

# 応用Javaプログラミング

## 第11回

1. チャット・システムの構築
2. Observerデザインパターン
3. Factory Methodデザインパターン

# チャット・システム

チャット・サーバー

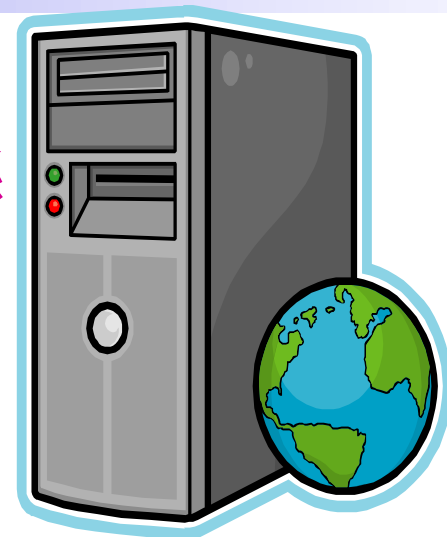
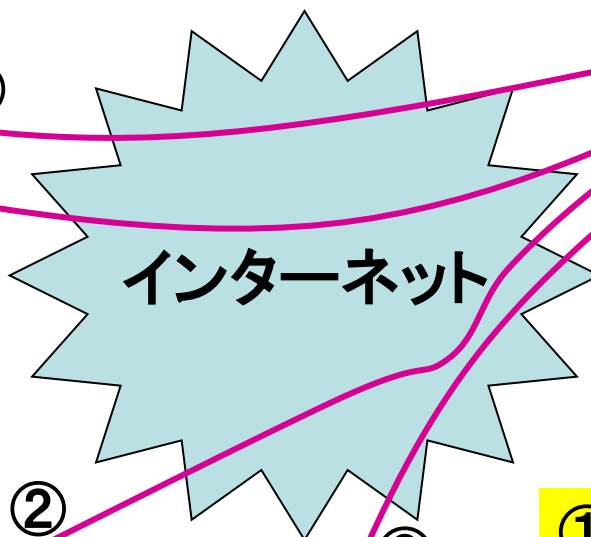
チャット・クライアント



チャット・クライアント



チャット・クライアント



- ① あるクライアントから会話をサーバーに送ると,
- ②サーバーがその会話をクライアント全員に送る.

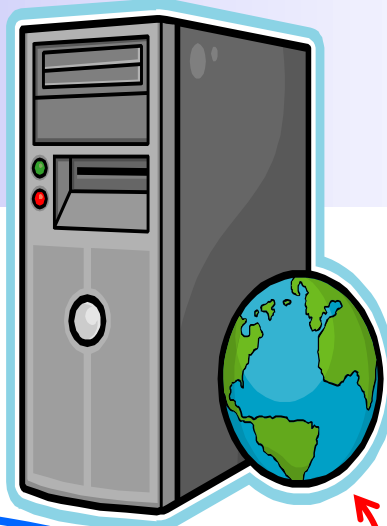
あたかもクライアントの間で直接会話が行われているように見える.

# ここで作るチャット・システムの構成

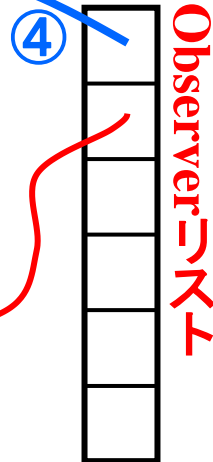
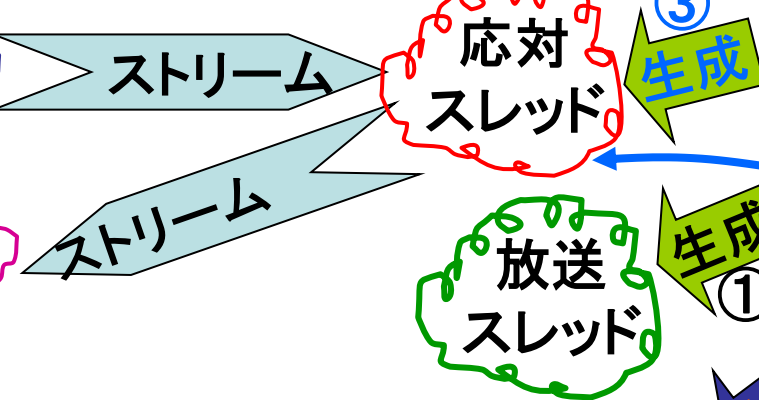
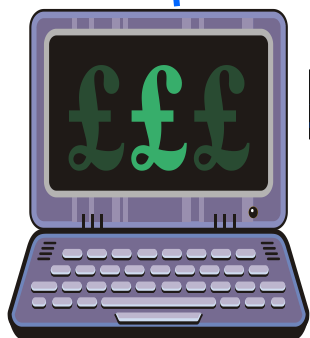
チャット・サーバー

