**REQUERIMIENTO CORBA JAVA**

**MANUAL DE INSTALACIÓN**



**Integrantes**

**Luis Manuel Arango Guauña**

**Profesor**

**Pablo Augusto Magé Imbachí**

**Universidad del Cauca**

**Facultad de Ingeniería Electrónica y Telecomunicaciones**

**Programa de Ingeniería de Sistemas**

**Curso: Sistemas Distribuidos**

**Popayán, Marzo de 2022**

**Tabla de contenido**

[**I.** **INTRODUCCIÓN** 3](#_Toc97643414)

[**II.** **Requerimientos del Sistema.** 4](#_Toc97643415)

[**III.** **Descripción y estructura de directorios** 4](#_Toc97643416)

[**IV.** **Descripción de los archivos fuente** 4](#_Toc97643417)

[**Descripción “gusuarios.idl”** 5](#_Toc97643418)

[**a.** **Descripción personalDTO** 6](#_Toc97643419)

[**b.** **Descripción usuarioDTO** 6](#_Toc97643420)

[**c.** **Descripción credencialDTO** 6](#_Toc97643421)

[**d.** **Descripción AdmCllbckInt** 6](#_Toc97643422)

[**e.** **Descripción FapCllbckInt** 6](#_Toc97643423)

[**f.** **Descripción Interface GestionPersonalInt** 7](#_Toc97643424)

[**Descripción “gpacientes.idl”** 7](#_Toc97643425)

[**a.** **Descripción ValoracionDTO** 8](#_Toc97643426)

[**b.** **Descripción EjercicioDTO** 8](#_Toc97643427)

[**c.** **Descripción ProgramaDTO** 8](#_Toc97643428)

[**d.** **Descripción ProgramaFísicoDTO** 8](#_Toc97643429)

[**e.** **Descripción AsistenciaDTO** 8](#_Toc97643430)

[**f.** **Descripción notificaciónDTO** 9](#_Toc97643431)

[**g.** **Descripción Interface GestionPacientesInt** 9](#_Toc97643432)

[**Descripción s\_gestion\_usuarios** 9](#_Toc97643433)

[**a.** **Implementación GestionPersonalImpl** 9](#_Toc97643434)

[**b.** **Implementación del servidor de Objetos** 14](#_Toc97643435)

[**Descripción s\_seguimiento\_usuarios** 15](#_Toc97643436)

[**a.** **Implementación GestionPacientesImpl.java** 15](#_Toc97643437)

[**b.** **Implementación AdmCllbckImpl** 20](#_Toc97643438)

[**Descripción cliente** 20](#_Toc97643439)

[**a.** **Implementación AdmCllbckImpl** 20](#_Toc97643440)

[**V.** **Descripción de compilación y ejecución** 21](#_Toc97643441)

[**VI.** **Repositorio GitHub** 21](#_Toc97643442)

# **INTRODUCCIÓN**

Este manual de instalación está diseñado para poder instalar correctamente un requerimiento de sistema distribuido utilizando el CORBA en el lenguaje Java para la asignatura de Sistemas distribuidos del Grupo A del programa de ingeniería de Sistemas de la Universidad del Cauca. Se presenta la elaboración que se utilizó para cubrir dicho requerimiento orientado al ejemplo del programa estilos de vida saludable especializándose en el programa de hora saludable enfocada para docentes y administrativos de las diferentes facultades de la Universidad del Cauca.

El programa hora saludable tendrá como clientes al administrador, quien será el encargado de registrar el personal y consultar dicho personal, otro de los clientes es la secretaria quien registra usuarios y puede consultarlos, el PAF es el usuario que puede hacer valoraciones sobre los pacientes registrados por la secretaria, finalmente los pacientes pueden ingresar para consultar las valoraciones, su programa físico y asistencias.

El programa está diseñado con un cliente de objetos y dos servidores de objetos, para la creación de ellos es necesario tener conocimientos básicos de compilación y mapeo de archivos IDL, para la implementación de este sistema usaremos 2 archivos IDL para la gestión del personal y la gestión de los pacientes.

# **Requerimientos del Sistema.**

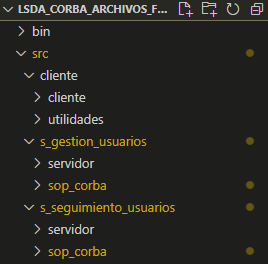
Para la aplicación distribuida debe tener en cuenta los siguientes aspectos generales sobre las tecnologías utilizadas:

|  |  |
| --- | --- |
| TECNOLOGÍA | NECESARIO |
| Sistema Operativo | Windows 10 / Linux |
| Lenguaje de Programación | JAVA |
| Java versión | Java 8 |
| Java JDK versión | 1.8.0\_202 |
| Modelo | CORBA de la OMG para GNU/Linux o Windows |

Debemos disponer de un editor como lo es VS Code o un entorno para java de su preferencia, está desarrollado para mostrar resultados mediante la consola. Para esta guía presentaré un modelo a seguir utilizando Visual Studio Code.

# **Descripción y estructura de directorios**

Iniciemos por ubicar nuestro proyecto en una carpeta que será nuestro directorio principal, dele un nombre apropiado para la aplicación que se va a implementar, en este caso por regla de entrega de este requerimiento la he nombrado “lsdA\_corba\_archivos\_funte”, dentro de ella dispondremos de la siguiente organización como lo muestra la fig. 1



*Fig. 1 Estructura de directorios*

# **Descripción de los archivos fuente**

A continuación, se describen las interfaces, clases y métodos utilizados para desarrollar el requerimiento. Empezaremos primero por definir los servidores que se van a ubicar en las carpetas “s\_gestion\_usuarios” y “s\_seguimiento\_usuarios” y “cliente”.

Primero en la carpeta raíz crearemos 2 nuevos archivos con la extensión “.idl”, estos nos permitirán en un paso próximo crear los archivos del soporte de CORBA.

