneousCurrents().send();
```

この device とは、Echo に指定した EventListener(今回は onNewPowerDistributionBoardMetering)の引数に渡ってくる値で、機器そのものを表すオブジェクトです。device.get()で、その機器の Getter を取得します。Getter とは、その機器に対して取得リクエスト可能な値をメソッドの形で表現したもので、そのメンバ関数を呼び出すことで機器に値取得リクエストを送信することができます。具体的には Getter に送りたいリクエストの関数を呼び出し、その返り値に対して.send()を呼び出すことで機器オブジェクトにリクエストを送信することができます。実際のコードでは、IOException を try-catch しておく必要があります。なお、Getter は、機器固有のものであり、実際に送ることができるリクエストは、機器の実装に依存することに注意してください。ベンダ固有のリクエストがあるかもしれませんが、送っても返答のないリクエストもあるかもしれません。

また、複数のリクエストを同時に送ることもできます。それには以下のようにメソッドをつなげていきます。(reqGetMeasuredInstantaneousCurrents()の返り値に対して、さらに reqGetOperationStatus()を呼び出すことになります)

```
try{
    device.get().reqGetMeasuredInstantaneousCurrents().reqGetOperationStatus().send();
} catch(IOException e){
    e.printStackTrace();
}
```

また、今回は get を使いましたが、それ以外にも set と inform があります。同種のリクエストであれば、やはり同じようにつなげていき、最後に send()を呼び出すことで、同時送信することができます。この 3 種類のリクエストは、それぞれ以下のような基本的な意味を持っています。

- get : 特定の機器オブジェクトに情報を問い合わせるリクエストです。返答は自分のみに送信されてきます。
- set : 特定の機器オブジェクトへの情報送信 (書き込み)です。デフォルトでは成功したかどうかの返答も受けとることができます。
- inform : 特定の機器オブジェクトに情報を問い合わせるリクエストです。返答は自分を含む、ネットワーク内全てのノードに送られることが get との違いです。ただし、対象オブジェクトが返答を生成できない、いわゆる「不可応答」の場合は、その情報は送り元のみにエラーとして返答されてきます。

ECHONET Lite において各機器オブジェクトに含まれるのは「プロパティ」と呼ばれるメンバ変数のようなものです。このプロパティに対して get、set、inform を行なうことで、状態の読みとりや制御を行うわけですが、今回は電力量の読み込みだったので get を用いましたが、例えば、照明のオンオフは書き込みなので、set リクエストを送ることによって機器を操作することができます。

さらに、これらのリクエストの送信先が特定の機器オブジェクトのみではなく、ネットワーク内に存在する同種類の機器オブジェクト全てとなる getG、setG、informG が存在しています(末尾の G は Group を表します)。これらは相手を特定しなくてよいためのスタティックメソッドになっており、クラス名から直接呼び出すことができます。詳細は三章で説明します。

機器オブジェクトに対してどのようなリクエストを送ることができるかは、分電盤の場合、PowerDistributionBoardMetering の Getter/Setter メソッド一覧を参照してく

ださい。他の機器オブジェクトに対しても同様に、ライブラリリファレンスから当該機器オブジェクトの Getter/Setter メソッド一覧を参照してください。

サンプルでは、最後にノード一覧をリクエストするコードで終わっています。これによりノード発見が行われ、見つかった時に `EventListener` が呼ばれるようになります。一章の例と違い、今回は何度もノード一覧リクエストを行う必要はありません。

もう一つの例：センサー情報の取得

本質的には分電盤と変わりませんが、もう一つ利用価値の高いサンプルとして、温度センサーと湿度センサーの値を読み取るサンプルプログラムを以下に載せます。

```
import com.sonyosl.echo.Echo;
import com.sonyosl.echo.EchoProperty;
import com.sonyosl.echo.eoj.EchoObject;
import com.sonyosl.echo.eoj.device.DeviceObject;
import com.sonyosl.echo.eoj.profile.NodeProfile;
import com.sonyosl.echo.eoj.device.sensor.TemperatureSensor;
import com.sonyosl.echo.eoj.device.sensor.HumiditySensor;
import com.sonyosl.echo.processing.defaults.DefaultNodeProfile;
import com.sonyosl.echo.processing.defaults.DefaultController;

int bti(byte[] b){
    int ret = 0;
    for( int bi=0;bi<b.length;++bi ) ret = (ret<<8)|(int)(b[bi]&0xff);
    return ret;
}

void setup(){
    // System.out にログを表示するようにします。
    //Echo.addListener( new Echo.Logger(System.out) );

    Echo.addListener(new Echo.EventListener() {
        public void onNewTemperatureSensor (TemperatureSensor device){
            println( "Temperature sensor found." );
            device.setReceiver( new TemperatureSensor.Receiver(){
                protected void onGetMeasuredTemperatureValue(EchoObject eoj, short tid, byte esv,
                                                                EchoProperty property, boolean success){
                    super.onGetMeasuredTemperatureValue(eoj, tid, esv, property, success);
                    int ti = bti(property.edt);
                    //ECHONET Lite では温度の 10 倍の値が返ってくるので、1/10 倍する
                    float tmpe = ti*0.1;
                    println("Temperature : "+ tmpe + " degree");
                }
            });
        }
    });
    try {
        device.get().reqGetMeasuredTemperatureValue().send();
    } catch( IOException e){ e.printStackTrace(); }
```

```

    }

    public void onNewHumiditySensor (HumiditySensor device){
        println( "Humidity sensor found.");
        device.setReceiver( new HumiditySensor.Receiver(){
            protected void onGetMeasuredValueOfRelativeHumidity(EchoObject eoj, short tid
                , byte esv, EchoProperty property, boolean success) {
                super.onGetMeasuredValueOfRelativeHumidity(eoj, tid, esv, property, success);
                println("Humidity : "+property.edt[o]+"%");
            }
        });
        try {
            device.get().reqGetMeasuredValueOfRelativeHumidity().send();
        } catch( IOException e){ e.printStackTrace(); }
    }
};
try {
    Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
    NodeProfile.getG().reqGetSelfNodeInstanceListS().send();
} catch( IOException e){ e.printStackTrace(); }
println("Started");
}

```

このプログラムを実行すると、温度センサーが見付かれれば"Temperature sensor found."、湿度センサーが見付かれれば"Humidity sensor found."と表示し、それぞれの計測値を表示します。

本章の締めくくりとして、本章で用いた機器オブジェクトの EOJ と、アクセスしたプロパティの ID(EPC)を記します。これらの情報は、OpenECHO for Processing の通常の使用において明示的に指定する必要はありませんが、将来 ECHONET Lite のシステムのデバッグをパケットモニタリングなどで行う場合に知っておくと便利です。OpenECHO で用いているメソッド名は、ECHONET Lite 英語版規格書からの自動処理によって生成されていますが、プロトコル上は EOJ や EPC といったコードのみで通信がなされています。

分電盤(PowerDistributionBoardMetering)の EOJ : [02.87] (インスタンスコードは省略しています)。

動作状態(OperationStatus)の EPC : 80

瞬時電流計測値(MeasuredInstantaneousCurrents)の EPC: C7

温度センサーの EOJ : [00.11]

温度計測値(MeasuredTemperatureValue)の EPC : E0

湿度センサーの EOJ : [00.12]

相対湿度計測値(MeasuredValueOfRelativeHumidity)の EPC : E0

第三章 他のノードを制御する

第二章では、分電盤オブジェクトに対して `get` リクエストを呼び出して情報を読み取りました。第三章では、`set` リクエストを使用して、他の機器の状態を変更することを学びます。今回使用する機器は、照明(`GeneralLighting`),エアコン(`HomeAirConditioner`)です。

対応するサンプルプログラム：Tutorial3a_AllLightsAirconOff
Tutorial3b_AllLightsAirconOff_Individual

複数機器の一括制御

まず、ネットワーク上に存在するすべての照明とエアコンをオフにするプログラムから作っていきましょう。例えば、家を出るときに全ての電源をワンタッチでオフにできると便利ですね。照明オブジェクトは、ECHONET Lite の規格書にはいくつかの種類がありますが、ここでは `GeneralLighting` というクラスを対象にします。

まずはこれまでのように `Echo.start` メソッドによりノードを作ります。

