Sesión L10- Cliente/Servidor

Para realizar esta práctica vamos a crear el directorio *L10-ClienteServidor*, dentro de nuestro directorio de trabajo. Por lo tanto tendremos que hacer:

```
> cd Practicas
> mkdir L10-ClienteServidor
> cd L10-ClienteServidor
```

Este será el subdirectorio en el que vamos a trabajar durante el resto de la sesión. Cualquier fichero o directorio que creemos en esta sesión estará dentro de *L10-ClienteServidor.* Copia las carpetas ***bucleInfinito***, ***sockets-ej2, sockets-ej3,*** y ***socketsConObjetos*** en tu directorio *L10-ClienteServidor*.

01 Procesos

Vamos a comenzar trabajando con el concepto de **proceso** (ejecución de un programa). Cuando ejecutamos cualquier aplicación haciendo doble click con botón derecho, cuando abrimos cualquier ventana, como por ejemplo un terminal, o lanzamos un comando en el terminal, estamos iniciando procesos (la ejecución de los programas correspondientes que hemos invocado, bien desde línea de comandos, o desde el ratón).

Comenzaremos por el comando **top**

Proporciona una vista en tiempo real de las tareas (procesos) en ejecución del sistema

Para ver toda la información relativa a este comando puedes consultar https://linux.die.net/man/1/top , o teclear desde el terminal **man top**.

El comando **man**, seguido de un nombre de comando, proporciona información sobre dicho comando: su propósito, opciones, ejemplos de uso, otros comandos relacionados, etc. El comando *man* nos va mostrando la información pantalla a pantalla:

- Pulsando la ***barra espaciadora***: avanzamos a la siguiente pantalla,
- Para volver atrás tenemos que pulsar la tecla "**b**" (*backward*).
- Para terminar, pulsaremos "**q**". No hay que poner comillas en ningún caso.

Ejecutaremos el comando top desde un terminal:

```
> top
```

Este comando muestra la información en varias columnas, y se actualiza en tiempo real. Cada una de las **filas** se corresponde con un determinado proceso. De las **columnas** mostradas nos interesan especialmente las siguientes:

- *PID*: Process IDentifier: es un número que utiliza el sistema para identificar de forma única a cada proceso (no hay dos procesos con el mismo pid),
- *USUARIO*: Es el nombre del usuario que está ejecutando el proceso. En nuestro caso, si estás utilizando la máquina virtual, verás que hay dos usuarios: el usuario **root**, y el usuario **pa**
- *ORDEN*: es el nombre del proceso (comando) que estamos ejecutando

De forma interactiva, podemos realizar varias acciones, como por ejemplo:

- Pulsando la tecla "**u**" podemos seleccionar un usuario, para que sólo nos muestre los procesos de dicho usuario. Prueba a mostrar sólo los procesos del usuario "pa" (no hay que poner comillas)

- Pulsando la tecla "**k**" (kill) podemos enviar una señal a un proceso para detenerlo. Prueba a enviar la señal SIGTERM al proceso top (para ello primero tienes que estar atento a que aparezca en pantalla el proceso top, y a continuación teclear k, seguidamente introducimos su pid, y cuando nos pida la señal que queremos enviar teclearemos 15 (es equivalente a teclear "sigterm"). Verás que acabamos de "matar" al proceso top que estábamos ejecutando. Debes llevar cuidado cuando detengas los procesos, puesto que puedes "matar" al propio proceso de la máquina virtual.

SIGTERM proviene de "SIGnal TERMinated". Se usa para detener la ejecución de un proceso. En este caso, el proceso podría ignorar la señal y continuar su ejecución. La forma de enviar SIGTERM a través de un comando desde el terminal es : kill ‹pid_proceso›.

SIGKILL proviene de "SIGnal KILLed". Se usa para detener la ejecución de un proceso. En este caso, el proceso NO puede ignorar la señal. La forma de enviar SIGKILL a través de un comando desde el terminal es : kill -9 ‹pid_proceso›

(ver (https://linuxhandbook.com/sigterm-vs-sigkill/)

Para terminar la ejecución del comando top utilizaremos la tecla "**q**"

De forma alternativa podemos ejecutar el comando top, utilizando opciones como:

> `top -u pa` (para mostrar sólo los procesos del usuario "pa")

Otro comando similar es el comando **ps** (***p**rocess **s**tatus*):

Proporciona información sobre una selección de procesos activos (podemos seleccionar todos los procesos o no

Puedes consultar la información relativa a este comando en: https://manpages.ubuntu.com/manpages/xenial/en/man1/ps.1.html , o teclear desde el terminal **man ps**.

Algunas opciones posibles son:

> `ps` (Por defecto muestra cuatro columnas con información sobre los procesos, entre ellas el PID, y el nombre del proceso (CMD))

> `ps -ef` ("**e**" muestra todos los procesos; "**f**" muestra 6 columnas de información en lugar de 4). Como la lista de procesos puede ser muy larga, utiliza en su lugar el comando `ps -ef | less`. En este caso te mostrará la información pantalla a pantalla, para movernos arriba o abajo, o salir, se utilizan las mismas teclas que hemos indicado antes para el comando man.

