応用Javaプログラミング
第10回

1. クライアント／サーバー・モデル
2. ソケット・プログラミング入門
3. Template Methodデザインパターン（続）

サーバーのポート番号（port number）

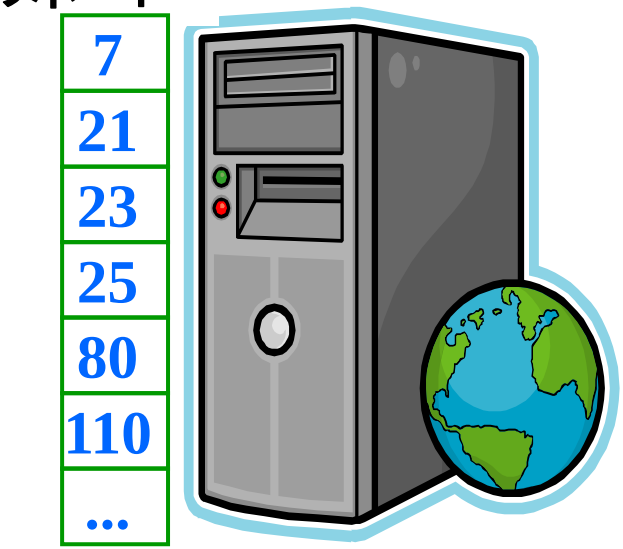
サーバーで色々なネットワークサービスを提供している．各サービスを特定する識別子があり，その識別子のことを**ポート番号**と呼ぶ．

よく知られているポート番号と
それらに対応するネットワークサービス：

ポート番号	ネットワークサービス
7	echo（エコー）
21	FTP（ファイル転送）
23	Telnet（遠隔ログイン）
25	SMTP（電子メール送信）
80	HTTP（WWW）
110	POP3（電子メール受信）
…	…

サーバーのポート

陳研サーバー
(133.14.32.91)



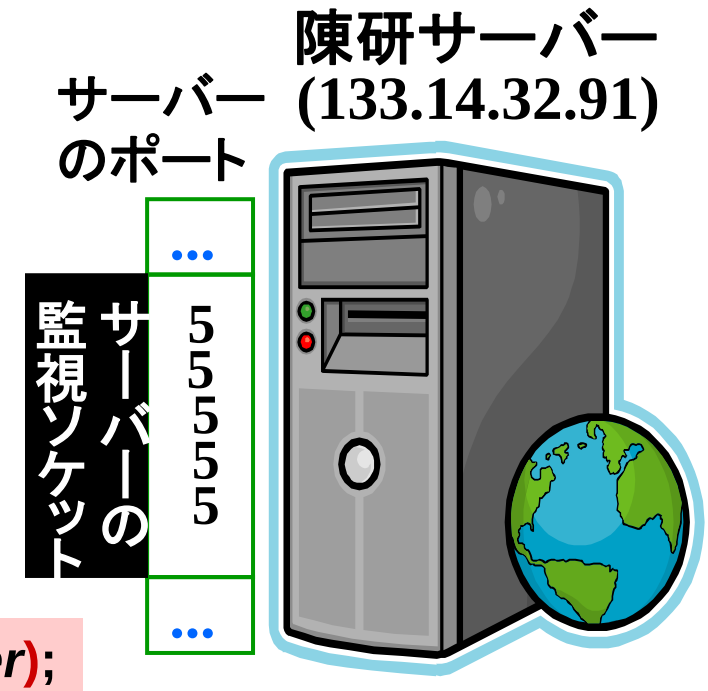
各ポート番号は
0以上65535以下
の整数である．

あるポートに届いた要求は，そのポートに対応するサービスプログラムによって処理される．

ソケット（socket）と サーバー・ソケット（server socket）

IPアドレスとポート番号の組をソケットと呼ぶ.

クライアント側からの接続要求を監視するためのソケットをサーバー・ソケットと呼ぶ.



JavaでServerSocketクラスが用意されており，サーバー・ソケットの生成に使われる:

```
ServerSocket ss = new ServerSocket(portNumber);
```

例: あるサーバーで新規のサービスを提供したい場合，まずそのサービスに対応するポート番号を選び，そのポートの監視用ソケットを生成する:

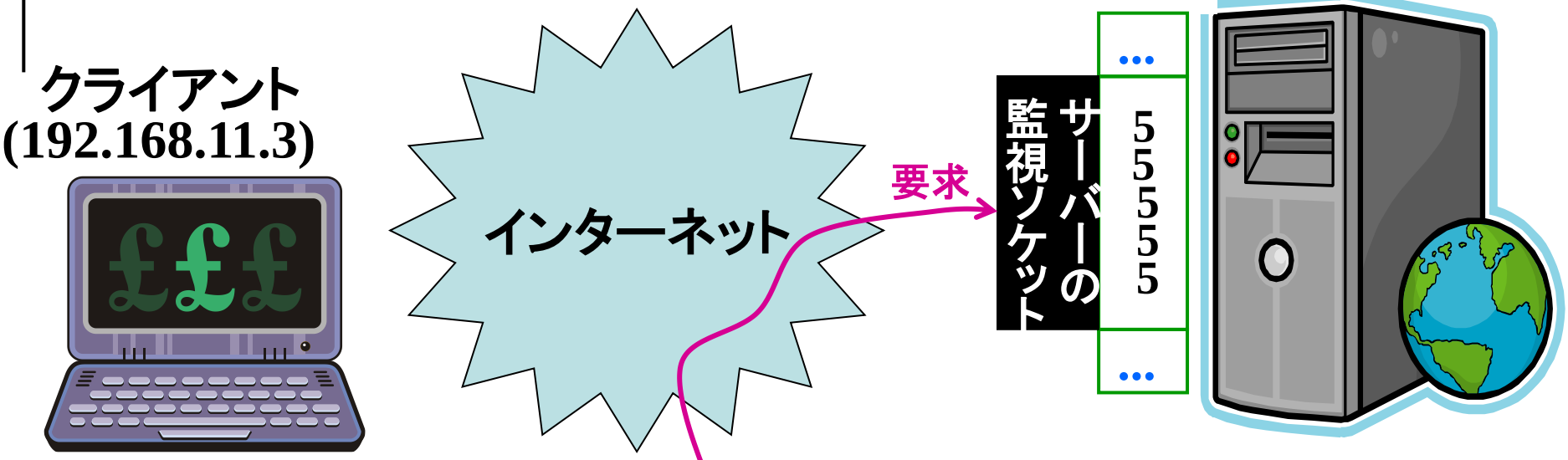
```
final int NEW_PORT = 55555; //まだ使われていない番号を選ぶ
ServerSocket ss = new ServerSocket(NEW_PORT) ;
```

java.net.ServerSocketクラス

主なコンストラクタ
public ServerSocket(int port)
指定されたポート番号にバインド（bind）されたソケットを生成する.

主なメソッド
public Socket accept( )
このソケットに対する接続要求を待機し，それを受け取り，新ソケットを作成する．このメソッドは接続が行われるまでブロックされる.

