Chat-TCP-UDP

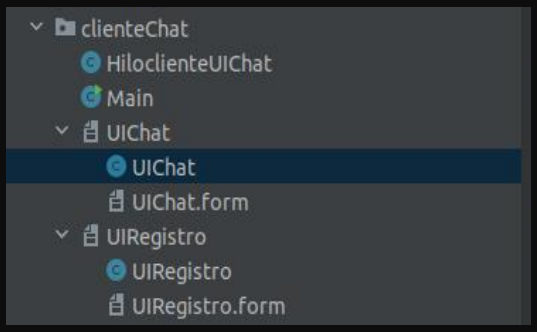
Alejandro Morón Turiel

Tabla de contenido

Pasos previos:

En esta practica se mostrara la estructura y creación de una base de un Chat utilizando las arquitecturas TCP y UDP respectivamente , en esta memoria se enseñaran los diversos fragmentos de código , así como la funcionalidad de los mismos y el motivo del uso de los mismos.

En este caso se han decidido crear 2 bloques principales de código dentro del proyecto , donde estarán guardados todos los datos , así como las clases referentes a un bloque del programa en concreto.



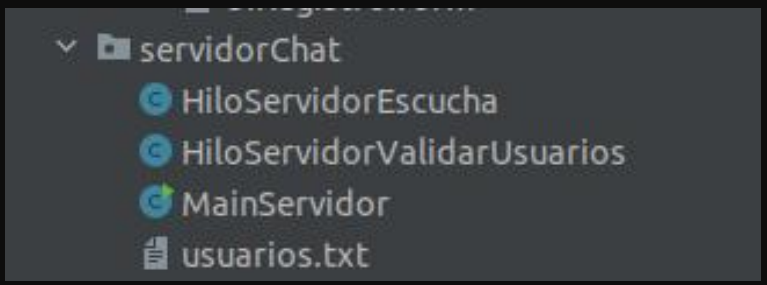
En el bloque **clienteChat**: dentro de este bloque se encuentran todos los datos necesarios para el correcto funcionamiento del cliente , que sera el encargado de gestionar los datos recibidos por el servidor y de enviar los mismos.

Donde podemos ver 2 directorios administrativos los cuales albergan:

-> UIChat: consiste en el directorio donde estarán los elementos encargados de gestionar las comunicaciones dentro del chat

->UIRegistro: consiste en el directorio donde estarán los elementos encargados de gestionar la entrada con una conexión individual con el servidor.

En el bloque **ServidorChat**: dentro de este bloque se encuentran todos los datos necesarios para el correcto funcionamiento del servidor , que sera el encargado de gestionar los datos recibidos por el cliente y de enviar los mismos.

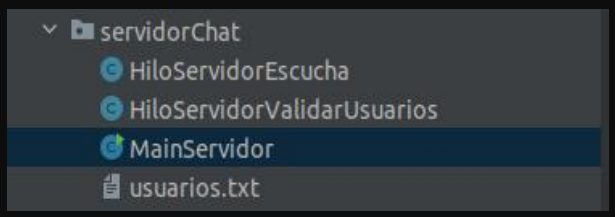


También existe internamente una función la cual gestionara los usuarios que conoce el servidor para así autenticar los usuarios que quieran entrar al servidor por parte de los clientes.

Todos los datos usados en este proyecto se pueden obtener siguiendo el siguiente enlace: https://github.com/alejandroMortur/Chat-TCP

ServidorChat:

Dentro del bloque del ***ServidorChat*** se podrán apreciar múltiples clases, segregadas por funcionalidad , a continuación se expondrán el motivo y función de las mismas , contando con las siguientes clases:



MainServidor : dentro de esta clase de código se encuentran los lanzadores necesarios para cada parte responsable del servidor que se pueden subdividir en los siguientes componentes:

Bloque de escucha UDP:

Aunque parezca contraproducente antes de cualquier conexión directa con el servidor por vía TPC, en el arranque del servidor se lanzara un único hilo donde se escucharan los mensajes ***multicast*** del servidor, esto se hace por la naturaleza de la conexión TCP que se lanzara posteriormente , el funcionamiento es simple, debido a que se requiere de un ***login*** previo entre el cliente y el servidor, para poder redireccionar al cliente aportándole el grupo y el puerto ***multicast*** al que añadirse tras el ***login,*** debe existir previamente.

```
ServerSocket serverSocket = new ServerSocket(PuertoServer);
System.out.println("\n----------------------------------------\n");
System.out.println("Servidor TCP esperando conexiones en el puerto: " + PuertoServer);
System.out.println("\n----------------------------------------\n");

Thread hiloClienteEscuchar = new Thread(new HiloServidorEscucha());
hiloClienteEscuchar.start();
```

Bloque de escucha TCP:

Una vez sabiendo que obligatoria mente es necesario crear el canal ***multicast*** vació, posteriormente ya se crearan la conexiones individuales entre el cliente y el servidor para realizar el ***login*** de credenciales.

```
    while (true) {

        Socket socketCliente = serverSocket.accept();
        // Crear un nuevo hilo para manejar la conexión con el cliente
        Thread hiloCliente = new Thread(new HiloServidorValidarUsuarios(socketCliente));
        hiloCliente.start();

    }

} catch (IOException e) {

    System.out.println("Error en el servidor: " + e);

}
```

Este sistema es así debido a la naturaleza de la comunicación, que al ser sensible , no es propio que la información circule por un medio mas masivo como puede ser un ***multicast***, ademas de que a futuro sera mas fácil aplicar medidas de seguridad extra a este medio de comunicación como cifrado u otros medios.

HiloServidorValidarUsuarios :

Dentro de esta clase de código se encuentran todos los medios necesarios para gestionar las credenciales entre cliente y servidor , donde se validara si el usuario y contraseña son correctos y están registrados en un fichero TXT , que para esta prueba tiene una serie de usuarios guardados por defecto.

Bloque de Variables:

En este bloque de código se albergan todas las variables así como elementos compartidos, usados dentro del código gestor de credenciales, consta de:

```
4 usages
private Socket socketCliente;
private byte[] buffer = new byte[1024];
5 usages
private static String nombreCliente = "";
1 usage
private static String archivo = "./src/servidorChat/usuarios.txt";


1 usage  alex
public HiloServidorValidarUsuarios(Socket socket) {

    this.socketCliente = socket;

}
```

->Un socket encargado de gestionar la conexión de tu a tu con el cliente.

