



発展プログラミング演習II

12. ネットワーク

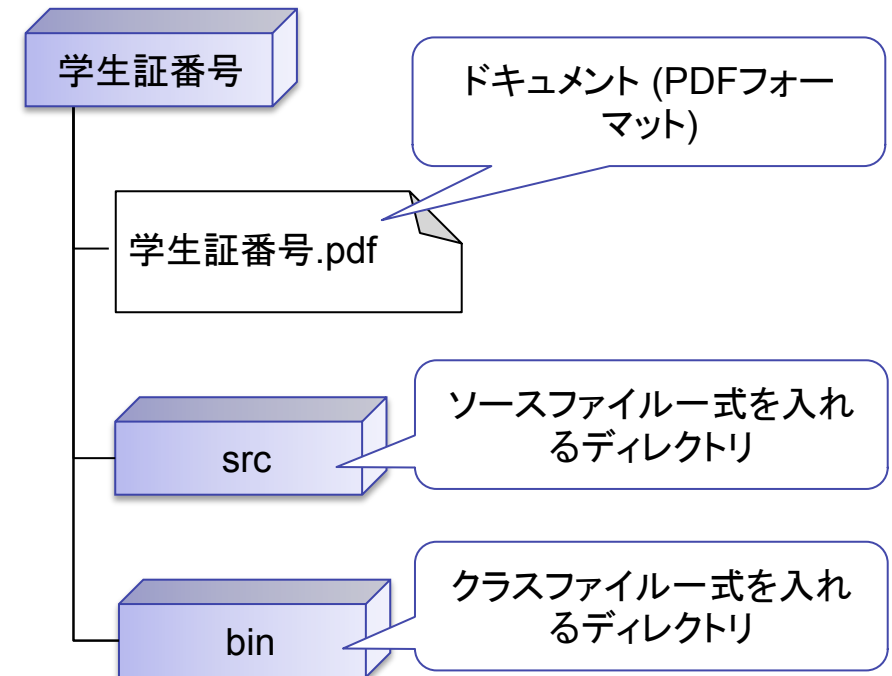
玉田 春昭 水口 充

概要

- チャットサーバを作成せよ.
 - プロトコルは自由. 自分たちで定義しても良いし, 既存のプロトコルを利用しても良い.
 - クライアントは不要.
 - telnetコマンドをクライアントとすること.
 - マルチクライアントを達成すること.

提出方法

- 6桁の学生証番号のディレクトリを用意し、そこに次のファイルを入れる.
 - srcディレクトリ
 - ソースファイル一式.
 - binディレクトリ
 - クラスファイル一式.
- これら全てをzip圧縮したファイル (学生証番号.zip) をMoodleに提出する.
 - 展開したとき、学生証番号のフォルダができるようにする.



注意点

- コピーしないこと.
 - 他人のソースコードのコピーはもちろん, Webページからの無断転載は禁止.
 - Webページのプログラムを参考にした場合は, 自分が作成した部分を明確にすること.
- クラス名を適切に変更すること.
 - EchoXxxxを変更して作成するのは可.
 - その場合であっても, チャットであるのに, クラス名がEchoXxxであるのは名前と実装が合っていないのでダメ.

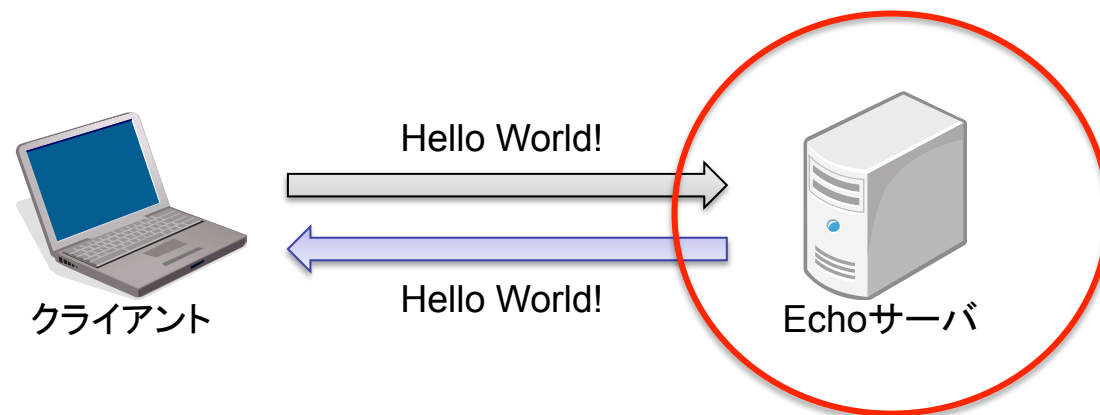



ネットワークプログラミング




目標: Echoサーバを作成する.

- 送られてきたデータをそのまま送り返すサーバのこと.
 - ネットワークにつながっているかを確認することができる.
 - 一番簡単なソケットプログラミング.