②接続要求

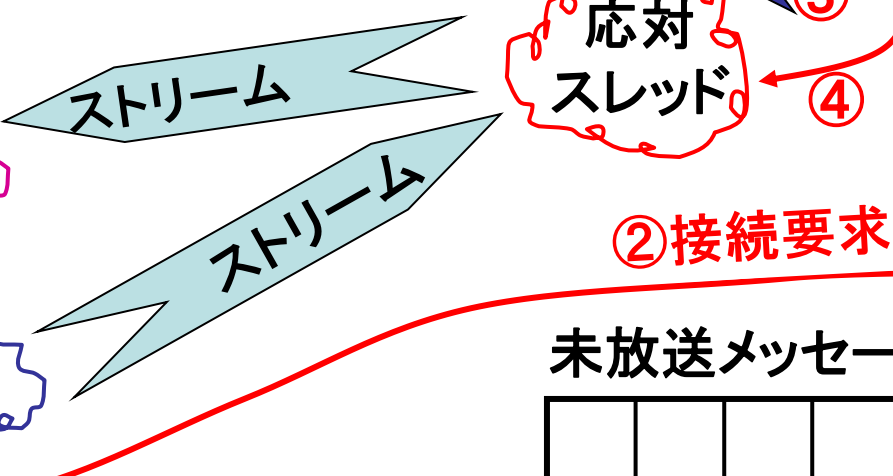
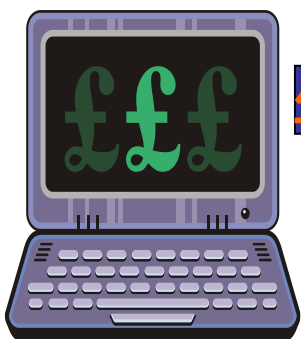
監視  
サーバーの  
ソケット



チャットクライアント



チャットクライアント

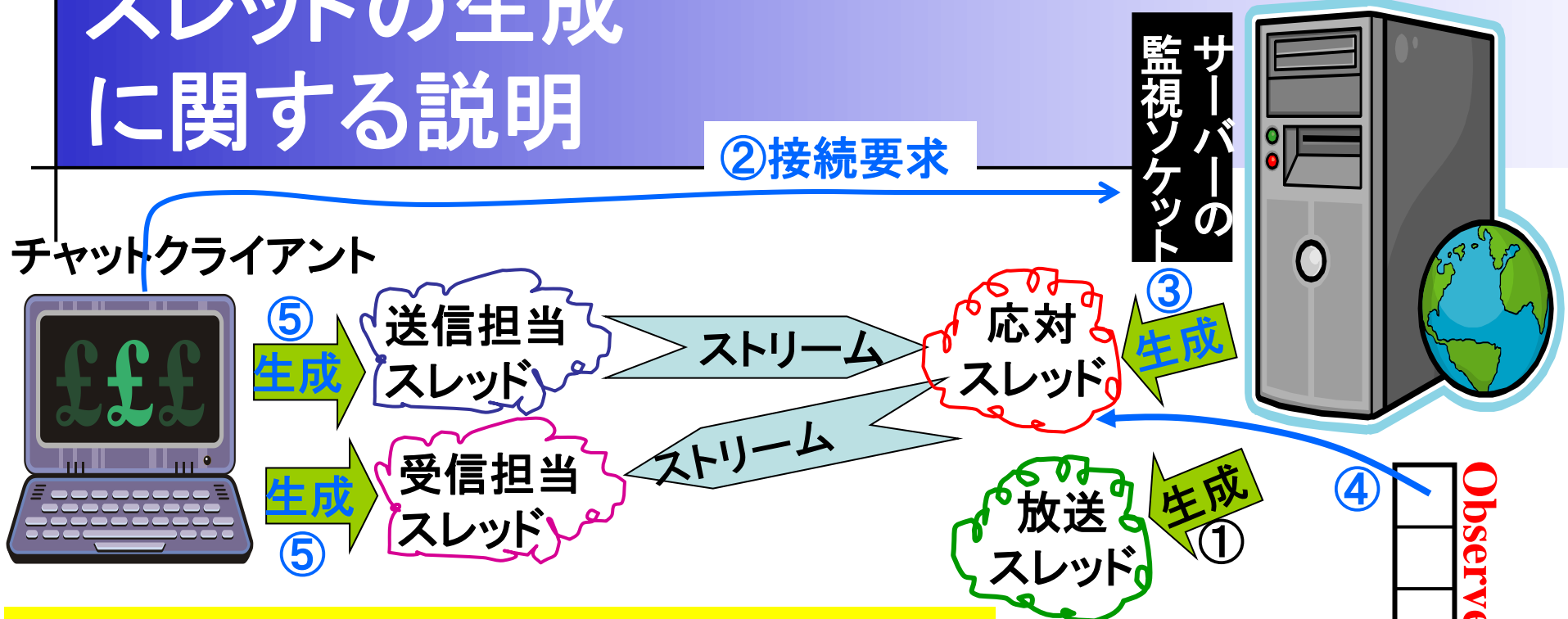


未放送メッセージのキュー



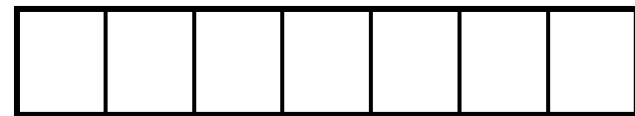
# スレッドの生成に関する説明

チャット・サーバー



- ①サーバーが放送スレッド(**Subject**役)を生成.
- ②サーバーが監視用ソケットを介してクライアントの接続要求を待ち受ける.
- ③クライアントの接続要求が来たら、サーバーはそれに対応するスレッドを生成する.
- ④サーバーは「応対スレッド」を生成したら、それを放送スレッドの**Observer**リストに追加.
- ⑤クライアント側で送受信のスレッドを生成.