->Un buffer encargado de guardar los datos enviados por el cliente

->Un string encargado de guardar el nombre y contraseña aportados por el cliente

->La URL de ruta relativa del fichero donde la clase leerá el contenido para comprobar si el usuario existe o no.

Bloque escuchador:

Dentro de este bloque de código existen todos los elementos necesarios para leer los mensajes enviados al servidor por parte del cliente, debido a esto y sabiendo que solo se enviara un paquete que contiene las credenciales del usuario, con una sola sentencia de ***BufferedReader y PrintWriter*** sera más que suficiente, también teniendo en cuenta que es el cliente quien cierra la conexión en caso de superar un cierto numero de intentos, no sera necesario un bucle.

```
BufferedReader lector = new BufferedReader(new InputStreamReader(socketCliente.getInputStream()));
PrintWriter escritor = new PrintWriter(socketCliente.getOutputStream(), autoFlush: true);

// Autenticación del cliente (puedes modificar esta parte según tus necesidades)
nombreCliente = lector.readLine();
String contraseñaCliente = lector.readLine();
```

Posteriormente, se leerán los datos aportados por el cliente donde estarán las credenciales a verificar si son correctas o no.

Posteriormente y una vez se tienen las credenciales guardadas y registradas, dichas credenciales se le pasaran a un método el cual sera el encargado de recorrer y comprobar si efectivamente existen dichas credenciales ***EN LA MISMA LINEA*** , esto es importante debido a que se considera que a cada usuario le pertenece una linea en el fichero

```
if (nombreCliente != null && buscarCredencialesEnArchivo(archivo, nombreCliente,contraseñaCliente) &&
        contraseñaCliente != null) {

    System.out.println("\n-----------------------------------------------------------\n");
    System.out.println("\nUsuario autenticado: " + nombreCliente);
    System.out.println("\nContraseña correcta: " + contraseñaCliente);
    System.out.println("\n-----------------------------------------------------------\n");

    escritor.println("Autenticación exitosa");
```

Posteriormente y siempre y cuando sean correctas ambas lineas permitirá al programa continuar, este mostrara un ***log*** de que el usuario se a autenticado correctamente, que en este caso sale por terminal, aunque de desearlo podría guardarse en un fichero.

Una vez que el usuario se a verificado que es quien dice ser, para que el cliente entienda que la verificación ha sido exitosa , como código interno se le mandara un entero con el valor 50, si el cliente recibe este entero sabrá que se puede conectar correctamente al grupo ***multicast.***

```
escritor.write( c: 50);

String mensaje = lector.readLine();
if (mensaje.startsWith("CONNECT")) {

    String nombreUsuario = mensaje.split( regex: " ")[1];

    System.out.println(nombreUsuario);

    System.out.println("\n-----------------------------------------------------------\n");
    System.out.println(mensaje+" correctamente");
    System.out.println("\n-----------------------------------------------------------\n");

}
```

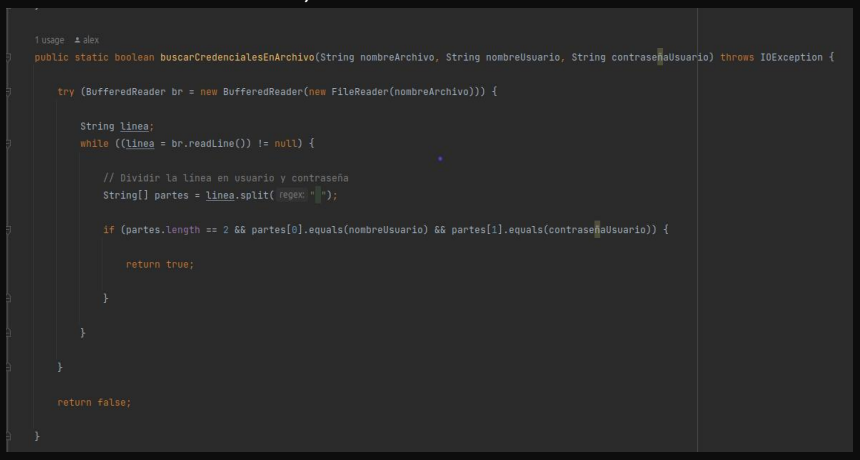
También como medio de gestión para poder la correcta entrada del cliente al grupo ***multicast,*** se verifica si este a enviado un mensaje con la muletilla ***"connect",*** ya que el cliente lo envía tanto al grupo como por el canal privado TCP, de ser cierto (que siempre lo sera) muestra otro mensaje de confirmación y finalmente cierra el ***socket*** ya que este no tendrá ningún uso mas.

En caso de algún fallo o en caso de que las credenciales no son correctas , se imprimirá por consola un mensaje de error en las credenciales como no se pasa al interior de la condición ***if***, se podría volver a intentar otra verificación, así seria hasta que el cliente se canse o se desconecte, que en ese caso lo trata la ***exception***

```
    } else {

        System.out.println("\n-----------------------------------------------------------\n");
        System.out.println("\nError de autenticación para el usuario: " + nombreCliente);
        System.out.println("\n-----------------------------------------------------------\n");

        escritor.println("Error de autenticación");
    }

} catch (IOException e) {

    System.out.println("Error: "+e);

}
```

Bloque lectura fichero:

Dentro de este código se gestiona la lectura del fichero de texto, donde se buscaran los usuarios dentro del servidor.



Para ello es necesario retirar con la sentencia Split() los posibles huecos que pudiesen albergar el fichero en cada linea.

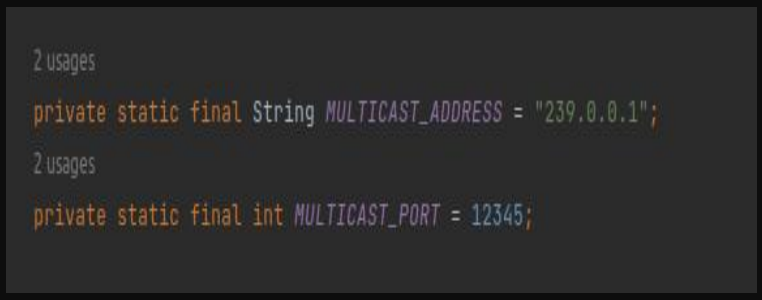
Una vez limpiada la linea se comprueba si efectivamente coincide los las credenciales aportadas por el usuario , de serlo devolverá un valor booleano verdadero, de no serlo a pesar de recorrer todo el fichero devolverá un valor booleano falso ***(contando estos valores como que esta registrado o no en el fichero)***

HiloServidorEscucha :

Dentro de esta clase de código se encuentran todos los medios necesarios para gestionar las comunicaciones internas entre la red multicast y los clientes, donde podemos destacar los siguientes bloques.