Step1. サーバを立てる



```
import java.net.*;
import java.io.*;

class EchoServer{
    ServerSocket server;
    void listen(){
        try{
            server = new ServerSocket(18080);
            System.out.println("Echoサーバをポート18080で起動しました. ");
            Socket socket = server.accept();
            System.out.println("クライアントが接続してきました. ");
        } catch(IOException e){
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        EchoServer echo = new EchoServer();
        echo.listen();
    }
}
```

確認ポイント

- 例外

- 想定外の動きがあったとき、例外が投げられる.
- 例外は受け取る処理を書いておかなければならない.
 - try-catch文.
 - RuntimeExceptionとそのサブクラスはtry-catchで囲んでおかなくても良い.
- もしくは、メソッドの呼び出し元に投げると宣言しなければならない.
 - メソッド宣言に throws をつける.

```
try{  
    // tryブロック  
    // 例外が投げられる可能性ある  
    // プログラムがここに書かれる.  
} catch (SomeException e) {  
    // tryブロック内でSomeExceptionが  
    // 投げられた場合、その時点で、  
    // このブロックに処理が移る.  
    // SomeExceptionと関係のない例外  
    // が投げられたとき、  
    // このブロックは実行されない.  
}  
finally{  
    // tryブロック、catchブロックの最後に  
    // 必ず1回だけ実行されるブロック.  
    // どのような場合でも実行されるため、  
    // 後処理が書かれることが多い.  
}
```


java.net.ServerSocketクラス

- Java言語でサーバを作成するときに用いるクラス.
 - ネットワークサーバのクラス.
 - **ポート番号**を指定してサーバを立てる.
 - acceptメソッドでクライアントからの接続を待つ.
 - クライアントが接続するとメソッドから制御が返る.
 - クライアントが接続するまで待ち続ける.

```
ServerSocket server = new ServerSocket(18080);  
Socket socket = server.accept();
```

クライアントが接続してくるまでこのメソッドからオブジェクトは返ってこない(ブロックされる).
クライアントが接続すると, クライアントを表すSocketオブジェクトを返す.

Echoサーバへの接続

telnetコマンド

- コンソールからtelnetコマンドを使用する.
 - telnet <ホスト名> [ポート番号]
 - ホスト名を指定する.
 - ポート番号はオプション. 指定しなければ23が指定されたことになる. (telnetポート)
- Echoサーバを起動すれば以下のコマンドでサーバに接続できる.

```
telnet localhost 18080
```

- WWWサーバにもtelnetで接続できる.

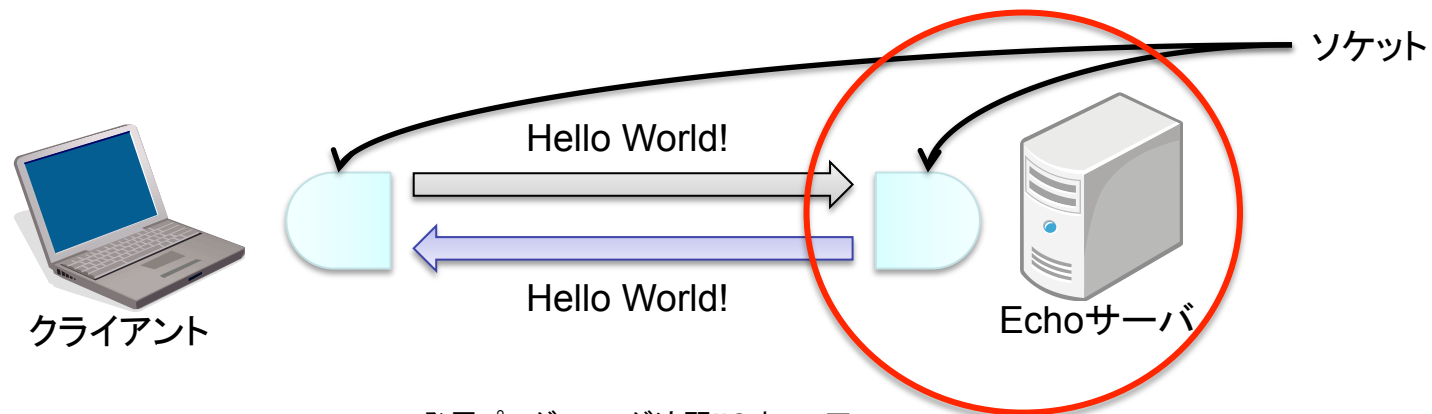
```
telnet www.google.co.jp 80  
GET / HTTP/1.0<ENTER><ENTER>
```

Step2. サーバへの接続

- `telnet`コマンドを使ってサーバに接続してみよう.
- Echoサーバを起動する.
 - EchoServerプログラムを実行する.
 - 別のターミナルを開き, 以下のコマンドを実行する.
 - `telnet localhost 18080`
 - 手元のPCでEchoサーバを起動し, 手元のPCから起動したEchoサーバに接続する.