> `ps -fu pa | less` ("**u**" muestra sólo los procesos del usuario indicado (en este ejemplo "pa"); "**f**" es lo mismo de antes). Si la lista de procesos ocupa más de una pantalla, el comando "less" nos permitirá ver la salida mostrando una pantalla cada vez.

> `ps -C chrome` ("**C**" muestra sólo los procesos con el nombre indicado, en este ejemplo "chrome").

> `ps -p 16804` ("**p**" muestra sólo los procesos con el pid indicado, en este caso "16804")

> `ps --ppid 16804` ("**ppid**" muestra sólo los procesos cuyo padre tiene el pid indicado, en este caso "16804")

Hay otro comando que muestra el PID a partir de un nombre, este comando es `pgrep`. Si queremos saber el PID del proceso con nombre chrome usaremos el comando `pgrep chrome`.

Prueba a abrir varios procesos, por ejemplo el navegador Firefox o el editor de textos *featherpad*, averigua sus pids utilizando el comando **ps**, y a continuación detén las aplicaciones abiertas invocando al comando `kill <pid>`, desde el terminal (<pid> es el identificador del proceso que

queremos detener). A continuación vuelve a ejecutar el comando ps y comprueba que dichos procesos ya no se muestran en la lista.

: para abrir eclipse desde el terminal debes indicar la ruta del ejecutable, que en la máquina virtual se encuentra en: `/home/pa/eclipse/jee-2022-03/eclipse/eclipse`

Un proceso A a su vez puede lanzar (crear) más procesos (por ejemplo B y C). En ese caso, los procesos B y C son procesos "hijos" de A, y por tanto, A es el proceso "padre" de B y C.

Por ejemplo, si lanzamos eclipse, y ejecutamos el comando "ps -ef | grep eclipse", veremos algo como:

```
UID  PID    PPID   ... CMD
pa   26348  1126   ... /home/pa/eclipse/jee-2022-03/eclipse/eclipse
pa   26361  26348  ... /usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/java ...
```

En la lista anterior, vemos que el proceso que se ejecuta cuando iniciamos eclipse es `/home/pa/eclipse/jee-2022-03/eclipse/eclipse`, y que dicho proceso tiene un PID con valor 26348.

La columna PPID (**P**arent **P**rocess) **ID**entifier muestra el PID del proceso padre. Fíjate que eclipse es "padre" del proceso "java" (eclipse ejecuta una máquina virtual de java).

También podemos averiguar los pid de los procesos hijos de uno dado usando el comando: `ps --ppid <pid_padre>`. En este caso necesitamos conocer el pid del proceso padre. Para el ejemplo anterior, si ejecutamos `ps --ppid 26348`, obtendremos:

```
PID   ... CMD
26361 ... java
```

Volviendo al comando **kill**, ya hemos visto que éste puede resultarnos útil cuando queremos detener la ejecución de un proceso y no podemos hacerlo desde otro medio. Por ejemplo, supón que estamos implementando un programa en java, y ejecutamos un bucle infinito. Podríamos detener el proceso desde el terminal utilizando primero el comando ps, para averiguar su pid, y a continuación el comando kill con el pid del proceso.

Ahora usaremos el código que habéis copiado en el directorio ***bucleInfinito***, aquí tenéis una clase java con un bucle infinito.

Prueba a compilar y ejecutar la clase desde línea de comandos con:

> `javac Bucle.java` (compilamos el programa)

> `java Bucle` (ejecutamos el programa)

Verás que el terminal se queda "bloqueado" ejecutando el programa, y que éste no termina nunca. Abre otra terminal, averigua el pid del proceso con el comando **ps** y utiliza el comando **kill** para detenerlo.

02 Comunicación de procesos cliente/servidor con *sockets:* el servidor

Para este ejercicio utilizaremos el proyecto ***servidor,*** que os hemos proporcionado en la carpeta ***sockets-ej2***. Importa el proyecto *servidor* en Eclipse. Este proyecto está formado por una única clase java denominada *GreetingServer*.

La clase ***GreetingServer*** tiene un único método estático *main( )* con la implementación de un proceso servidor que utiliza *sockets* para comunicarse con un cliente. Deberías abrir el editor de código y entender primero la implementación de nuestro servidor. Verás que hacemos uso del método *setSoTimeout( )* sobre un objeto de tipo *ServerSocket*. Tal y como se indica en los comentarios del código, este método crea un temporizador inicializado a 10 segundos sobre dicho objeto. El temporizador tiene efecto sobre la llamada a *accept()* de dicho *socket*: si en 10 segundos ningún cliente ha solicitado conectarse con el servidor, entonces se lanzará la excepción *SocketTimeoutException*.

También usamos las clases *DataOutputStream* y *DataInputStream*. Cada una de ellas se inicializa con el *OutputStream* e *InputStream* del *socket* en la parte del servidor (objeto *server*), que se encargará de comunicarse con el cliente. Éstas clases. nos facilitan las tareas tareas de escritura y lectura de objetos de tipo String, a través de los métodos *writeUTF()*, y *readUTF()* respectivamente.

