1.- Envío de objetos a través de Sockets

Los stream soportan diversos tipos de datos como son los bytes, los tipos de datos primitivos, caracteres y objetos.

El objetivo de este apartado es intercambiar objetos entre programas emisor y receptor o entre programas cliente y servidor usando sockets.

2.- Objetos en Sockets TCPejemplo1Cliente

Las clases **ObjectInputStream** y **ObjectOutputStream** nos permiten enviar objetos a través de socket TCP.

Utilizaremos los siguientes métodos :

- **readObject()** para leer el objeto del stream.
- **WriteObject ()** para escribir el objeto al stream.
- Usaremos el constructor que admite un **InputStream** y un **OutputStream**

Ejemplo de preparación el flujo de salida para escribir objetos

*ObjectOutputStream **ObjetoSalida** = new ObjectOutputStream(socket.getOutputStream());*

Ejemplo de preparación de flujo de entrada para leer objetos

*ObjectInputStream **ObjetoEntrada** = new ObjectInputStream(socket.getInputStream());*

Las clases a la que pertenecen estos objetos tienen que implementar la interfaz "Serializable".

Interfaz Serializable
Java ha añadido una interesante faceta al lenguaje denominada serialización de objetos que permite convertir cualquier objeto cuya clase implemente el interface *Serializable* en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original. Esta característica se mantiene incluso a través de la red, por lo que podemos crear un objeto en un ordenador que corra bajo Windows , serializarlo y enviarlo a través de la red a una estación de trabajo que corra bajo UNIX donde será correctamente reconstruido. No tenemos que procuparnos, en absoluto, de las diferentes representaciones de datos en los distintos ordenadores.

Para que un objeto se pueda serializar debe de implementar la interfaz Serializable.

Ejemplo de escritura de objeto al flujo de salida

1. Creamos un objeto de la clase Lista

   Lista lista1 = new Lista(new int[] {12 , 15 , 11 , 4 , 32 }

2. Creamos un flujo de salida a disco, pasandole el nombre del archivo en disco o un objeto de la clase File

   FileOutputStream ficheroSalida = new FileOutputStream ("media.obj");

3. El flujo de salida ObjectOutputStream es el que procesa los datos y se ha de vincular a un objeto ficheroSalida de la clase FileOutputStream

   ObjectOutputStream salida = new ObjectOutputStream(ficheroSalida);

4. El método writeObject escribe los objetos al flujo de salida y los guarda en un archivo en disco. Por ejemplo, un string y un objeto de la clase lista

   salida.writeObject("guardar este string y un objeto \n");

   salida.writeObject(lista1);

5. Por último se cierran los flujos

   salida.close();

Ejemplo de escritura de objeto al flujo de entrada

1. Creamos un flujo de entrada a disco, pasándole el nombre del archivo en disco o un objeto de la clase File

   FileInputStream ficheroEntrada = new FileInputStream("media.obj");

2. El flujo de entrada ObjectInputStream es el que procesa los datos y se ha de vincular a un objeto "ficheroEntrada" de la clase FileInputStream.

   ObjectInputStream entrada = new ObjectInputStream(ficheroEntrada);

3. El método readObject lee los objetos del flujo de entrada , en el mismo orden en el que ha sido escrito. Primero un string y luego, un objeto de la clase Lista.

   String str=(String) entrada.readObject();

   Lista obj1=(Lista)entrada.readObject();

4. Ahora podemos realizar tareas con dichos objetos, por ejemplo , desde el objeto obj1 de la clase Lista se llama a la función miembro "valorMedio" para obtener el valor medio del array de datos, o se muestran en la pantalla

   System.out.println("Valor medio "+obj1.valorMedio());

   System.out.println(str+obj1);

5. Finalmente, se cierra los flujos

```
entrada.close();
```

Por ejemplo, si tenemos un objeto como el siguiente:

```
public class Punto2D implements Serializable {
        private int x;
        private int y;

        public int getX() {
                return x;
        }
        public void setX(int x) {
                this.x = x;
        }
        public int getY() {
                return y;
        }
        public void setY(int y) {
                this.y = y;
        }
}
```