クライアントから サーバーへの要求



ネットワーク・パケット

133.14.32.91	55555	データ

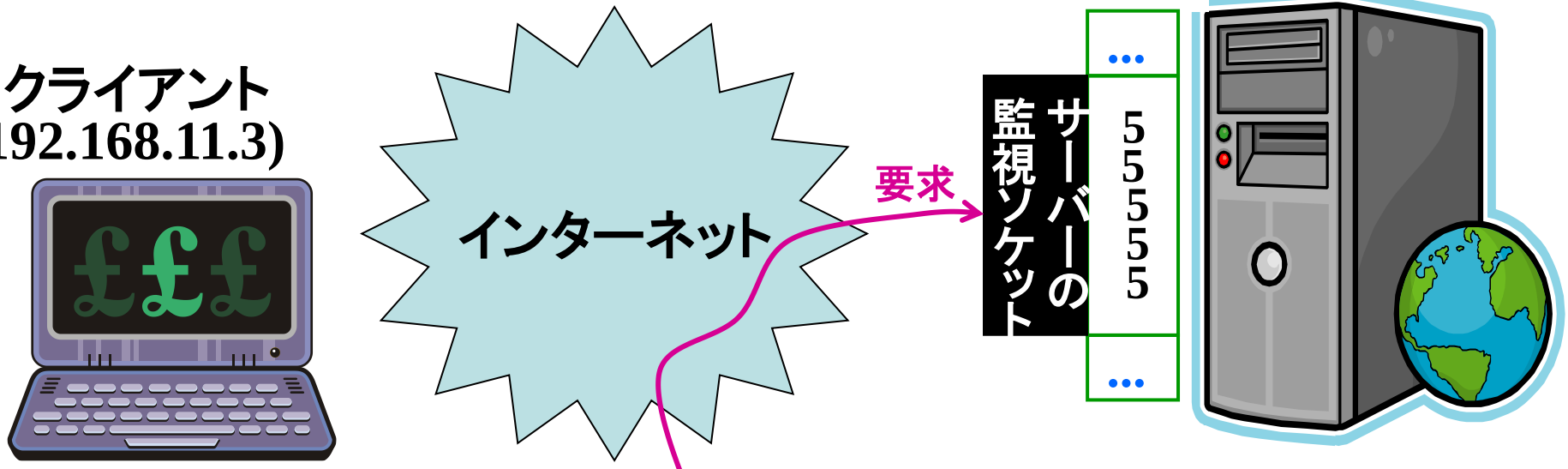
要求先のサーバーのIPアドレス　要求先のサーバーのポート番号　要求先のサーバーに送るデータ

サーバー側のソケット

クライアントからサーバーへの要求はネットワーク・パケットの形で送られる.

サーバーからクライアントへの応答も同じような形で行いたいならば,
クライアントからサーバーに送られるデータにクライアントのIPアドレスとポート番号を含める必要がある.

クライアント側にも ソケットが必要！



ネットワーク・パケット

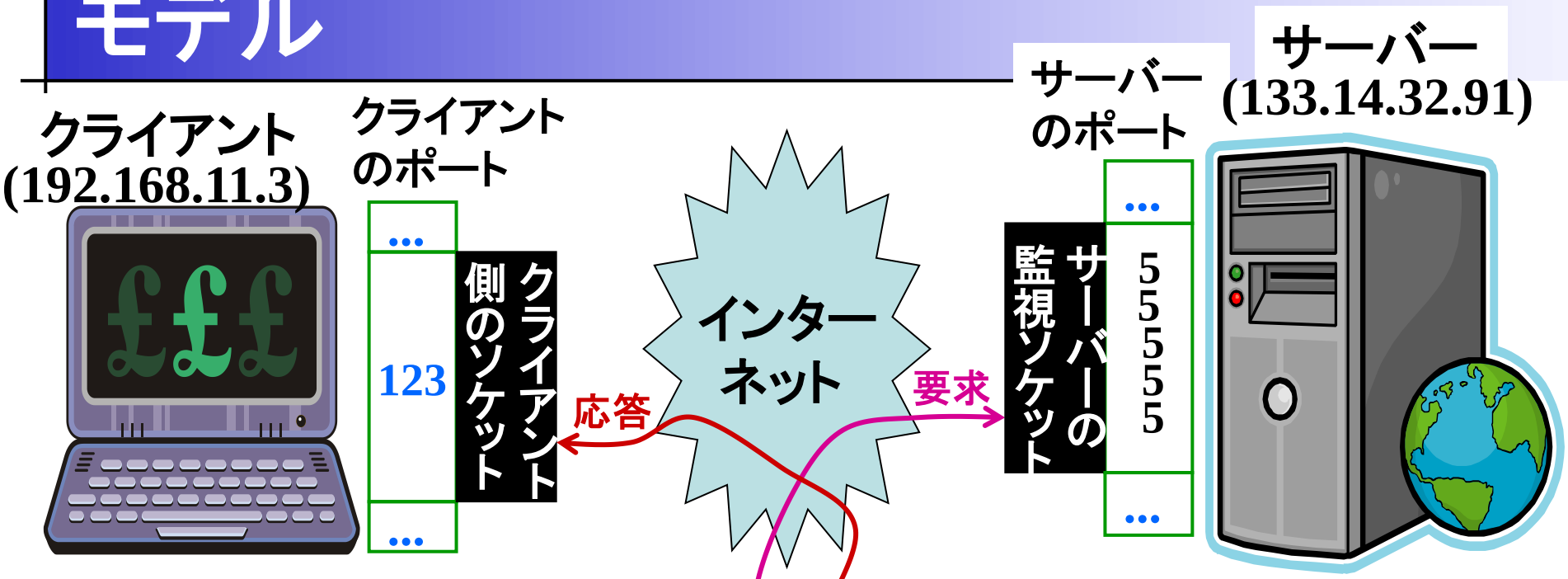
133.14.32.91	55555	データ

クライアントからサーバーへの要求は
ネットワーク・パケットの形で送られる.

サーバーのと同様に,
クライアントのIPアドレスと
ポート番号の組をソケットと呼ぶ.

サーバーからクライアントへの応答も
同じような形で行いたいならば,
クライアントからサーバーに送られる
データにクライアントのIPアドレスと
ポート番号を含める必要がある.

クライアント／サーバーモデル



ネットワーク・パケット

133.14.32.91	55555	データ

要求先のサーバーのIPアドレス

要求先のサーバーのポート番号

要求先のサーバーに送るデータ（クライアントのIPアドレスとポート番号を含む）

ネットワーク・パケット

192.168.11.3	123	データ

応答先のクライアントのIPアドレス

応答先のクライアントのポート番号

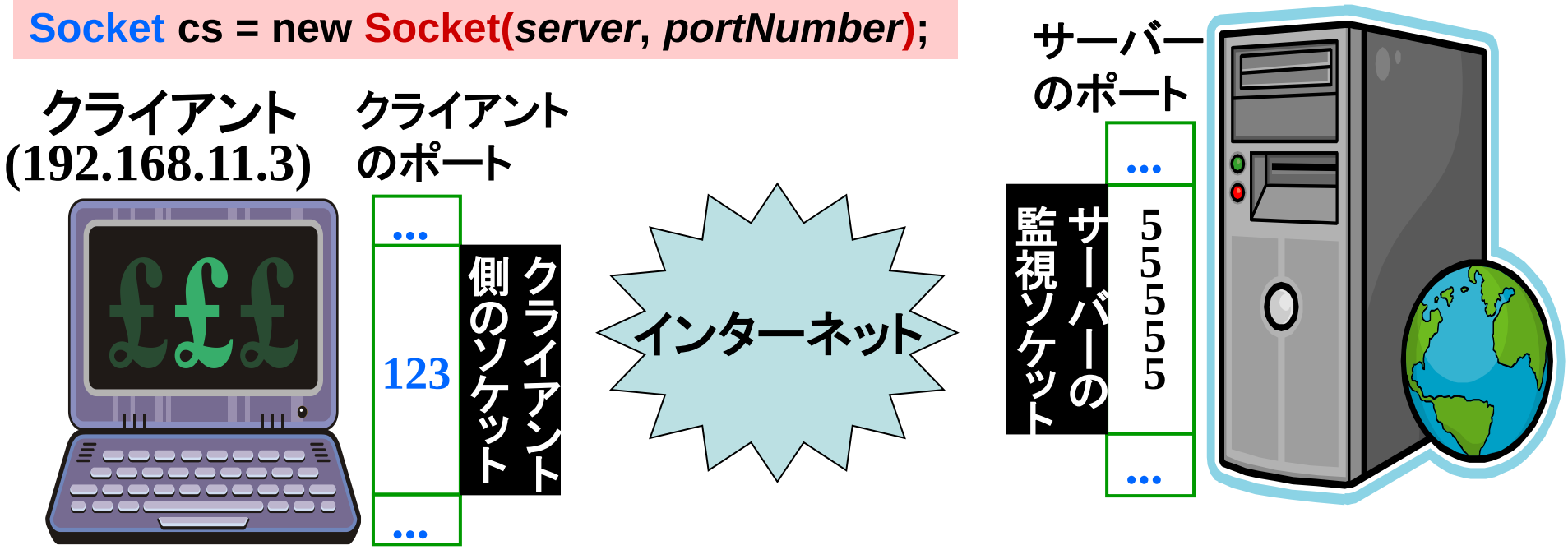
応答先のクライアントに送るデータ

クライアント側のソケットの生成

前頁から分かるように，サーバーからの応答を受けるためには，クライアントはサーバーへの要求を送る前に，自分側のソケットを生成する必要がある．

JavaでSocketクラスが用意されており，
クライアント側のソケットの生成に使われる:

```
Socket cs = new Socket(server, portNumber);
```