Una vez que te hayas familiarizado con el código, ejecuta el proceso servidor usando la *run configuration* proporcionada: ***greetingServer-L10-run***, y deja pasar el tiempo. Verás que transcurridos 10 segundos termina la ejecución del servidor.

El servidor se queda "bloqueado" en la línea 28, esperando una petición de conexión por parte de algún cliente. Si pasan 10 segundos y no ha habido ninguna petición,, el método accept() detecta que se ha agotado el temporizador y lanza la excepción *SocketTimeoutException*. En ese momento, la ejecución del programa salta a la línea 41, por lo que las líneas 29..40 NO se ejecutan. Después de ejecutar el catch de la línea 41, el programa continua en la línea 51.

Si por ejemplo, al crear el ServerSocket en la línea 20, hubiese saltado una excepción de tipo IOException, ésta sería capturada en el catch de la línea 46. En cualquier caso, la línea 51 se ejecuta siempre!!

Podemos ver qué sockets están siendo usados por nuestro sistema y su estado usando el comando:

```
> ss -ta
```

El comando ***ss*** *(**s**ocket **s**tatistics)*:

Muestra información sobre las conexiones de red con *sockets*, indicando el estado del *socket*, así como las direcciones IP de los dos *endpoints*.

Puedes consultar la información relativa a este comando en: https://linux.die.net/man/8/ss , http://www.binarytides.com/linux-ss-command/, o teclear desde el terminal **man ss**.

Algunas opciones posibles son:

- `> ss -ta` ("**t**" muestra sólo los sockets TCP, "**a**" muestra tanto los sockets que están estado LISTENING como en estado ESTABLISHED)
- `> ss -ltp` ("**l**" muestra sólo los sockets que están en estado LISTENING; "**t**" es lo mismo de antes; "**p**" muestra también información del proceso que está asociado al socket, visualizando el pid del dicho proceso).
- `> ss -tap` (Se pueden combinar las opciones en el orden que queramos, en este caso estaremos mostrando los sockets TCP, que estén en estado LISTENING o no, y mostraremos además el pid y nombre del proceso asociado a dichos sockets)

Dado que conocemos el número de puerto que queremos "investigar" (el 6066), otro comando interesante es el comando **lsof** (**l**i**s**t **o**pen **f**iles), con las siguiente opciones:


```
> lsof -i:6066
```

En este caso veremos por pantalla qué proceso está asociado al puerto 6066 (se trata del comando *java*), el *pid* del proceso, el usuario que ha iniciado el proceso, y el estado del socket asociado al puerto (LISTEN).

Puedes consultar más información sobre este comando desde: http://www.thegeekstuff.com/2012/08/lsof-command-examples/

Tanto si utilizamos el comando *lsof* como si utilizamos el comando ***ss*** con la opción "p", averiguaremos cuál es el *pid* del proceso que está todavía "acaparando" el *socket*. Una vez que conocemos el *pid* del proceso, podemos liberar los recursos del proceso "matándolo" con el comando ***kill***. Cuando un proceso termina, automáticamente se liberan los recursos. Cuando "matamos" un proceso, lo que hacemos es detener su ejecución.

Puedes comprobar esto ejecutando el proceso servidor desde Eclipse (o desde el terminal, como prefieras, con el comando maven correspondiente), y antes de 10 segundos usa los comandos

anteriores (ss y lsof) desde un terminal para comprobar que efectivamente hay un *socket* escuchando. Para que nos dé tiempo suficiente a ejecutar el comando, puedes cambiar el timeout a 30 segundos o más. Verás que durante todo este tiempo el *socket* está escuchando, y cuando termina la ejecución del servidor el *socket* se ha liberado.

También puedes hacer la prueba siguiente: lanza el proceso servidor desde eclipse, y desde un terminal comprueba usando el comando ***lsof*** que el *socket* está ocupado. A continuación mate el proceso y luego vuelve a comprobar el estado del *socket*. Después de matar al proceso el *socket* vuelve a estar libre para ser usado de nuevo.

Para trabajar este ejercicio anótate las pruebas que has hecho en un fichero de texto, y los resultados que has obtenido, así como cualquier otro comentario que te sirva para no tener que volver a realizar el ejercicio antes del examen.

Sube GitHub el trabajo realizado hasta el momento (en este caso deberás subir el fichero de texto con los comentarios indicados):

03 Comunicación de procesos cliente/servidor con *sockets*

Para este ejercicio utilizaremos el proyecto ***cliente-servidor-L10,*** que os hemos proporcionado en la carpeta ***sockets-ej3***.

Importa el proyecto desde Eclipse. En los paquetes ***pa.servidor*** y ***pa.cliente*** tenemos un código de un servidor y un cliente respectivamente. Observa que tenemos un único proyecto maven, pero tenemos dos métodos main(), que podemos ejecutar por separado. También podríamos haber separado físicamente los dos códigos en dos proyectos maven independientes. Razona qué ventaja tendría el tener dos proyectos maven en lugar de uno.