```
Echo.start( new DefaultNodeProfile()
            ,new DeviceObject[]{new DefaultController()});
```

次に、

```
GeneralLighting.setG().reqSetOperationStatus(new byte[]{0x31}).send();
```

を呼び出すことにより、`GeneralLighting` のインスタンス全てに対して `SetOperationStatus` リクエストを、`0x31` というデータを載せて送信します。`0x31` というデータは電源オフという意味です。

この呼び出し方法は、これまでノード一覧を得るために使用していた以下の呼び出し方法と同様のものです。

```
NodeProfile.getG().reqGetSelfNodeInstanceListS().send();
```

このノード一覧を得るためのコードは、実際には全ての `NodeProfile` のインスタンスに対し、機器オブジェクトのリストを通知するように要求するものです。`GeneralLighting.setG()`との共通点は、ネットワーク上に存在するもの全てにリクエストを送るということです。つまり、

```
(機器クラス).(リクエストの種類)G.req(リクエスト)((送信データ)).req...send();
```

というコマンドは、機器クラスのインスタンス全体にリクエストを送信します。機器クラスでなく `NodeProfile` でも全く同じ呼び出し方になります。「リクエストの種類」は `get|set|inform` で、「リクエスト」はそれに対応した `Get*|Set*|Inform*`です(*には `OperationStatus` などのプロパティ名が入ります)。「送信データ」は、`get` や `inform` では不要です。

`NodeProfile.getG()`とすることで、全ての `NodeProfile` に対して `inform` リクエストを同時に送ることができるわけです。`set` の場合は書き込みなので、引数に送信データを加えます。エアコンの場合もこの書式にしたがって、

```
HomeAirConditioner.setG().reqSetOperationStatus(new byte[]{0x31}).send();
```

などと書くことができます。

これにより、機器すべてをオフにすることができました。これまでのコードをおさらいすると、

```
import com.sonycs1.echo.Echo;
import com.sonycs1.echo.EchoProperty;
import com.sonycs1.echo.eoj.EchoObject;
import com.sonycs1.echo.eoj.device.DeviceObject;
import com.sonycs1.echo.eoj.profile.NodeProfile;
import com.sonycs1.echo.eoj.device.housingfacilities.GeneralLighting;
import com.sonycs1.echo.eoj.device.airconditioner.HomeAirConditioner;
import com.sonycs1.echo.processing.defaults.DefaultNodeProfile;
import com.sonycs1.echo.processing.defaults.DefaultController;

void setup(){
    try {
        Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
        // 全ての照明の Off
        GeneralLighting.setG().reqSetOperationStatus(new byte[]{0x31}).send();
        // 全てのエアコンの Off
        HomeAirConditioner.setG().reqSetOperationStatus(new byte[]{0x31}).send();
    } catch(IOException e){
```

```

        e.printStackTrace();
    }
    println("Started");
}

```

となります。

機器の個別制御

全ての機器ではなく特定の機器のみをオフにする場合は、分電盤から情報を取得したときと同様に、当該ノードが発見されたときの処理を *EventListener* に記述しておいた上でネットワーク上にノード一覧をリクエストします。

```

import com.sonyosl.echo.Echo;
import com.sonyosl.echo.EchoProperty;
import com.sonyosl.echo.eoj.EchoObject;
import com.sonyosl.echo.eoj.device.DeviceObject;
import com.sonyosl.echo.eoj.profile.NodeProfile;
import com.sonyosl.echo.eoj.device.housingfacilities.GeneralLighting ;
import com.sonyosl.echo.eoj.device.airconditioner.HomeAirConditioner;
import com.sonyosl.echo.processing.defaults.DefaultNodeProfile;
import com.sonyosl.echo.processing.defaults.DefaultController;

void setup(){
    Echo.addEventListener(new Echo.EventListener() {
        public void onNewGeneralLighting (GeneralLighting device){
            super. onNewGeneralLighting(device);
            println( "General Lighting found.");
            // ここで必要な照明のみを対象にするようなコードを加える
            try {
                device.set().reqSetOperationStatus( new byte[] {0x31}).send();
            } catch (IOException e){
                e.printStackTrace();
            }
        }
    });
    public void onNewHomeAirConditioner (HomeAirConditioner device){
        super. onNewHomeAirConditioner (device);
        println( "HomeAirConditioner found.");
        // ここで必要なエアコンのみを対象にするようなコードを加える
        try {
            device.set().reqSetOperationStatus( new byte[] {0x31}).send();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
try {

```

```

        Echo.start( new DefaultNodeProfile(),new DeviceObject[] {new DefaultController()});
        NodeProfile.informG().reqInformInstanceListNotification().send();
    } catch(IOException e){
        e.printStackTrace();
    }
}

```

なお、set の場合でもリクエストの送りっぱなしではなく、get リクエストと同じように返答が返ってきます。この返答を受けとるには *Receiver* を設定しなければならないのも get と同様ですが、get では、*device.Receiver.onGet...* をオーバーライドして設定したのに対し、set では、*device.Receiver.onSet...* をオーバーライドします。

例えば、全ての機器オブジェクトの *Receiver* には次のハンドラーが定義されています。

```

protected void onSetOperationStatus(EchoObject eoj, short tid, byte esv,
                                     EchoProperty property, boolean success)

```

これは、*reqSetOperationStatus* リクエストを送った際の返答時に呼ばれるハンドラーです。成功時には、*success* は *true*、*property* の中は空になります。失敗時には、*success* は *false*、*property* の中は、set 時に設定したデータがそのまま入っています。

なお、OpenECHO では *inform* に対する返答も *onGet...* に返ってくることになっているため、これらを別の *Receiver* ハンドラーで受けとることはできないことに注意してください。

最後に本章で用いた機器オブジェクトの EOJ と、アクセスしたプロパティの ID(EPC)を記します。