java.net.Socketクラス

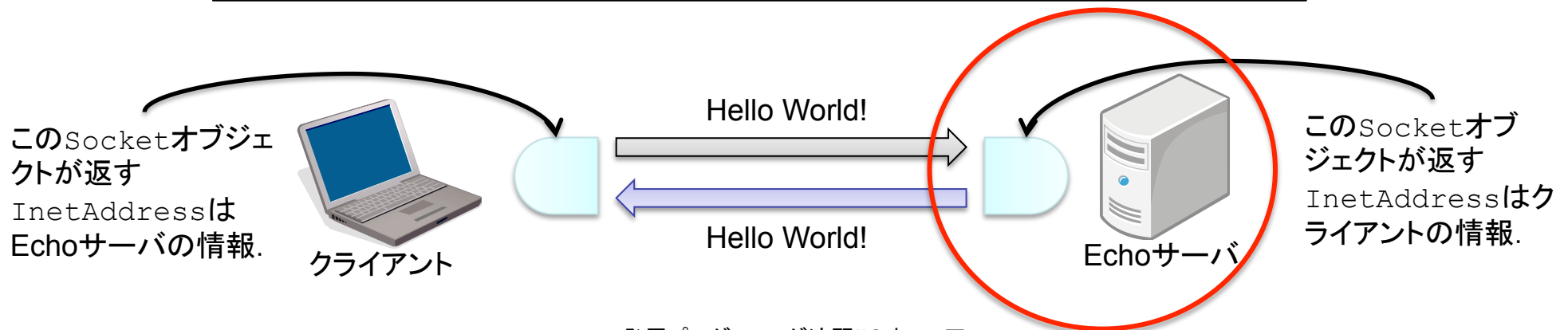
- ネットワークのソケットを表すクラス。
 - ネットワーク接続の端っこを表す。
- 以下の両方の場合で使われる。
 - クライアント側からサーバに接続する。
 - サーバ側で、クライアントが接続してきた。
- ネットワークプログラミングでは必須のクラス。



Step3. クライアント情報の取得

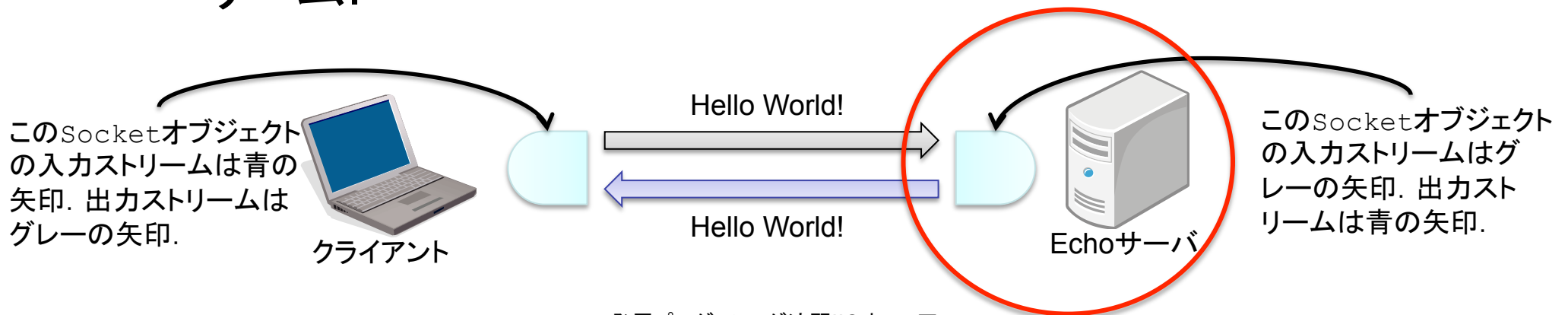
- クライアントの情報を表示しよう.
 - SocketクラスのgetInetAddressメソッドを使う.
 - getInetAddressメソッドは接続先のアドレスを表すオブジェクトであるInetAddressオブジェクトを返す.
 - 取得したInetAddressオブジェクトを出力する.

```
InetAddress address = socket.getInetAddress();  
System.out.println(address);
```