Una vez que te hayas familiarizado con el código, ejecuta los procesos servidor y cliente. En este caso, te será más cómodo ejecutarlos desde dos terminales diferentes, de forma que veas ambas ejecuciones a la vez. Desde Eclipse tendrás que ir cambiando de pestaña (desde *Console*) para ver las ejecuciones de ambos procesos.

Para ejecutar desde el terminal puedes usar los comandos (desde el directorio que contiene el pom.xml del proyecto):

```
> mvn clean compile //para eliminar el target y compilar tanto el cliente como el servidor
> mvn exec:java -Dexec.mainClass=pa.cliente.Cliente //para ejecutar el cliente
> mvn exec:java -Dexec.mainClass=pa.servidor.Servidor //para ejecutar el servidor
```

De forma alternativa, y una vez compilado el código, también podrías ejecutar directamente los ficheros *.class* con el comando java desde el directorio donde se encuentran dichos *.class* (***target/classes***):

```
> cd target/classes //nos situamos en el directorio donde se encuentran los .class
> java pa.cliente.Cliente  //ejecutamos el cliente
> java pa.servidor.Servidor //ejecutamos el servidor
```

Físicamente, se genera el fichero Cliente.class y Servidor.class en las subcarpetas target/classes/pa/cliente y traget/classes/pa/servidor, respectivamente.

Realiza las siguientes ejecuciones:

- Ejecuta primero el cliente y comprueba que éste termina, ya que el servidor no está "escuchando"
- Cambia el *timeout* del servidor a 30 segundos para que nos dé tiempo a ejecutar los dos procesos (cliente y servidor). Ahora ejecuta primero el servidor, y antes de que transcurran 30 segundos ejecuta el cliente. Verás que el cliente envía un mensaje al servidor, a continuación contesta al cliente y termina la comunicación.

Comprueba que se han liberados los puertos después de terminar los dos procesos.

Ahora modificaremos el código del cliente y del servidor para que mantengan una conversación "más larga".

Crea dos nuevas clases llamadas ***pa.cliente.TalkCliente*** y ***pa.servidor.TalkServidor***. Copia los métodos *main()* de las clases *pa.cliente.Cliente* y *pa.servidor.Servidor*, en la correspondientes clases *TalkCliente* y *TalkServidor.*

Comencemos por el **código del servidor**. Para poder permitir que el cliente nos envíe más de un mensaje, tendremos que introducir un bucle, de forma que repitamos las acciones de leer del *socket* el mensaje del cliente, y responderle. El servidor siempre va a dar la misma respuesta: "*Recuerda que debes teclear Bye para terminar la conversación...*". El bucle termina cuando el cliente envíe *"Bye",* de forma que el servidor ya no sigue leyendo más mensajes del cliente. En realidad lo que implementamos no es una conversación, ya que el servidor "escucha" el "monólogo" del cliente, hasta que éste decide terminar y le envía el mensaje "Bye".

Ahora modificaremos el **código del cliente**. En este caso, el cliente va a enviar al servidor un número indeterminado de mensajes, finalizando cuando el cliente envíe el mensaje "Bye" (el cliente no tiene que poner comillas en los mensajes).

En lugar de crear una colección del elementos con los mensajes que va a enviar el cliente, vamos a leerlos desde el teclado. Para ello debes usar la clase java ***Scanner*** que ya conoces de prácticas anteriores. La clase Scanner divide la entrada en partes (también llamadas *tokens*) usando como delimitador por defecto un espacio en blanco. En nuestro caso, para poder leer cadenas de caracteres que contengan espacios en blanco, cambiaremos el delimitador por el "retorno de carro" de la siguiente forma:

```
Scanner sc = new Scanner(System.in);
/*El mensaje puede incluir espacios en blanco
  y termina con un retorno de carro*/
sc.useDelimiter(Pattern.compile("\\n"));
sc.next(); //leemos el mensaje del cliente
```

Una vez que hayas realizado las modificaciones vuelve a ejecutar tanto el servidor como el cliente. Puedes crear dos nuevas run configurations con nombres: ***talkServidor-L10-run*** y ***talkCliente-L10-run***. En la **Figura 1** mostramos un ejemplo de ejecución desde el terminal.

Terminal del servidor

```
//desde cliente-servidor-L10
> mvn clean compile
> mvn exec:java -Dexec.mainClass=pa.cliente.TalkServidor
Server says: Esperando al cliente en el puerto 6066...
Server says: Acabo de conectar con /127.0.0.1:64042
Client says: Hola servidor
Client says: Te envio el primer mensaje
Client says: Te envio un segundo mensaje
Client says: Deberias contestarme!!!
Client says: Como no me contestas voy a terminar
Client says: Bye
Servidor detenido
```

Figura 1. Ejecuciones de los procesos cliente y servidor

Terminal del cliente

```
//desde cliente-servidor-L10
> mvn exec:java -Dexec.mainClass=pa.cliente.TalkCliente
Client says: Intentando conectar con localhost en el puerto 6066
Client says: Acabo de conectar con localhost/127.0.0.1:6066
Hola servidor
Server says: Recuerda que debes teclear Bye para terminar la conversación...
Te envio el primer mensaje
Server says: Recuerda que debes teclear Bye para terminar la conversación...
Te envio un segundo mensaje
Server says: Recuerda que debes teclear Bye para terminar la conversación...
Deberias contestarme!!!
Server says: Recuerda que debes teclear Bye para terminar la conversación...
Como no me contestas voy a terminar
Server says: Recuerda que debes teclear Bye para terminar la conversación...
Bye
El cliente termina
```