例：rnc.r.dendai.ac.jpのポート番号55555に対応するサービスに接続するには

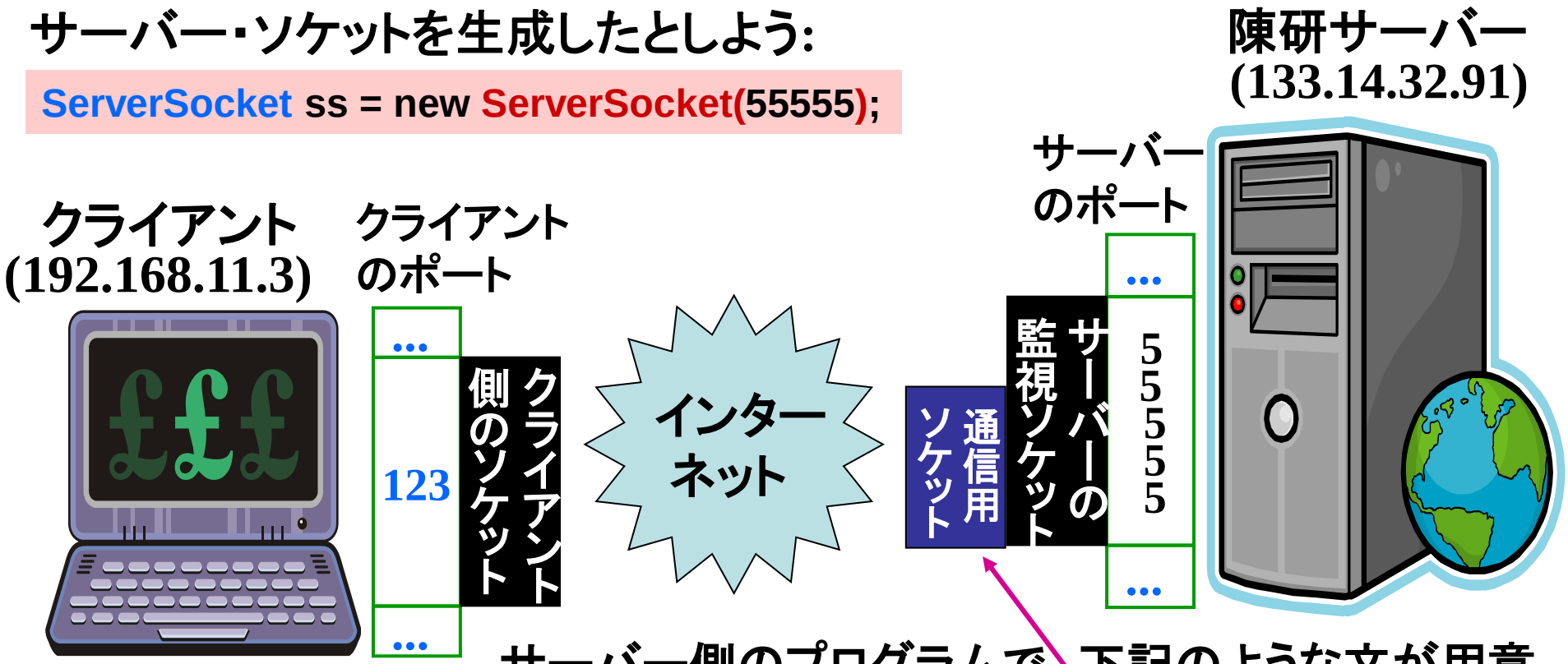
```
final  int  SERVER_PORT = 55555 ;
Socket cs = new Socket("rnc.r.dendai.ac.jp", SERVER_PORT);
```

左記のように自分のソケットを生成．

サーバー側で，接続の要求をしてきたクライアントとの通信用ソケットの生成

サーバー側のプログラムで下記のように
サーバー・ソケットを生成したとしよう:

```
ServerSocket ss = new ServerSocket(55555);
```