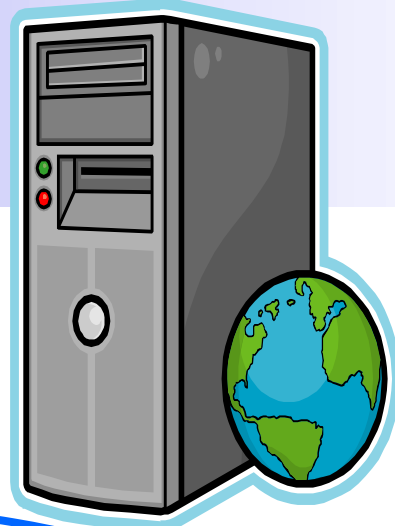
未放送メッセージのキュー



- ① クライアントの送信担当スレッドがサーバーの応対スレッドにメッセージmを送る.
- ② 応対スレッドがmを未放送メッセージキューに置く.
- ③ サーバーの放送スレッドがmをキューから取り出す.
- ④ 放送スレッドがmを受信担当スレッドに送ってもらう.

チャット・サーバー

サーバーの  
監視  
ソケット



チャットクライアント



送信担当  
スレッド

① メッセージm

ストリーム

応対  
スレッド

受信担当  
スレッド

ストリーム

④ メッセージm

放送  
スレッド

Observerコンス  
ト

## メッセージの流れ

チャットクライアント



受信担当  
スレッド

④ メッセージm

ストリーム

ストリーム

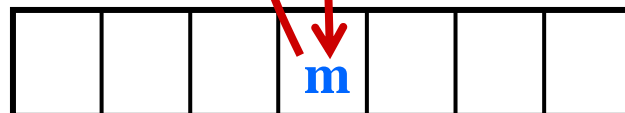
応対  
スレッド

送信担当  
スレッド

③

②

未放送メッセージのキュー



# チャット・サーバーのポイント

「未放送メッセージのキュー」は、「生産者／消費者」モデルで製品を格納するために使われるブロッキングキューを使えばよい。ここでの「製品」は未放送メッセージで、「生産者」はサーバー側の「応対スレッド」全体で、「消費者」はサーバー側の「放送スレッド」である。

# 放送スレッドと応対スレッドの関係 (Observerデザインパターン)

(1) 「放送スレッド」はSubject役.

JavaでSubject役を作るためにObservableクラスを用意しているが、非推奨になっている. その代わりに、PropertyChangeSupportクラスを使う.

(2) 「応対スレッド」はObserver役.

JavaでObserver役を作るためにObserverクラスを用意しているが、非推奨になっている. その代わりに、PropertyChangeListenerクラスを使う.

# Observerデザインパターン

**ポイント:** 複数の観察者が1人の観察対象に登録し、その観察対象の状態が変化すると、登録している観察者全員に通知が行くような仕組みを提供する手法.

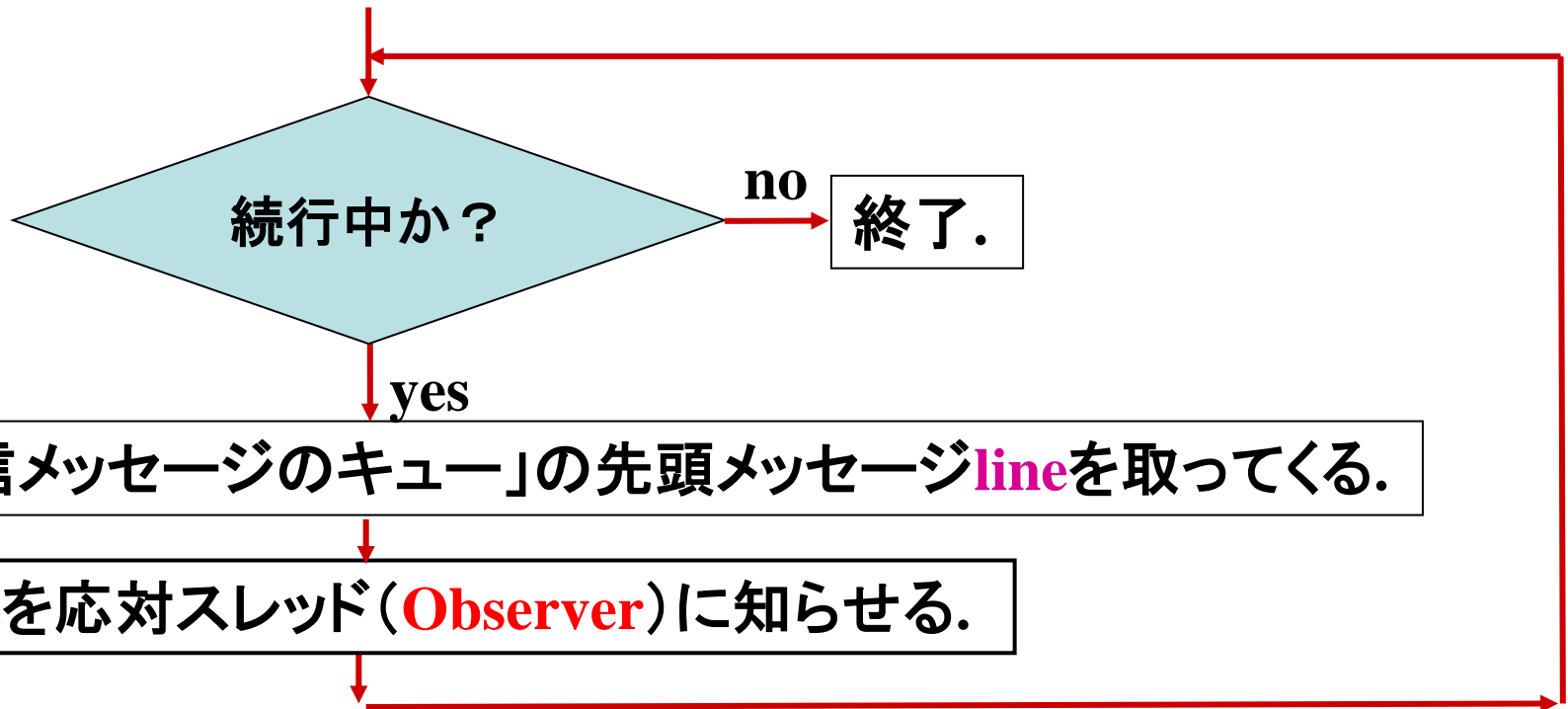
**Subject**(被験役): 観察対象を表す役で、観察者である**Observer**役を登録メソッドと、削除するメソッドを持っている. また、「現在の状態を取得する」メソッドも宣言されている. 状態が変化したら、そのことを登録されている**Observer**役に伝える. 例では、**Broadcaster**クラスがこの役をつとめている.