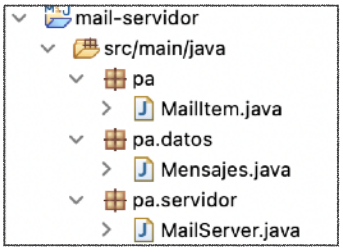
Recuerda finalmente subir tu trabajo a GitHub.

04 Servidor de correo con sockets

Para este ejercicio utilizaremos el código de la carpeta ***socketsConObjetos***.

Se proporcionan dos proyectos maven: ***mail-servidor*** y ***mail-cliente***. El código te resultará familiar puesto que ya lo hemos usado en el proyecto *Mail-System* de la práctica L03

SERVIDOR



La clase ***MailItem*** representa el mensaje enviado por un cliente.

La clase ***Mensajes*** almacena los mensajes enviados por los clientes. De momento los mensajes no se guardan de forma persistente, e inicialmente el servidor contiene tres mensajes.

La clase ***MailServer*** representa el servidor de correo y gestiona el envío y consultas de mensajes de los clientes.

CLIENTE

La clase ***MailItem*** es idéntica a la del servidor

Proporcionamos el paquete ***pa.cliente*** que contendrá la implementación de nuestro cliente de correo usando sockets.

Disponemos también de las ***run configurations:***

- ***mail-servidor-L10-compile, mail-servidor-L10-run,***
- ***mail-cliente-L10-compile*** y ***mail-cliente-L10-run***, para compilar y ejecutar cada uno de los proyectos maven.

En este caso tenemos dos proyectos maven, de forma que estamos separando físicamente el código del cliente y del servidor en dos proyectos diferentes que se podrán ejecutar en distintas máquinas. Fíjate que la clase MailItem es una clase que se utiliza tanto en el cliente como en el servidor, por lo que debe estar en ambos proyectos.

IMPORTANTE: En esta práctica vamos a enviar a través de los sockets los objetos de tipo *MailItem*, que representan los mensajes de los clientes. Para que un objeto pueda transmitirse a través del socket es NECESARIO que la clase correspondiente sea Serializable, para lo cual debe implementar la interfaz ***Serializable*** (definida en el paquete ***java.io***).

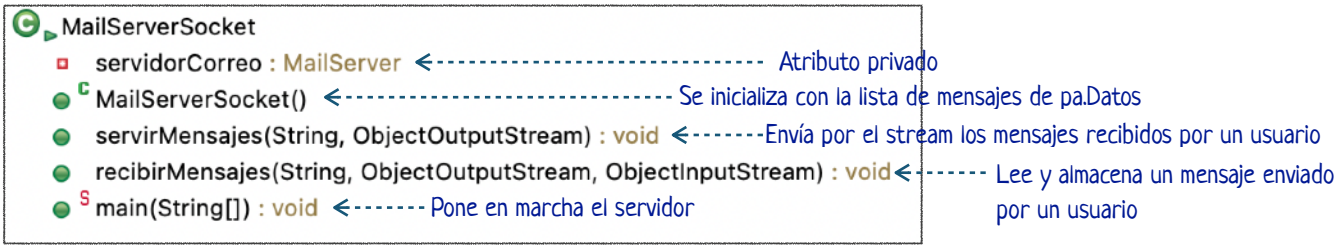
La interfaz Serializable es una interfaz algo "particular" ya que no contiene atributos ni métodos, por lo que no necesitaremos implementar ningún método adicional. Debemos declarar el atributo serialVersionUID, en este caso, vamos a darle el valor: 1234567L. Recuerda que dicho valor debe ser idéntico tanto en la clase del servidor como en la del cliente.

El proceso de serialización consiste en convertir el objeto java en una serie de bytes que pueden transmitirse a través del stream que corresponda. Dicha secuencia de bytes será "deserializada" cuando llegue al otro extremo y será convertida de nuevo en objeto java para poder usarse por el proceso correspondiente.

Por tanto, lo primero que haremos será modificar convenientemente la clase ***Mailitem*** para que pueda serializarse, tanto en el cliente como en el servidor.

MODIFICACIONES EN EL SERVIDOR

Implementa la clase ***pa.servidor.MailServerSocket***. A continuación mostramos su representación abstracta:



- El método ***servirMensajes*** recibe como parámetros el **nombre** del usuario que ha solicitado ver los mensajes que ha recibido, y el ***stream*** por el que se enviarán los datos a dicho usuario.