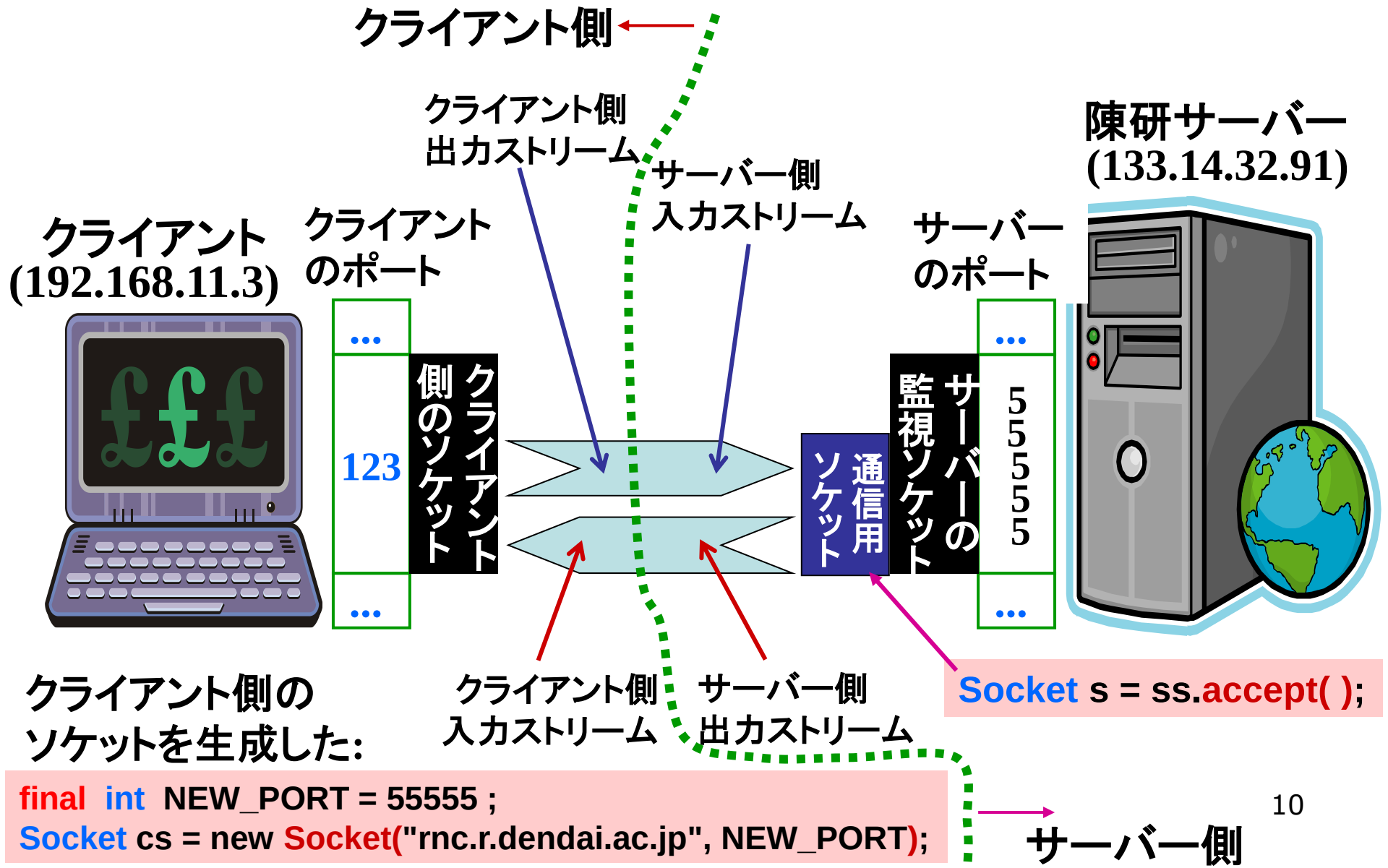


サーバー側のプログラムで，下記のような文が用意されていなければならない:

```
Socket s = ss.accept( );
```

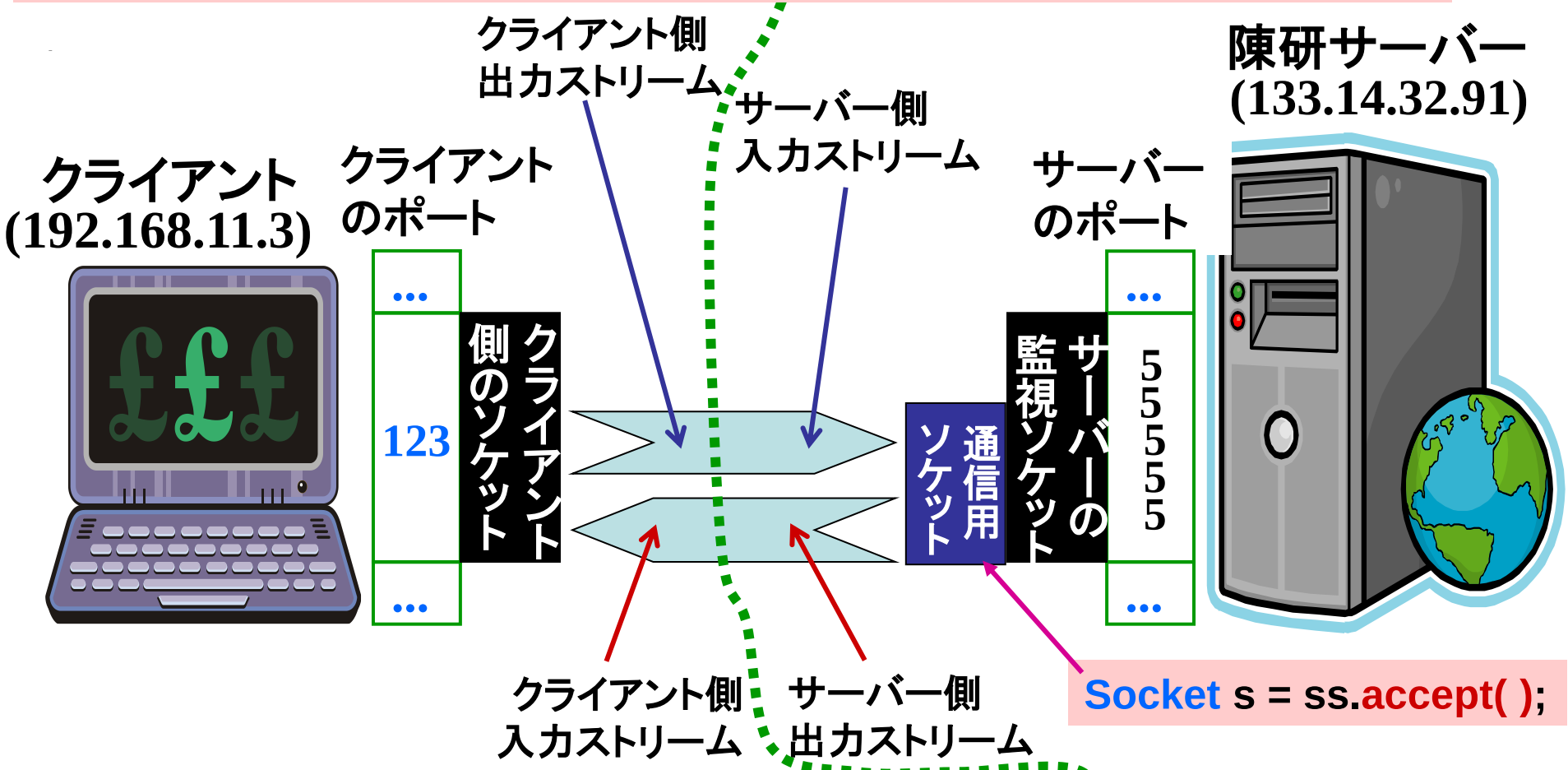
この文で，接続要求をしてきたクライアントとの通信を行うためのソケットを生成.

ソケットで通信が確立されたときの様子



クライアント側の入出力ストリームを得るには？

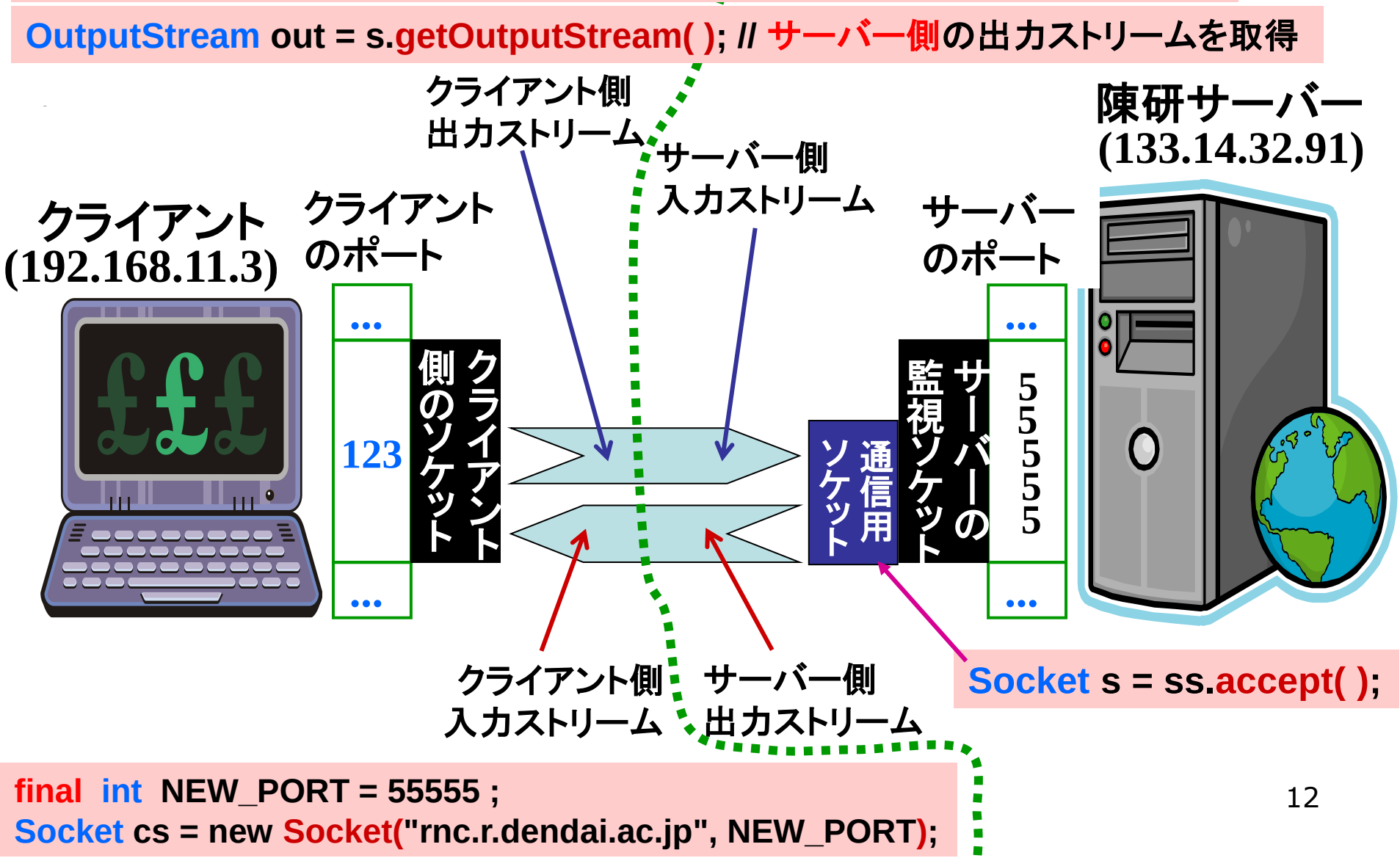
```
OutputStream out = cs.getOutputStream( ); // クライアント側の出力ストリームを取得
InputStream in = cs.getInputStream( ); // クライアント側の入力ストリームを取得
```




```
final  int  NEW_PORT = 55555 ;
Socket cs = new Socket("rnc.r.dendai.ac.jp", NEW_PORT);
```