Bloque de Variables:

En este bloque de código se albergan todas las variables así como elementos compartidos, usados dentro del código gestor de credenciales consta de:

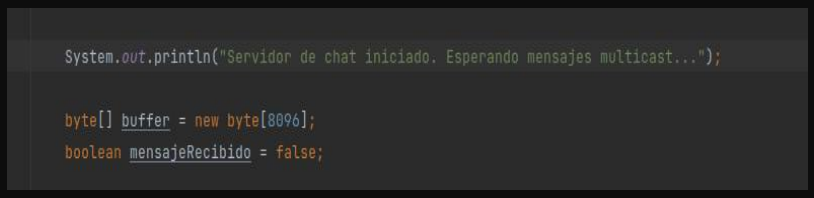


->Un String que guarda la ip del grupo multicast

->Un entero que indicara el puerto al que se deben conectar los clientes.

->Un Buffer encargado de guardar los datos de los mensajes

->Un booleano encargado de gestionar cuando se ha recibido un mensaje (para evitar duplicados)



Bloque escuchador:

Una vez que se han generado todos los medios necesarios para la correcta lectura del grupo ***multicast***, sera necesario permanecer a la escucha de forma indefinida , para ello se ha hecho uso de un bucle ***while***, donde de forma reiterada se obtienen paquetes de la red ***multicast***.

```
boolean mensajeRecibido = false;

while (true) {
    // Recibir información multicast
    DatagramPacket paquete = new DatagramPacket(buffer, buffer.length);
    redBroadcast.receive(paquete);
```

Bloque gestor mensajes:

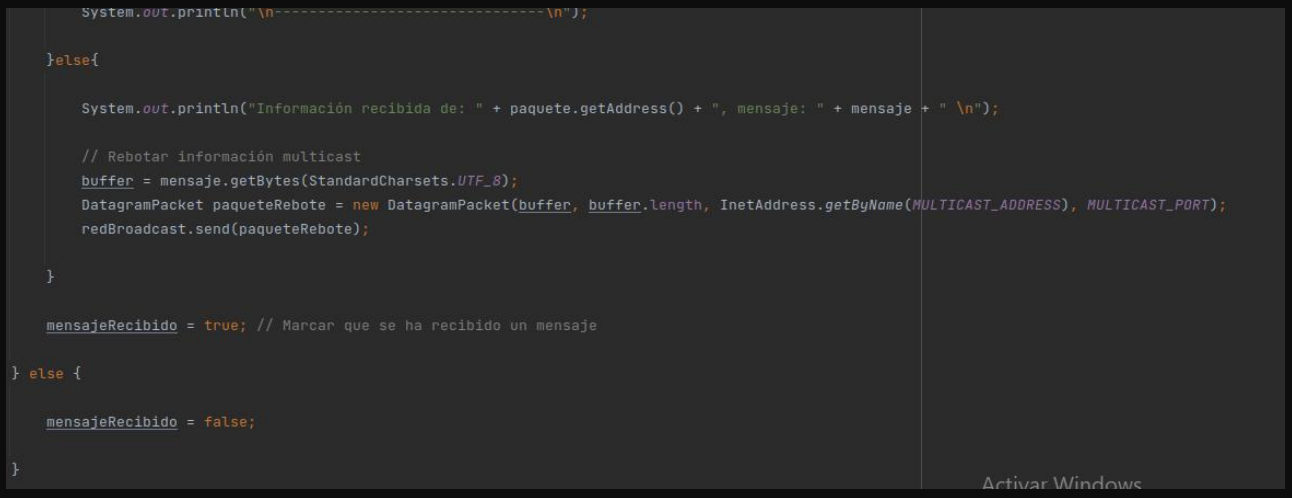
Una vez que se ha recibido el paquete gracias a un ***booleano*** se comprueba si este no se ha repetido por error , esto se hace debido a la manipulación del ***booleano*** que debe de ser distinta a verdadero para que se considere un mensaje como nuevo.

De ser este considerado como uno nuevo, se saca su contenido, para acto seguido gestionar si este contiene alguna “muletilla” , en caso de contener alguna de las 2 posibles ***(offline y online)*** se considera la ***desconexión y conexión*** explicita en cada caso, mostrando por terminal la salida correspondiente con sus datos.

```
if (!mensajeRecibido) {

    int length = paquete.getLength(); // Obtener el tamaño real del mensaje recibido
    String mensaje = new String(paquete.getData(),  offset: 0, length, StandardCharsets.UTF_8);

    if (mensaje.contains("offline")) {

        System.out.println("\n------------------------------\n");
        System.out.println(mensaje);
        System.out.println("\n------------------------------\n");

    }
    if (mensaje.contains("online")){

        System.out.println("\n------------------------------\n");
        System.out.println(mensaje);
        System.out.println("\n------------------------------\n");
```

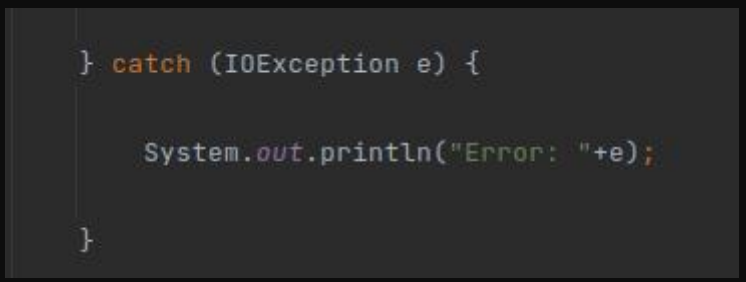
Aunque de todas maneras , de mostrar los mensajes de log anteriormente citados, aun asi sera necesario gestionar dicho mensaje para que el servidor cumpla su función de ***multicast,*** independientemente del mensaje.

Una vez que recibe el mensaje y tras verificar que es nuevo , se mostrara los datos del mensaje enviado, así como quien lo ha hecho por ip dentro de la red, tras hacerlo con un ***buffer*** se generara un nuevo paquete a partir del anterior y se reenviara dicho paquete al resto de la red, es en este momento cuando el ***booleano*** se setea a verdadero, para evitar que en el siguiente ciclo del bucle ***while*** trate de nuevo un mensaje que ya ha tratado anteriormente.