  Al usuario se le envía (mira la *traza de ejecución* de ejemplo al final del enunuciado):

  - el número de mensajes que ha recibido, y a continuación se mostrará por pantalla "`Enviando <numMensajes>` mensajes para: `<nombreUsuario>`", con el número de mensajes que se le van a enviar al cliente y el nombre del usuario al que se le envían los mensajes.
  - cada uno de dichos mensajes (en el caso de que haya recibido algúno). Cada mensaje es un objeto de tipo *MailItem*. Después del envío de cada mensaje al cliente, se mostrará por pantalla "`Mensaje: <contenidoDelMensaje>`", con el texto del contenido del mensaje enviado (que será de tipo String). Para enviar una cadena de caracteres por el stream debes usar el método `writeUTF(String)`

  La excepción ***IOException*** que puede lanzar la escritura en el stream no la capturaremos, sino que vamos a propagarla.

- El método ***recibirMensajes*** recibe como parámetros el **nombre** del usuario que envia un mensaje desde el proceso cliente, y los streams de entrada y salida a través de los cuales nos vamos a comunicar con dicho usuario para que nos haga llegar los datos de dicho mensaje. Una vez que lo hayamos recibido, lo almacenaremos en nuestra lista de mensajes del servidor. Los pasos a seguir son:

  - Mostramos por pantalla "`Pedimos los datos`" y enviamos al cliente dos mensajes: "`Destinatario mensaje: `" y "`Escriba su mensaje entre comillas dobles y con un sólo retorno de carro al final: `". Usa el método flush() después de enviar cada mensaje.
  - Leemos el mensaje del cliente por el stream (recibiremos un objeto de tipo MailItem) y mostramos por pantalla "`Mensaje recibido`".
  - Añadimos el mensaje a la lista de mensajes del servidor y mostramos por pantalla "`Mensaje añadido`"
  - Enviamos al cliente el mensaje "`Mensaje enviado con éxito`"

  La excepción ***IOException*** que puede lanzar la lectura/escritura en el stream no la capturaremos, sino que vamos a propagarla.

  La excepción ***ClassNotFoundException***, que puede lanzar la lectura de un objeto del stream la capturaremos y mostraremos por pantalla el mensaje asociado con la excepción.

NOTA: Cuando escribamos en el stream, para asegurarnos de que los datos se escriben inmediatamente en él usaremos el método `flush()` sobre dicho stream de salida.

- El método ***main()*** realizará lo siguiente:
  - El servidor tendrá asignado el puerto 6066. Una vez creado el objeto ServerSocket imprimirá por pantalla el mensaje: "`Servidor de correo en marcha en el puerto: <numeroDePuertoDelServidor>`", indicando el número de puerto en el que estará escuchando el servidor (método *getLocalPort()*).
  - El servidor quedará en espera y después de aceptar una petición del cliente, imprimirá por pantalla el mensaje: "`Acabo de conectar con: <numeroDePuertoDelCliente>`", indicando el número del puerto del proceso cliente.

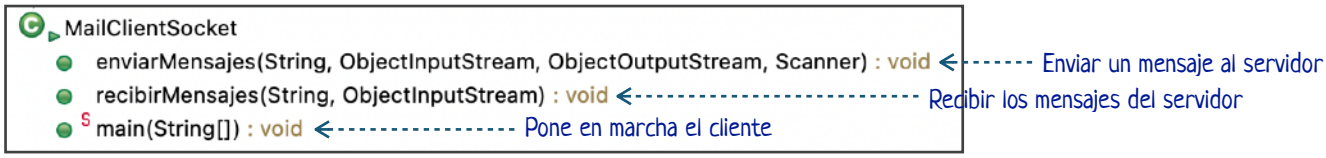
- Leeremos del stream el nombre del cliente (con el método readUTF()), y mostramos por pantalla el mensaje "`<nombre> ha accedido al sistema`", indicando el nombre del usuario que se ha conectado
- Enviamos al cliente el menú de opciones:

```
Indique la operación a realizar:
1. consultar mensajes recibidos
2. enviar un mensaje
Teclee opción: (para terminar teclee cualquier otro valor)
```

- Leemos la opción que nos envía el cliente (valor entero)
- Dependiendo de la opción que solicita el cliente consultaremos los mensajes recibidos (*servirMensajes()*), o leeremos y almacenaremos el mensaje a enviar por el cliente (*recibirMensajes()*)
- Repetiremos el proceso desde la lectura del nombre del cliente, hasta que desde el cliente se indique una opción diferente de los valores 1 ó 2. En cuyo caso el servidor detiene su ejecución y muestra por pantalla el mensaje: "`Servidor detenido`".

MODIFICACIONES EN EL CLIENTE

Implementa la clase ***pa.cliente.MailClientSocket***. A continuación mostramos su representación abstracta:



- El método ***enviarMensajes*** recibe como parámetros el **nombre** del usuario que quiere enviar un mensaje, los **streams** asociados al socket del cliente y un objeto de tipo ***Scanner*** que usaremos para leer desde el teclado.