## **Descripción “gusuarios.idl”**

IDL es una extensión es un lenguaje de descripción de interfaz, por lo que este archivo nos ayudará a definir las interfaces y estructuras que utilizaremos. Aquí tendremos la gestión del personal y el registro para los usuarios, así como también las estructuras necesarias para definir los objetos con los que trabajaremos. Así pues, crear primero un archivo llamado “gusuarios.idl” en la carpeta raíz(src), en seguida está definido nuestro IDL:

module sop\_corba{

interface AdmCllbckInt{

void notificar(in string nombreCompleto, in long id );

};

interface FapCllbckInt{

void informarIngreso(in string nombreCompleto, in long id);

};

struct personalDTO{

string tipo\_id;

long id;

string nombreCompleto;

string ocupacion;

string usuario;

string clave;

};

struct usuarioDTO{

long id;

string nombreCompleto;

string facultad;

string tipo;

string fechaIngreso;

string patologia;

string usuario;

string clave;

};

struct credencialDTO{

string usuario;

string clave;

long id;

};

interface GestionPersonalInt{

void registrarPersonal(in personalDTO objPersonal, out boolean res);

boolean consultarPersonal(in long id, out personalDTO objPersonal);

void registrarUsuario(in usuarioDTO objUsuario, out boolean res);

boolean consultarUsuario(in long id, out usuarioDTO objUsuario);

boolean abrirSesion(in credencialDTO objCredencial);

void registrarCallback(in AdmCllbckInt objCllbck);

void registrarCallbackFap(in FapCllbckInt objCllbck);

};

};

Como se puede observar no es un lenguaje de programación, sino que describe las interfaces y estructuras que hemos de utilizar para la construcción de este sistema distribuido.

Una vez creado el IDL procederemos a compilarlo y generar los soportes de CORBA para el servidor de gestión de usuario haciendo uso del siguiente comando:

*idlj -fall -pkgPrefix sop\_corba s\_gestion\_usuarios gusuarios.idl*

El comando anterior genera los soportes de CORBA en la carpeta “sop\_corba” que está en “s\_gestion\_usuarios” utilizando el archivo IDL “gusuaios.idl”.

Ahora, describamos los struct e interfaces definidas en el archivo IDL anterior.

### **Descripción personalDTO**

Este struct nos permite almacenar los datos referentes a un personal, dichos campos corresponden al tipo de identificación que puede tomar uno de los siguientes tres valores: CC, TI o PP, el número de la identificación que debe ser mayor a cero, el nombre completo del personal con una longitud mínima de 2 caracteres y máxima de 64, la ocupación del personal que en este caso solo pueden existir de 2 tipos: una secretaria (SEC) o un profesional de acondicionamiento físico (PAF), además solo puede existir un personal de cada ocupación, por último el personal tendrá asociado un nombre usuario y la clave, en ambos casos su longitud debe ser mínimo de 8 caracteres alfanuméricos.

### **Descripción usuarioDTO**

Este struct nos permite almacenar los datos referentes a un usuario, dichos campos corresponden a el número de la identificación que debe ser mayor a cero, el nombre completo del personal con una longitud mínima de 2 caracteres y máxima de 64, la facultad a la que pertenece que puede tomar los siguientes valores: FACARTES, FACAGRO, FSALUD, FHUMANAS, FCCEA, FACNED, FDERECHO, FCIVIL, FIET; un campo que describe el tipo de cargo que tiene el usuario con dos posibles valores “Administrativo” o “Docente”, la fecha de ingreso en el que se registra, una patología que puede ser ingresada u omitida, por último el usuario tendrá asociado un nombre usuario y la clave, en ambos casos su longitud debe ser mínimo de 8 caracteres alfanuméricos.

### **Descripción credencialDTO**

En credenciaDTO almacenamos los datos correspondientes para poder acceder al sistema, esos datos son el usuario, contraseña y el ID asociado a los 2 anteriores.

### **Descripción AdmCllbckInt**

Es interfaz es utilizada para hacer un callback al administrador cuando un personal PAF inicia sesión en el sistema, se notificara al administrador por medio del método notificar().

### **Descripción FapCllbckInt**

Es utilizada para hacer un callback y notificar el ingreso de un nuevo paciente al PAF cuando una secretaria registra a un nuevo paciente.

### **Descripción Interface GestionPersonalInt**

Describe los métodos usados para gestionar el registro y consulta de un personal, el registro y consulta de un usuario, el acceso (login) al sistema, y el registro de los callback para el administrador y el FAP.

## **Descripción “gpacientes.idl”**

Aquí tendremos la gestión de los pacientes, así como también las estructuras necesarias para definir los objetos con los que trabajaremos. Así pues, crear primero un archivo llamado “gpacientes.idl” en la carpeta raíz(src), en seguida está definido nuestro IDL:

module sop\_corba{

struct AsistenciaDTO{

long idPaciente;

string fechaAsistencia;

string observacion;

};

typedef sequence<AsistenciaDTO> ArrayAsistencia;

struct EjercicioDTO{

string nombreEjercicio;

long repeticiones;

long peso;

};

typedef sequence<EjercicioDTO> ArrayEjercicios;

struct ProgramaDTO{

long dia;

ArrayEjercicios listaEjercicios;

};

typedef sequence<ProgramaDTO> ArrayPrograma;

struct ProgramaFisicoDTO{

long idPaciente;

string fechaInicio;

ArrayPrograma listaProgramaSemana;

};

struct ValoracionDTO{

long idPaciente;

string fechaValoracion;

long fecCardiacaReposo;

long fecCardiacaActiva;

long estatura;

long brazo;

long pierna;

long pecho;

long cintura;

string estado;

};

struct notificacionDTO{

string nombreCompleto;

string ocupacion;

};

interface GestionPacientesInt{

void registrarValoracion(in ValoracionDTO objValoracion, out boolean res);

boolean consultarValoracion(in long id, out ValoracionDTO objValoracion);

void registrarProgramaFisico(in ProgramaFisicoDTO objProgramaFisico, out boolean res);

boolean consultarProgramaFisico(in long id, out ProgramaFisicoDTO objProgramaFisico);

void registrarAsistencia(in AsistenciaDTO objAsistencia, out boolean res);

boolean consultarAsistencia(in long id, out ArrayAsistencia lstAsistencia);

void contarFaltas(in long id, out long numFaltas);

void enviarNotificacion(in notificacionDTO objNotificacion);

};

};

Una vez creado el IDL procederemos a compilarlo y generar los soportes de CORBA para el servidor de gestión de pacientes haciendo uso del siguiente comando:

*idlj -fall -pkgPrefix sop\_corba s\_seguimiento\_usuarios gpacientes.idl*

El comando anterior genera los soportes de CORBA en la carpeta “sop\_corba” que está en “s\_seguimiento\_usuarios” utilizando el archivo IDL “gusuaios.idl”.

Explicamos a continuación los struct del IDL.

### **Descripción ValoracionDTO**

Ester struct nos permite almacenar la información de una valoración realizada por el PAF a un paciente, contiene el id del paciente, la fecha de la valoración, la frecuencia cardiaca en reposo (pulsaciones por minuto), frecuencia cardiaca activa, las mediciones en centímetros de: estatura, brazo, pierna, pecho y cintura, finalmente un campo en el que se almacena con base en los campos anteriores el estado físico del paciente.