サーバー側の入出力ストリームを得るには？

```
InputStream in = s.getInputStream( ); // サーバー側の入力ストリームを取得
OutputStream out = s.getOutputStream( ); // サーバー側の出力ストリームを取得
```




```
final  int  NEW_PORT = 55555 ;
Socket cs = new Socket("rnc.r.dendai.ac.jp", NEW_PORT);
```

java.net.Socketクラス

主なコンストラクタ
public Socket(String host, int port)
指定されたホスト上の指定されたポート番号に接続するためのソケットを生成.
public Socket(InetAddress address, int port)
指定されたIPアドレスの指定されたポート番号に接続するためのソケットを生成.

主なメソッド
public InputStrem getInputStream( )
このソケットの入力ストリームを返す.
public OutputStrem getOutputStream( )
このソケットの出力ストリームを返す.
public void close( )
このソケットを閉じる.
public void setTimeout(int timeout)
このソケットの入力ストリームから読み出そうとすると，設定した時間だけブロックされる.

ソケットの入出力ストリームともバイト・ストリーム.

大抵の場合，他のストリーム（たとえば文字ストリーム）に変換して使う.

2つの便利なツールをまとめたクラス

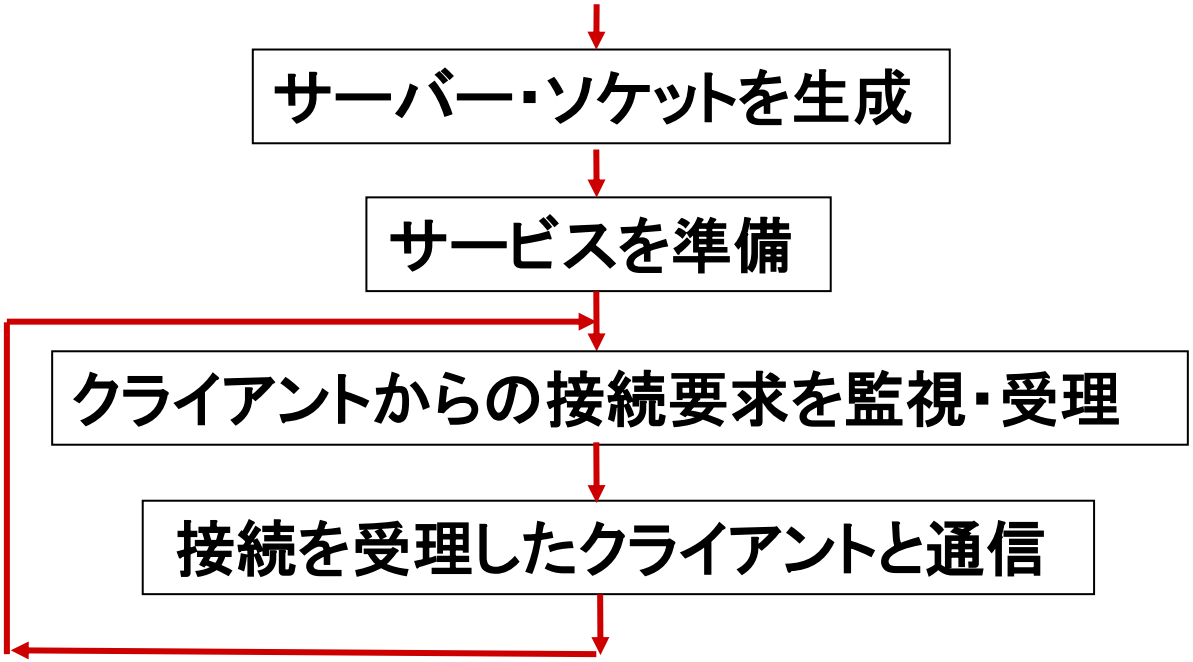
エコーサーバーやチャットサーバーの場合，テキストストリームを使うし，
漢字コードの問題を考慮する必要があるので，下記のクラスを用意しておく.

```
package network ;

import …(略)

public class SocketTools {
    private SokectTools( ) { } // 他のクラスでのインスタンス生成を禁止

    //ソケットから入力用のScannerを取得するためのメソッド
    public static Scanner getScanner( Socket s ) throws IOException {
        InputStream in = s.getInputStream( ); //ソケットから入力ストリームを取得.
        Reader r = new InputStreamReader(in, "UTF-8"); // 漢字コード対策.
        return new Scanner( r ) ; //ストリームにスキャナーをかぶせて返す
    }

    //ソケットから出力用のPrintWriterを取得するためのメソッド
    public static PrintWriter getPrintWriter( Socket s ) throws IOException {
        OutputStream out = s.getOutputStream( ); //ソケットから出力ストリームを取得.
        Writer w = new OutputStreamReader(out, "UTF-8"); // 漢字コード対策.
        return new PrintWriter( w, true); //ストリームにPrintWriterをかぶせて返す
    }
}
```

行の自動フラッシュを行う．各行をすぐに
目的地（e.g., クライアント 側）に送るようになる.

SocketTools.java

サーバーのtemplate



Socketを使うサーバーのtemplate

```
public abstract class Server {
    private ServerSocket ss ; //サーバーソケット
    public Server( ) { } // 何もしないコンストラクタ
    public Server( int portNo ) throws IOException {
        ss = new ServerSocket( portNo ); // サーバーソケットを生成.
    } //引数ありのコンストラクタ
    public void shutdown( ) throws IOException
    { if (ss != null) ss.close( ) ; } //サーバーを止めるメソッド
    public static void disconnect(Socket s) throws IOException
    { if (s != null && !s.isClosed()) s.close( ) ; } //clientとの通信を止めるためのメソッド
    public void startService( ) throws IOException, InterruptedException {
        prepareForService( ); //子クラスで定める必要のあるメソッドを呼び出す
        while ( !ss.isClosed( ) ) { // ループ
            try {
                Socket s = ss.accept( ); // クライアントからの接続要求を監視・受理する.

                //ソケットsを使って受理されたクライアントとの通信を行う.
                startService(s); //子クラスで定める必要のあるメソッドを呼び出す
            } catch (Exception e)  {   e.printStackTrace( ) ;   }
        } // ループの終わり
    }
    public void prepareForService( ) {  } //クライアント全体にサービスを提供する前の準備
    public abstract void startService(Socket s) throws IOException,
        InterruptedException ; //ソケットを使って受理されたクライアントとの通信を行う.
}
```