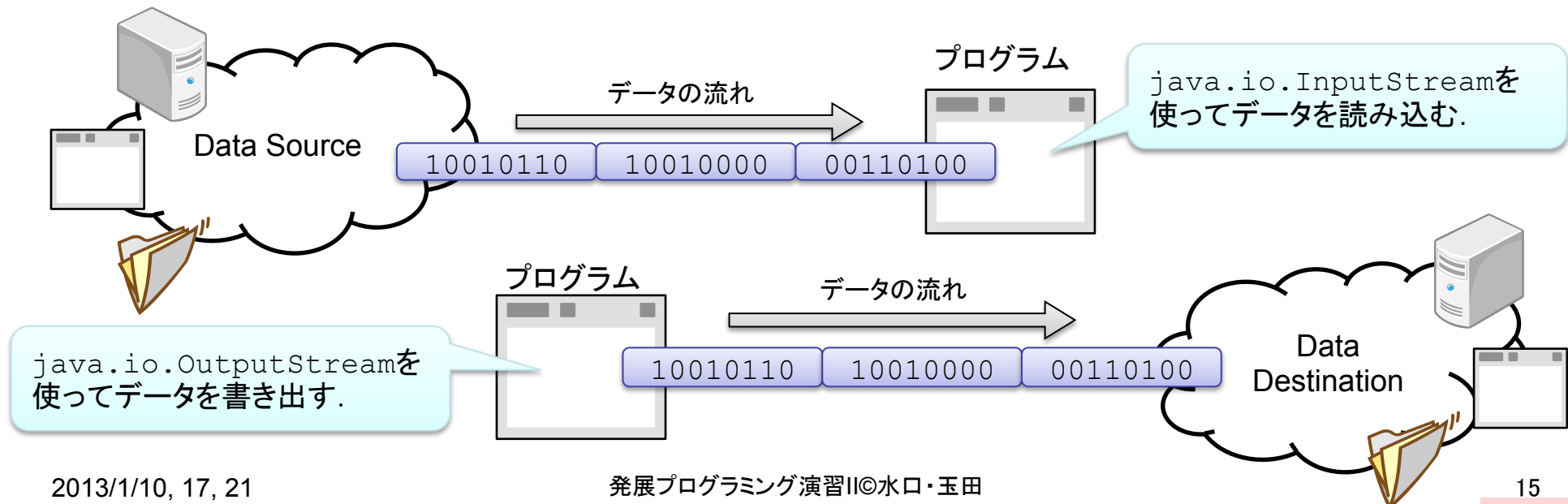
Socketからデータ読み取り

- Socketからは出力ストリーム(OutputStream), 入力ストリーム(InputStream)の両方が取得できる.
 - 入力ストリーム: もう一つの端点から送り出されたデータを読むためのストリーム.
 - 出力ストリーム: もう一つの端点へデータを送り出すストリーム.



第21, 22回目の復習 ストリーム

- データを流れとみなす考え方.
 - データは, Data SourceからData Destinationへ流れていく.
 - Data Sourceはデータを送りだすものであれば, なんでも良い.
 - Data Destinationはデータを受け取ればなんでも良い.



Step4. クライアントから送られてきたデータを出力する.

- 手順

- Socketから入力ストリームを取得する(getInputStreamメソッド)
- 取得した入力ストリームからデータを読む.
 - 空行が送られてくるまで繰り返す.
 - 空行が送られてきたら接続を切るためにループを抜ける.
- 読み込んだデータを画面に出力する.

```
Socket socket = server.accept();
InputStream socketIn = socket.getInputStream();
BufferedReader in = new BufferedReader(new InputStreamReader(socketIn));
while(true){
    String line = in.readLine();
    if(line.equals("")) break;
    System.out.println(line);
}
```




確認ポイント


- `break`の使い方.
- 入力ストリームの使い方.
 - 色々なストリームを覆って(ラップして)使える.

Step5. クライアントにデータを送り返す

- 手順

- Socketから出力ストリームを取得する(getOutputStreamメソッド).
- 取得した出力ストリームにデータを書き出す.
 - 書き出すデータはStep4で取得した(入力ストリームから読み込んだ)データ.

```
OutputStream socketOut = socket.getOutputStream();
...
BufferedWriter out = new BufferedWriter(new OutputStreamWriter(socketOut));
While(true){
    String line = in.readLine();
    ...
    out.write(line);
    out.write("\r\n");
    out.flush();
}
```



確認ポイント

- 出力ストリームの使い方.
 - 入力ストリームと似た使い方ができる.
 - データの流れる方向が異なるだけ.


Step6. 入出力ストリームを閉じる

- ストリームを開いた後, 必ず閉じなければならない.
- ソケットも開いた後, 閉じなければならない.

```
Socket socket = null;
BufferedReader in = null;
BufferedWriter out = null;
try{
    ...
} catch(IOException e){
    e.printStackTrace();
} finally{
    if(in != null){ try{ in.close(); } catch(IOException e){ } }
    if(out != null){ try{ out.close(); } catch(IOException e){ } }
    if(socket != null){ try{ socket.close(); } catch(IOException e){ } }
}
```

try節の中で宣言すると, finally節で参照できないため, tryの外で宣言する.
ローカル変数は初期値が決まらないとコンパイルできないので, nullを代入しておく.

closeメソッドもIOExceptionを投げる可能性があるので, tryで囲む.



確認ポイント

- ストリームは使ったら閉じなければならない。
 - 例外が起こった場合も、そうでない場合も閉じなければならないので、`finally`ブロックに記述する。
 - `finally`ブロックから入出力ストリームの変数を参照するため、`try`ブロックの外で変数を宣言する。



シングルクライアント

- 作成したEchoServerは1台のクライアントにし
かサービスを提供できない.
 - クライアントを1台acceptしたら、後はそのクライア
ントにかかりきりになってしまう.
- マルチクライアント
 - 一度クライアントからの接続を受けたら、そのクライ
アントの処理を行いつつ、また別のクライアントの接続を
待つ.

Step7-1 クライアントの対応を 別クラスに任せる

- クライアントに対応する部分を別のクラスに抽出する.
 - さらに処理の意味ごとにメソッドに分ける.

```
Socket socket = server.accept();

EchoClientHandler handler = null;
try{
    handler = new EchoClientHandler(socket);
    handler.open();
    String message = handler.receive();
    handler.send(message);
} catch(IOException e){
    e.printStackTrace();
} finally{
    if(handler != null){
        handler.close();
    }
}
```