### **Descripción EjercicioDTO**

Nos permite almacenar la información de un ejercicio que se compone del nombre del mismo, la cantidad de repeticiones a realizar y el peso en kilogramos con el que se ejecuta el ejercicio.

### **Descripción ProgramaDTO**

Contiene la información del programa de ejercicios para un día en específico, los atributos que lo describen son el número del día y una lista de ejercicios que se realizan para ese día.

### **Descripción ProgramaFísicoDTO**

Describe el programa físico debe desarrollar un paciente, y se identifica por el ID del paciente, la fecha de inicio del programa físico y una lista del tipo Programa DTO que contiene los días y los ejercicios a realizar.

### **Descripción AsistenciaDTO**

Almacena la información de la asistencia al programa físico para un paciente, por el id del paciente se identifica y se registra la asistencia, la fecha en la que se registra la asistencia y una observación que puede ser una excusa, una no asistencia, o ninguna en caso de asistir.

### **Descripción notificaciónDTO**

Contiene el nombre y la ocupación de un personal con permiso para ingresar al servidor de seguimiento de usuarios.

### **Descripción Interface GestionPacientesInt**

Describe los recursos que van a consumir el PAF y el paciente, dichos recursos los definen los métodos para registrar y consultar una valoración, registrar y consultar un programa físico, registrar y consultar asistencias, un método para hacer el conteo del número de faltas de un paciente por su ID y por último un método para notificar el acceso del PAF.

## **Descripción s\_gestion\_usuarios**

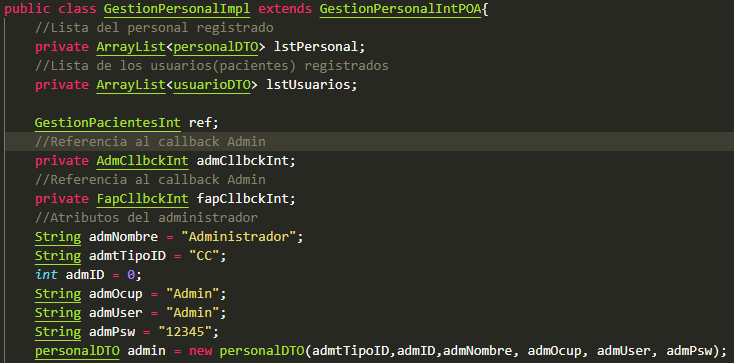
Pasamos ahora a implementar los soportes que hemos generado con los IDL, ubicados en la carpeta *src/s\_gestion\_usuarios/servidor* creamos un nuevo archivo java que nombraremos como “*GestionPersonalImpl”,* y otro más *“ServidorDeObjetos”*.

### **Implementación GestionPersonalImpl**

Lo primero que hacemos es extender esta clase de GestionPersonalIntPOA (Esta clase se habrá generado al momento de compilar con el comando del IDL gusuarios.idl)

#### Atributos

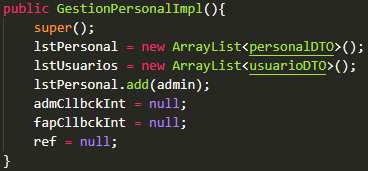
Los Atributos que debemos definir en esta clase serán los siguientes:



*Fig. 2 – Atributos GestionPersonalImpl*

#### Constructor

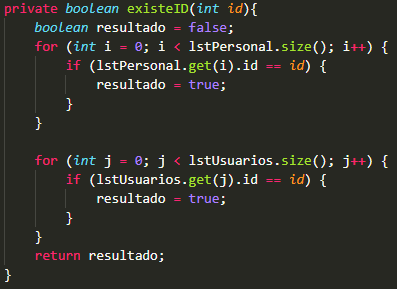
Inicializamos el constructor con los parámetros definidos en el paso anterior.



*Fig. 3 – Constructor de la clase GestionPersonalImpl*

#### Método auxiliar existeID

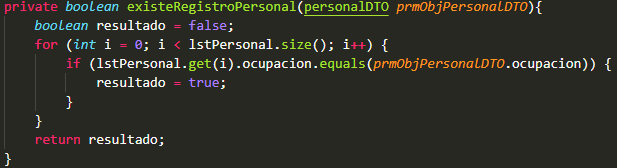
Este método está hecho para verificar si ya existe un id registrado en el sistema, esto será importante para poder tener integridad en los datos y que no existan ID repetidos en los registros del personal y el usuario. True si ya existe o false de lo contrario.

**

*Fig. 4 – Método auxiliar existeID()*

#### Método auxiliar existeRegistroPersonal

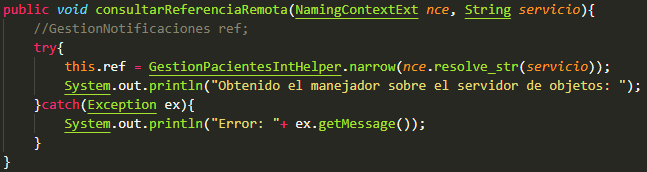
Este método está hecho para verificar si ya existe un personal con una ocupación específica, ya que se tiene la restricción de que solo pueden existir una secretaria y un PAF. True si ya existe un personal con esa ocupación y false de lo contrario



*Fig. 5 – Método auxiliar existeRegistroPersonal()*

#### Método auxiliar existeRegistroPersonal

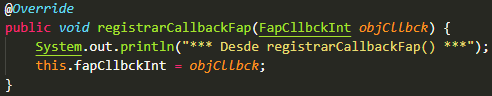
Este método nos será útil en un paso posterior, nos permitirá hacer la conexión con el segundo servidor de seguimiento a usuarios al obtener la referencia remota.



*Fig. 6 – Método auxiliar consultarReferenciaRemota()*

#### Método Override registrarCallbackFap

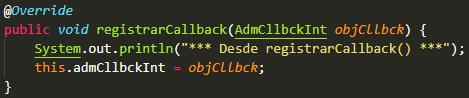
Nos permite asignar la referencia para el FAP que inicializamos en nulo en el constructor de esta clase.



*Fig. 7 – registrarCallbackFap()*

#### Método Override registrarCallback

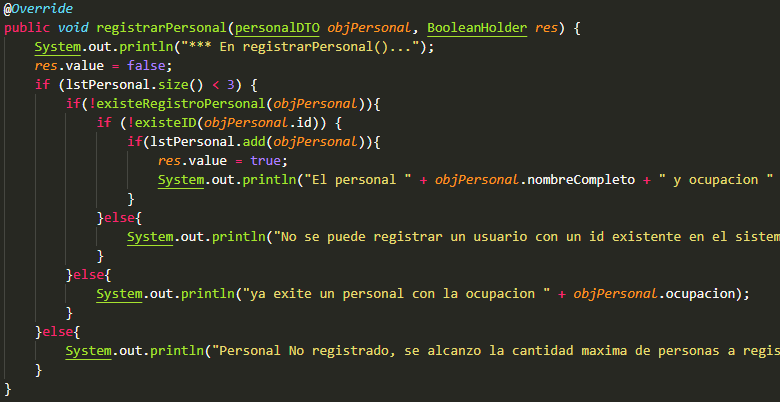
Este método nos permite asignar la referencia para el Administrador que inicialmente esta en nulo en el constructor de esta clase.