エコーサーバーにしろ
チャットサーバーにしろ,
このクラスがtemplateになる

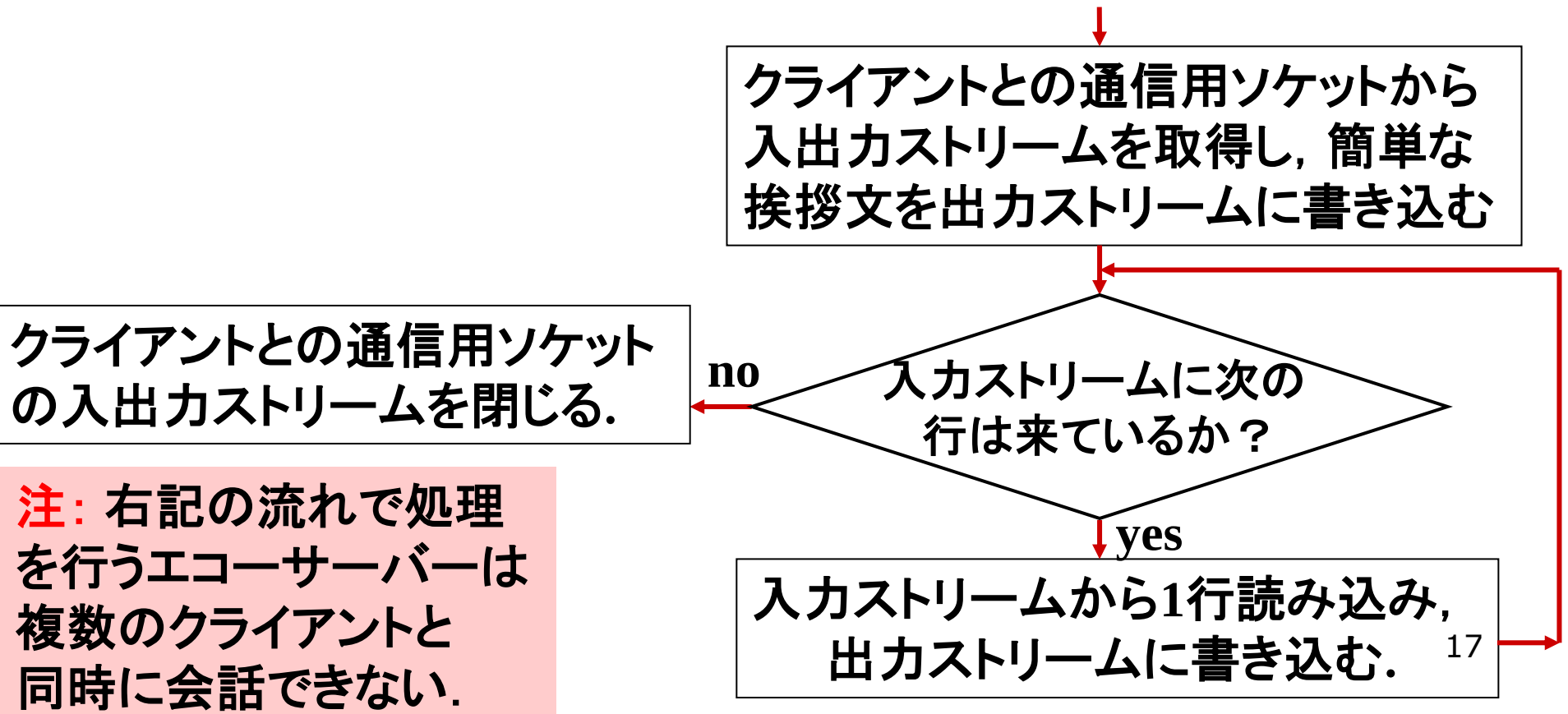
Server.java

エコー・サーバーを作ろう

概要: エコーサーバーは，クライアント側から送られてきた各行をそのまま返すだけ．また，クライアント側からの送信が なくなったら，そのクライアントとの通信をやめる．

prepareForServiceメソッドで何もしなくていい．

startServiceメソッドで行う処理の流れ：



注: 右記の流れで処理を行うエコーサーバーは複数のクライアントと同時に会話できない．

エコーサーバーを作ろう（続）

```
Scanner scan = SocketTools.getScanner(s);
PrintWriter p = SocketTools.getPrintWriter(s);
p.println("Welcome!!!");
```

```
while (scan.hasNextLine( ) )
```

クライアントとの通信用ソケットから入出力ストリームを取得し，簡単な挨拶文を出力ストリームに書き込む

```
scan.close( );
p.close( );
```

クライアントとの通信用ソケットの入出力ストリームを閉じる.

no

入力ストリームに次の行が来ているか？

yes

```
String line = scan.nextLine( );
p.println("Server：" + line);
```

入力ストリームから1行読み込み，出力ストリームに書き込む.

エコー・サーバーを表すクラス

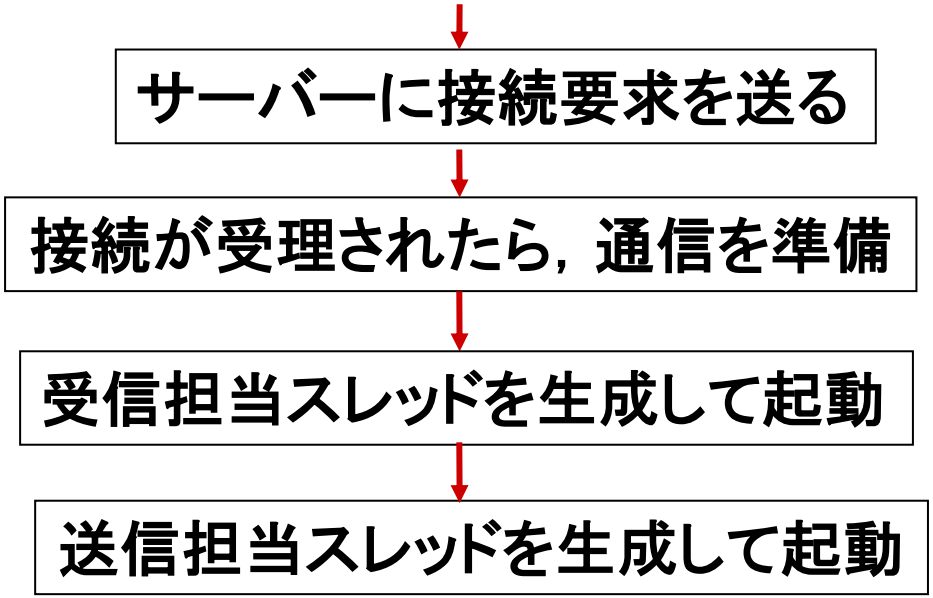
```
public class  EchoServer extends  Server {
   public EchoServer( ) { } // 何もしないコンストラクタ

   //ポート番号を受け取り，クライアントからの接続要求を受け付けるためのコンストラクタ
   public EchoServer( int portNo ) throws IOException
   {   super( portNo );  } //引数ありコンストラクタの終わり

   //ソケットを受け取ってクライアントと通信を行うための自作メソッド.
   public void startService(Socket s) throws IOException {
      try ( Scanner scan = SocketTools.getScanner(s) ;
                  PrintWriter p = SocketTools.getPrintWriter(s)  ){
         p.println("Welcome!!!") ; // 最初の挨拶文を表示
         while ( scan.hasNextLine( ) ) {    // クライアントから会話が来ている間.
            String line = scan.nextLine( ) ; // クライアントからの次の会話文を読み込む
            p.println("Server: " + line); // クライアントからの会話文にエコーをつけて返す.
            if (line.trim().equals("BYE")) break;
         }
      } finally { disconnect( s ) ; } // 通信用ソケットを閉じる.
   }
   //mainメソッドは次頁にある.
}
```