Step7-2. EchoClientHandler

- 処理内容ごとにメソッドに分けて定義する.

```
import java.io.*;
import java.net.*;
class EchoClientHandler{
    Socket socket; // クライアントを表すソケット.
    BufferedReader in;
    BufferedWriter out;
```

```
    EchoClientHandler(Socket sock){
        this.socket = sock;
    }
```

```
    void open() throws IOException{ // クライアントとのデータのやり取りを行うストリームを開くメソッド.
        ...
    }
```

```
    String receive() throws IOException{ // クライアントからデータを受け取るメソッド.
    }
```

```
    void send(String message) throws IOException{ // クライアントとのデータのやり取りを行うメソッド.
        ...
    }
```

```
    void close(){ // クライアントとの接続を閉じるメソッド.
        ...
    }
```

201 }

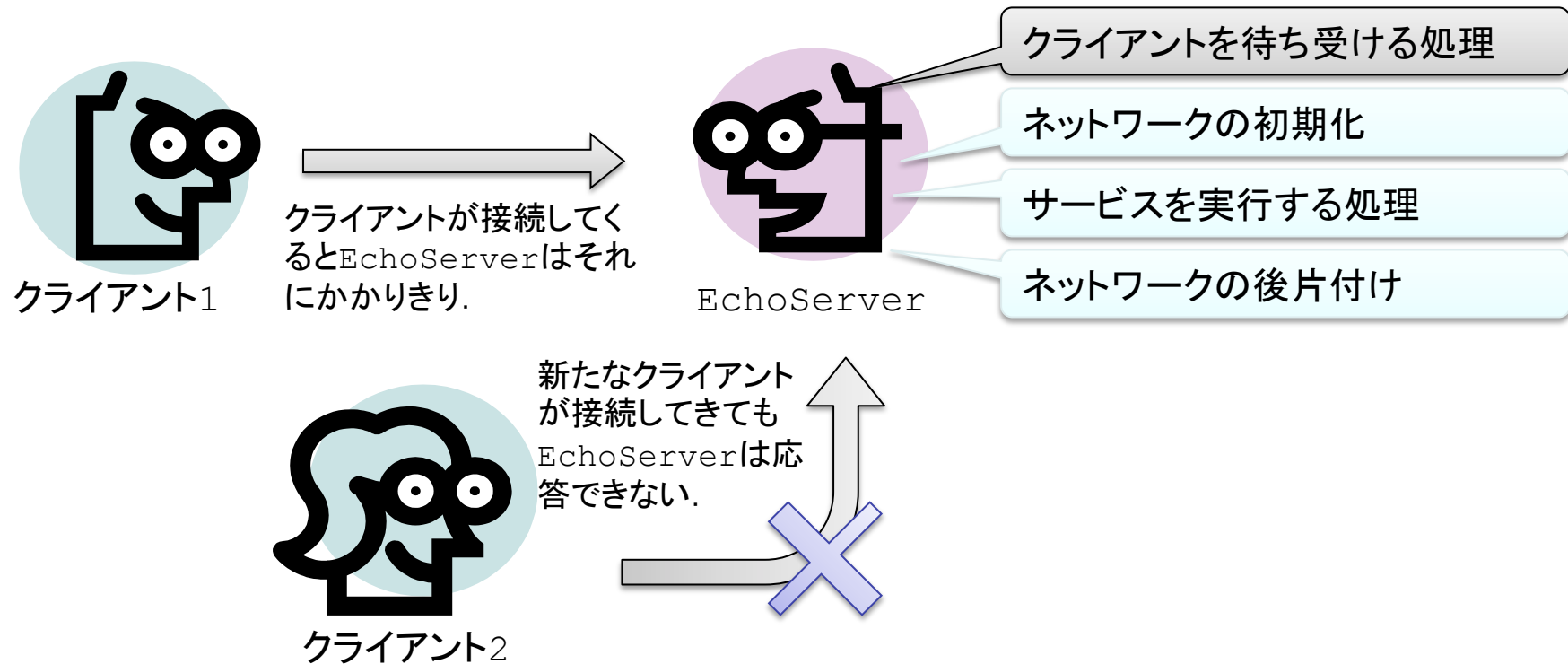
- クライアントとの接続を開く処理.
- クライアントとデータのやり取りを行う処理.
- クライアントとの接続を閉じる処理.