Podríamos enviarlo a través de un flujo, independientemente de su destino, de la siguiente forma:

```
Punto2D p = crearPunto();
FileOutputStream fos = new FileOutputStream(FICHERO_DATOS);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(p);
oos.close();
```

En este caso hemos utilizado como canal de datos un flujo con destino a un fichero, pero se podría haber utilizado cualquier otro tipo de canal (por ejemplo para enviar un objeto Java desde un servidor web hasta una máquina cliente). En aplicaciones distribuidas los objetos serializables nos permitirán mover estructuras de datos entre diferentes máquinas sin que el desarrollador tenga que preocuparse de la codificación y transmisión de los datos.

Muchas clases de la API de Java son *serializables*, como por ejemplo las colecciones. Si tenemos una serie de elementos en una lista, podríamos serializar la lista completa, y de esa forma guardar todos nuestros objetos, con una única llamada a writeObject.

Ejercicio 1:

Sea la clase Persona con 2 atributos , nombre y edad, 2 constructores y los métodos get y set correspondiente

```
import java.io.Serializable;
@SuppressWarnings("serial")

public class Persona implements Serializable {
	String nombre;
	int edad;
	public Persona(String nombre, int edad) {
		super();
		this.nombre = nombre;
		this.edad = edad;
	}
	public Persona() {super();}

	public String getNombre() {return nombre;}
	public void setNombre(String nombre) {this.nombre = nombre;}
	public int getEdad() {return edad;}
	public void setEdad(int edad) {this.edad = edad;}
}
```

Crear una clase cliente y otra servidor para intercambiar objetos Personas. Se deben de utilizar Socket TCP .

1. El programa servidor (TCPObjetoServidor1) crea un objeto Persona, dándole valores , mostrándolo por pantalla y se lo envía al programa cliente .
2. El programa cliente (TCPObjetoCliente1) realiza los cambios oportunos en el objeto y se lo devuelve modificado al servidor que lo muestra por pantalla.

SOLUCIÓN

Clase Servidor.

```
import java.io.*;
import java.net.*;

public class TCPObjetoServidor1 {
  public static void main(String[] arg) throws IOException,ClassNotFoundException {

		int numeroPuerto = 6000;// Puerto
		ServerSocket servidor =  new ServerSocket(numeroPuerto);
		System.out.println("Esperando al cliente.....");
		Socket cliente = servidor.accept();

   // Se prepara un flujo de salida para objetos

   // Se prepara un objeto y se envía

 //enviando objeto
```

```
   // Se obtiene un stream para leer objetos


   // CERRAR STREAMS Y SOCKETS
  }
}//..
```

Clase Cliente

```
import java.io.*;
import java.net.*;

public class TCPObjetoCliente1 {
  public static void main(String[] arg) throws IOException,ClassNotFoundException {
    String Host = "localhost";
    int Puerto = 6000;//puerto remoto

    System.out.println("PROGRAMA CLIENTE INICIADO....");
    Socket cliente = new Socket(Host, Puerto);

    //Flujo de entrada para objetos

    //Se recibe un objeto


    //Modifico el objeto


    //FLUJO DE salida para objetos


    // Se envía el objeto


    // CERRAR STREAMS Y SOCKETS

  }
}//..
```

Ejercicio 2

Socket TCP

Crea una clase Java Numeros que defina 3 atributos, uno de ellos entero, y los otros dos de tipo long

```
int numero;
long cuadrado;
long cubo;
```

1. Define un constructor con parámetros y otro sin parámetros.
2. Define los métodos get y set de los atributos.
3. Crea un programa cliente que introduzca por teclado un número e inicialice un objeto "Números", el atributo "numero" debe contener el número introducido por teclado. Debe

enviar ese objeto al programa servidor. El proceso se repetirá mientras el número introducido por teclado sea mayor que 0.

4. Crea un programa servidor que reciba un objeto "Numeros". Debe calcular el cuadrado y el cubo del atributo "numero" y debe enviar el objeto al cliente con los cálculos realizados, el cuadrado y el cubo del número introducido por teclado. El programa servidor finalizará cuando el número recibido en el objeto "Numeros" sea menor o igual que 0.
5. Controlar posibles errores, por ejemplo si ejecutamos el cliente y el servidor no está inicializado, o si estando el servidor inicializado ocurre algún error en el cliente, o este finaliza inesperadamente, etc.