```

一般照明(GeneralLighting)の EOJ  :[02.90]
動作状態(OperationStatus)の EPC : 80

```

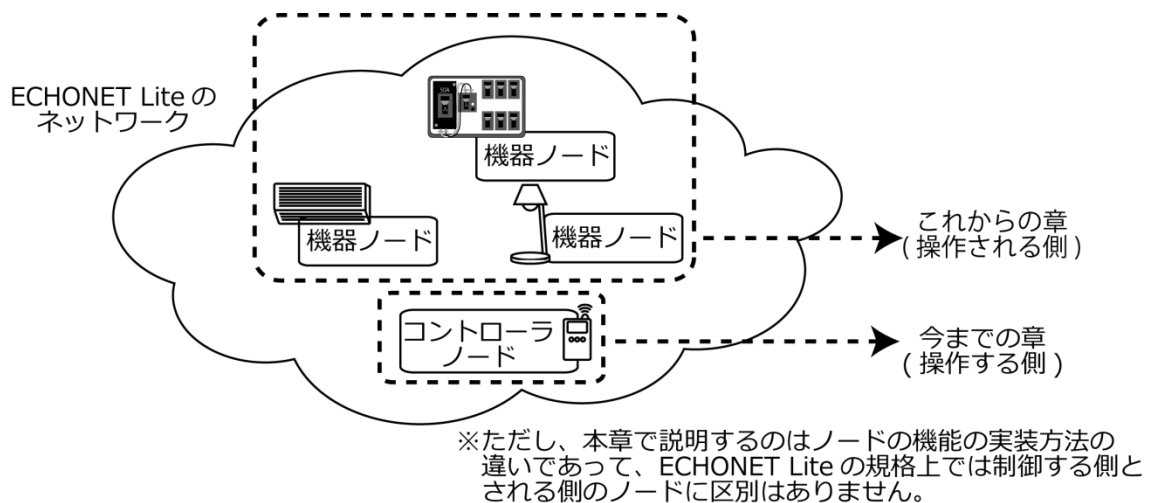
```

家庭用エアコン(HomeAirConditioner)の EOJ : [01.30]
動作状態(OperationStatus)の EPC : 80

```


第四章 機器オブジェクトの実装

ここまで作ってきたノードは、機能的には家電機器ではなく、それらを制御する側である「コントローラ」のみを持つノードでした。本章では、これまでとは逆に、コントロールされる対象側の機器を持つノードを作成し、ネットワーク上の他のノードから制御できるようにしてみましょう。



これができるようになると、ECHONET Lite 対応の機器を持っていなくても、自作機器を ECHONET Lite 対応機器としてネットワークに参加させ、コントローラノードから制御できるようになります。例えば市販のモーターを使って小さい扇風機を作ってみるのはどうでしょうか？

対応するサンプルプログラム：Tutorial4_LightEmulator

機器エミュレータ

本章では、実際の機器を真似するエミュレータを作ってみましょう。それには、使用する機器のクラスを継承した独自のクラスを作成します。今回は、*GeneralLighting* をエミュレートするクラスを作成してみます。

GeneralLighting の場合、

- `setOperationStatus`
- `getOperationStatus`
- `setInstallationLocation`
- `getInstallationLocation`
- `getFaultStatus`
- `getManufacturerCode`

をオーバーライドしなければなりません。これらは、親クラスで *abstract* で宣言されているので、オーバーライドしなければコンパイルエラーとなります。これらのメソッドは、ECHONET Lite で必須と定義されているものです。オーバーライドすべきメソッドが多いのですが、まずは実装例を以下に示します。

```
public class LightEmulator extends GeneralLighting {
    byte[] mStatus = {0x31}; // 電源状態を格納する変数です。デフォルトは OFF と仮定します。
    byte[] mLocation = {0x00}; // 機器の置き場所を格納する変数です。
    byte[] mFaultStatus = {0x42}; // 機器に問題が発生した時に、そのコードを格納します。
    byte[] mManufacturerCode = {0, 0, 0}; // ベンダー固有のメーカーコードです。

    protected boolean setOperationStatus(byte[] edt) {
        mStatus[0] = edt[0];
        //電源状態が変化したことを他のノードに通知します
        try {
            inform().reqInformOperationStatus().send();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return true;
    }

    protected byte[] getOperationStatus() {
        return mStatus;
    }

    protected boolean setInstallationLocation(byte[] edt) {
        mLocation[0] = edt[0];
        try {
            inform().reqInformInstallationLocation().send();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return true;
    }
    protected byte[] getInstallationLocation() {
        return mLocation;
    }

    protected byte[] getFaultStatus() {
```

```

        return mFaultStatus;
    }
    protected byte[] getManufacturerCode() {
        return mManufacturerCode;
    }
}

```

ここで、オーバーライドしたメソッドについて、簡単に説明しておきます。

- `setOperationStatus`
- `getOperationStatus`

これらは、第一章、二章でも出てきた、稼動状態の読み書きメソッドです。`set*`は他のノードなどから値を設定するリクエストを受けとった時に呼びだされるメソッド、`get*`は他のノードから値を取得したいとリクエストされた時に呼び出されるメソッドです。

- `setInstallationLocation`
- `getInstallationLocation`

これらは、機器の設置箇所の読み書きメソッドです。すべての機器の親クラスとなる、`DeviceObject` に `abstract` 宣言されているため、すべての機器でオーバーライドしなければなりません。

- `getFaultStatus`

異常状態か、そうでないかです。ここで返すデータにより、どのような異常が発生しているかを表します。こちらでも `DeviceObject` で `abstract` 宣言されています。

- `getManufacturerCode`

これは ECHONET Lite コンソーシアムの会員になった企業に割り当てられる、メーカーコード(製造社番号)です。やはり `DeviceObject` で `abstract` 宣言されています。

これらの返すべきデータの詳細は、ECHONET Lite の規格書や OpenECHO のリファレンスを参照してください。

ここで注意すべきことがあります。今回、自分のノードの下に機器を作成しましたが、その情報取得や制御などをする目的で実装した `setOperationStatus` や、`getInstallationLocation` 等を自分で直接呼び出してはいけません。操作対象が自分のノード

ド内にあっても情報の取得や制御には必ず `set().reqSet*` や `get().reqGet*` を使用してください。なぜなら、これらのメソッドを使用した場合、OpenECHO が想定する方法ではない方法で機器の状態が変更されることになり、これらの変更が OpenECHO ライブラリに通知されずに不具合が生じる可能性があるからです。

ところで `set` メソッドの中で、

```
inform().reqInformOperationStatus().send();
```

という行があります。これは、他のノードに対し、機器オブジェクトが状態変化したということを伝えるものです。なぜこれが `inform` になるかというと、この機器オブジェクトからネットワーク全体に対して通知を送ってほしいからです。このように、機器オブジェクトが自ノードに含まれていようがいまいが、とにかくそこからネットワーク全体へ通知をさせるには `inform` を使うことになります。状態変化した時にネットワークに対してそれを通知すべきかどうかは、ECHONET Lite の規格書に載っています。各機器オブジェクトのプロパティ一覧表において「状態時アナウンス」という列があるので、これに○がついているプロパティが変化した場合には、状態変化を `inform()` してください。

今回の場合 `OperationStatus` を `Inform` しているので、もしこの通知を OpenECHO で作成された他のノードが受けとったとすると、`EventListener` の、

```
public void onSetProperty(EchoObject eo, short tid, byte esv, EchoProperty property, boolean success)
```

というハンドラーが呼び出されます。

なお、本章では、必須となっているメソッドのみを実装します。必須でないメソッドの場合、メソッドをオーバーライドするだけでは不十分で、実装したメソッド情報について「プロパティマップ」というものに設定しておく必要があります。必須でないメソッドの実装とプロパティマップの設定方法については、第六章で解説しています。

定義したクラスを使用した完全なプログラムが以下になります。動作状態(電源の ON/OFF)に応じて、ウィンドウの背景を変更するようにしています。