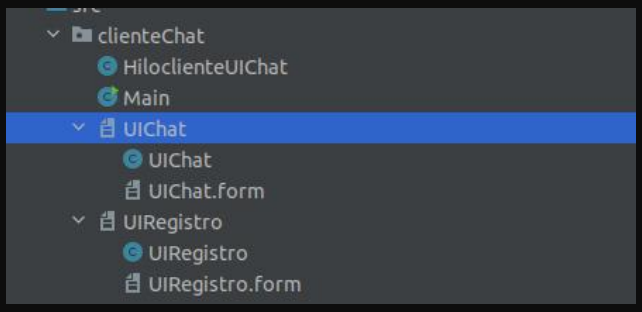
Como el mensaje al ser reenviado por ***multicast*** lo vuelve a recibir otra vez el servidor , gracias al ***booleano*** se evitara que lo use de nuevo, forzando al servidor a saltar a la linea donde se setea el ***booleano*** a false otra vez, para evitar que el siguiente mensaje ***QUE SI ES NUEVO***, se descarte cuando no se debe hacer.

En caso de algún fallo ocurra con la lectura del fichero , se trata la ***exception*** con un mensaje de error para que en todo momento se tenga consciencia de que ha ocurrido internamente en el servidor



ClienteChat:

Dentro del bloque del ***ClienteChat*** se podrán apreciar múltiples clases, segregadas por funcionalidad , a continuación se expondrán el motivo y función de las mismas , contando con las siguientes clases:



->Bloque main del cliente: consiste en la clase principal encargada de gestionar el orden de llamada de todos los medios necesarios dentro del cliente del chat.

->Bloque UIregistro: consiste en todos los medios necesarios para gestionar, el ingreso correcto de un usuario en el servidor, es aquí donde se realiza una conexión TCP directa entre el cliente y el servidor

->Bloque UiChat: consiste en todos los medios necesarios para gestionar el funcionamiento interno de la interfaz gráfica del servidor, aunque no de forma directa existe otro fichero también ligado a este bloque, el archivo "hiloclienteUiChat" encargado de gestionar la escucha y el envió de la información hacia al servidor.

MainCliente :

Dentro del código del fichero main, aprovechando su lanzador es donde se crearan los medios comunes menos específicos dentro del programa del cliente, entre estos incluye elementos como: ***el socket, las interfaces y sus configuraciones , diversas variables....***

Este código se subdivide en diversos puntos:

Bloque de Variables:

En dicho bloque se encuentran todas las variables necesarias para el correcto funcionamiento de las interfaces gráficas, puesto que algunas de ellas gestionan conexiones y demás datos, se almacenan en esta clase para tenerlos más centralizados, contando con los siguientes:

```
//variables gestoras conexión TCP Cliente-Servidor
1 usage
private static final String ipServer = "127.0.0.1";
1 usage
private static final int puertoServer = 6001;

//variables gestoras registro y UI del chat
9 usages
private static UIRegistro UIRegistro = null;
9 usages
private static UIChat UI = null;

//variables gestoras grupo multicast
2 usages
private static final String ipMulticast = "239.0.0.1";
2 usages
private static final int multicastPort = 12345;
```

->Un String que guarda la ip del grupo multicast y de la ip del server

->Un entero que indicara el puerto al que se deben conectar los clientes tanto en el multicast como en el TCP nominal.

->Los objetos interfaces necesarios para la correcta gestión de la interfaz gráfica del cliente.

```
//variables gestoras conexión TCP Cliente-Servidor
1 usage
private static final String ipServer = "127.0.0.1";
1 usage
private static final int puertoServer = 6001;

//variables gestoras registro y UI del chat
9 usages
private static UIRegistro UIRegistro = null;
9 usages
private static UIChat UI = null;

//variables gestoras grupo multicast
2 usages
private static final String ipMulticast = "239.0.0.1";
2 usages
private static final int multicastPort = 12345;
```

En Posteriormente y a necesidad se irán entregando dichos valores a las partes necesarias desde el bloque main.

Bloque lanzamiento interfaz registro:

Dentro de este bloque se prepara con las variables anteriores todos los elementos necesarios para un lanzamiento correcto de la interfaz de usuarios, en este caso, al necesitar leer los paquetes enviados por el servidor en una conexión TCP, primeramente se creara un sockets y unos inputs y outputs,

```
// Crear el socket TCP y obtener los streams de entrada y salida
Socket clienteSocket = new Socket(ipServer, puertoServer);
InputStream input = clienteSocket.getInputStream();
OutputStream output = clienteSocket.getOutputStream();
```

Los cuales posteriormente se le entregaran como argumentos de constructor a la clase encargada de gestionar esta interfaz, para así evitar repetir el mismo código.

```
// Interfaz de registro de usuario
UIRegistro = new UIRegistro(clienteSocket, input, output);
UIRegistro.setContentPane(UIRegistro.panel_Registro);
UIRegistro.setTitle("Entrada chat");
UIRegistro.setResizable(false);
UIRegistro.setSize( width: 400,  height: 500);
UIRegistro.setVisible(true);
UIRegistro.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

También sera en este punto donde se adecuara el ***Frame*** de la interfaz gráfica para que el usuario pueda hacer uso del mismo, seteando valores como: ***el tamaño de ventana, la visibilidad, el nombre....***

Bloque de errores:

En todo momento se tienen en cuenta las posibles excepciones que se pueden generar durante la primera conexión del servidor, ante esto y un posible fallo , se ha rodeado todo el primer bloque de código con una expresión ***try-catch*** que en caso de error mostrara un mensaje al usuario para darle a entender que debe reiniciar el programa para probar a conectarse de nuevo.