*Fig. 8 – RegistrarCallbackFap()*

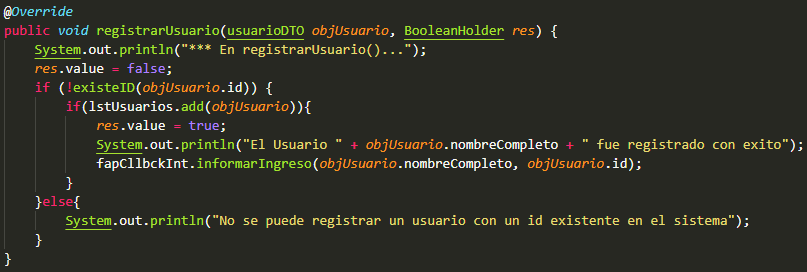
#### Método Override registrarPersonal

Nos permite registrar un objeto de tipo personalDTO, primero verifica si hay espacio en la lista del personal seguido a ello verifica si el personal con la ocupación entrante ya se encuentra previamente registrado, y finalmente si ese personal con el ID no existe en los registros del personal y usuarios se procede a hacer la adición de ese objPersonal a la lista del personal. En el parámetro de salida *res* se almacena el resultado de la operación, true de ser exitosa y false de lo contrario

  
*Fig. 9 – registrarPersonal()*

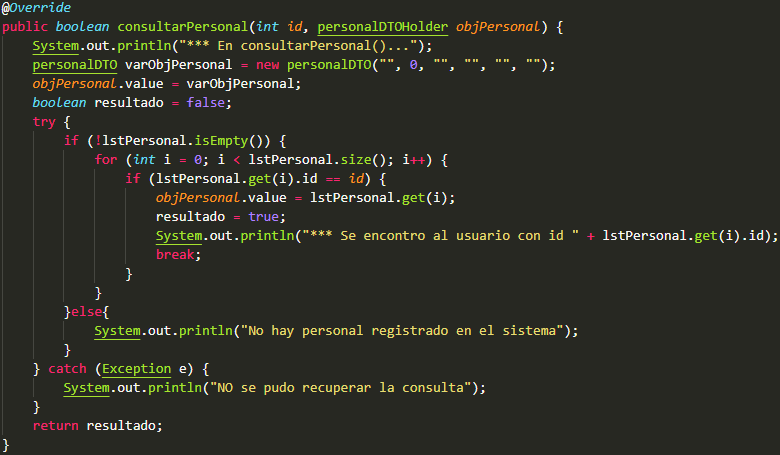
#### Método Override registrarUsuario

Nos permite registrar un objeto de tipo usuarioDTO, primero verifica si ya existe algún usuario con el id proporcionado y si la inserción es exitosa el usuario podrá ser registrado en el sistema. En el parámetro de salida *res* se almacena el resultado de la operación, true de ser exitosa y false de lo contrario

*  
Fig. 10 – registrarUsuario()*

#### Método Override consultarPersonal

Nos permite consultar si un personal está registrado en el sistema, la consulta se realiza partiendo del ID, si hay coincidencia se retorna el objeto de tipo personal y true, de lo contrario si no existe se retornará un objeto de tipo personal con parámetros por defecto y false.



*Fig. 11 – consultarPersonal()*

#### Método Override consultarUsuario

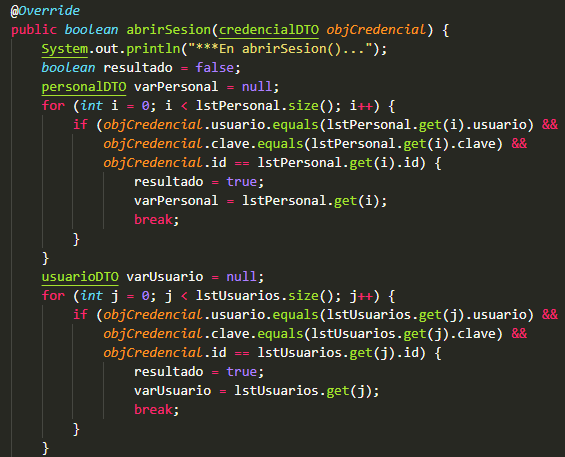
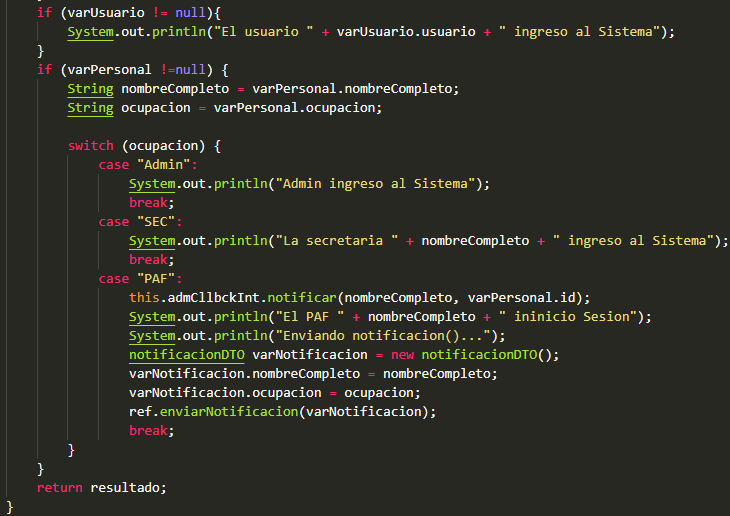
Nos permite consultar si un usuario está registrado en el sistema, la consulta se realiza partiendo del ID, si hay coincidencia se retorna el objeto de tipo usuario y true, de lo contrario si no existe se retornará un objeto de tipo usuario con parámetros por defecto y false.