```
import java.io.IOException;
import processing.net.*;
import controlP5.*;
```

```

import com.sonycs1.echo.Echo;
import com.sonycs1.echo.node.EchoNode;
import com.sonycs1.echo.eoj.profile.NodeProfile;
import com.sonycs1.echo.eoj.device.DeviceObject;

import com.sonycs1.echo.processing.defaults.DefaultNodeProfile;
import com.sonycs1.echo.eoj.device.housingfacilities.GeneralLighting;

color backgroundLightOnColor = color(255, 204, 0);
color backgroundLightOffColor = color(0, 0, 0);
color backgroundNow = backgroundLightOffColor;

// 照明クラスを実装します。
public class LightEmulator extends GeneralLighting {
    byte[] mStatus = {0x31}; // 電源状態を格納する変数です。デフォルトは OFF と仮定します。
    byte[] mLocation = {0x00}; // 機器の置き場所を格納する変数です。
    byte[] mFaultStatus = {0x42}; // 機器に問題が発生した時に、そのコードを格納します。
    byte[] mManufacturerCode = {0, 0, 0}; // ベンダー固有のメーカーコードです。

    protected boolean setOperationStatus(byte[] edt) {
        mStatus[0] = edt[0];
        //背景色を変更
        if(mStatus[0] == 0x30){
            backgroundNow = backgroundLightOnColor;
        }else{
            backgroundNow = backgroundLightOffColor;
        }
        //電源状態が変化したことを他のノードに通知します
        try {
            inform().reqInformOperationStatus().send();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return true;
    }

    protected byte[] getOperationStatus() {
        return mStatus;
    }

    protected boolean setInstallationLocation(byte[] edt) {
        mLocation[0] = edt[0];
        try {
            inform().reqInformInstallationLocation().send();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return true;
    }

    protected byte[] getInstallationLocation() {
        return mLocation;
    }
}

```

```

    }
    protected byte[] getFaultStatus() {
        return mFaultStatus;
    }
    protected byte[] getManufacturerCode() {
        return mManufacturerCode;
    }

    public String toString() {
        if (mStatus[0] == 0x31) {
            return "Light Emulator(Off)";
        }
        else {
            return "Light Emulator(On)";
        }
    }
}

ControlP5 cp5 ;
LightEmulator light ;
String[] btnStrs = {
    "SWITCH_ON", "SWITCH_OFF"
};

void setup() {
    size(210, (btnStrs.length)*30);
    frameRate(30);

    // 次に、学習と再生のユーザーインターフェースを作成します。
    cp5 = new ControlP5(this);
    // 送信用のボタンを左に、学習用のボタンを右に表示します。
    for ( int bi=0;bi<btnStrs.length;++bi ) {
        cp5.addButton(btnStrs[bi], 0, 0, (bi)*30, 100, 25);
    }

    // System.out にログを表示するようにします。
    //Echo.addEventListener( new Echo.Logger(System.out) );

    // 自分自身が LightEmulator を含むノードになることにしましょう。
    try {
        light = new LightEmulator();
        Echo.start( new DefaultNodeProfile(), new DeviceObject[] {
            light
        } );
    }
    catch( IOException e) {
        e.printStackTrace();
    }
}

void draw() {
    background(backgroundNow);

```

```

}

// ボタンが押された時の処理です。
// ※ControlP5 ではボタンのラベルがそのまま関数名になります。
public void SWITCH_ON(int theValue){
    try {
        light.set().reqSetOperationStatus(new byte[] {0x30}).send();
    }
    catch(IOException e){
        e.printStackTrace();
    }
}
public void SWITCH_OFF(int theValue){
    try {
        light.set().reqSetOperationStatus(new byte[] {0x31}).send();
    }
    catch(IOException e){
        e.printStackTrace();
    }
}
}

```

このコードを実行した上で、他のマシン上で他のノードの情報を得るようなプログラム(第一章で作成したようなもの)を実行すると、ここで作成した照明オブジェクトが見つかるはずです。

本章で用いた機器オブジェクトの EOI と、アクセスしたプロパティの ID(EPC)を記します。

```

一般照明(GeneralLighting)の EOI : [02.90]
  動作状態(OperationStatus)の EPC : 80
  設置場所(InstallationLocation)の EPC : 81
  異常発生状態(FaultStatus)の EPC : 88
  メーカーコード(ManufacturerCode)の EPC : 8A

```

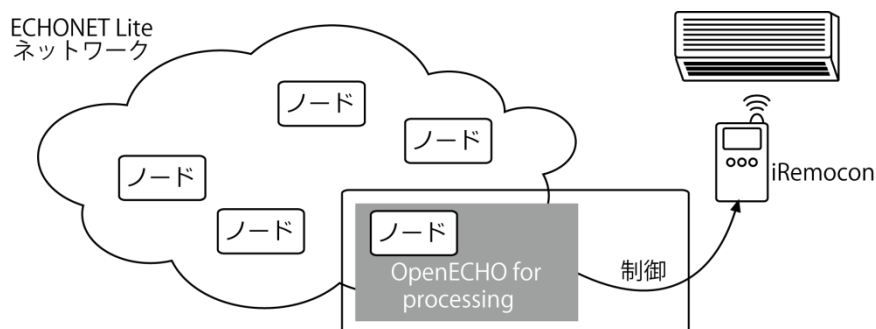
第五章 赤外線を利用した実機器オブジェクト作成

次に、エミュレーションではなく、実際の機器を動かしてみしましょう。しかし、実際の機器を動かすといっても、Processing から機器を制御する方法はハードウェア的な実装によって全く異なるでしょう。例えば自分が持っている ECHONET Lite 非対応エアコンを、OpenECHO for Processing を用いて ECHONET Lite 対応エアコンにすることはできないのでしょうか？

対応するサンプルプログラム： Tutorial5a_iRemoconLight
Tutorial5b_iRemoconAircon

iRemocon：ネットワークから制御可能な学習リモコン

ここでは、機器を制御する方法として、グラモ社の iRemocon というネットワーク対応の赤外線学習リモコンを使用します。本項で説明する方法は、iRemocon 向けの部分をシリアル通信や ZigBee 通信などに書き換えることで、様々な既存のネットワーク家電を ECHONET Lite に対応させるために応用することができます。



※神奈川工科大学、株式会社ソニーコンピュータサイエンス研究所は iRemocon およびグラモ社とは何の関係もなく、公開情報を用いてサンプルを作成しているに過ぎません。不具合があっても我々やグラモ社などには問い合わせないでください。自己責任でお願いします。

赤外線学習リモコンとは、既存の家電についている赤外線リモコンの信号を覚えこませると、それをそのまま出力することでリモコンの肩代りをさせることができるデバ

イスです。いくつものリモコンを一つのデバイスに覚えさせておくことで、リモコンの数が増えるのを抑えることができます。

通常の学習リモコンは、ユーザーが指でボタンを押すことで覚えた信号を出力することになりますが、iRemocon はインターネットからの指令を受けとって動作するというのが大きな特徴になっており、スマートフォンなどから遠隔利用できるということで人気の機器となっています。

iRemocon のホームページ <http://i-remocon.com/>

iRemocon を制御するにはソケット通信を行い、コマンドとその結果を送受信することで行います。iRemocon の制御に関する情報は、下記のリンク先から取得できます。

<http://i-remocon.com/development/>

赤外線リモコンでの制御は単方向の通信(送りっぱなしの通信)となるため、機器の電源の ON,OFF をすることは可能ですが、現在の機器の状況を取得することなどはできません。しかし、ECHONET Lite では、現在の状態を取得し、返答しなければならないことがあります。そこで今回は、前回送ったコマンドを覚えておき、それを返答することにします。プログラムからのみ制御される場合はこれでほとんど問題はないと考えられますが、同時に他の赤外線リモコン(例えば元々家電機器についているもの)からも操作されてしまうような場合はプログラムの中に保存している状態と現実の状態の間に食い違いが発生してしまうことはご了承ください。

ではまず、iRemocon を Processing からどのように制御するかについて説明します。

まず、iRemocon の IP アドレスを調べておきます。これについては、Android や iPhone の公式アプリを使用してください。

今回使用する iRemocon のコマンドは以下の 2 つです。

- *ic* (リモコン学習開始)
- *is* (赤外線発光)

これらのコマンドは、iRemocon へのソケットを開き、以下の文字列を書き込むことで実行されます。ポート番号は、51013 です。

- **ic;XX;\r\n*
- **is;XX;\r\n*

XX は、1 から 1500 までの整数で、iRemocon 内部に保存される赤外線パターンの ID を表しています。例えば、319 番に保存する場合には

