

PROGRAMACIÓN DE COMUNICACIONES EN RED.

SOCKETS II.

Índice de contenido

1. SOCKETS MULTICAST.....	2
2. ENVÍO DE OBJETOS A TRAVÉS DE SOCKETS.....	7
2.1. OBJETOS EN SOCKETS TCP.....	7
2.2. OBJETOS EN SOCKETS UDP.....	11

1. SOCKETS MULTICAST

La clase **MulticastSocket** es útil para enviar paquetes a múltiples destinos simultáneamente.

Para poder recibir estos paquetes es necesario establecer un **grupo multicast**, que es un grupo de direcciones IP que comparten el mismo número de puerto. Cuando se envía un mensaje a un grupo de multicast, todos los que pertenezcan a ese grupo recibirán el mensaje; la pertenencia al grupo es transparente al emisor, es decir, el emisor no conoce el número de miembros del grupo ni sus direcciones IP.

Un grupo multicast se especifica mediante una dirección IP de clase D y un número de puerto UDP estándar. Las direcciones desde la 224.0.0.0 a la 239.255.255.255 están destinadas para ser direcciones de multicast. La dirección 224.0.0.0¹ está reservada y no debe ser utilizada.

La clase **MulticastSocket** tiene varios constructores (pueden lanzar la excepción *IOException*):

CONSTRUCTOR	MISIÓN
MulticastSocket ()	Construye un multicast socket dejando al sistema que elija un puerto de los que están libres
MulticastSocket (int port)	Construye un multicast socket y lo conecta al puerto local especificado.

Algunos métodos importantes son (pueden lanzar la excepción *IOException*):

MÉTODO	MISIÓN
joinGroup(InetAddress mcastaddr)	Permite al socket multicast unirse al grupo de multicast
leaveGroup(InetAddress mcastaddr)	El socket multicast abandona el grupo de multicast
send(DatagramPacket p)	Envía el datagrama a todos los miembros del grupo multicast.
receive(DatagramPacket p)	Recibe el datagrama de un grupo multicast

El esquema general para un **servidor multicast** que envía paquetes a todos los miembros del grupo es el siguiente:

```
//Se crea el socket multicast. No hace falta especificar puerto:
MulticastSocket ms = new MulticastSocket ();
//Se define el Puerto multicast:
int Puerto = 12345;
//Se crea el grupo multicast:
InetAddress grupo = InetAddress.getByName("225.0.0.1");
String msg = "Bienvenidos!!";
//Se crea el datagrama:
DatagramPacket paquete = new DatagramPacket(msg.getBytes(), msg.length() ,grupo, Puerto);
//Se envía el paquete al grupo:
ms.send (paquete);
//Se cierra el socket:
```

¹ El bloque 224.0.0.0/24 es solo para enlaces multicast locales. Se usa para cosas como protocolos de enrutado. Los datagramas a estas direcciones no deberían ser reenviados por los routers.

```
ms.close();
```

Para que un cliente se una al grupo multicast primero crea un **MulticastSocket** con el puerto deseado y luego invoca al método *joinGroup()*. El cliente multicast que recibe los paquetes que le envía el servidor tiene la siguiente estructura:

```
//Se crea un socket multicast en el puerto 12345:
MulticastSocket ms = new MulticastSocket (12345);
//Se configura la IP del grupo al que nos conectaremos:
InetAddress grupo = InetAddress.getByName ("225.0.0.1");
//Se une al grupo
ms.joinGroup (grupo);
//Recibe el paquete del servidor multicast:
byte[] buf = new byte[1000];
DatagramPacket recibido = new DatagramPacket(buf, buf.length);
ms.receive(recibido) ;
//Salimos del grupo multicast
ms.leaveGroup (grupo);
//Se cierra el socket:
ms.close ();
```

En el siguiente ejemplo tenemos un **servidor multicast** que lee datos por teclado y los envía a todos los clientes que pertenezcan al grupo multicast, el proceso terminará cuando se introduzca un asterisco:

```

import java.io.*;
import java.net.*;
}
public class servidorMC1 {
}
public static void main(String args[]) throws Exception {
    // FLUJO PARA ENTRADA ESTANDAR
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    //Se crea el socket multicast.
    MulticastSocket ms = new MulticastSocket();
    int Puerto = 12345; //Puerto multicast
    InetAddress grupo = InetAddress.getByName("225.0.0.1"); //Grupo
    String cadena="";
    //
    while(!cadena.trim().equals("")) {
        System.out.print("Datos a enviar al grupo: ");
        cadena = in.readLine();
        // ENVIANDO AL GRUPO
        DatagramPacket paquete = new DatagramPacket(cadena.getBytes(),
            cadena.length(), grupo, Puerto);
        ms.send(paquete);
        String msg = new String(paquete.getData());
        System.out.println ("Envío: " + msg.trim());
    }
    ms.close (); //cierro socket
    System.out.println ("Socket cerrado ... ");
}
}