*Fig. 12 – ConsultarUsuario()*

#### Método Override abrirSesion

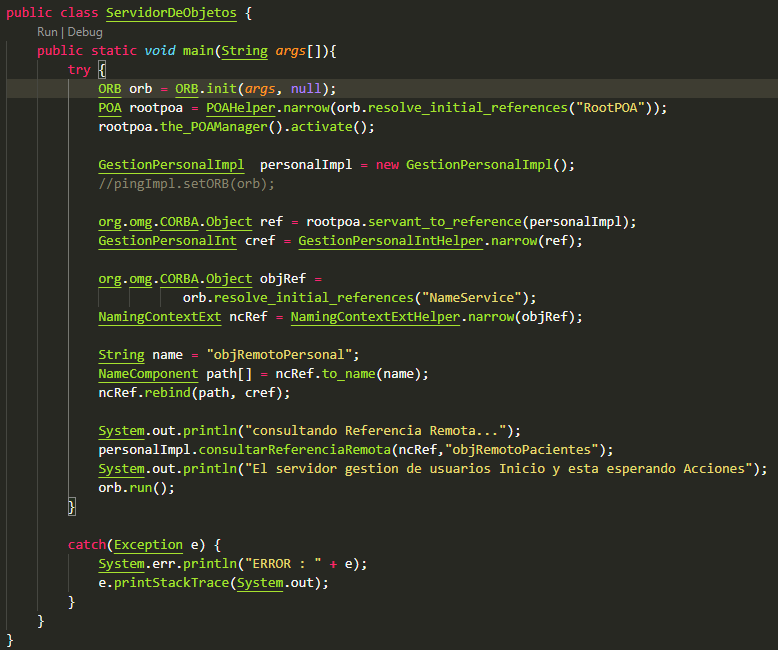
En este método retornamos una respuesta true o false respecto a una petición de acceso al sistema, mediante un objeto de tipo credencial verificamos que el usuario que hace la petición esté autorizado para ingresar al sistema. True en caso de que la operación sea exitosa y false de lo contrario. En el caso del FAP se envía una notificación al Admin y se construye un objeto *notificacionDTO* para informar al servidor de su ingreso al sistema mediante a referencia remota *ref*.

*Fig. 13 – abrirSesion()*

### **Implementación del servidor de Objetos**

Ubicados en la siguiente ruta de la carpeta raíz de nuestro directorio *src/s\_gestion\_usuario/servidor* creamos una clase denominada *ServidorDeObjetos,* esta clase será el punto de arranque del servidor actual. Definamos a continuación esta clase:

  
*Fig. 14 – ServidorDeObjetos()*

Como pueden observar, esta clase instancia el orb y es registrado con el nombre *“objRemotoPersonal”,* y luego de ello consulta la referencia remota del servidor 2 que estaremos describiendo en la sección posterior, con esto el servidor queda en escucha. Aclarar que primero se debe lanzar el ORBD y el servidor 2, está secuencia se describirá posteriormente.

## **Descripción s\_seguimiento\_usuarios**

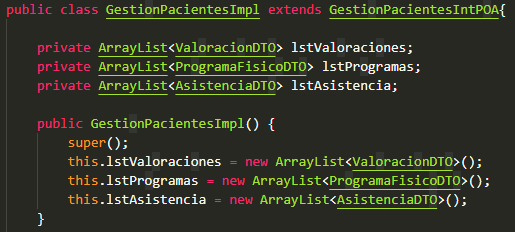
Pasamos ahora a implementar los soportes que hemos generado con el IDL *“gpacientes.idl”*, ubicados en la carpeta *src/s\_seguimiento\_usuarios/servidor* creamos un nuevo archivo java que nombraremos como “*GestionPacientesImpl”,* y otro más *“ServidorDeObjetos”*.

### **Implementación GestionPacientesImpl.java**

Lo primero que hacemos es extender esta clase de GestionPacientesIntPOA (Esta clase se habrá generado al momento de compilar con el comando del IDL gpacientes.idl).

#### Atributos y constructor

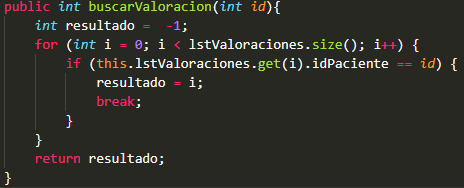
Los Atributos que debemos definir en esta clase serán los siguientes y en el constructor inicializamos dichos atributos:



*Fig. 15 - GestionPacientesImpl*

#### Método auxiliar buscarValoracion

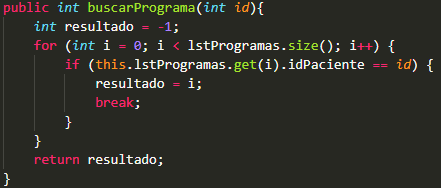
Este método busca mediante un id de paciente una valoración asociada a ese ID, si la encuentra retornara la posición, es decir, el índice de la lista de valoraciones donde se encuentra ubicada esa valoración, y devolverá -1 en caso de que la búsqueda no arroje coincidencias.



*Fig. 16 – buscarValoracion()*

#### Método auxiliar buscarPrograma

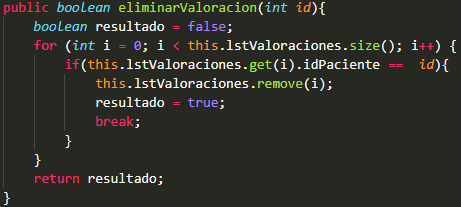
Este método busca en la lista de programas un objeto asociado al id de un paciente, de haber coincidencia se retorna el índice de la lista donde se encuentra ubicado, si la operación no encuentra ningún resultado retornará el valor -1.

**

*Fig. 17 – bucarPrograma()*

#### Método auxiliar eliminarValoracion

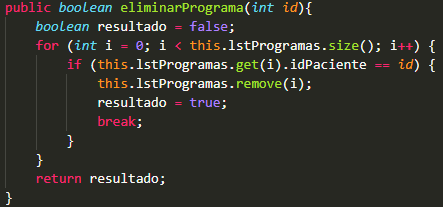
Este método busca en la lista de valoraciones un objeto asociado al id de un paciente, de haber coincidencia se procede a eliminar el objeto de la lista, retornará true si la eliminación tuvo éxito o false de lo contrario.



*Fig. 18 – eliminarValoracion()*

#### Método auxiliar eliminarPrograma

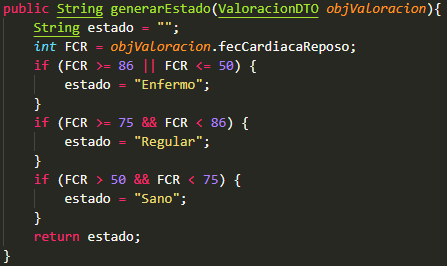
Este método busca en la lista de programas un objeto asociado al id de un paciente, de haber coincidencia se procede a eliminar el objeto de la lista, retornará true si la eliminación tuvo éxito o false de lo contrario.