3.- Objetos en Sockets UDP

Para intercambiar objetos en socket UDP utilizaremos las clases ByteArrrayOutputStream y ByteArrayInputStream. Se necesita convertir el objeto a un array de bytes. Por ejemplo, para convertir un objeto "Persona" a un array de bytes escribimos las siguientes líneas:

```
Persona persona = new Persona ("Juan" , 58);

// CONVERTIMOS OBJETO A BYTES

ByteArrayOutputStream bs = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream (bs);

// ESCRIBIMOS EL OBJETO PERSONA EN EL STREAM

out.writeObject(persona);

// CERRAMOS EL STREAM

out.close();

// OBJETO EN BYTES

byte[]  bytes = bs.toByteArray();
```

Para convertir los bytes recibidos por el datagrama en un objeto "Persona" escribimos:

```
// RECIBO DATAGRAMA

byte[]  recibidos = new byte [1024];
DatagramPacket paqRecibido = new DatagramPacket (recibidos, recibidos.length);

// RECIBO EL DATAGRAMA

socket.receive(paqRecibido);
```

```
// CONVERTIMOS BYTES A OBJETO

ByteArrayInputStream  bais = new  ByteArrayInputStream(recibidos);
ObjectInputStream in = new ObjectInputStream(bais);

// OBTENEMOS EL OBJETO

Persona persona = (Persona) in.readObject();
in.close();
```

Ejercicio 3

Usando sockets UDP realiza un programa servidor que espere un datagrama de un cliente . El cliente le envía un objeto "Persona" que previamente había inicializado. El servidor modifica los datos del objeto "Persona" y se lo envía de vuelta al cliente. Visualiza los datos del objeto "Persona" tanto en el programa cliente cuando los envía y los recibe como en el programa servidor cuando los recibe y los envía modificados.

Ejercicio 4

Realiza el ejercicio 2 con socket UDP

4.- Conexión de múltiples clientes: hilos

Hasta ahora los programas servidores que hemos creado solo son capaces de atender a u cliente en cada momento, pero lo más típico es que un programa servidor pueda atender a muchos clientes simultáneamente. La solución para poder atender a múltiples clientes está en el "multihilo", cada cliente será atendido en un hilo.

Ejercicio 5

Crear un programa Servidor_Cliente : El cliente le tiene que enviar una cadena al servidor y este la convertirá a mayúscula y se la enviará al cliente. Esta aplicación debe de soportar conexiones de múltiples clientes. Para probar la aplicación pasa a dos de tus compañeros la clase Cliente con la IP del equipo donde corre la clase Servidor.

Debes de ejecutar la clase Servidor antes de la clase Cliente. La clase HiloServidor debe de estar en el paquete con la clase Servidor.

La aplicación constará de tres clases:

- Clase Servidor
- Clase HiloServidor
- Clase Cliente.

Clase Servidor

El esquema básico en sockets TCP sería construir un único servidor con la clase "**ServerSocket**" e invocar al método "**accept ()**" para esperar las peticiones de conexión de los clientes.

Cuando un cliente se conecta, el método **accept()** devuelve un objeto "**Socket**", éste se usará para crear un hilo cuya misión es atender a ese cliente. Después se vuelve a invocar a "**accept()**" para esperar a un nuevo cliente; habitualmente la espera de conexiones se hace dentro de un bucle infinito.

```
public class Servidor {
	public static void main(String args[]) throws IOException  {

		// Creamos el socket en el puerto 6000

	}
}
```

Clase HiloServidor que extiende a Thread

Todas las operaciones que sirven a un cliente en particular quedan dentro de la clase hilo . El hilo permite que el servidor se mantenga a la escucha de peticiones y no interrumpa su proceso mientras los clientes son atendidos.