EchoServer.java

エコー・サーバーを表すクラス[続き]

```
public static void main( String [ ] args ) throws IOException {
    if ( args.length != 1) { //使い方が正しくなければ，エラーを表示して終了
        System.out.println("Usage: java EchoServer portNo") ;
        return ;
    }
    try {
        //エコーサーバーを生成してサービスを開始する
        new  EchoServer( Integer.parseInt(args[0]) ).startService( ) ;
    } catch (Exception e) {
        e.printStackTrace( ) ;
    }
} //mainメソッドの終わり
```

クライアントのtemplate



Socketを使うクライアントのtemplate

エコークライアントにしろ
チャットクライアントにしろ,
下記のクラスがtemplateになる.

```
public abstract class Client {
    private Socket s ; // ソケット
    public Client( ) { } // 何もしないコンストラクタ
    public Client(String serverName, int portNo)
                throws IOException  //引数ありのコンストラクタ
    {    s = new Socket( serverName, portNo ); // サーバーに接続.  }

    public void disconnect( ) throws IOException
     { if (s != null && !s.isClosed()) s.close( ) ; } //serverとの通信を止めるためのメソッド

    public void prepareForService(){ System.out.println("type ¥"BYE¥" to quit!"); }

    //PrintWriterを受け取ってサーバーにメッセージを送るためのメソッド. 子クラスで定義.
    public abstract void send(PrintWriter p) throws IOException ;

    //Scannerを受け取ってサーバーからメッセージを受信するためのメソッド. 子クラスで定義.
    public abstract void receive(Scanner scan) throws IOException ;

    //Accessorメソッド.
    public Socket getSocket( ) { return s ; }
    public void setSocket(Socket s) { this.s = s ; }
```

Client.java

次頁へ

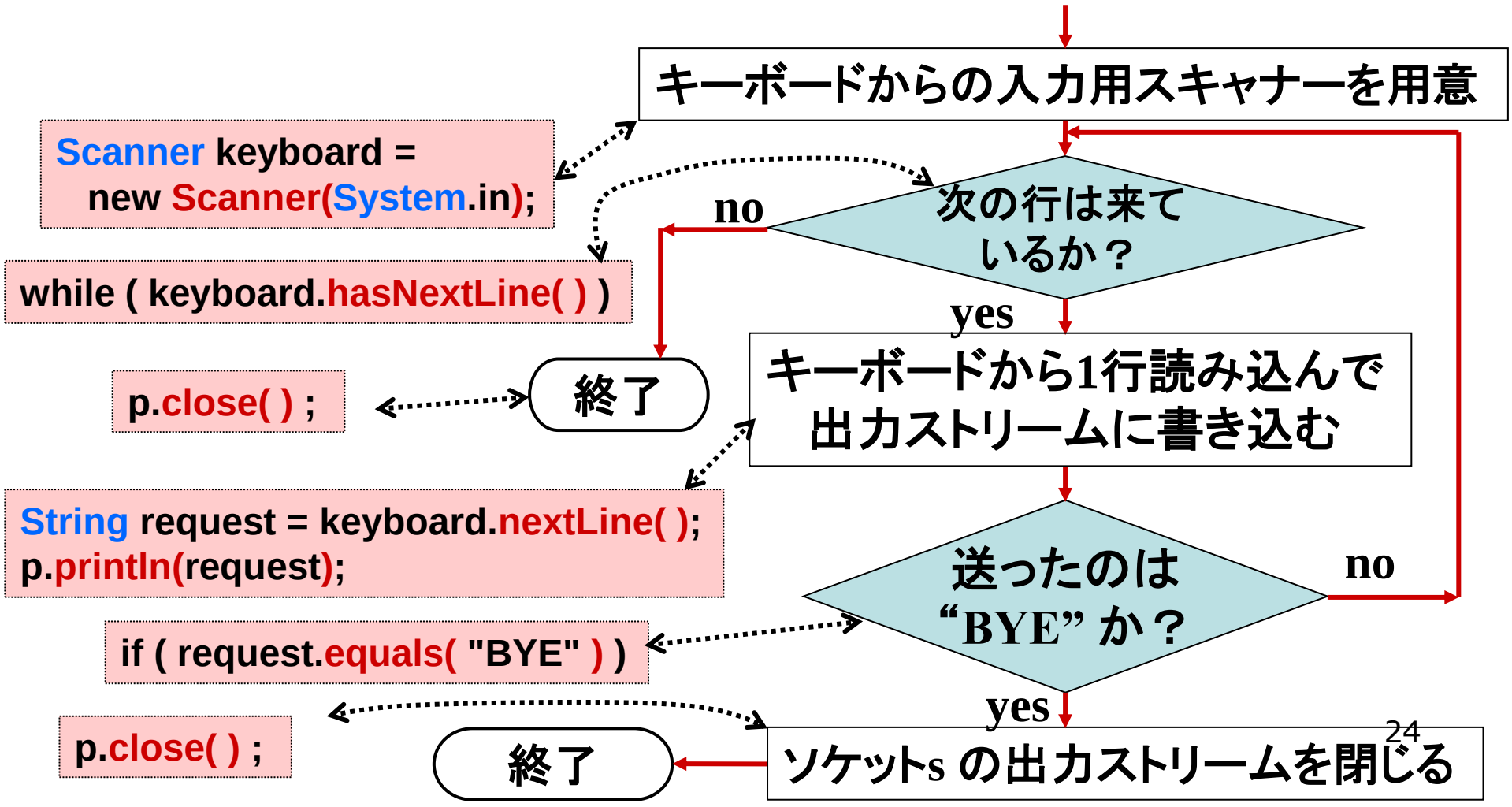
Socketを使うクライアントのtemplate（続）

```
//送受信を担当するスレッドをそれぞれ生成して起動する自作メソッド.
public void startService( ) throws IOException {
    prepareForService( ) ;

    new Thread(new Runnable() { //受信用スレッドを生成して起動
        public void run( ) {    // 仕事の内容
            try (Scanner scan = SocketTools.getScanner(s) ) {
                receive(scan) ; //子クラスで定める自作メソッド
            } catch ( IOException e ) { e.printStackTrace() ;  }
        } // runメソッドの終わり
    }).start( ) ; //受信を担当するスレッドを生成して起動.

    new Thread(new Runnable() { //送信用スレッドを生成して起動
        public void run( ) {    // 仕事の内容
            try (PrintWriter p = SocketTools.getPrintWriter(s) ) {
                send(p) ; //子クラスで定める自作メソッド
            } catch ( IOException e ) { e.printStackTrace() ;  }
        } // runメソッドの終わり
    }).start( ) ; //送信を担当するスレッドを生成して起動.
  }
} //Client.javaの終わり
```