*Fig. 19 – eliminarPrograma()*

#### Método auxiliar generarEstado

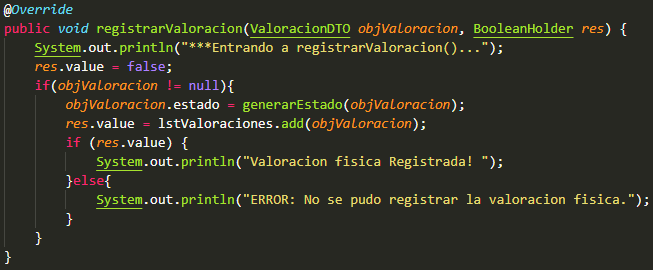
Este método define el estado de una persona, dependiendo de las pulsaciones por minuto de la frecuencia cardiaca en estado de reposo se puede considerar el estado de salud de un paciente, los posibles valores que puede retornar son: Enfermo, Regular o Sano.



*Fig. 20 – generarEstado()*

#### Método Override registrarValoracion

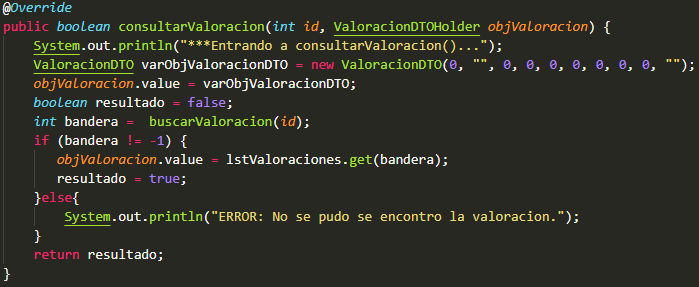
Este método registra una valoración en la lista de valoraciones, de ser exitoso el proceso de registro retornará en su parámetro de salida *res* true, false de lo contrario.

**

*Fig. 21 – registrarValoracion()*

#### Método Override consultarValoracion

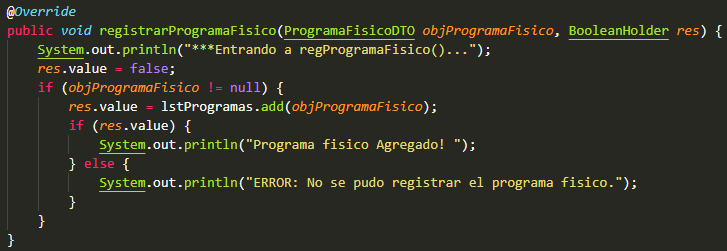
Este método consulta una valoración por el id de paciente asociado a ella, retornará True y el objeto de tipo valoración en el parámetro objValoracion si se encontró una coincidencia, y si falla retornará false y el objValoracion tendrá valores por defecto.

**

*Fig. 22 – consultarValoracion()*

#### Método Override registrarPrograma

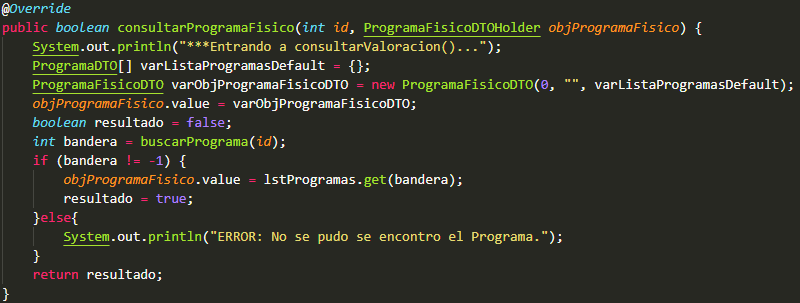
Este método registra una valoración en la lista de valoraciones, de ser exitoso el proceso de registro retornará en su parámetro true, y false de lo contrario.

**

*Fig. 23 – regstrarProgramaFisico*

#### Método Override consultarProgramaFisico

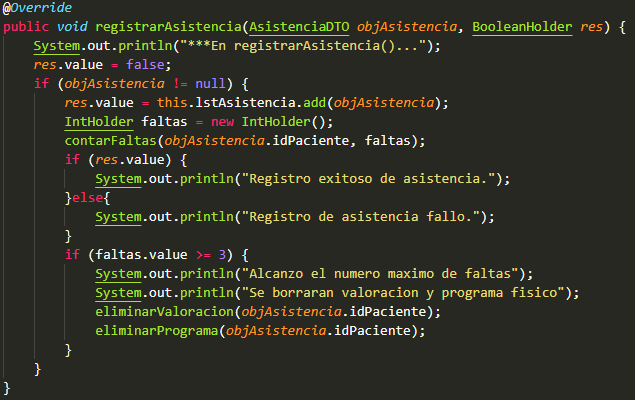
Este método consulta un programa por el id de paciente asociado a ella, retornará True y el objeto de tipo ProgramaFisico en el parámetro objProgramaFisico si se encontró una coincidencia, y si falla retornará false y el objProgramaFisico tendrá valores por defecto.



*Fig. 24 – ConsultarProgramaFisico()*

#### Método Override registrarAsistencia

Este método registra una asistencia en la lista de Asistencias, de ser exitoso el proceso de registro retornará en su parámetro *res* true, y false de lo contrario. Hace uso del conteo de faltas, y cuando las son iguales a 3 hace uso de los métodos auxiliares para borrar la Valoración y el programa físico del Paciente con el ID asociado a la asistencia.



*Fig. 25 – registrarAsistencia*

#### Método Override consultarAsistencia

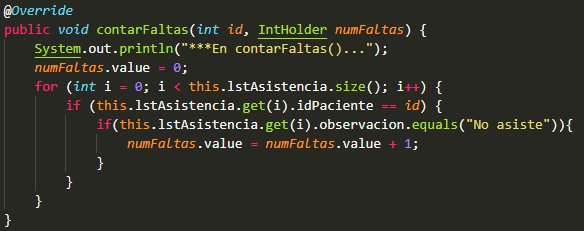
Este método consulta las asistencias de un paciente por su ID, si el paciente tiene registros se irán almacenando en el parámetro lstAsistencia y retornará true, en caso contrario lstAsistencia tendrá un objeto de tipo lista de asistencias con parámetros por defecto y el retorno será false.