**Observer**役(観察者役): **Subject**役から「状態が変化したよ」と教えてもらう役. そのためのメソッドが **update**メソッド. そのメソッドが呼び出されると、そのメソッドの中で**Subject**役の現在の状態を取得する. 例では、**Connection**クラスがこの役をつとめている.



# 放送スレッド(Subject役)

放送スレッドのrunメソッド内での処理の流れ:

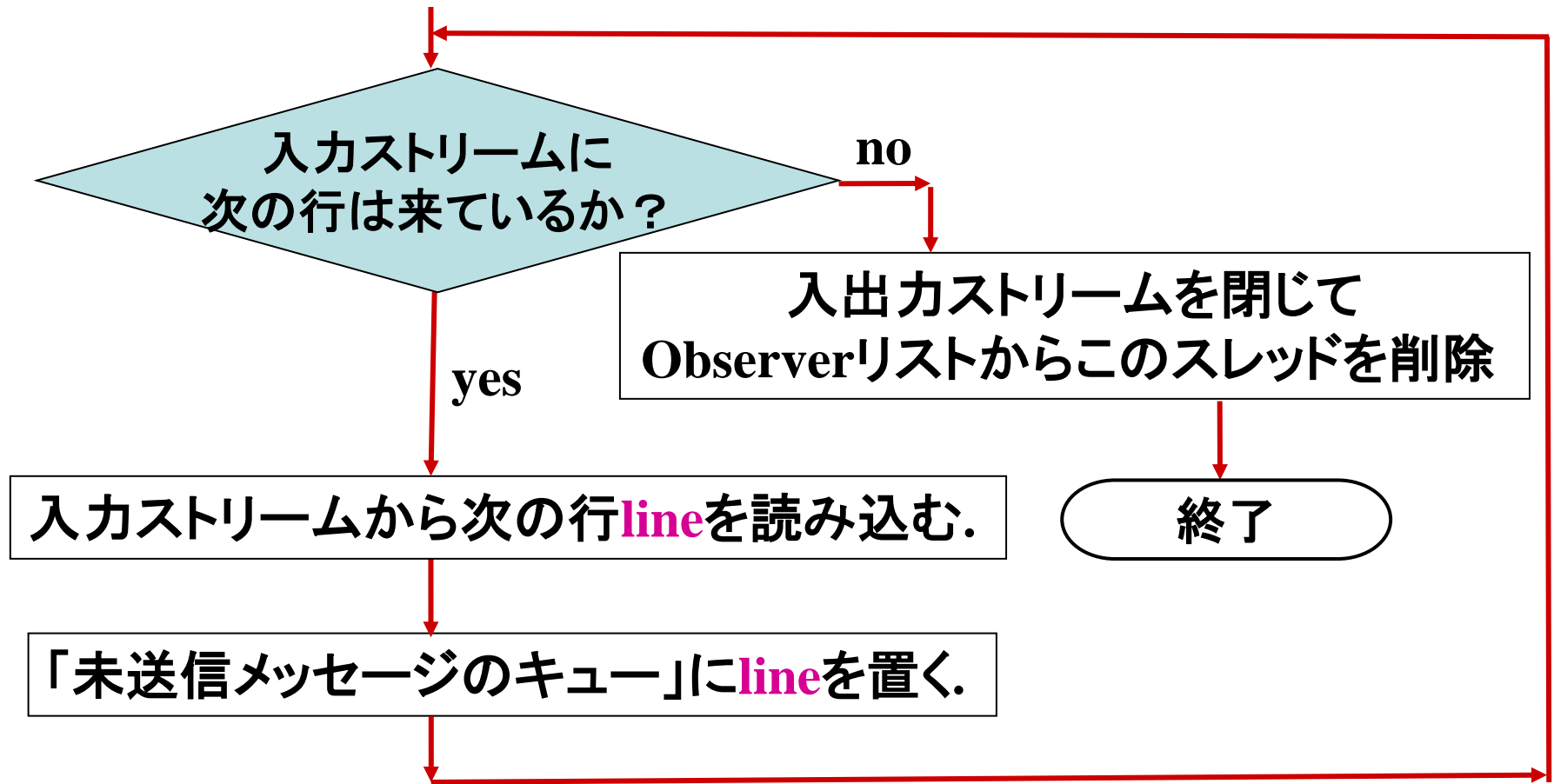


ソースコードは `BroadCaster.java` にある.