  El método realiza lo siguiente (mira la *traza de ejecución* de ejemplo al final del enunuciado):
  - lee del stream el mensaje del servidor y lo imprime por pantalla (el mensaje recibido será para solicitar el nombre del destinatario).
  - lee desde teclado el nombre del destinatario usando el objeto *Scanner* (con el método *nextLine()*)
  - lee del strem el mensaje del servidor y lo imprime por pantalla (el mensaje recibido será para solicitar el contenido del mensaje a enviar)
  - lee desde teclado el contenido del mensaje usando el objeto *Scanner*. A continuación crea el objeto MailItem y lo envía al servidor.
  - lee del stream la confirmación del servidor de que el mensaje se ha enviado con éxito y escribe por pantalla dicho mensaje.

  La excepción ***IOException*** que puede lanzar la escritura en el stream no la capturaremos, sino que vamos a propagarla.
- El método ***recibirMensajes*** recibe como parámetros el **nombre** del usuario que recibe los mensajes y el stream de entrada por el que recibimos los mensajes del servidor.

  Los pasos a seguir son:
  - Leemos del servidor el número de mensajes que tiene pendientes de leer el usuario (valor entero) y mostramos por pantalla "`Mensajes recibidos = <numeroDeMensajes>`".
  - Si el usuario tiene mensajes pendientes de leer, entonces mostramos por pantalla "`<nombre>, aquí tiene sus mensajes`".
  - Leemos los mensajes del servidor y los vamos mostrando por pantalla precedidos por la línea "`leyendo mensaje: <i>`". Siendo `<i>` el número de la iteración en la que se lee el mensaje. Y a continuacíon imprimimos el mensaje por pantalla (recuerda que ya tienes un método en la clase MailItem para hacer eso). Si al leer algún objeto de tipo *MailItem* del servidor se produce la excepción ***ClassNotFoundExcepcion*** la capturaremos e imprimiremos el mensaje asociado a dicha excepción.

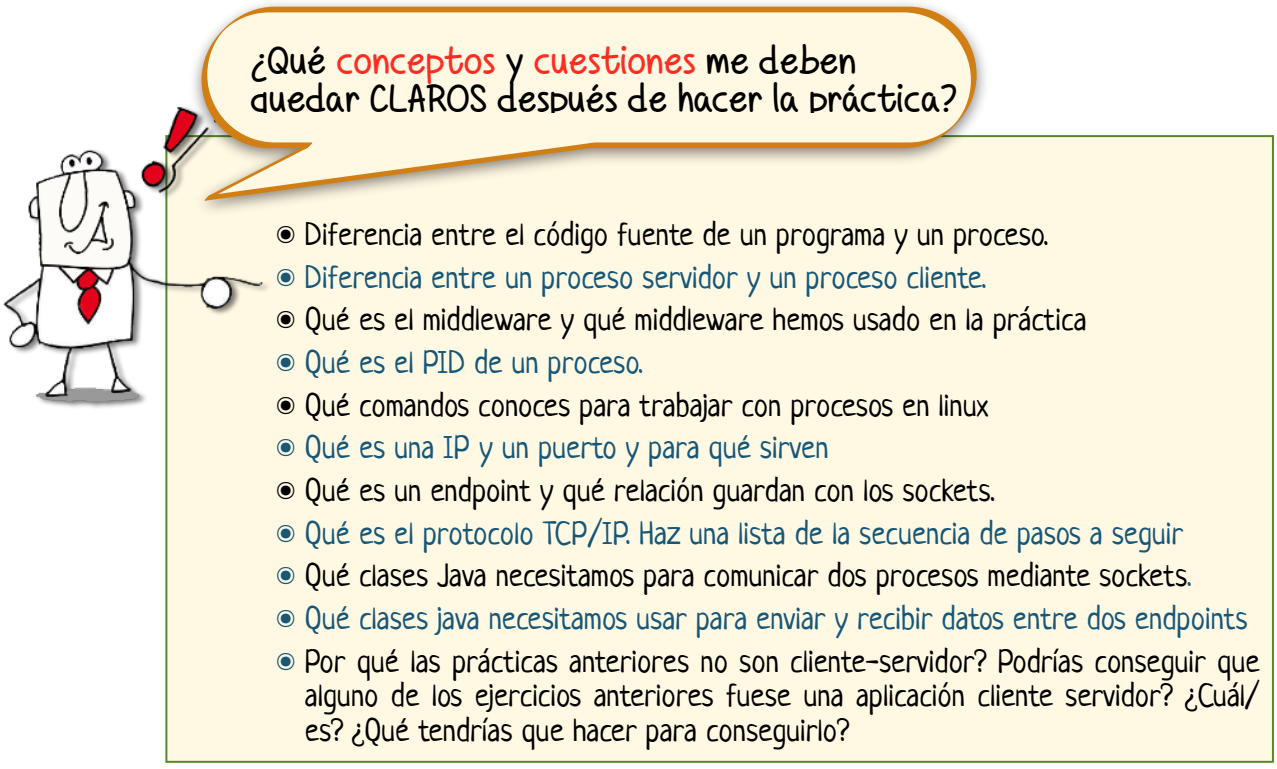
La excepción ***IOException*** que puede lanzar la lectura en el stream no la capturaremos, sino que vamos a propagarla.