**

*Fig. 26 – consultarAsistencia ()*

#### Método Override contarFaltas

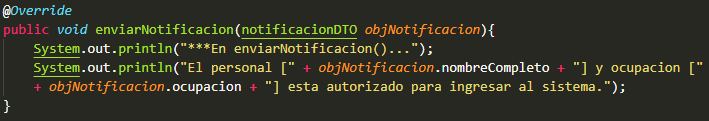
Este método cuenta en las faltas de un paciente por su ID, lo realiza haciendo una búsqueda cíclica en la lista de asistencia, retorna el número de faltas para el paciente y 0 cuando no se encuentran faltas reportadas

**

*Fig. 27 – contarFaltas()*

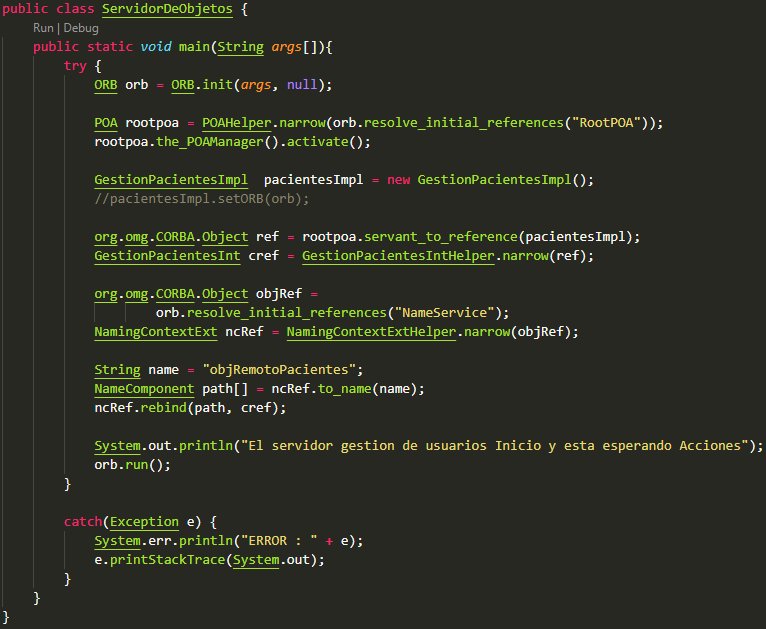
#### Método Override enviarNotificacion

Envía una notificación generando un eco al servidor de seguimiento de usuarios, este se utiliza cuando el FAP inicia sesión, notificando así que ha ingresado al sistema que está habilitado para él

**

*Fig. 28 – enviarNotificacion()*

### **Implementación Servidor de Objetos**



## **Descripción cliente**

Pasamos ahora a implementar los soportes que hemos generado con los IDL, ubicados en la carpeta *src/cliente/cliente* creamos un nuevo archivo java que nombraremos como “*AdmCllbckImpl” y* “*FapCllbckImpl”,* y en *src/cliente* creamos un archivo nombrado *“ClienteDeObjetos”*.

### **Implementación AdmCllbckImpl**

Extendemos de AdmCllbckIntPOA (Archivo generado en el soporte de CORBA) y procedemos a implementar el método notificar, que es una alerta que imprime en consola para el administrador cuando un FAP ingresa

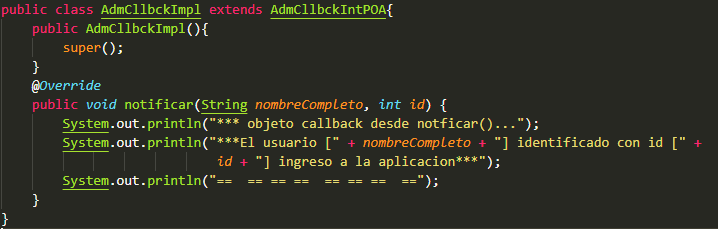
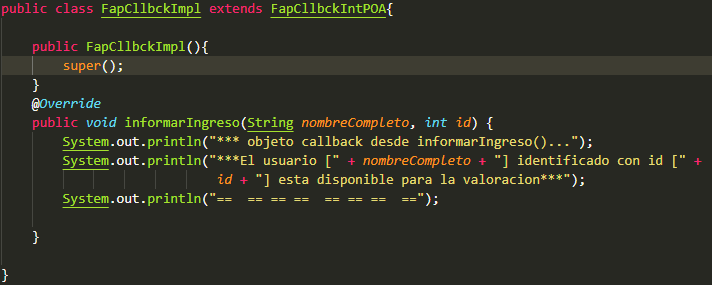


Fig. 29 – Class AdmCllbckImpl

### **Implementación FapCllbckImpl**

Extendemos de FapCllbckIntPOA (Archivo generado en el soporte de CORBA) y procedemos a implementar el método Informar ingreso, que se hace cuando un nuevo paciente es registrado, se le notifica así pues al PAF para que pueda hacer la respectiva valoración.

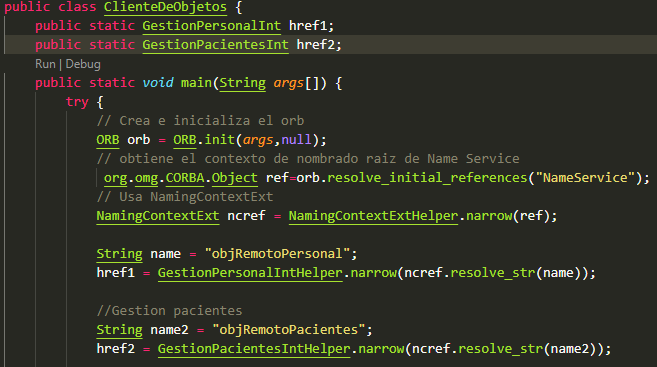
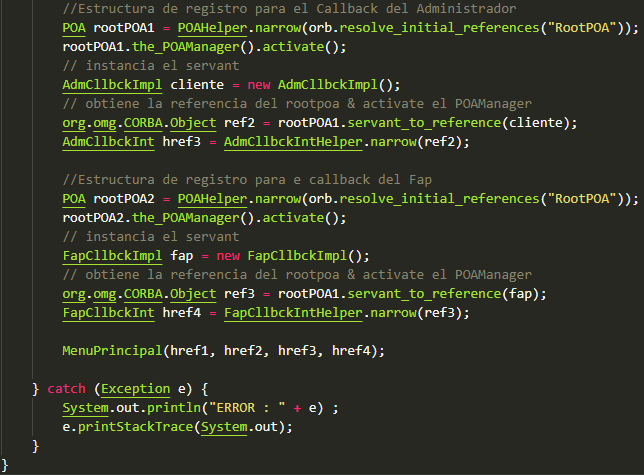


*Fig. 30 – Class FapCllbckImpl*

### **Implementación Cliente de Objetos**

En esta clase tendremos primero el arranque del cliente además de los menús y el paso de información a los servidores. En pocas palabras esta clase consume todos los servicios de los servidores dependiendo del tipo de usuario, aquí se arman los paquetes de datos de informan a pasar en cada uno de los métodos. También se controlan las restricciones como longitudes de las cadenas y valores nulos o negativos, para que la información pase filtrada a los servidores.