Client.java

エコークライアントを作ろう

まず，send(PrintWriter p)メソッドで行う処理の流れ：

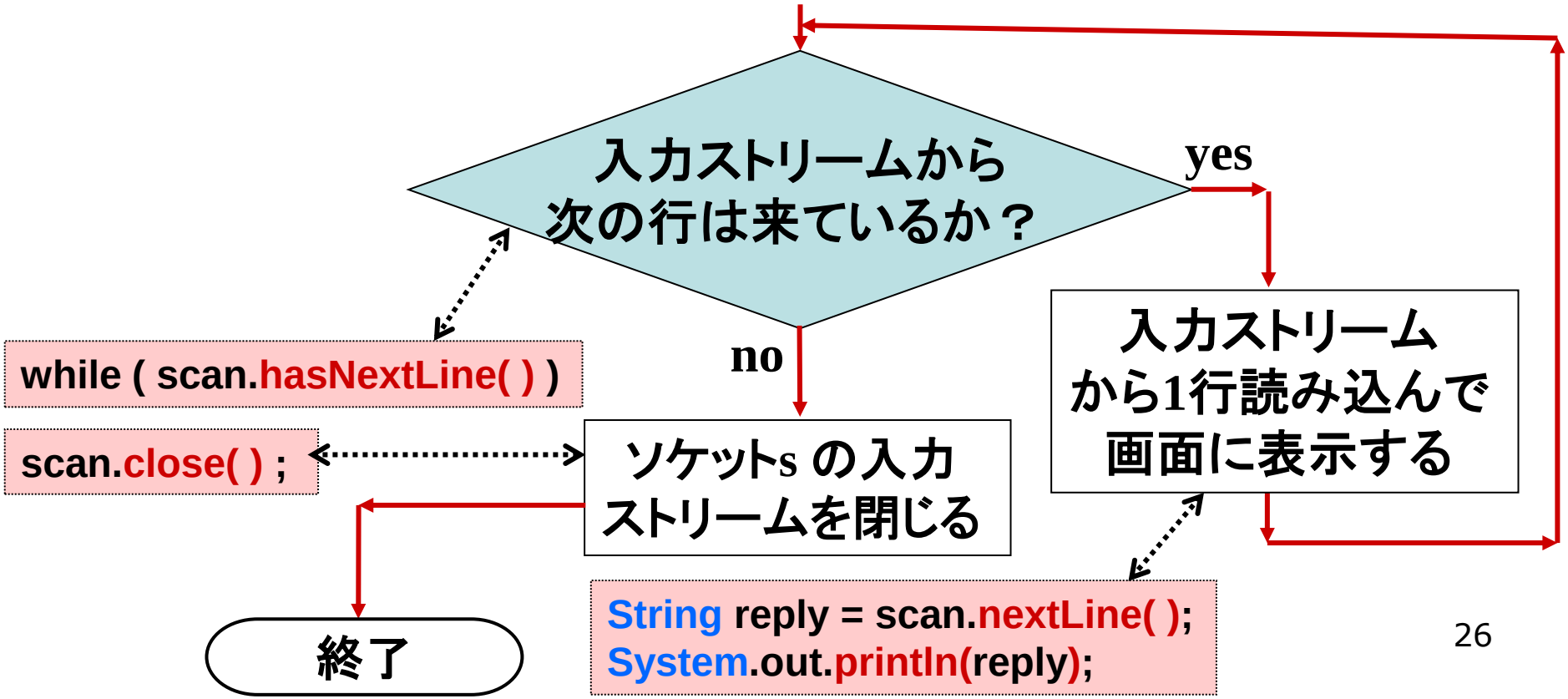


send(PrintWriter p)メソッド

```
//printWriterを受け取ってサーバーにメッセージを送るための自作メソッド.
@SuppressWarnings("resource") //キーボードからの入力用Scannerを閉じたくない
 public void send(PrintWriter p) throws IOException {
    Scanner keyboard = new Scanner(System.in); // キーボードからの入力用.
    String request ; // サーバーに送られる一文を覚える変数.
    try {
       while ( !getSocket().isClosed( ) && keyboard.hasNextLine( ) ) {
          request = keyboard.nextLine( ); //キーボードから次の会話文を読み込む.
          p.println(request); // 読み込んだ会話文をサーバーに送る.
          if ( request.equals("BYE") )  break ;
       }
    } fianally {
        disconnect( ) ;
    }
 } // sendメソッドの終わり
```

エコークライアントを作ろう（続）

次に，receive(Scanner scan)メソッドで行う処理の流れ：



receive(Scanner scan)メソッド

```
//scannerを受け取ってサーバーからメッセージを受信するための自作メソッド.
public void receive(Scanner scan) throws IOException {
    String reply ; // サーバーからの返事（一文）を覚える変数.
    try {
        while ( scan.hasNextLine( ) ) {
            reply = scan.nextLine( ); //入力ストリームから次の会話文を読み込む.
            System.out.println(reply); // 読み込んだ会話文を画面に表示する.
        }
    } fianally {
         disconnect( ) ;
    }
} // receiveメソッドの終わり
```

エコークライアントを表すクラス

```
public class  EchoClient extends  Client {
   public EchoClient( ) { } //何もしないコンストラクタ

   public EchoClient( String serverName, int portNo ) throws IOException {
      super(serverName, portNo);
   } // 引数ありのコンストラクタの終わり

   public void send(PrintWriter p) throws IOException { …（略）}
   public void receive(Scanner scan) throws IOException { …（略）}

   public static void main( String [ ] args ) throws IOException {
      if ( args.length != 2) { //使い方が正しくなければ，エラーを表示して終了
         System.out.println(“Usage: java EchoClient serverName portNo”);
         return;
      }
      try {
         new  EchoClient(args[0], Integer.parseInt(args[1])).startService( ) ;
      } catch (Exception e) {
         e.printStackTrace( ) ;
      }
   } //mainメソッドの終わり
} //EchoClient.javaの終わり
```

EchoClient.java

先ほど作ったエコーサーバーを見直そう

先ほどのエコー・サーバーは複数のクライアントと同時に会話できない．

この欠点を改善するためには，マルチスレッドを利用すればよい．

具体的には，新しいstartServiceメソッド内での処理を下記のように変更：

```
// このクライアントとの会話 を担当するスレッドを生成して起動する.
new Thread(new  Runnable() {
    public void run() {
        try {
            new EchoServer().startService(s) ;
        } catch ( IOException e) {
            e.printStackTrace() ;
        }
    }
}).start() ;
```

マルチスレッドを用いたエコーサーバーを表すクラス

```
public class  EchoServer2 extends  Server {
    public EchoServer2( int portNo ) throws IOException {
        super( portNo ) ; // スーパークラスの引数ありコンストラクタを呼び出す.
    } // 引数ありコンストラクタの終わり

    public void startService(Socket s) {
        new Thread(new Runnable() {
            public void run( ) {
                try {
                    new EchoServer().startService(s) ;
                } catch ( IOException e) { e.printStackTrace() ; }
            }
        }).start( );
    } // startServiceメソッドの終わり

    public static void main( String [ ] args ) throws IOException {
        if ( args.length != 1) { //使い方が正しくなければ，エラーを表示して終了
        {     System.out.println("Usage: java EchoServer2 portNo") ;   return ; }
        new  EchoServer2( Integer.parseInt(args[0]) ).startService( ) ;
    } //mainメソッドの終わり
} //EchoServer2.javaの終わり
```

EchoServer2.java

演習課題

EchoServer2クラスでサーバーがクライアントを受理するたびにスレッドを新たに生成して応答を担当してもらった．クライアントとの会話を終えると，応答スレッドが終了してしまう．スレッドの生成と終了にはオーバーヘッドが生じてしまうのが問題．

上記の問題を改善するためには，スレッドのプールを利用すればよい．この方法では，サーバーがまずスレッドのプールを生成しておいて，クライアントを受理するたびに，「それに応答する」という仕事をスレッドのプールに投げる．ただし，サーバーの終了時に，スレッドのプールをシャットダウンしなければならない．

```
ExecutorService pool =
        Executors.newFixedThreadPool(NUMBER_OF_THREADS) ;
```

課題：EchoServer2クラスを参考にして，スレッドのプールを利用するEchoServerPoolクラスを作成せよ．

Template Methodデザインパターン（再）

ポイント：
親クラスで処理の枠組みを定め，子クラスでその具体的内容を定める．

AbstractClass役（抽象クラス役）：テンプレートメソッド（電卓の例では，setUpメソッド）を実装する役．また，そのテンプレートメソッドで使っている抽象メソッドを宣言する．この抽象メソッドは，子クラスである**ConcreteClass役**によって実装される．
今回の例では，**Server**クラスと**Client**クラスがこの役をつとめている．

ConcreteClass役（具象クラス役）：**AbstractClass役**で定義された抽象メソッドを具体的に実装する役．ここで実装したメソッドは，**AbstractClass役**のテンプレートメソッドから呼び出される．
今回の例では，**EchoServer**クラスや**EchoServer2**クラスや**EchoClient**クラスがこの役この役をつとめている．