Ejemplo: Supongamos que el cliente envía una cadena de caracteres al servidor y el servidor se la devuelve en mayúsculas, hasta que recibe un asterisco que finalizará la comunicación con el cliente. El proceso de tratamiento de la cadena se realiza en un hilo, en este caso se llama "HiloServidor".

```
public class HiloServidor extends Thread {

	// Declaramos el bufer de entrada, el de salida y el socket

	// CONSTRUCTOR
	public HiloServidor(Socket s) throws IOException {

		// se crean flujos de entrada y salida
	}

	// tarea a realizar con el cliente
	public void run() {

	}
}// ..
```

Clase Cliente

El programa cliente "Cliente.java" se conectará con el servidor en el puerto 6000 y le enviará cadenas introducidas por teclado; cuando le envíe un asterisco el servidor finalizará la comunicación con el cliente.

public class Cliente {

```
  public static void main(String[] args) throws IOException {

        // CREO FLUJO DE SALIDA AL SERVIDOR

        // CREO FLUJO DE ENTRADA AL SERVIDOR

        // FLUJO PARA ENTRADA ESTANDAR

        }//
}//
```

```
COMUNICO CON: Socket[addr=/127.0.0.1,port=49745,localport=6000]
FIN CON: Socket[addr=/127.0.0.1,port=49745,localport=6000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=49746,localport=6000]
COMUNICO CON: Socket[addr=/192.168.0.13,port=51434,localport=6000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=49747,localport=6000]
COMUNICO CON: Socket[addr=/192.168.0.12,port=50743,localport=6000]
```

```
<terminado> Cliente [Aplicación Java] C:
Introduce cadena: ee
  =>ECO: EE
Introduce cadena: *
  =>ECO: *
Fin del envío...
```

5.- Hilos compartiendo objetos

Ejercicio 6

En la aplicación, se tiene que desarrollar un servidor que define un número (por ejemplo el 100) que los clientes que se conecten tendrán que adivinar . Los clientes tendrán hasta 5 intentos para adivinar el número.

Cada cliente se debe de tratar en un hilo y todos compartirán un objeto que contendrá información sobre el estado del juego:

- El número a adivinar.
- Si el juego ha terminado o no.
- Quien es el jugador ganador.

Observación: El método que comprueba la jugada se debe de declarar como **synchronized**, para que la comprobación de la jugada no interfiera entre los jugadores y se haga de forma unívoca.

El servidor enviará a cada cliente que se conecte un identificador y el objeto compartido. El cliente enviará al servidor números leídos por teclado, el servidor le devolverá información sobre si ha ha

acertado el número o no , si el número es mayor o menor que el que hay que adivinar, si el juego ha finalizado , los intentos que le quedan , etc.
Todo el control de juego del jugador se hará en el hilo.

Clase ServidorAdivina.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class ServidorAdivina {
  public static void main(String args[]) throws IOException  {

        ServerSocket servidor = new ServerSocket(6001);
        System.out.println("Servidor iniciado...");

        // Numero a adivinar entre 1 y 25
        int numero = (int) (1 + 25*Math.random());
        System.out.println("NUMERO A ADIVINAR=> " +numero);

        // Creamos el objeto

        while (true) {
              //Creamos el socket cliente

               //esperando cliente

              //identificador para el cliente

        }
  }
}//..
```

Clase ObjetoCompartido

Observación: El método "nuevaJugada()" que recibe el identificador del jugador y el número a adivinar , compara el número del jugador con el número a adivinar y devuelve una cadena indicando lo que ha pasado en la comparación. Cuando un jugador adivina el número se cambia el estado del juego mediante los atributos "acabo" y "ganador", de esta manera sabremos cuando el juego ha finalizado:

```
public class ObjetoCompartido {
        private int numero; // número a adivinar
        private boolean acabo; // true cuando se haya terminado el juego
        private int ganador; // jugador ganador

        public ObjetoCompartido(int numero) {

        }

        public boolean seAcabo() {

        }
```

```
        public int getGanador() {

        }

        public synchronized String nuevaJugada(int jugador, int suNumero) {


        }

}
```

Clase Datos

La comunicación entre el cliente y el servidor se realiza mediante objetos de tipo "Datos". En este se definen una serie de atributos, constructores y los métodos get y set . El atributo cadena se utilizará para intercambiar mensajes entre el servidor y cliente y para que el cliente envíe el número a adivinar al servidor.