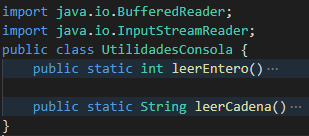
En primera medida declaramos las referencias de los servidores y obtenemos las referencias remotas mediante el nombre que hemos configurado en los servidores de objetos anteriormente, luego obtenemos la referencia para los callback y hacemos llamado al método de menú principal.

*Fig. 31 – Cliente de Objetos*

### **Implementación Utilidades de consola**

Dentro de la carpeta “Utilidades” creamos una nueva clase “UtilidadesConsola.java” e implementamos los métodos para leer un entero y leer una cadena. Imagen de referencia a continuación:



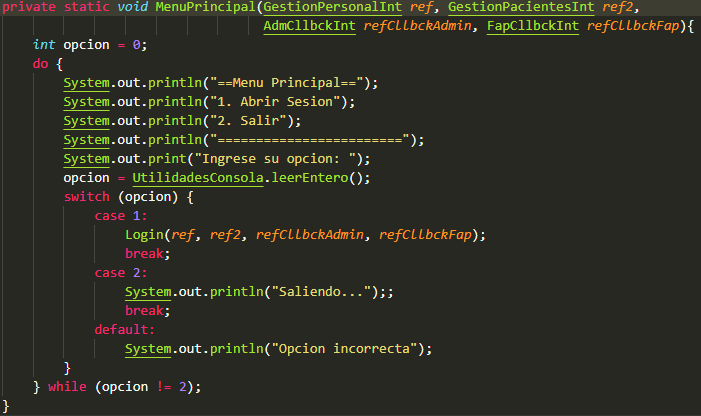
*Fig. 32 clase UtilidadesConsola.java*

### **Implementación Métodos en el Cliente de Objetos**

Aquí tendremos definidos luego del *main* los métodos para los menús y la captura de información para posteriormente utilizar las referencias remotas y consumir los servicios de los servidores.

#### Método Menu Principal

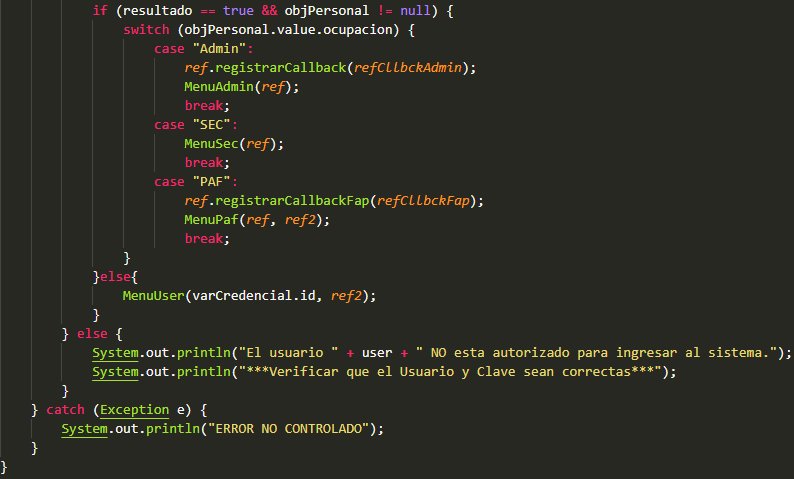
Creamos un método estático como se define en la siguiente figura. Este menú será la vista principal al momento de iniciar la aplicación del cliente.

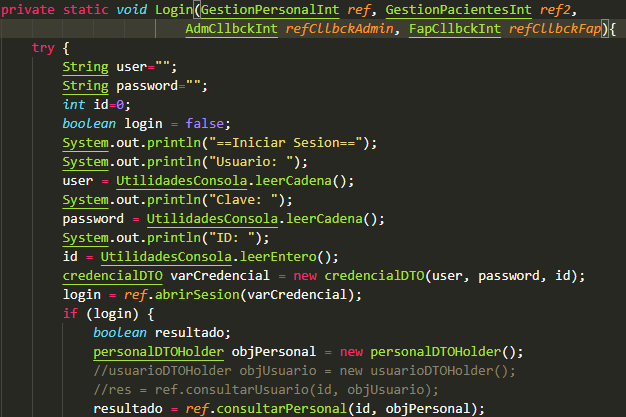


*Fig. 33 – MenuPrincipal()*

#### Método Login

Este método es una fachada para desplegar los menús de los usuarios finales, procesamos las credenciales y con la respuesta del servidor mostramos la vista pertinente dependiendo del tipo de usuario que está haciendo una petición de acceso al servidor. Para el caso del Admin y del PAF se registra el correspondiente callback.





*Fig. 34 – Login()*

#### Método Menú administrador

Este método es una fachada para desplegar los menús de los usuarios finales, procesamos las credenciales y con la respuesta del servidor mostramos la vista pertinente dependiendo del tipo de usuario que está haciendo una petición de acceso al servidor. Para el caso del Admin y del PAF se registra el correspondiente callback.

# **Descripción de compilación y ejecución**

Para compilar nuestro proyecto nos ubicamos en la carpeta “src”, y a continuación compilamos cada uno de los servidores y el cliente de objetos con los comandos siguientes:

* Compilar cliente
  + *javac –d ../bin cliente/\*.java*
* Compilar s\_gestion\_usuarios
  + *javac –d ../bin s\_gestion\_usuarios/\*.java*
* Compilar s\_seguimiento\_usuarios
  + *javac –d ../bin s\_seguimiento\_usuarios/\*.java*

Para ejecutar la aplicación se deben seguir los siguientes pasos, primero se lanza el servidor de objetos 2, luego el servidor de objetos 1 y por último el cliente. Las operaciones descritas anteriormente se realizan con los siguientes comandos:

Esta vez nos ubicamos en la carpeta “bin” de nuestro proyecto.

* Ejecutar Servidor de objetos 2
  + *java s\_seguimiento\_usuarios.ServidorDeObjetos2*
* Ejecutar Servidor de objetos 1
  + *java s\_gestion\_usuarios.ServidorDeObjetos1*
* Ejecutar Cliente de objetos
  + *java cliente.ClienteDeObjetos*

Cada ejecución requiere que ingresemos una dirección IP, en este momento ingresamos “localhost”, también nos solicitará un número de puerto, aquí ingresamos un número superior a 1024, en los tres debemos ingresar los mismos datos para tener una conexión exitosa.

# **Repositorio GitHub**

Repositorio de GitHub: [AppDistribuida-JAVA.RMI](https://github.com/lmarango/AppDistribuida-JAVA-RMI)