```
} catch (IOException e) {

    //en caso de no poder conectarse al servidor mensaje de error
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Error al conectarse al servidor\n\nReinicie el programa y intentelo de nuevo",  title: "Error de c
    System.out.println("Error: " + e);

}
```

Bloque lanzarChat:

Forma parte del main de la clase, esta implementación de código es usada, una vez que el cliente certifica que correctamente el usuario a aportado las credenciales correctas , antes de finalizar el código de la interfaz se llama a este trozo de código, donde se le aportara una string, ***esta string es la encargada de aparecer posteriormente como el nombre del usuario aceptado con el que hemos realizado la conexión***, apareciendo en diversos puntos como:

->Una string dentro de la interfaz gráfica que recuerda en todo momento con quien nos hemos conectado.

->La composición de diversos mensajes

Es aquí donde primeramente como en el caso anterior se le aportaran los medios necesarios para que la clase funcione correctamente creando elementos como el grupo al que ya esta conectado el cliente:

```
// Crear un nuevo socket multicast y unirse al grupo multicast
MulticastSocket redBroadcast = new MulticastSocket(multicastPort);
InetAddress grupo = InetAddress.getByName(ipMulticast);
redBroadcast.joinGroup(grupo);

// Cerrar la ventana de registro
UIRegistro.dispose();

// Inicializar la interfaz de chat
UI = new UIChat(nombre, redBroadcast);
UI.setContentPane(UI.panel_chat);
```

Posteriormente y teniendo en cuenta que el gestor de la interfaz gráfica del cliente no es el encargado directo de la ***lectura de y escritura de paquetes*** en el grupo multicast, si no que solo gestiona los eventos y medios de la interfaz, sera necesario crear por detrás un hilo que si que sera el encargado de gestionar todo esto.

```
//lanzamiento hilo gestor UI del chat
HiloclienteUIChat hiloClienteUIChat = new HiloclienteUIChat(UI, redBroadcast,ipMulticast ,multicastPort);
Thread hilo = new Thread(hiloClienteUIChat);
hilo.start();
```

Tal y como se hizo con el anterior lanzamiento, por medio del constructor se aportaran todos los objetos necesarios , para que dicha clase funcione sin tener que escribirlos de nuevo, posteriormente se adapta como un hilo y se lanza el mismo para asi poder recibir y enviar mensajes correctamente.

Bloque de errores:

En todo momento se tienen en cuenta las posibles excepciones que se pueden generar durante la segunda conexión del servidor, ante esto y un posible fallo , se ha rodeado todo el primer bloque de código con una expresión ***try-catch*** que en caso de error mostrara un mensaje al usuario para informarle de lo sucedido.

```
} catch (IOException e) {

    //En caso de error al conectarse al grupo multicast, mensaje de error
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Error al unirse al chat: " + e.getMessage(),  title: "Error de conexión", JOptionPane.ERROR_MESSA
    System.out.println("Error al unirse al chat: " + e.getMessage());

}
```

Bloque de errores:

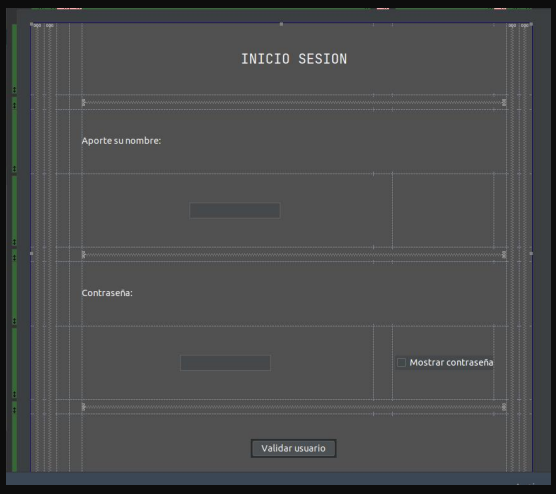
Sabiendo que ya estan creadas tanto el hilo como la interfaz gráfica , finalmente ambas necesitan de un método intermedio para poder finalizar sin problemas sus funciones.

```
1 usage   alex +2
public static void detenerhilo(){

    //en caso de que el cliente cierre la ventana de la UI, forzar cese hilo
    HiloclienteUIChat.detener();

}
```

Teniendo en cuenta que la interfaz gráfica sera la encargada de gestionar de manera visual el cierre del programa , sera esta la encargada de hacer uso de este trozo de código , con esto se evitara tener problemas con el cierre de hilos a la hora de cerrar la aplicación.

Este mismo llamara a una función interna del hilo la cual cesara la actividad del mismo

UIRegistro :



Dentro del código de este apartado se encuentran todos los puntos necesarios para que el usuario pueda interactuar directamente con el servidor, esta compuesta de ***2 campos donde se aportaran el nombre y la contraseña del usuario, un botón , encargado de gestionar el envió directo al servidor y un checkbox, que sera el encargado de gestionar el campo de la contraseña para ocultar o no los dígitos del mismo , para mayor seguridad.***

Este código se subdivide en diversos puntos:

Bloque de Variables:

En dicho bloque se encuentran todas las variables necesarias para el correcto funcionamiento de la interfaz gráfica, siendo necesarias para la comunicación efectiva con el servidor en lo que a credenciales se refiere

```
//-------variables locales:---------
1 usage
private Socket clienteSocket = null;
1 usage
private InputStream input = null;
1 usage
private OutputStream output = null;
2 usages
private int intentos = 3;

1 usage  alex +1
public UIRegistro(Socket clienteSocket, InputStream input, OutputStream output) {

    //Aprovechado de elemento bloque main
    this.clienteSocket = clienteSocket;
    this.input = input;
    this.output = output;
```

->Un Socket con el cual el cliente se comunicara con el servidor directamente.

->Un entero que indica el numero de intentos que tiene el usuario antes de bloquearse el formulario de acceso

->Los objetos necesarios para enviar y recibir los paquetes del servidor (input y output)

Dichos valores como se indico anteriormente seran tomados como argumentos de constructor

Bloque del constructor:

En dicho debido a la naturaleza del formulario , es donde residen los gestores de eventos del propio formulario, ya que como es lógico, es interesante que al crear la instancia de la interfaz gráfica y por ende su constructor, ya este implícito la creación de dichos eventos.

Debido a esto es en el constructor donde se generara todo el código necesario para que dichos eventos funcionen , creando un total de 2 eventos:

-> un evento gestor del botón de registro

-> un evento encargado del ocultamiento del contenido del texto de la contraseña

Donde nuestro checkbox es el elemento ***"mostrarContraseña"*** el cual esta definido en la interfaz gráfica, el funcionamiento del evento es simple, este comprueba si esta seleccionado o chekeado el mismo, de estarlo cambia todos los caracteres a propiamente caracteres mostrando la contraseña si estaba ocultada, en el caso contrario, de no estarlo marcado el checkbox, ocultara los caracteres sustituyéndolos con un simple "*"

```
alex +1
mostrarContraseña.addActionListener(new ActionListener() {
    alex +1
    @Override

    public void actionPerformed(ActionEvent e) {

        // Verificar si la casilla de verificación está marcada
        if (mostrarContraseña.isSelected()) {

            // Si está marcada, mostrar el campo de contraseña como un JTextField
            campo_contraseña.setEchoChar((char) 0); // Mostrar caracteres

        } else {

            // Si no está marcada, mostrar el campo de contraseña como un JPasswordField
            campo_contraseña.setEchoChar('*'); // Ocultar caracteres

        }

    }
});
```

Otro evento importante consta del evento de envió de las credenciales al darle a un botón ***"pulsa_registro",*** el cual estará ligado a este botón, una vez presionado , de los 2 campos se sacara su contenido el cual sera guardado en una variable individual para cada campo.