- El método ***main()*** realizará lo siguiente:
  - Mostramos por pantalla el mensaje:"`Intentando conectar con <nombreServidor> en el puerto <puertoServidor>`". El nombre del servidor será "localhost" y el puerto el 6066.
  - Una vez que se lleve a cabo la conexión con el servidor, mostramos por pantalla: "`Acabo de conectar con: <numeroDePuertoDelServidor>`", indicando el número del puerto del proceso servidor.
  - Mostramos por pantalla el mensaje: "`Introduce tu nombre`" y leemos el nombre usando un objeto de tipo ¨Scanner.
  - Enviamos el nombre del usuario al servidor (método *writeUTF()*). Después de cada escritura usa el método *flush()* para enviar inmediatamente la información al stream.
  - Leemos el menú de opciones que nos envía el servidor y lo mostramos por pantalla.
  - Leemos la opción (valor entero) del usuario usando el objeto Scanner
  - Enviamos dicha opción al servidor.
  - En función de la opción deseada, procedemos a recibir los mensajes solicitados (método ***recibirMensajes()***), o enviamos el mensaje al servidor (método ***enviarMensajes()***).
  - Repetimos de nuevo el proceso volviendo a preguntar el nombre del cliente ("`Introduce tu nombre`").
  - El proceso termina cuando el usuario introduce una opción diferente de 1 ó 2. Entonces mostraremos por pantalla "`El cliente termina`".

  La excepción ***IOException*** que puede lanzarse durante todo este proceso la capturaremos y mostraremos "`Error durante la conexión (<mensajeDeLaExcepcion>)`

Para ejecutar la aplicación, te será más cómodo si usas dos terminales.
En la siguiente página mostramos un ejemplo de ejecución:

Recuerda guardar tu trabajo en GitHub



```
TERMINAL DEL CLIENTE
>mvn clean compile
>mvn exec:java -Dexec.mainClass=pa.cliente.MailClientSocket
Intentando conectar con localhost en el puerto 6066
Acabo de conectar con localhost/127.0.0.1:6066
Introduce tu nombre: Marta
Indique la operación a realizar:
1. consultar mensajes recibidos
2. enviar un mensaje
Teclee opción: (para terminar teclee cualquier otro valor) 1
Mensajes recibidos = 3
Marta, aquí tiene sus mensajes
leyendo mensaje: 0
From: Carlos
To: Marta
Message: Mensaje número 1

leyendo mensaje: 1
From: Pedro
To: Marta
Message: Mensaje número 2

leyendo mensaje: 2
From: Luis
To: Marta
Message: Mensaje número 3

Introduce tu nombre: Marta
Indique la operación a realizar:
1. consultar mensajes recibidos
2. enviar un mensaje
Teclee opción: (para terminar teclee cualquier otro valor 1
Mensajes recibidos = 0
Introduce tu nombre: Pedro
Indique la operación a realizar:
1. consultar mensajes recibidos
2. enviar un mensaje
Teclee opción: (para terminar teclee cualquier otro valor) 2
Destinatario mensaje: Lucas
Escriba su mensaje entre comillas dobles y con un sólo
retorno de carro al final:
"Hola Lucas, estoy de vacaciones!!"
Mensaje enviado con éxito
Introduce tu nombre: Lucas
Indique la operación a realizar:
1. consultar mensajes recibidos
2. enviar un mensaje
Teclee opción: (para terminar teclee cualquier otro valor) 1
Mensajes recibidos = 1
Lucas, aquí tiene sus mensajes
leyendo mensaje: 0
From: Pedro
To: Lucas
Message: "Hola Lucas, estoy de vacaciones!!"

Introduce tu nombre: Ana
Indique la operación a realizar:
1. consultar mensajes recibidos
2. enviar un mensaje
Teclee opción: (para terminar teclee cualquier otro valor) 3
El cliente termina
```

```
TERMINAL DEL SERVIDOR
>mvn clean compile
>mvn exec:java
-Dexec.mainClass=pa.servidor.MailServerSocket
Servidor de correo en marcha en el puerto: 6066...
Acabo de conectar con /127.0.0.1:64547
Marta ha accedido al sistema
Petición del cliente: 1
Enviando 3 mensajes para: Marta
Mensaje: Mensaje número 1
Mensaje: Mensaje número 2
Mensaje: Mensaje número 3
Marta ha accedido al sistema
Petición del cliente: 1
Enviando 0 mensajes para: Marta
Pedro ha accedido al sistema
Petición del cliente: 2
Pedimos los datos
Mensaje recibido
Mensaje añadido
Lucas ha accedido al sistema
Petición del cliente: 1
Enviando 1 mensajes para: Lucas
Mensaje: "Hola Lucas, estoy de vacaciones!!"
Ana ha accedido al sistema
Petición del cliente: 3

Servidor detenido
```

NOTAS:

- El texto en rojo son las entradas del usuario por teclado.
- Cuando desde el cliente se elige la opción 1. El proceso cliente se encarga de crear el objeto MailItem con el mensaje a enviar y lo envía al servidor.
- Cuando desde el cliente se elige la opción 2. El proceso cliente se encarga de leer los objetos de tipo MailItem enviados por el servidor y los imprime por pantalla