```
import java.io.Serializable;

@SuppressWarnings("serial")
public class Datos implements Serializable {
        String cadena; //cadena que se intercambia con el servidor
        int intentos;  //intentos que lleva el jugador
        int identificador; //id del jugador
        boolean gana; //true si el jugador adivina el numero
        boolean juega; //true si el jugador juega, false juego finalizado


        public Datos(String cadena, int intentos, int identificador) {

        }

        public Datos() {

        }

        public boolean isJuega() {

        }

        public void setJuega(boolean juega) {

        }

        public boolean isGana() {

        }

        public void setGana(boolean gana) {

        }
```

```
        public int getIdentificador() {

        }

        public void setIdentificador(int identificador) {

        }

        public String getCadena() {

        }
        public void setCadena(String cadena) {

        }
        public int getIntentos() {

        }
        public void setIntentos(int intentos) {

        }

}
```

Clase HiloServidorAdivina

En el hilo, cuando el jugador (o cliente) se conecta, se le envía un objeto de este tipo con el identificador que le corresponde, un mensaje, el número de intentos, inicialmente 0 , y el estado de juego. Habrá un proceso repetitivo en el que se recibe el número del jugador y se comprueba en el método "nuevaJugada()" del objeto compartido. Después de la comprobación se enviará de nuevo al cliente un objeto "Datos" con la información de lo que ha ocurrido y el estado del juego. El proceso repetitivo del cliente finaliza cuando se acaba el juego o cuando el número de intentos de adivinar el número es 5, entonces el servidor cerrará la conexión con el cliente.

```
import java.io.*;
import java.net.*;

public class HiloServidorAdivina extends Thread {
        ObjectInputStream fentrada;
        ObjectOutputStream fsalida;

        Socket socket = null;

        ObjetoCompartido objeto;
        int identificador;
        int intentos = 0;

        public HiloServidorAdivina(Socket s, ObjetoCompartido objeto, int id) {

                try {
```

```
                } catch (IOException e) {
                        System.out.println("ERROR DE E/S en HiloServidor");
                        e.printStackTrace();
                }
        }// ..

        // ---------------------------------------------------------------
        public void run() {



                //PREPARAR PRIMER ENVIO AL CLIENTE



                try {

                } catch (IOException e1) {
                        System.out.println("Error en el Hilo al realizar " +  "el primer envio al jugador: " +
identificador);
                        return;
                }

                 // mientras no se haya acabado el juego


                try {
                        fsalida.close();
                        fentrada.close();
                        socket.close();
                } catch (IOException e) {
                        System.out.println("Error en Hilo al cerrar cliente: " + identificador);
                        e.printStackTrace();
                }

        }// run

}//
```

Clase JugadorAdivina

Se debe considerar los posibles errores de comunicación entre cliente y servidor, visualizando en cada mensaje el identificador del cliente que produce la situación de error . En el programa cliente se introducen números por teclado y se envían al servidor. Se recibe del servidor el estado del juego y se muestra por pantalla lo que ha ocurrido. **El proceso de entrada finaliza cuando se lea un "*".**

```
import java.io.*;
import java.net.*;
import java.util.Scanner;
```

```
public class JugadorAdivina {
        public static void main(String[] args) throws IOException, ClassNotFoundException {
                String Host = "localhost";
                int Puerto = 6001;// puerto remoto


                // FLUJO PARA ENTRADA ESTANDAR


                //OBTENER PRIMER OBJETO ENVIADO POR EL SERVIDOR

        }

}//JugadorAdivina
```

Ejercicio 7

Adapta el ejercicio 6 para que la entrada de datos del cliente se realice usando una pantalla gráfica. El aspecto de la pantalla se muestra en la siguiente figura . Se debe mostrar el identificador del jugador y el número de intentos que lleva , inicialmente es 0. El botón "Enviar" realiza el envío del número introducido al servidor y el servidor le devolverá el estado del juego, se visualizará en la pantalla el mensaje que indica lo que ha ocurrido en el juego.