```
alex +1
pulsa_registro.addActionListener(new ActionListener() {
    alex +1
    @Override

    public void actionPerformed(ActionEvent e) {

        String nombre = campo_nombre.getText();
        String contraseña = campo_contraseña.getText();
```

Posteriormente se comprueba ***si ambas cadenas están escritas y no están vacías*** , de ser correcto este hecho , posteriormente se comprueba si los intentos por arranque del cliente aun son superiores a cero, de serlo se procede a enviar las credenciales formateadas de una forma concreta, la cual el servidor sabe interpretar.

Este formato consta de las credenciales separadas por un espacio entre medias de las mismas, cuando estas son enviadas , posteriormente se espera el código del servidor, que de ser uno concreto significara que el usuario y contraseña existen en el servidor:

```
//comprueba si las entradas no estan vacias
if (comprobarEntrada(nombre) && comprobarEntrada(contrasena)) {

    try {

        //gestiona hasta un total de 3 intentos antes de bloquear el programa por fallo credenciales 3 -> 2 -> 1 -> 0
        if (intentos >= 0) {

            // Envío de datos al servidor con un separador
            output.write((nombre + "\n" + contrasena + "\n").getBytes());

            // Espera la respuesta del servidor
            int respuesta = input.read();

            System.out.println(respuesta);
```

En caso de que el código sea el que el cliente entiende como una verificación correcta , se encargara de limpiar los campos por donde obtuvo las credenciales, posteriormente mostrara por terminal un mensaje de confirmación para certificar que el servidor a aceptado las credenciales correctamente.

```
System.out.println(respuesta);

//si la respuesta del servidor es el 65 , las credenciasles con correctas
if (respuesta == 65) {

    //limpieza de campos
    campo_contrasena.setText("");
    campo_nombre.setText("");

    //mensaje de confirmación por terminal
    System.out.println("\n-----------------------------------------\n");
    System.out.println("CREDENCIALES ACEPTADAS POR EL SERVIDOR");
    System.out.println("\n-----------------------------------------\n");

    //enviar mensaje confirmación conexión
    String mensaje = "CONNECT " + nombre;
    output.write(mensaje.getBytes());

    // Lanzar el chat del usuario
    lanzarChat(nombre);

    //Cierra la conexión TCP por desuso
    clienteSocket.close();
```

Posteriormente sabiendo que el servidor necesita una confirmación el también para cercionar que se va a desconectar, el cliente manda un mensaje que posteriormente es re aprovechado como saludo para la entrada en el chat multisockets, de esta manera el servidor se entera que esta por desconectarse y ademas certifica que ha entrado en el grupo correctamente, finalmente antes de cerrar el socket***, hace uso de un método interno el cual pondrá a punto el lanzamiento de la otra interfaz del chat.***

En todo momento se tiene en cuenta si el usuario falla tanto la contraseña como el nombre del usuario a la hora de hacer login, que en ese caso como es propio no solo se le muestra un mensaje de su equivocación si no que también se le restaran intentos

```
        clienteSocket.close();

    } else {

        //en caso de fallar el intento , se resta un turno al intento
        --intentos;

        //Se muestra mensaje de error
        JOptionPane.showMessageDialog(pulsa_registro,  message: "Error: Usuario o contraseña incorrectos");

        //Confirmación por terminal
        System.out.println("\n-----------------------------------------\n");
        System.out.println("CREDENCIALES NO ACEPTADAS POR EL SERVIDOR");
        System.out.println("\n-----------------------------------------\n");

    }
```

En el fatídico caso en el que , el usuario consume todos los intentos sin básicamente acertar, también se le muestra un mensaje certificando que se ha bloqueado la aplicación por motivos de seguridad y que debe reiniciar el programa.

```
    } else {

        //En caso de fallar todos los intentos , mensaje de error
        JOptionPane.showMessageDialog(pulsa_registro,  message: "Error: Ha alcanzado el máximo de intentos");

        //Confirmación maximo intentos por terminal
        System.out.println("\n-----------------------------------------\n");
        System.out.println("BLOQUEO INTERFAZ LOGIN POR EXCESO DE INTENTOS");
        System.out.println("\n-----------------------------------------\n");

        //limpieza y bloqueo de campos
        campo_contraseña.setEnabled(false);
        campo_nombre.setEnabled(false);
        campo_contraseña.setText("");
        campo_nombre.setText("");

        //Cierre socket con el servidor
        clienteSocket.close();

    }
```

Para evitar problemas, en este caso también se cierra el socket puesto que al sufrir de un bloqueo , este ya no tendrá ningún uso más hasta que el cliente reinicie el programa.

Bloque de comprobarEntrada:

En este bloque es donde con un método se comprueba que efectivamente tanto la contraseña como el usuario cumplen 2 criterios básicos, que son: estar escritas y contar con una mayúscula, e*ste método es el encargado de leer dichos valores y devolver un verdadero en caso de cumplirlos o un falso, en caso de no cumplirlos.*

```java
//comprobar si la entrada no está vacía y tiene al menos una letra mayúscula
2 usages  ▲ alex
public boolean comprobarEntrada(String entrada) {