```
*ic;319;¥r¥n
```

を送信し、iRemocon に赤外線を当てて学習させた後、

```
*is;319;¥r¥n
```

を送信することで、機器を制御できます。

また、Processing からソケット通信を行うためには、Processing 標準ライブラリの Client クラスを使用します。すると、上記コマンドは以下のような関数にすることができます。iRemoconIP という変数は iRemocon の公式 Android アプリなどで調べておいたアドレスにしてください。

```
final String iRemoconIP = "192.168.126.101";
final int iRemoconPort = 51013;
void iRemoconLearn(int id){
    Client c = new Client(this,iRemoconIP,iRemoconPort);
    c.write("*ic;" + id + "\r\n");
    c.stop();
}

void iRemoconSend(int id){
    Client c = new Client(this,iRemoconIP,iRemoconPort);
    c.write("*is;" + id + "\r\n");
    c.stop();
}
```

※上記のコードは、iRemocon が存在しない環境で用いると、ソケットの接続先が存在しないためエラーになります。

では、これらを使って、iRemocon を使った機器クラスを作成します。

まずは、iRemocon で今回使用する赤外線パターンの ID を定数で宣言しておきましょう。この ID は、自分で他の目的に使っていないものなら何でも良いのですが、わかりやすいように連続した番号にしておきます。

```
final int iBase = 100;
final int SWITCH_ON = 0 + iBase;
```

```
final int SWITCH_OFF = 1 + iBase;
```

では、iRemocon を使えるようになったので、LightEmulator を作った時のように、GeneralLighting を継承した iRemoconLight を作成してみましょう。

```
// 照明クラスを実装します。
public class iRemoconLight extends GeneralLighting {
    byte[] mStatus = {0x31}; // 初期の電源状態は OFF だと仮定します。
    byte[] mLocation = {0x00};
    byte[] mFaultStatus = {0x42};
    byte[] mManufacturerCode = {0,0,0};

    protected boolean setOperationStatus(byte[] edt) {
        iRemoconSend( edt[0] == 0x30 ? SWITCH_ON : SWITCH_OFF );
        mStatus[0] = edt[0];
        // 電源状態が変化したことを他のノードに通知します
        try {
            inform().reqInformOperationStatus().send();
        } catch (IOException e) { e.printStackTrace(); }
        return true;
    }
    // 現在の電源状態を問われたら、前回送ったコマンドをそのまま返します。
    protected byte[] getOperationStatus() { return mStatus; }
    protected boolean setInstallationLocation(byte[] edt) {
        mLocation[0] = edt[0];
        try {
            inform().reqInformInstallationLocation().send();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return true;
    }
    protected byte[] getInstallationLocation() { return mLocation; }
    protected byte[] getFaultStatus() { return mFaultStatus; }
    protected byte[] getManufacturerCode() { return mManufacturerCode; }
}
```

LightEmulator の場合とほとんど同じですが、setOperationStatus のところのみが、iRemocon から実際に赤外線を送るように変更されています。

このクラスを使用したプログラムが以下になります。少々長いのですが、全て掲載します。

```
import java.io.IOException;
import processing.net.*;
import controlP5.*;
```

```

import com.sonycsl.echo.Echo;
import com.sonycsl.echo.node.EchoNode;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.DeviceObject;

import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.GeneralLighting;

// iRemoconIP は iRemocon アプリで検索された IP アドレスに書き換えてください。
final String iRemoconIP = "192.168.126.101";
final int iRemoconPort = 51013;

// このアプリで用いるコマンドの番号を扱いやすいように定数にしておきます。
// ベースの番号を適当に定義しました。本来はすでに定義されている番号と
// 合わない番号にしないといけません。
final int iBase = 100;
final int SWITCH_ON = 0 + iBase;
final int SWITCH_OFF = 1 + iBase;
// ボタンを作成する時の文字列です。
String[] btnStrs = {"SWITCH_ON", "SWITCH_OFF"};

void iRemoconLearn(int id){
    println( "iRemoconLearn : "+btnStrs[id-SWITCH_ON] );
    Client c = new Client(this,iRemoconIP,iRemoconPort);
    c.write("*ic;" + id + "\r\n");
    c.stop();
}

void iRemoconSend(int id){
    println( "iRemoconSend : "+btnStrs[id-SWITCH_ON] );
    Client c = new Client(this,iRemoconIP,iRemoconPort);
    c.write("*is;" + id + "\r\n");
    c.stop();
}

// 照明クラスを実装します。
public class iRemoconLight extends GeneralLighting {
    byte[] mStatus = {0x31}; // 初期の電源状態は OFF だと仮定します。
    byte[] mLocation = {0x00};
    byte[] mFaultStatus = {0x42};
    byte[] mManufacturerCode = {0,0,0};

    protected boolean setOperationStatus(byte[] edt) {
        iRemoconSend( edt[0] == 0x30 ? SWITCH_ON : SWITCH_OFF );
        mStatus[0] = edt[0];
        //電源状態が変化したことを他のノードに通知します
        try { inform().reqInformOperationStatus().send(); } catch (IOException e)
            { e.printStackTrace(); }
        return true;
    }
}

// 現在の電源状態を問われたら、前回送ったコマンドをそのまま返します。

```

```

protected byte[] getOperationStatus() { return mStatus; }
protected boolean setInstallationLocation(byte[] edt) {
    mLocation[0] = edt[0];
    try {
        inform().reqInformInstallationLocation().send();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    return true;
}
protected byte[] getInstallationLocation() {return mLocation;}
protected byte[] getFaultStatus() { return mFaultStatus;}
protected byte[] getManufacturerCode() {return mManufacturerCode;}
}

```