24

確認ポイント

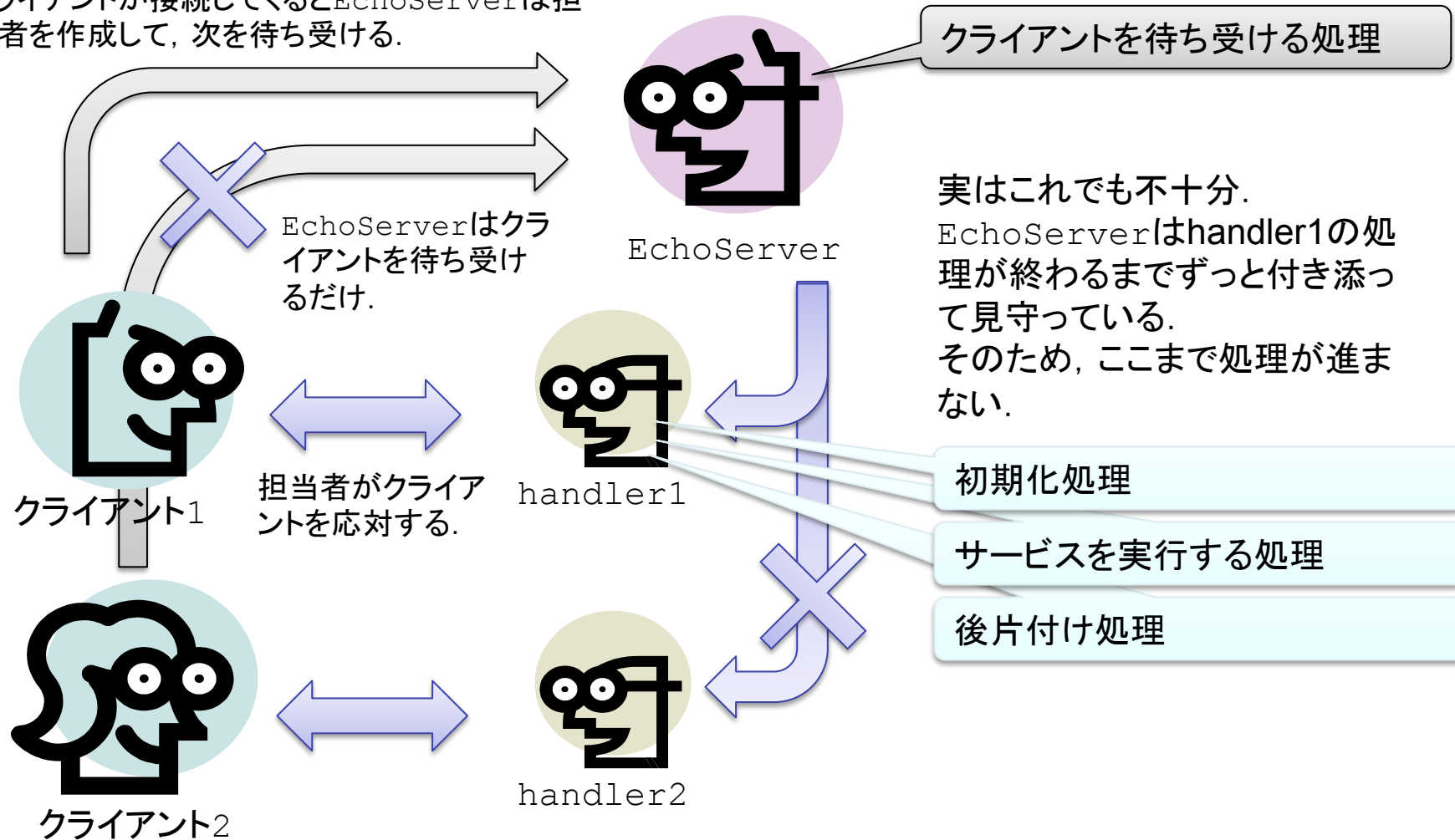
- 例外を投げる可能性のあるメソッドでは「throws 例外クラス名」を宣言しなければならない。
 - そのメソッド内では例外に対応しなくても良い。
 - メソッドの呼び出し元で対応しなければならない。
- try-catchの中にtry-catchを入れられる。
 - 入れ子にできる。
- フィールドとローカル変数の区別。
 - フィールドとローカル変数に同じ名前が付けられる。
 - 「this.」を付けるとフィールドを参照することになる。
 - 付けなければローカル変数を参照することになる。

今までのプログラム



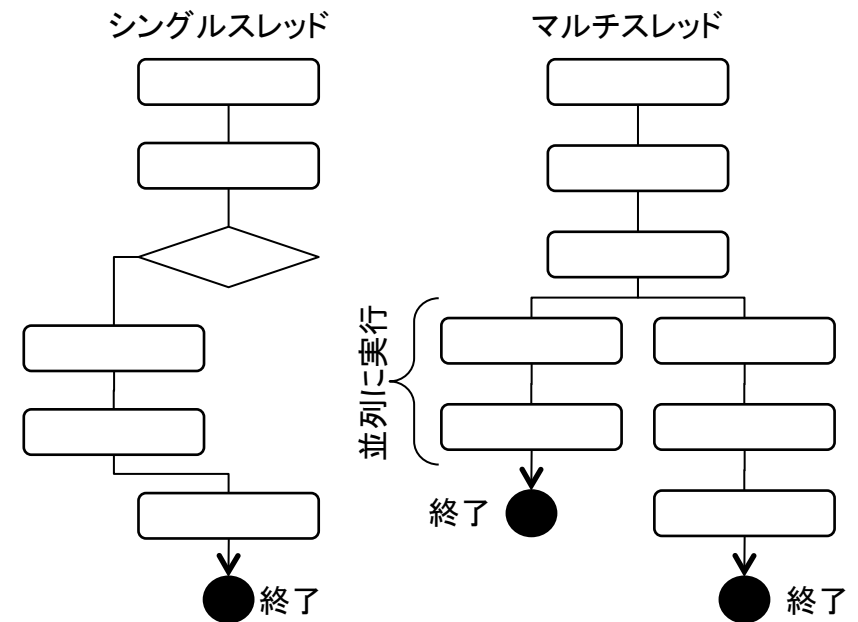
修正後のプログラム


クライアントが接続してくるとEchoServerは担当者を作成して、次を待ち受ける。



処理を並行して同時に行う

- 別のクラスが処理を実行しているときに、そのままでは、並行して処理を行えない。
- Java言語には、並行して処理を行うための機構が用意されている。
 - 並行して処理を行うには `java.lang.Thread` を用いる。
- `EchoServer` をスレッドに対応させる。
 - サーバにクライアントが接続してくれば、スレッドに処理を委譲し、サーバは次のクライアントを待ち受ける。





java.lang.Threadクラス

- Threadクラスを継承して, runメソッド内に並行して行いたい処理を書く.
- Threadクラスのstartメソッドが呼び出されると, runメソッドが並行して実行される.