    if (!entrada.isBlank()) {

        for (char caracter : entrada.toCharArray()) {

            if (Character.isUpperCase(caracter)) {

                return true; // La cadena tiene al menos una letra mayúscula

            }

        }

        return false; // La cadena no tiene letras mayúsculas

    } else {

        return false; // La cadena está vacía

    }

}
```

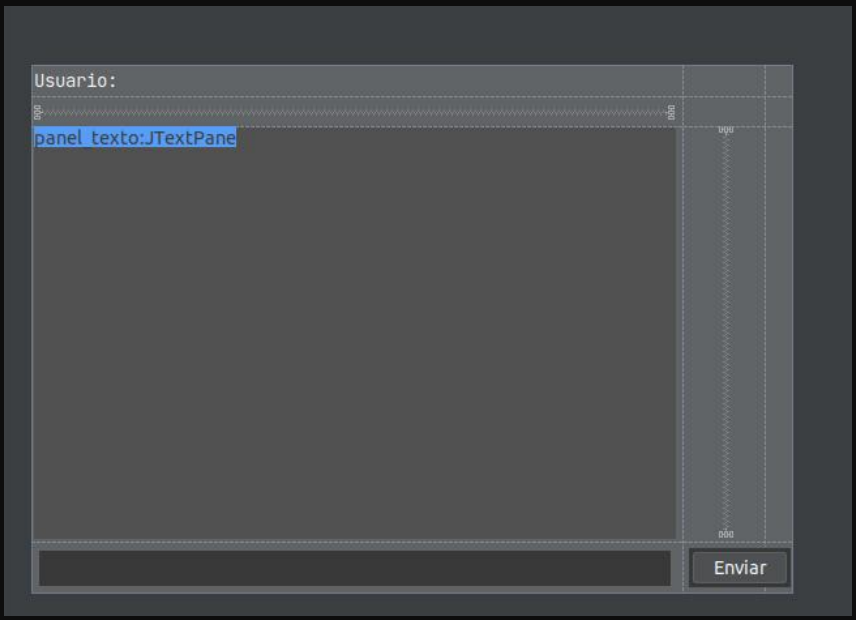
Bloque Lanzamiento de la UI del chat:

Como se cito anteriormente ya que esta clase proviene de la clase *principal main*, la cual cuenta con un método activador de la interfaz del chat en si, es en este momento cuando ya se sabe el usuario correcto con el que se quiere entrar, cuando se llama a este fragmento de código con el nombre correcto del ***usuario que a aceptado el servidor.***

```java
}

// Método para lanzar la ventana de chat
1 usage  ▲ alex +1 *
public void lanzarChat(String nombre) {

    //lanza el sistema del chat
    Main.lanzarChat(nombre);

}
```

UIRegistro :

Dentro del código de este apartado se encuentran todos los puntos necesarios para que el usuario pueda interactuar directamente con el servidor, por una interfaz amigable esta compuesta de ***4 campos, una interfaz que mostrara el texto en pantalla , mostrando el dialogo fluido entre los diversos usuarios del chat, un campo en el cual se podrá escribir lo que el usuario quiera enviar al chat para que todos reciban, y un botón de enviar.***



Este código se subdivide en diversos puntos:

Bloque de Variables:

En dicho bloque se encuentran todas las variables necesarias para el correcto funcionamiento de la interfaz gráfica, siendo necesarias para la comunicación efectiva con el servidor en lo que a dialogo se refiere

```
//variables locales
4 usages
private MulticastSocket redBroadcast = null;
2 usages
private static final String ipMulticast = "239.0.0.1";
2 usages
private static final int multicastPort = 12345;
```

Contando con los siguientes elementos:

->Un MultiSocket con el cual el cliente se comunicara con el grupo.

->Un entero que indica el puerto con el que se comunicara el cliente

->Una string con la dirección ip del grupo multicast

Bloque del constructor:

En dicho bloque debido a la obtención de datos por argumentos de constructor, la clase cuenta con los datos necesarios que ella precisa para operar correctamente, también es en este bloque de código donde se setea el nombre del usuario en la interfaz , marcando con que usuario se esta escribiendo en ese momento en concreto, esto es posible gracias a una etiqueta de texto llamada ***"etiqueta_usuario"*** donde se le añade a su texto el nombre del usuario, posteriormente se hace uso de un método el cual automáticamente setea los eventos necesarios para que la interfaz funcione.

```
//Constructor
1 usage   alejandroMorTur +2
public UIChat(String nombre, MulticastSocket redBroadcast){

    this.nombre = nombre;
    this.redBroadcast = redBroadcast;

    //Setea autoscroll pare evitar perder texto cuandl el dialogo se llena
    panel_texto.setAutoscrolls(true);

    //Setea la etiequeta con el nombre del usuario que ha iniciado sesión
    etiqueta_usuario.setText("Usuario: " + nombre);

    //Setea los eventos
    setEventos();

}
```

Bloque de eventos:

En dicho bloque debido a la naturaleza del formulario , es donde residen los gestores de eventos del propio formulario para darle las funcionalidades necesarias para el correcto funcionamiento de la interfaz, contando con los siguientes eventos

-> un evento gestor del botón de envió.

-> un evento encargado de la gestion del cierre de la interfaz

Un evento importante consta del evento de envió del texto al darle a un elemento ***"boton_enviar",*** el cual estará ligado a este botón, una vez presionado , de la texto en la que el usuario puede aportar texto se sacara la información necesaria que se quiere enviar , evidentemente se debe comprobar si la información no esta vacía, de no estarlo , se prepara una ***string compuesta por el nombre de usuario y junto el texto a enviar,*** posteriormente se prepara un datagram packet y se envía al grupo para que el servidor pueda rebotar la información.

```java
public void actionPerformed(ActionEvent e) {

    //saca el texto del textbox de la interfaz
    texto = entrada_texto.getText();

    if(!Objects.equals(texto, b: "") && texto != null){

        try {

            //prepara el texto para enviarlo por el multicast
            texto = nombre +": "+ texto;

            // Crear el DatagramPacket con los datos a enviar
            byte [] buffer = texto.getBytes();
            DatagramPacket paqueteRebote = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(ipMulticast), multicastPort);

            // Enviar el paquete a través del socket multicast
            redBroadcast.send(paqueteRebote);

            // Mostrar mensaje de éxito en la consola
            System.out.println("Mensaje: " + texto + " enviado correctamente al servidor \n");

            // Limpiar el campo de entrada
            entrada_texto.setText("");

        } catch (IOException ex) {

            // Manejar errores de E/S
            System.out.println("Error al enviar el mensaje: " + ex.getMessage());

        }
```

Posteriormente al envió del mensaje. Para evitar que el usuario tenga que sobreescribir el campo ***textbox***, se limpia el mismo, también en todo momento se tiene en cuenta las posibles complicaciones que pueden llegar a ocurrir dentro del sistema de envió , ***en caso de ocurrir algún error o excepción esta se prepara para posteriormente poder ver el error concreto de lo ocurrido.***

Un evento importante consta del evento de la gestión del cierre del formulario, dentro de este evento teniendo en cuenta que el servidor lee una frase muy concreta para entender que el cliente se ha desconectado del grupo:

Es en este momento cuando se aprovecha para notificar al resto de usuarios y al servidor de que se ha desconectado por una cadena de texto muy concreta .