```

ControlP5 cp5 ;
iRemoconLight light ;

```

```

void setup(){
    size(210,(btnStrs.length)*30);
    frameRate(30);

    // 次に、学習と再生のユーザーインターフェースを作成します。
    cp5 = new ControlP5(this);
    // 送信用のボタンを左に、学習用のボタンを右に表示します。
    for( int bi=0;bi<btnStrs.length;++bi ){
        cp5.addButton(btnStrs[bi],0,0,(bi)*30,100,25);
        cp5.addButton("LEARN_"+btnStrs[bi],0,110,(bi)*30,100,25);
    }

    // System.out にログを表示するようにします。
    // Echo.addEventListener( new Echo.Logger(System.out) );

    try {
        light = new iRemoconLight();
        Echo.start( new DefaultNodeProfile(),new DeviceObject[] {light});
    } catch( IOException e){ e.printStackTrace(); }
}

// ボタンを描画するためだけに定義しています。
void draw(){}

// ボタンが押された時の処理です。
// 送信用の関数と、学習用の関数を交互に定義しています。
// ※ControlP5 ではボタンのラベルがそのまま関数名になります。
// light.setOperationStatus(new byte[] {0x30});としてはいけません！！
public void SWITCH_ON(int theValue){
    try{
        light.set().reqSetOperationStatus(new byte[] {0x30}).send();
    }catch(IOException e){e.printStackTrace();}
}
public void LEARN_SWITCH_ON(int theValue){

```

```

    iRemoconLearn(SWITCH_ON);
}
public void SWITCH_OFF(int theValue){
    try{
        light.set().reqSetOperationStatus(new byte[]{0x31}).send();
    }catch(IOException e){e.printStackTrace();}
}
public void LEARN_SWITCH_OFF(int theValue){
    iRemoconLearn(SWITCH_OFF);
}
}

```

このプログラムでは、赤外線パターンを学習させるためのボタンも配置しました。Processing で表示されるボタンを押すと、iRemocon 上のランプが点灯するので、iRemocon に手持ちの赤外線リモコンを向けてそのボタンを押し、学習を行ってください。その後は、操作用のボタン、他のノードからのリクエストに応じて動作します。

本章で実装したのは *GeneralLighting* ですが、例えばエアコンの場合も、実装しなければならないメソッドの量は増えますが基本的には同様です。このサンプルについては Tutorial5b_iRemoconAircon をご参照ください。

本章で用いた機器オブジェクトの EOI と、アクセスしたプロパティの ID(EPC)は前章と同じです。

```

一般照明(GeneralLighting)の EOI   :[02.90]
  動作状態(OperationStatus)の EPC : 80
  設置場所(InstallationLocation)の EPC :81
  異常発生状態(FaultStatus)の EPC :88
  メーカーコード(ManufacturerCode)の EPC :8A

```

第六章 ノードを正しく作成する

本章では、今までデフォルトのまま使ってきたノードプロファイルおよび機器オブジェクトの詳細実装方法について説明します。これらのデフォルトのクラスは、あくまで本チュートリアルをわかりやすくする目的に限ってそれらを仮実装したものですので、他のノードからの様々なリクエストに対して正しく返答できる保証はありません。OpenECHO を用いたプログラムを公開する時などには、本章で説明する内容をくまなく実装して戴けるようお願いいたします。

対応するサンプルプログラム：Tutorial6a_ImplementRealNode
Tutorial6b_ElectricLock

ノードプロファイル

ノードプロファイルは、前章までで説明したように、ネットワークに自分が所属しているノードの情報を伝える役割を持っています。従って、*NodeProfile* クラスを継承し、自分が作成しているノードの情報に応じて独自の実装を行う必要があります。

NodeProfile では以下のメソッドが必須となっています。

- *getManufacturerCode*
ECHONET コンソーシアムから与えられる製造者コード。メーカーコードとも言う。3 バイトの値です。
- *getOperatingStatus*
電源が ON か OFF かを表します。ON は 0x30、OFF は 0x31 です。
- *getIdentificationNumber*
オブジェクトをドメイン内で一意に識別するための 17 バイトの識別番号です。
この値のフォーマットについては ECHONET Lite 規格書を参照してください。
- *setUniquelIdentifierData*
個体識別番号と呼ばれる 2 バイトの値です。変更可能なので *set* があります。
この値のフォーマットについても ECHONET Lite 規格書を参照してください。
- *getUniquelIdentifierData*
個体識別番号の取得リクエストを受けとった時に返答するためのメソッドです。

実装例を示します。基本的に既定値を *return* するだけです。

```

public class MyNodeProfile extends NodeProfile {
    byte[] mManufactureCode = {0, 0, 0}; // Given by ECHONET Consortium
    byte[] mStatus = {0x30}; // 0x30:ON 0x31:OFF
    byte[] mIdNumber = {(byte)0xFE, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    byte[] mUniqueId = {0, 0};

    protected byte[] getManufacturerCode() {return mManufactureCode;}
    protected byte[] getOperatingStatus() { return mStatus; }
    protected byte[] getIdentificationNumber() {return mIdNumber;}
    protected boolean setUniqueIdentifierData(byte[] edt) {
        if((edt[0] & 0x40) != 0x40) return false;
        mUniqueId[0] = (byte)((edt[0] & (byte)0x7F) | (mUniqueId[0] & 0x80));
        mUniqueId[1] = edt[1];
        return true;
    }
    protected byte[] getUniqueIdentifierData() {return mUniqueId;}
}

```

必須でないメソッドのオーバーロードについて

第四章では、*GeneralLighting* を例にして独自の機器オブジェクトを定義しました。その際には、必須となっているメソッドのみしかオーバーロードしませんでした。ここでは、オプションとなるメソッドの実装方法を説明します。この方法は、ノードプロファイルと機器オブジェクトの両方で共通しています。

プロパティ

プロパティとは、Java で言えばクラスのメンバ変数のようなもので、機器がサポートしている機能を表現しています。例えば、*OperationStatus* 等のことを言います。OpenECHO では、それらを変数として表現するわけではなく、それらに対するいわゆるアクセサ、つまり *get* や *set* といったメソッドを用いて読みこんだり書きこんだりします。

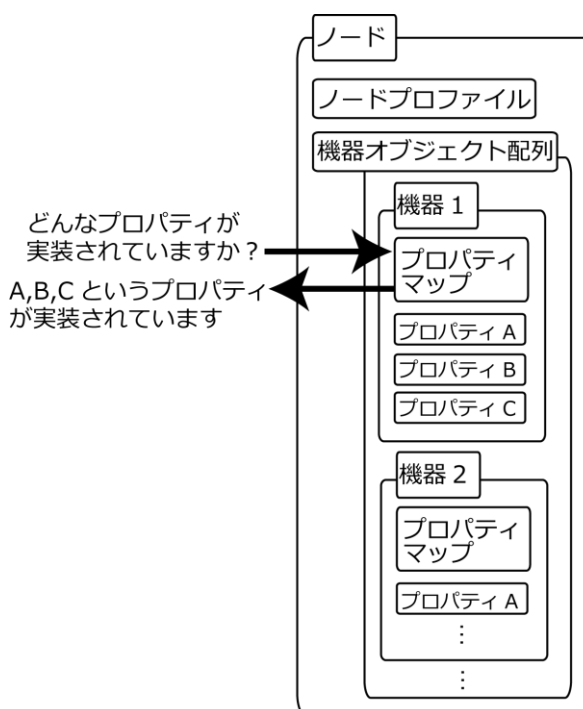
それらのメソッドには、必須であり先祖クラスで *abstract* 宣言されているものと、必須ではなく実装するかどうか任意に決められるものが存在することは第四章で説明した通りです。どのような機能が必須であるか、オプションであるかは、OpenECHO リファレンス、および ECHONET Lite 仕様書をご参照ください。

プロパティマップ

プロパティマップとは、機器オブジェクトのプロパティのうち、どれが実装されているかを示す表です。必須なものは、常に実装されているはずなので敢えて明示的に設定する必要はありませんが、必須でないプロパティは定義されていることを他のノードに伝えるためにこれを設定する必要があります。

ここでは、プロパティマップの使用例として、電子錠 (ElectricLock) クラスを実装します。ElectricLock クラスには、中に人がいるかどうかを示すプロパティである `OccupantNonOccupantStatus` が存在します。このプロパティの実装は必須ではなく、また `get` のみしか存在しません。

以下の例では、何らかの方法で室内に人がいるかどうかを知ることができる装置を持っているとします。それは手でボタンを押す自作機器でも構いませんし、前章で使ったようなエミュレータでも、センサーを使った本格的な物でも構いません。



では、今までと同様に、`ElectricLock` クラスを継承したクラスを作しましょう。その時に、`getOccupantNonOccupantStatus` メソッドをオーバーライドします。また、他の必須なメソッドも実装しておきます。