Step8-1. EchoClientHandlerの Thread対応



- EchoClientHandlerのスーパークラスをThreadにする.
- EchoClientHandlerにrunメソッドを追加する.
 - runメソッドの中ではopen, receive, send, closeメソッドを呼び出す.

```
class EchoClientHandler  Thread{  
    ...  
    public void run(){  
        try{  
            open();  
            String message = receive();  
            send(message);  
        } catch (IOException e){  
            e.printStackTrace();  
        } finally{  
            close();  
        }  
    }  
    ...  
}
```



Step8-2. EchoServerで Threadの作成

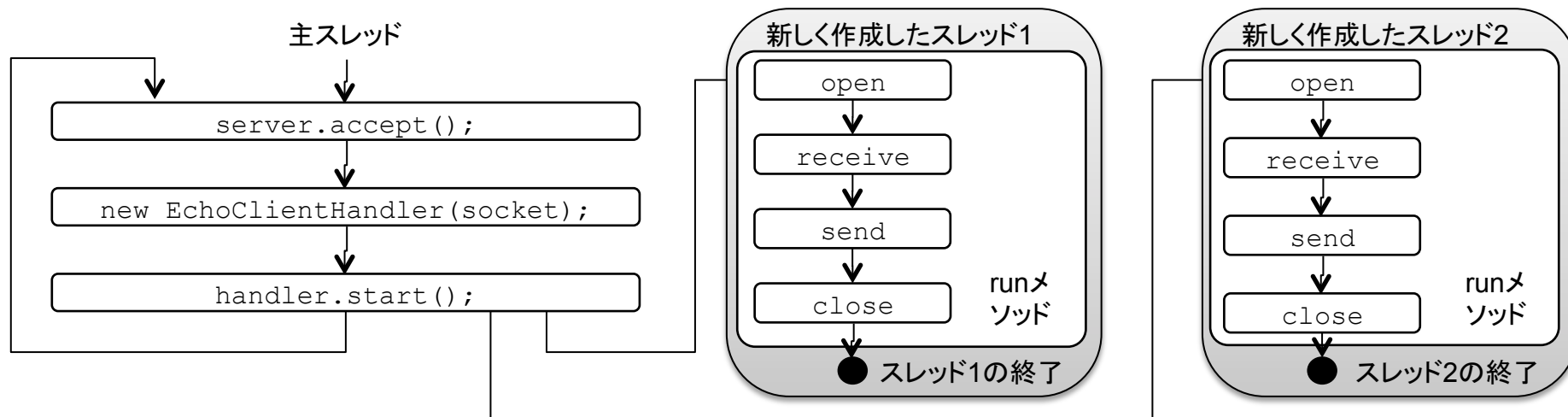
- performメソッドの呼び出しをstartメソッドの呼び出しに置き換える.
 - startメソッドには引数はない.
- closeメソッドの呼び出しを削除する.
 - startメソッドからは即座に処理が戻ってくる. runメソッドの実行途中でcloseしてしまうことになる.
- 無限ループ内でクライアントからの接続を待ち受ける.

```
while(true){  
    Socket socket = server.accept();  
    EchoClientHandler handler = new EchoClientHandler(socket);  
    handler.start(); // handler.receive(), send() の代わりに呼び出す.  
    // handler.close() を呼び出さないようにする.  
}
```