# 応対スレッド(Observer役)

Javaで**Observer役**のために**PropertyChangeListener**インターフェースが用意されている。**Thread**クラスを拡張して**PropertyChangeListener**を実装するクラスで応対スレッドを定義。

応対スレッドの**run**メソッド内での処理の流れ:



ソースコードは **Connection.java** にある。

# チャット・サーバーを作ろう

**概要:** チャットサーバーは、クライアント側から送られてきた各行をすべてのクライアントに転送する。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**メソッドで行う処理の流れ:

↓  
未放送メッセージのキューを生成

↓  
放送スレッドを生成して起動。

```
public void prepareForService( ) {  
    // 「未放送メッセージのキュー」を生成する. LinkedBlockingQueueクラスを使用している.  
    queue = new LinkedBlockingQueue<String>( );  
    // 「放送スレッド」を生成して起動する.  
    BroadCaster caster = new BroadCaster(queue);  
    new Thread(caster).start( );  
}
```

# チャットサーバーを作ろう(続)

**startService**メソッドで行う処理の流れ:

↓  
「新メンバーが入った」というメッセージを未放送メッセージキューに置く

↓  
「応対スレッド」を生成して「**Observerリスト**」に追加してから起動する.

```
public void startService(Socket s) throws IOException, InterruptedException{  
    // メッセージを「未放送メッセージのキュー」に置き, 「放送スレッド」にチャット・  
    queue.put("新メンバーが入った"); // クライアント全員に送ってもらう.  
  
    // 受け付けたばかりのクライアントに対応する「応対スレッド」を生成して,  
    // 「応対スレッドのリスト」に追加してから, 起動する.  
    Connection connection = new Connection(s, queue, caster); // 「応対スレッド」を生成  
    caster.addPropertyChangeListener(connection); // この応対スレッドを  
                                                // 「Observerリスト」に登録.  
    connection.start(); // 「応対スレッド」を起動する.  
}
```

# チャット・サーバーを表すクラス

```
public class ChatServer extends Server { //getterとsetterを省略
    private BroadCaster caster ; //放送スレッドを覚えるフィールド
    private BlockingQueue<String> queue ; //クライアントからの一文を覚えるため

    public ChatServer(int portNo) throws IOException, InterruptedException {
        super( portNo );
    }

    public void prepareForService( ) { ...(前述) }
    public void startService(Socket s) throws IOException, InterruptedException{...}

    public static void main( String [ ] args ) {
        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 ChatServer portNo");
            return ;
        }
        try {
            new ChatServer( Integer.parseInt(args[0]) ).startService( ) ;
        } catch (Exception e) {
            e.printStackTrace( ) ;
        }
    } //mainメソッドの終わり
} //ChatServer.javaの終わり
```

# チャットクライアントを作ろう

エコークライアントとほぼ同じなので、それを拡張。

違いは、クライアントのIDを覚えるためのフィールドidを新たに用意すること  
と下記のように **send**メソッドをoverrideすることだけ。

## **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。

@Override

```
public void send(PrintWriter p) throws IOException {  
    Scanner keyboard = new Scanner(System.in);  
    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( id + " > " + request); // 読み込んだ会話文をサーバーに送る。  
            if ( request.equals("BYE") ) break ;  
        }  
        } finally { disconnect( ) ; keyboard.close(); }  
} // sendメソッドの終わり
```

# チャット・クライアントを表すクラス

```
public class ChatClient extends echo.EchoClient { //getterとsetterを省略
    private String id ; //クライアントのID

    public ChatClient( ) { } //何もしないコンストラクタ.
    public ChatClient(String serverName, int portNo, String id) throws IOException{
        super( serverName, portNo );
        this.id = id ;
    } // 引数ありのコンストラクタの終わり

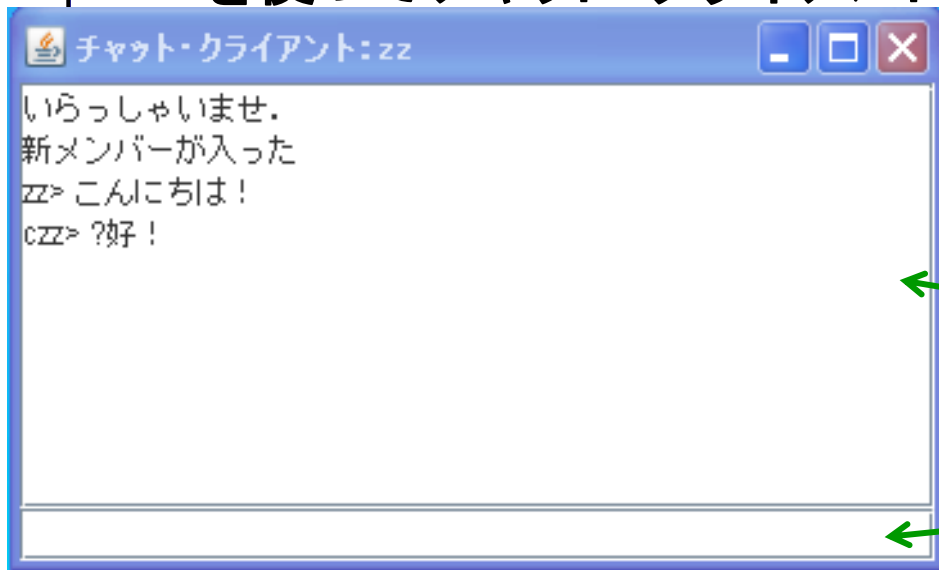
    @Override
    public void send(PrintWriter p) { ... (前述) }

    public static void main( String [ ] args ) {
        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 ChatClient(args[0], Integer.parseInt(args[1])).startService( ) ;
        } catch (Exception e) {
            e.printStackTrace( ) ;
        }
    } //mainメソッドの終わり
} //ChatClient.javaの終わり
```