```
public class MyElectricLock extends ElectricLock {
    byte[] mStatus = {0x30};
    byte[] mLocation = {0x00};
    byte[] mFaultStatus = {0x42};
    byte[] mManufacturerCode = {0,0,0};

    // 今鍵が閉まっているかどうかを保持する変数。
    byte[] mLockStatus = {0x30};
    // 誰かが在室しているかどうかを保持する変数。
    byte[] mOccupantStatus = {0x42};
    // 以上 2 つの変数は、本来はここに変数として持っておくよりも実際の
```

```

// 機器の状態を読み書きする方が望ましいと考えられますが、サンプルを
// 動作させる都合上メソッドを用意しています。

// 電源が入っているかどうかを返す
protected byte[] getOperationStatus() { return mStatus; }

// 機器が設置されている場所を設定する
protected boolean setInstallationLocation(byte[] edt) {
    if(mLocation[0] == edt[0]) return true;
    mLocation[0] = edt[0];
    try {
        inform().reqInformInstallationLocation().send();
    } catch (IOException e) { e.printStackTrace(); }
    return true;
}
// 機器が設置されている場所を返答する
protected byte[] getInstallationLocation() {return mLocation;}
protected byte[] getFaultStatus() { return mFaultStatus;}
protected byte[] getManufacturerCode() {return mManufacturerCode;}
//ここで setLockSetting1 となっている理由は、この機器オブジェクトでは
//複数の鍵を制御できるようになっているためです。
protected boolean setLockSetting1(byte[] edt) {
    mLockStatus = edt;
    return true;
}
protected byte[] getLockSetting1() {
    return mLockStatus;
}
// 在室・不在状態を得るメソッド(必須ではないメソッド)をオーバーライド
// します。
// 本来はこの関数の中でセンサーにアクセスして実際の在室状態を調べ、
// その結果を返答するように実装する必要があります。
protected byte[] getOccupantNonOccupantStatus() {return mOccupantStatus;}
}

```

必須でないメソッドを実装したので、このクラスの中で `setPropertyMaps` メソッドをオーバーライドしてそのメソッドを教えてやる必要があります。これがプロパティマップの設定です。コードは以下のようになります。

```

protected void setupPropertyMaps() {
    super.setupPropertyMaps();
    addGetProperty(EPC_OCCUPANT_NON_OCCUPANT_STATUS);
}

```

このように、まず親クラスのメソッドを呼び出す必要があります。この中で、必須プロパティの情報を作成しています。次に `addGetProperty` で実装したオプションメソッドを指定します。この引数はプロパティごとに割り振られた ID(EPC と呼ばれていま

す)で、各機器クラスの中に接頭詞 *EPC_* を持つ *final* な定数として定義されています。
get メソッドを実装した時には、*addGetProperty* を呼び出し、その引数に実装したプロパティの ID を与えます。set メソッドを実装した場合は、*addSetProperty* を用います。

完成したクラスを使ったプログラムが以下になります。

```
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.EchoProperty;
import com.sonycsl.echo.eoj.EchoObject;
import com.sonycsl.echo.eoj.device.DeviceObject;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.ElectricLock;
import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;

public class MyElectricLock extends ElectricLock {
    byte[] mStatus = {0x30};
    byte[] mLocation = {0x00};
    byte[] mFaultStatus = {0x42};
    byte[] mManufacturerCode = {0,0,0};
    byte[] mLockStatus = {0x30};
    byte[] mOccupantStatus = {0x42};

    // setupPropertyMaps メソッド内で add(Get|Set)Property を呼び出すことにより
    // 必須でないが実装したプロパティを登録します。
    // 必須のプロパティは、super.setupPropertyMaps()で登録されます。
    protected void setupPropertyMaps() {
        super.setupPropertyMaps();
        // 状態変化時アナウンスプロパティマップに登録する際は
        // addStatusChangeAnnouncementProperty,
        // Set プロパティマップに登録する際は addSetProperty,
        // Get プロパティマップに登録する際は addGetProperty を使います。
        addGetProperty(EPC_OCCUPANT_NON_OCCUPANT_STATUS);
    }

    protected byte[] getOperationStatus() { return mStatus; }
    protected boolean setInstallationLocation(byte[] edt) {
        changeInstallationLocation(edt[0]);
        return true;
    }
    protected byte[] getInstallationLocation() {return mLocation;}
    public void changeInstallationLocation(byte location) {
        if(mLocation[0] == location) return ;
        mLocation[0] = location;
        try {
            inform().reqInformInstallationLocation().send();
        } catch (IOException e) { e.printStackTrace(); }
    }
    protected byte[] getFaultStatus() { return mFaultStatus;}
    protected byte[] getManufacturerCode() {return mManufacturerCode;}
    protected boolean setLockSetting1(byte[] edt) {
```

```

        mLockStatus = edt;
        return true;
    }
    protected byte[] getLockSetting1() {
        return mLockStatus;
    }

    // 在室・不在状態を得るメソッドをオーバーライドします。
    // 実際に在室かどうかを確かめて返答するコードに置き換えてください。
    protected byte[] getOccupantNonOccupantStatus() {return mOccupantStatus;}