java.lang.Threadクラス

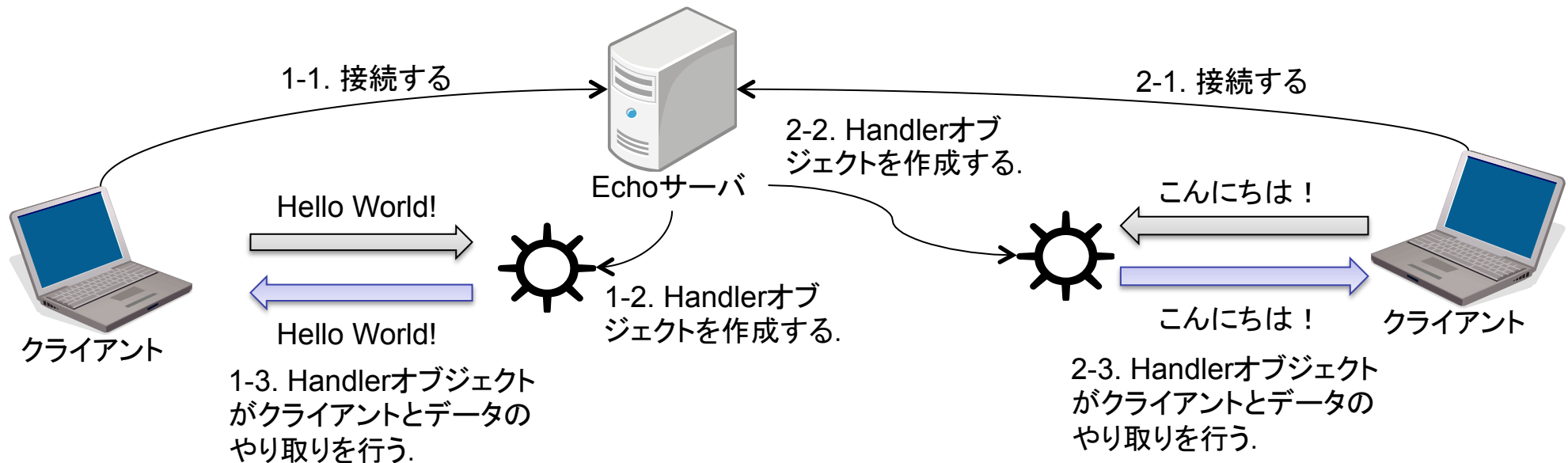
- Threadクラスを継承して, runメソッド内に並行して行いたい処理を書く.
- Threadクラスのstartメソッドが呼び出されると, runメソッドが並行して実行される.

Thread導入後のEchoClientHandlerの動作イメージ



完成したEchoServerの模式図

- EchoServerはクライアントが接続してくれば、Handlerオブジェクトを生成する。
 - EchoServerはその後、処理をHandlerにまかせ、次の接続を待つ。
- Handlerオブジェクトはクライアントとデータのやり取りを行う。

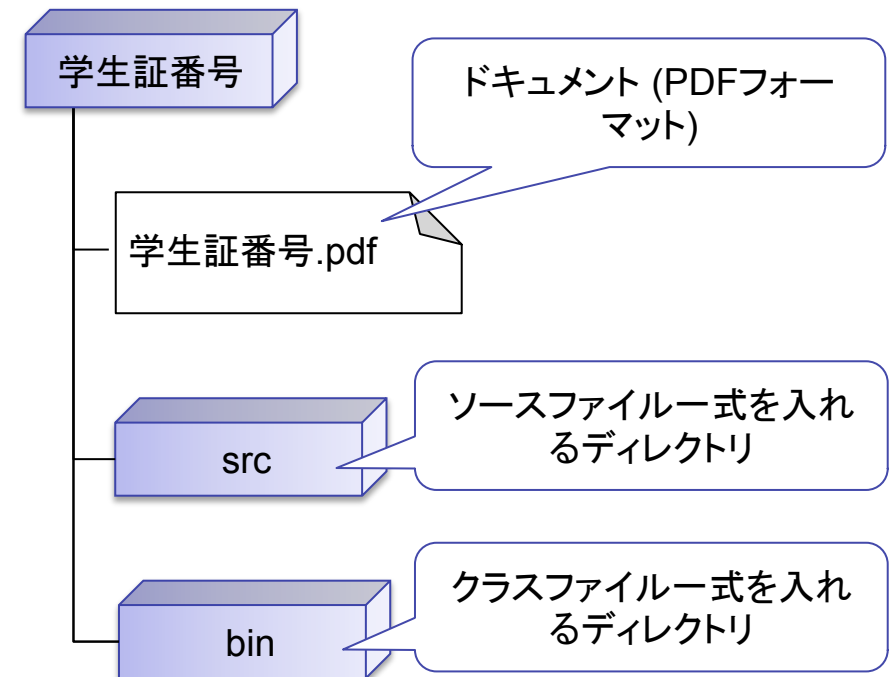


概要

- チャットサーバを作成せよ.
 - プロトコルは自由. 自分たちで定義しても良いし, 既存のプロトコルを利用しても良い.
 - クライアントは不要.
 - telnetコマンドをクライアントとすること.
 - マルチクライアントを達成すること.

提出方法

- 6桁の学生証番号のディレクトリを用意し、そこに次のファイルを入れる.
 - srcディレクトリ
 - ソースファイル一式.
 - binディレクトリ
 - クラスファイル一式.
- これら全てをzip圧縮したファイル (学生証番号.zip) をMoodleに提出する.
 - 展開したとき、学生証番号のフォルダができるようにする.



注意点

- コピーしないこと.
 - 他人のソースコードのコピーはもちろん, Webページからの無断転載は禁止.
 - Webページのプログラムを参考にした場合は, 自分が作成した部分を明確にすること.
- クラス名を適切に変更すること.
 - EchoXxxxを変更して作成するのは可.
 - その場合であっても, チャットであるのに, クラス名がEchoXxxであるのは名前と実装が合っていないのでダメ.



チャットサーバのヒント

- Echoサーバとの相違点
 - 自分の送ったデータは返送されない.
 - 他人の送ったデータが返送される.
 - サーバが全クライアントの参照(`Handler`オブジェクト)を保持している.
 - `close`メソッドの呼び出しは特定のコマンド(`quit`)が送られてくることで, 行うようにする.
 - `Handler`の`close`メソッドが呼び出されるときに, サーバから登録を削除する.