ChatClient.java

# チャット・クライアントを見直そう

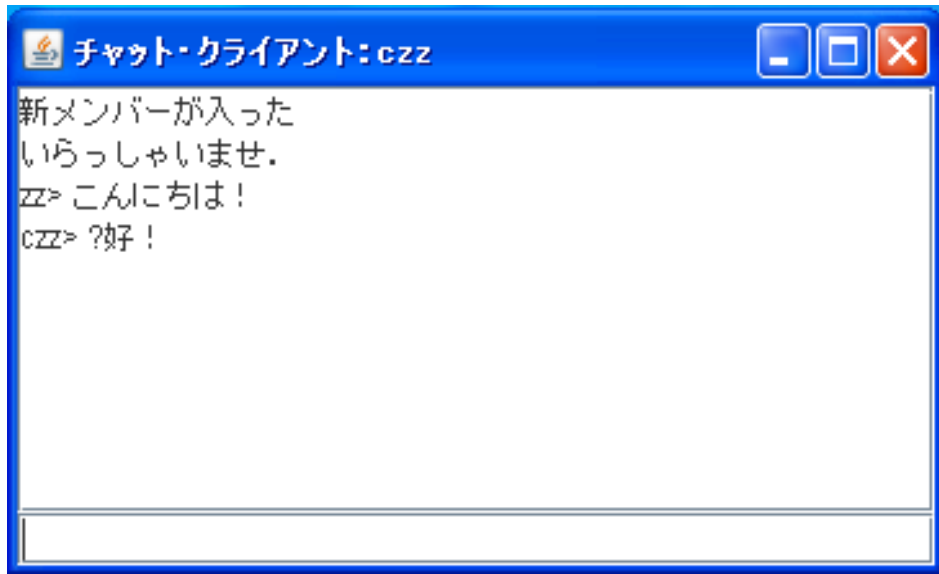
GUIを使ってチャット・クライアントを見やすくすることにする。



フレーム クライアントの  
会話ウィンドウ

テキスト  
・エリア サーバーから受け取った  
内容(会話の内容)を  
表示するテキストエリア

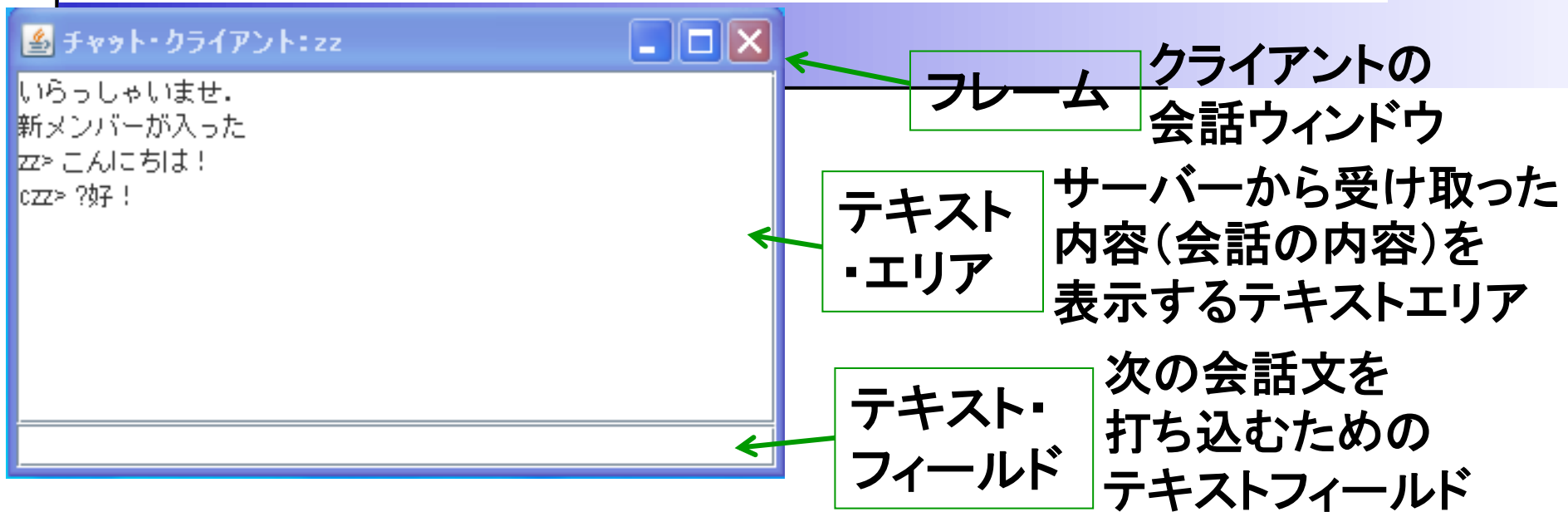
テキスト・  
フィールド 次の会話文を  
打ち込むための  
テキストフィールド