}

void setup() {
    // System.out にログを表示するようにします。
    Echo.addEventListener( new Echo.Logger(System.out) );
    try {
        Echo.start( new DefaultNodeProfile()
                    , new DeviceObject[] { new MyElectricLock() } );
        NodeProfile.getG().reqGetSelfNodeInstanceListS().send();
    }
    catch( IOException e ) {
        e.printStackTrace();
    }
    println("Started");
}

```

このプログラムを実行しても特に何も起こりませんが、コントローラのノードなどから電子錠として発見され、かつ人が在室しているかを問い合わせられた時に不可応答ではなく適切な返答を返すことができます。

本章で用いた機器オブジェクトの EOJ と、アクセスしたプロパティの ID(EPC)を記します。

電気錠(ElectricLock)の EOJ : [02.6F]

動作状態(OperationStatus)の EPC : 80

設置場所(InstallationLocation)の EPC : 81

異常発生状態(FaultStatus)の EPC : 88

メーカーコード(ManufacturerCode)の EPC : 8A

施錠設定 1(LockSetting1)の EPC : E0

在室・不在状態(OccupantNonOccupantStatus)の EPC : E4

第七章 すべての機能を使う

このチュートリルのまとめとして、電子錠をかけると、住人が在室かどうかを取得し、不在であれば全ての照明をオフにする、というプログラムを紹介します。

対応するサンプルプログラム：Tutorial7_AllLightsOff

行うべき処理の概要

このプログラムの処理の概要は以下になります。

1. まず、電子錠が見つかった時にいろいろと設定を行いたいので、`EventListener` を登録してその中で `onNewElectricLock` メソッドをオーバーライドします。これについては二章で解説しました。
2. 電子錠が見つかったら、その `Receiver` を設定します。`Receiver` についても第二章で解説しました。これを設定しておく、と、電子錠の状態が変化した時に自動的に行う処理を記述することができます。具体的には、施錠や解錠がされた時や、施錠状態のリクエストに対して応答があった場合のハンドラー (`onGetLockSetting1`)を追加し、その中で在室状態を確認する処理を行います。

この在室状態リクエストに対する返答も `onGetOccupantNonOccupantStatus` という `Receiver` のハンドラーで受けとります。もし不在ならば `GeneralLighting.setG().reqSetOperationStatus` を用いて照明を消します。この呼び出し方法については、第三章で解説しました。

これらの `Receiver` を設定しおわったら、電子錠の施錠状態の初期状態の問い合わせを行います。これも第三章で解説しました。ECHONET Lite の規格書によれば、`ElectricLock` の施錠状態は、状態が変更された時に「状態時アナウンス」を送らなければならないので、この設定が終われば状態が変更になれば自動的に `Receiver` が呼ばれることになります。状態時アナウンスについては第四章で解説しました。

3. 全てのノードに対し、機器のリスト一覧を要求します。これは第一章からの皆勤賞です。

```
NodeProfile.getG().reqGetSelfNodeInstanceListS().send()
```

これにより、新たに施錠状態が確認された場合、および人が鍵をかけたときに、在室状態を確認、不在ならば照明を消すというプログラムを作成することができます。

ソースコード

ソースコードは以下の通りです。このプログラムはネットワーク内のどこかに電子錠を含むノードが存在するという仮定で書かれています。ECHONET Lite 対応電子錠をお持ちでない方は、第六章で作成した、*OccupantNonOccupantStatus* プロパティを持つ電子錠のプログラムを自分で作成した上で他のマシンで実行しておいてください。

```
import com.sonyosl.echo.Echo;
import com.sonyosl.echo.EchoProperty;
import com.sonyosl.echo.eoj.EchoObject;
import com.sonyosl.echo.eoj.device.DeviceObject;
import com.sonyosl.echo.eoj.profile.NodeProfile;
import com.sonyosl.echo.eoj.device.housingfacilities.ElectricLock;
import com.sonyosl.echo.eoj.device.housingfacilities.GeneralLighting;
import com.sonyosl.echo.processing.defaults.DefaultNodeProfile;
import com.sonyosl.echo.processing.defaults.DefaultController;

void setup(){
    // System.out にログを表示するようにします。
    Echo.addEventListener( new Echo.Logger(System.out) );

    Echo.addEventListener(new Echo.EventListener() {
        public void onNewElectricLock (ElectricLock device){
            println( "ElectricLock sensor found." );
            device.setReceiver( new ElectricLock.Receiver(){
                protected void onGetLockSetting1 (EchoObject eoj, short tid, byte esv
                                                , EchoProperty property, boolean success) {
                    super.onGetLockSetting1(eoj, tid, esv, property, success);
                    if( !success ){ println( "error in call reqGetLockSetting1" ); return ; }
                    if(property.edt[0]==0x42) { println("unlock"); return ; }

                    // 主電気錠が施錠されると在室状態を得るために電文を送ります。
                    // 但し、在室状態のプロパティは ECHONET Lite 規格上必須項目では
                    // ないため確実に取得できるとは限りません。
                    try {
                        ((ElectricLock)ej).get().reqGetOccupantNonOccupantStatus().send();
                    } catch(IOException e){e.printStackTrace();}
                }
            });
            protected void onGetOccupantNonOccupantStatus (EchoObject eoj, short tid
                                                , byte esv, EchoProperty property, boolean success) {
                super.onGetOccupantNonOccupantStatus(eoj, tid, esv, property, success);
                if( !success ){ println( "error in call reqGetOccupantNonOccupantStatus" ); return ; }
                if(property.edt[0]==0x41) { println("occupant"); return ; }
            }
        }
    });
}
```

```

        // 不在状態であるならば全ての照明をオフにする電文をマルチキャストで
        // 送ります.
        try {
            GeneralLighting.setG().reqSetOperationStatus(new byte[]{0x31}).send();
        } catch (IOException e) { e.printStackTrace(); }
    }
});

// 施錠・解錠は状態が変化すると通知されます.
// なので最初だけ get のリクエストを送信します.
try {
    device.get().reqGetLockSetting1().send();
} catch (IOException e) { e.printStackTrace(); }
}
});
try {
    Echo.start( new DefaultNodeProfile(), new DeviceObject[]{new DefaultController()});
    NodeProfile.getG().reqGetSelfNodeInstanceListS().send();
} catch (IOException e) { e.printStackTrace(); }
println("Started");
}

```

おわりに

以上でこのチュートリアルは終わりです。ECHONET Lite はまだ生まれて間もないプロトコルで今後仕様が変更されていく可能性もあり、また、2013 年初頭時点では対応機器も多いとは言えません。しかし、これほど多くの生活家電やセンサーなどを規格化したオープンなプロトコルはこれまでになく、各社の参入表明も相次いでおり、大変将来性があると考えられます。現時点で家電ネットワークを利用するサービスを開発したいと思ったら、通信レイヤーに ECHONET Lite を用いておくことで、今後対応機器が増えてきた時にシームレスにサービスを移行することが可能になるでしょう。OpenECHO for Processing が ECHONET Lite の普及の一助となることを願ってやみません。

OpenECHO もまだ開発途中のソフトウェアであり、今後様々な改良や仕様変更がなされていく可能性があります。最新版は常に GitHub に存在していますので、こちらも適宜ご確認戴ければ幸いです。OpenECHO for Processing や本チュートリアルも今後は OpenECHO のディストリビューション内に含まれ、OpenECHO 本体と並行してメンテナンスされていく予定です。

OpenECHO の配布元 <https://github.com/SonyCSL/OpenECHO/>

OpenECHO / OpenECHO for Processing / 本チュートリアルの中にバグや改善が望まれる点、ご不明な点などございましたら、ご遠慮なくソニーコンピュータサイエンス研究所までお問い合わせください。(info@kadecot.net)

※本文書内に記載されている会社名および商品名は、各社の商標または登録商標です。