```java
//Creación evento cierre de ventana por parte del usuario
this.addWindowListener((WindowAdapter) windowClosing(e) -> {

        try {

            //mensaje de consola confirmación cierre
            System.out.println("La ventana se está cerrando...");

            String mensajeDesconectado = nombre + " esta offline   ";

            // Convertir el mensaje a bytes utilizando UTF-8
            byte[] bufferMensaje = mensajeDesconectado.getBytes(StandardCharsets.UTF_8);

            // Crear el DatagramPacket con los datos a enviar
            DatagramPacket paqueteRebote = new DatagramPacket(bufferMensaje, bufferMensaje.length, InetAddress.getByName(ipMulticast), multicastPort);

            redBroadcast.send(paqueteRebote);

            //cese hilo
            Main.detenerhilo();

            //desconexion socket multiple
            redBroadcast.close();

        } catch (IOException ex) {

            //tratamiento de errores
            System.out.println("Error: "+ex);

        }
```

Se crea un datagram packet preparado con dicha cadena para posteriormente cerrar el socket multicast, todo esto justo antes de cerrar el formulario, para así salir de forma ordenada del cliente.

Bloque de añadirTexto:

En este bloque aunque sea un método es usado por el hilo principal gestor de las comunicaciones del socket multicast, ***el funcionamiento de este método es simple, cuando es llamado se le aporta un string (el contiene el texto que se quiere añadir al dialogo) y este es proceso y indexado para que el cliente pueda ver el mensaje correctamente.***

```
//metodo encargado de añadir el texto al dialogo del chat siempre que reciba un mensaje nuevo
1 usage   alejandroMorTur +2
public void añadirTexto(String texto) {

    StyledDocument doc = panel_texto.getStyledDocument();

    try {

        //añade el texto a la interfaz
        doc.insertString(doc.getLength(), str: texto + "\n", a: null);

    } catch (BadLocationException e) {

        System.out.println("Error al añadir texto: " + e);

    }

}
```

HiloclienteUiChat :

Dentro de esta clase , su funcionamiento consta de la lectura constante de los mensajes enviado por el sistema multisocket , ***que alguien debe leer para poder administrarlos correctamente, es esta clase la que administra cuando un mensaje merece ser mostrado y cuando no, llamando al método correcto en cada situación.***

La clase esta compuesta de los siguientes componentes:

Bloque de Variables:

En dicho bloque se encuentran todas las variables necesarias para el correcto funcionamiento de la lectura de los datagramas del sistema multisockets, siendo necesarias para la comunicación efectiva con el servidor en lo que a dialogo se refiere

Contando con los siguientes elementos:

->Un MultiSocket con el cual el cliente se comunicara con el grupo.

```
//variables gestoras interfaz UI
3 usages
private UIChat UIChat = null;
2 usages
private static boolean enChat = true;
3 usages
private MulticastSocket redBroadcast = null;

//btención datos grupo multicast
3 usages
private static String MULTICAST_ADDRESS;
3 usages
private static int MULTICAST_PORT;

//Seteo buffer
5 usages
private static  byte[] buffer = new byte[8096];
```

->Una string con la dirección ip del grupo multicast

->Un booleano encargado de gestionar si se debe escuchar constantemente o por el contrario parar

-> Un objeto que hace referencia a la interfaz del chat, siendo este necesario para el correcto uso de sus métodos en el caso concreto

Bloque del constructor:

En dicho bloque es donde recibe por parámetros todos los valores necesarios para que la clase y por ello el hilo, pueda funcionar de manera correcta .

```
1 usage   alex
public HiloclienteUIChat(UIChat UIChat, MulticastSocket redBroadcast,String ipMulticast,int multicastPort){

    this.UIChat = UIChat;
    this.redBroadcast = redBroadcast;
    this.MULTICAST_ADDRESS = ipMulticast;
    this.MULTICAST_PORT = multicastPort;

}
```

Bloque run:

Siendo aquí donde se lee de forma constante por un *bucle while*, controlado por un booleano, se registran todos los paquetes y se añaden respectivamente a su lugar correcto, todo eso en segundo plano debido a la naturaleza del hilo.Siendo el mecanismo del hilo el siguiente:

Primeramente se toma un paquete de la red, ya que la función de este hilo consiste en eso escuchar atentamente al grupo multicast constantemente, dicha información se guardara en un datagrama de nombre ***"paqueteRebote"***.

```
//recibe información
paqueteRebote = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(MULTICAST_ADDRESS), MULTICAST_PORT);
redBroadcast.receive(paqueteRebote);
```

Posteriormente gracias a un ***booleano*** interno, se gestiona si el mensaje se ha duplicado o no, es decir si se esta leyendo un mensaje que ya ha sido leído o no.

```
if (!mensajeRecibido) {
```

De ser correcto que el mensaje es nuevo, se readaptara dicho mensaje a una string para poder indicarle posteriormente a la interfaz, que es el contenido de ese mensaje el que se quiere añadir al panel del chat de dialago, de ahi la necesidad de tener una referencia a la interfaz del chat.

```
String mensaje = new String(paqueteRebote.getData(),  offset: 0, paqueteRebote.getLength(), StandardCharsets.UTF_8);

System.out.println("Información recibida de: " + paqueteRebote.getAddress() + ", mensaje: " + mensaje + " \n");

//añade mensaje a la interfaz del dialogo chat
UIChat.añadirTexto(mensaje);

//setea el booleano de recibido el mensaje a true, para considerar el mensaje ya recibido
mensajeRecibido = true;
```

Una vez que este mensaje ha sido añadido al panel del chat, se le considerara como tratado , por eso el ***booleano se setea a true***, para evitar re lectura de mensajes y duplicados en el panel del chat.

En caso de que el mensaje ya se haya leido y ya se haya tratado , sera necesario realizar la operación inversa en lo que a booleanos se refiere***, seteando el booleano a falso, para que la próxima vez que se lea un mensaje no sea descartado.***

```
    mensajeRecibido = true;

}else {

    //setea el booleano de recibido el mensaje a false
    mensajeRecibido = false;

}
```

Bloque método detener :

Siendo este un método accesible por el resto de las clases , es el encargado de frenar el hilo solo cuando el cliente decide cerrar la interfaz del usuarios del chat, este método se compone de un seteo de un booleano ***"enchat"*** que es indispensable que este a true, para que el hilo lea constantemente, una vez llamado a este método , el hilo terminara de forma exitosa.

```
// Método para detener el hilo
1 usage   alex
public static void detener() {

    enChat = false;

}
```