# チャット・クライアントを見直そう(続き)

GUIを使ってチャット・クライアントを見やすくすることにする。



 テキスト・フィールドに会話文を打ち込むと、その文は、「送信」を担当するスレッドによってサーバーに送られるようにしたい。

 テキスト・フィールドに **ActionListener** を付ける必要がある。

 テキスト・フィールドの **ActionEvent** を処理する「event dispatch thread」と「送信」を担当するスレッドはまさに、**生産者／消費者**の関係にある。

∴「次の会話文」の受け渡しにブロッキング・キューを使えばよい。

**Exchanger**を使う方法もある。

# チャット・ウィンドウの持つべき最低限機能

```
public abstract class ChatWindow extends JFrame {  
    private BlockingQueue<String> queue ; // サーバーに送る次の文を覚えるキュー。  
    public ChatWindow(String title, BlockingQueue<String> queue) { // コンストラクタ  
        super(title) ; // 親クラスのコンストラクタを呼び出してタイトルを設定する。  
        this.queue = queue ;  
    }  
  
    public void showMe( ) { // このウィンドウを表示するメソッド  
        // ... 部品の配置とリスナーの貼り付けは子供クラスで行う。  
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) ;  
        pack( ) ; // レイアウトに合わせてサイズを変更するように設定。  
        setLocationByPlatform(true); // 表示場所をプラットフォームに任せる。  
        setVisible(true); // フレームを開く。  
        setFocusInWindow( ) ; // フォーカスを要求する。  
    }  
  
    // ChatWindowが開いたときに最初にキーボードからの入力が必要な部品に  
    // フォーカスを設定するメソッド  
    public abstract void setFocusInWindow( ) ;  
  
    // サーバーからのメッセージを表示するメソッド  
    public abstract void append(String message) ;  
  
    public BlockingQueue<String> getQueue( ) { return queue ; } // getterメソッド  
}
```

ChatWindow.java

# 具体的なチャット・ウィンドウを表す単純なクラス

```
public class SimpleChatWindow extends ChatWindow {  
    private JTextArea area; // サーバーからのメッセージの表示用テキスト・エリア.  
    private JTextField field; // サーバーに送る次の文を打つためのテキスト・フィールド.  
  
    public SimpleChatWindow(String title, BlockingQueue<String> queue)  
    {    super(title, queue) ; // 親クラスのコンストラクタを呼び出す. }
```

@Override

```
public void showMe() {  
    arrangeComponents(); // 部品を生成してフレームに配置する.  
    addListeners(); // 一部の部品にリスナーを付ける.  
    super.showMe();  
}
```

次頁に続く...

```
public void arrangeComponents() { //部品を生成してフレームに配置するメソッド  
    setLayout( new BorderLayout() ); // フレーム内のレイアウトを設定.  
    area = new JTextArea(10,32); // 受信メッセージ表示用のテキストエリアを生成.  
    area.setLineWrap(true); //会話文の長さがテキストエリアの横幅を越えたら自動折り返し  
    area.append("いらっしやいませ!" + System.getProperty("line.separator"));  
    add("Center", new JScrollPane(area));  
    area.setEditable(false); //受信メッセージ表示用のテキストエリアを編集不可にする.  
    field= new JTextField(32); // 次の1文を打ち込むためのテキストフィールド.  
    add("South", field); //テキストフィールドをフレームの下側に配置する.  
} // 自作arrangeComponentメソッドの終わり
```

SimpleChatWindow.java

# チャット・ウィンドウを表す単純なクラス(続き)

```
public void addListeners() { //一部の部品にリスナーを付けるメソッド
    field.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
            String message = field.getText();
            try {
                getQueue().put(message); // 打ち込んだ1文をブロッキングキューに置く
                if ( message.equals("BYE") )
                    System.exit(0); // クライアントを終了する
            } catch (InterruptedException ex) { ex.printStackTrace(); }
            field.setText( "" ); // テキストフィールドの中身をクリアする.
            field.requestFocusInWindow(); // フォーカスを要求する.
        }
    }); // 無名クラスのインスタンス(リスナー)を生成してフィールドに付ける.
} // 自作addListenersメソッドの終わり

public void append(String message) { //サーバーからのメッセージを表示
    area.append(message + System.getProperty("line.separator")); //会話文を表示
    area.setCaretPosition(area.getText().length()); //キャレット位置を最後に移動
}

public void setFocustInWindow() { //必要な部品にフォーカスを設定するメソッド
    field.requestFocusInWindow(); // テキストフィールドにフォーカスを要求する
}
```

# チャット・クライアントを表す抽象クラス

```
package chat ;
import java.util.Scanner ;
import java.io.* ; // InputStreamクラスやOutputStreamクラスなどを使うので必要
import java.util.concurrent.* ; // BlockingQueueインターフェースなどを使うので必要
import java.awt.* ; // BorderLayoutクラスなどを使うので必要
import java.awt.event.* ; //(ActionEvent)クラスなどを使うので必要
import javax.swing.* ; // JFrameクラスなどを使うので必要

public abstract class AbstractChatClient extends ChatClient { //getterとsetterを省略
    private ChatWindow chatWindow // チャットウインドウを参照する変数.
    private BlockingQueue<String> queue ; // サーバーに送る次の文を覚えるキュー.

    public AbstractChatClient( ) { } //何もしないコンストラクタ

    public AbstractChatClient(String server, int portNo, String id) throws IOException {
        setSocket( new Socket(server, portNo) ); //サーバーに接続.
        setID( id ) ; //クライアントのIDをセットする
        queue = new ArrayBlockingQueue<String>(1) ; //1文を覚えるキューを生成.
    } // 引数ありのコンストラクタの終わり

    public void startService( ) throws IOException {
        EventQueue.invokeLater ( new Runnable( ) { public void run( ) {
            chatWindow = createChatWindow( ) ; // ここでfactoryメソッドを呼び出す
            chatWindow.showMe( ) ;
        } } ) ;
        try { super.startService( ) ; } catch (IOException e) { e.printStackTrace( ) ; }
    } // runメソッドの終わり
```

次頁に続く...

AbstractChatClient.java

@Override

```
public void send(PrintWriter p) throws IOException {
    try {
        while ( !getSocket().isClosed( ) ){//ソケットが閉じられていない間, 繰り返して送信.
            String request = queue.take(); //ブロッキングキューから次の会話文を読み込む
            p.println( getID() + "> " + request); // 読み込んだ会話文をサーバーに送る.
            if ( request.equals("BYE") ) break ;
        }
    } catch (InterruptedException ex) {
        disconnect( ) ; Thread.currentThread().interrupt( ) ;
    }
} // sendメソッドの終わり
```

チャット・クライアントを  
表す抽象クラス(続)

@Override

```
public void receive(Scanner scan) throws IOException {
    try {
        while ( scan.hasNextLine( ) ) { // 入力ストリームから会話文を繰り返して受信.
            String reply = scan.nextLine( ) ; //入力ストリームから次の会話文を読み込む
            EventQueue.invokeLater ( new Runnable( ) { public void run( ) {
                chatWindow.append(reply + System.getProperty("line.separator"));
            } } ) ;
        } catch (InterruptedException ex) { disconnect( ) ; }
    } // receiveメソッドの終わり
```

```
public abstract ChatWindow createChatWindow( ) ; //抽象factoryメソッド
```

AbstractChatClient.java



# チャット・クライアントを表す具象クラス

```
package chat ;
import java.io.IOException ;
public class SimpleChatClient extends AbstractChatClient {
    public SimpleChatClient( ) { } //何もしないコンストラクタ

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

    public void createChatWindow( ) {
        return new SimpleChatWindow("チャット・クライアント:" + getID( ), getQueue( ) ) ;
    } //具象factoryメソッド

    public static void main( String [ ] arg ) {
        if ( arg.length != 3 ) {
            System.out.println("Usage: java SimpleChatClient serverName portNo ID");
            return ; // 使い方が正しくないときに、プログラムを終了する。
        }
        try {
            new SimpleChatClient(arg[0], Integer.parseInt(arg[1]), arg[2]).startService();
        } catch (Exception ex) { ex.printStackTrace( ) ; }
    } // mainメソッドの終わり
} // SimpleChatClient.javaの終わり
```

SimpleChatClient.java

# Factory Methodデザインパターン

**ポイント:** インスタンスの作り方を親クラスで定めるが、具体的なクラス名までは定めず、具体的な肉付けはすべて子クラスで行う手法。

**Product**(製品役): このパターンで生成されるインスタンスが持つべきインターフェース(API)を定める抽象クラス. 具体的な内容は、子クラスである**ConcreteProduct**役が定める. 例では、**ChatWindow**クラスがこの役をつとめている.

**ConcreteProduct**役(具体的製品役): **Product**役で定めた抽象メソッドを実装したクラス. 例では、**SimpleChatWhindow**クラスがこの役をつとめている.

**Creator**(作成者役): **Product**役を生成する抽象クラス. 具体的な内容は、子クラスである**ConcreteCreator**役が定める. 例では、**AbstractChatClient**クラスがこの役をつとめている.

**ConcreteCreator**役(具体的作成者役): **Creator**役で定めた抽象メソッドを実装したクラス. 例では、**SimpleChatClient**クラスがこの役をつとめている.



# Factory Methodデザインパターン(続)

このパターンでは、生成するインスタンスの種類が多くなりそうな場合に、インスタンス生成の責務を外に切り出して、保守性を高める。

Factory Methodデザインパターンの**使うべき場合**：

インスタンスの種類が将来的に増えそうな場合や、  
コンストラクタではない場所でインスタンスの生成を行いたい  
場合(例:コンストラクタを直接呼ばせたくはないが、インスタンスは  
作成したい時)や、  
ポリモーフィズムを使いたいのだが、インスタンスの生成は共通化  
したい場合(例:我々のチャットクライアント)や、  
あるオブジェクトに関連するオブジェクトを、自身で作成させたい  
場合(例:クラスAのオブジェクト1はクラスBのオブジェクト1を  
必要とし、クラスAのオブジェクト2はクラスBのオブジェクト2を  
必要とするとき、Factory Methodパターンを使うと、クラスBの  
オブジェクト1・2を作成する責務を持つのは、それぞれクラスAの  
オブジェクト1・2ということになる)、  
など。

# 演習課題

`ChatServer.java`クラスを継承して、サーバーを起動すると、下記を満たすフレーム(モニタ画面)が画面の中央に表示されるようにせよ。

- ① フレームに「テキストエリア」と「停止ボタン」を配置する。
- ② 「テキストエリア」に接続を許可した時間の一覧を表示する。
- ③ 「停止ボタン」が押されると、現在接続中のクライアント全体に終了通知を送ってからサービスを終了する。

できれば、チャットサーバーのモニタ画面の持つべき最低限の機能を見出して抽象クラスを定義し、それを継承した具象クラスを使ってチャットサーバーの実際のコントロール画面を定めよ。