```

El **programa cliente** visualiza el paquete que recibe del servidor, su proceso finaliza cuando recibe un asterisco:

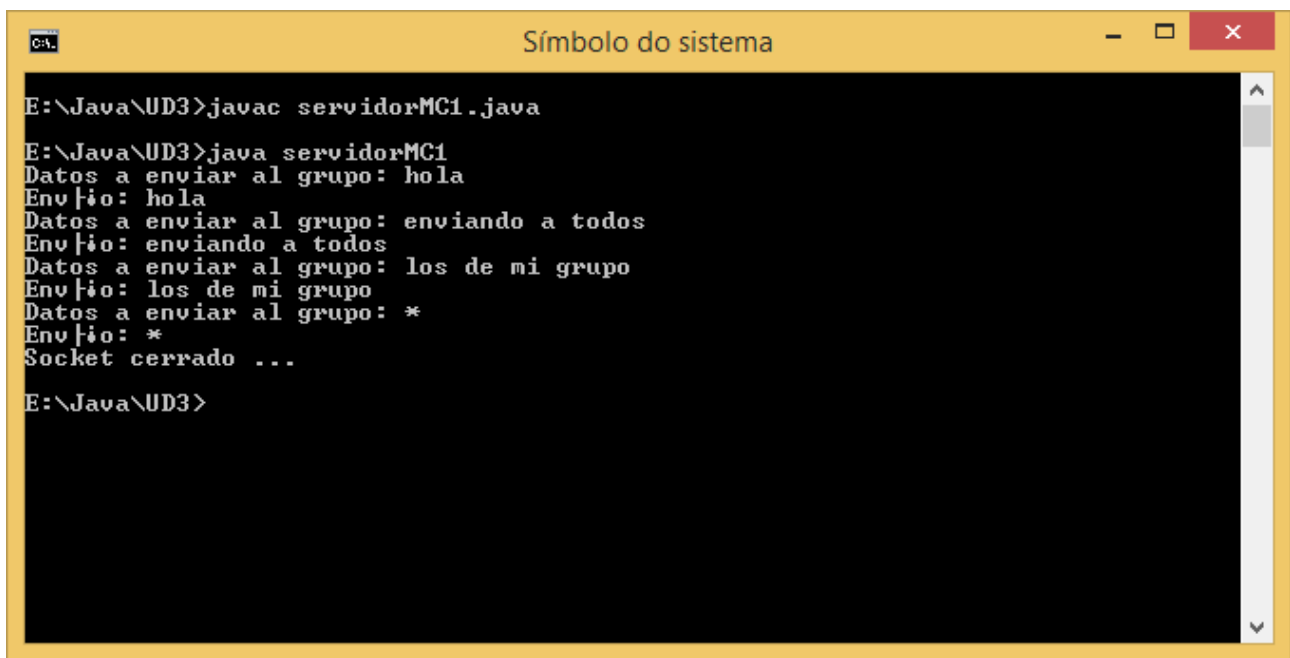
```

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

public class clienteMC1 {
    public static void main(String args[]) throws Exception {
        //Se crea el socket multicast
        int Puerto = 12345; //Puerto multicast
        MulticastSocket ms = new MulticastSocket(Puerto) ;
        InetAddress grupo = InetAddress.getByName("225.0.0.1") ; //Grupo
        //Nos unimos al grupo
        ms.joinGroup (grupo);
        String msg="";
        byte[] buf = new byte[1000];
        //
        while(!msg.trim().equals("")) {
            //Recibe el paquete del servidor multicast
            DatagramPacket paquete = new DatagramPacket(buf, buf.length);
            ms.receive(paquete);
            //msg = new String(paquete.getData());
            //En lugar de hacerlo como en la línea anterior comentada,
            //lo hago así (Obtener la cadena) especificando que sólo
            //admito los bytes de la longitud correspondiente, para que
            //no lea bytes que están por debajo.
            msg = new String(paquete.getData(), 0, paquete.getLength());
            System.out.println ("Recibo: " + msg.trim());
        }
        ms.leaveGroup (grupo); //abandonamos grupo
        ms.close (); //cierra socket
        System.out.println ("Socket cerrado ... ");
    }
}

```

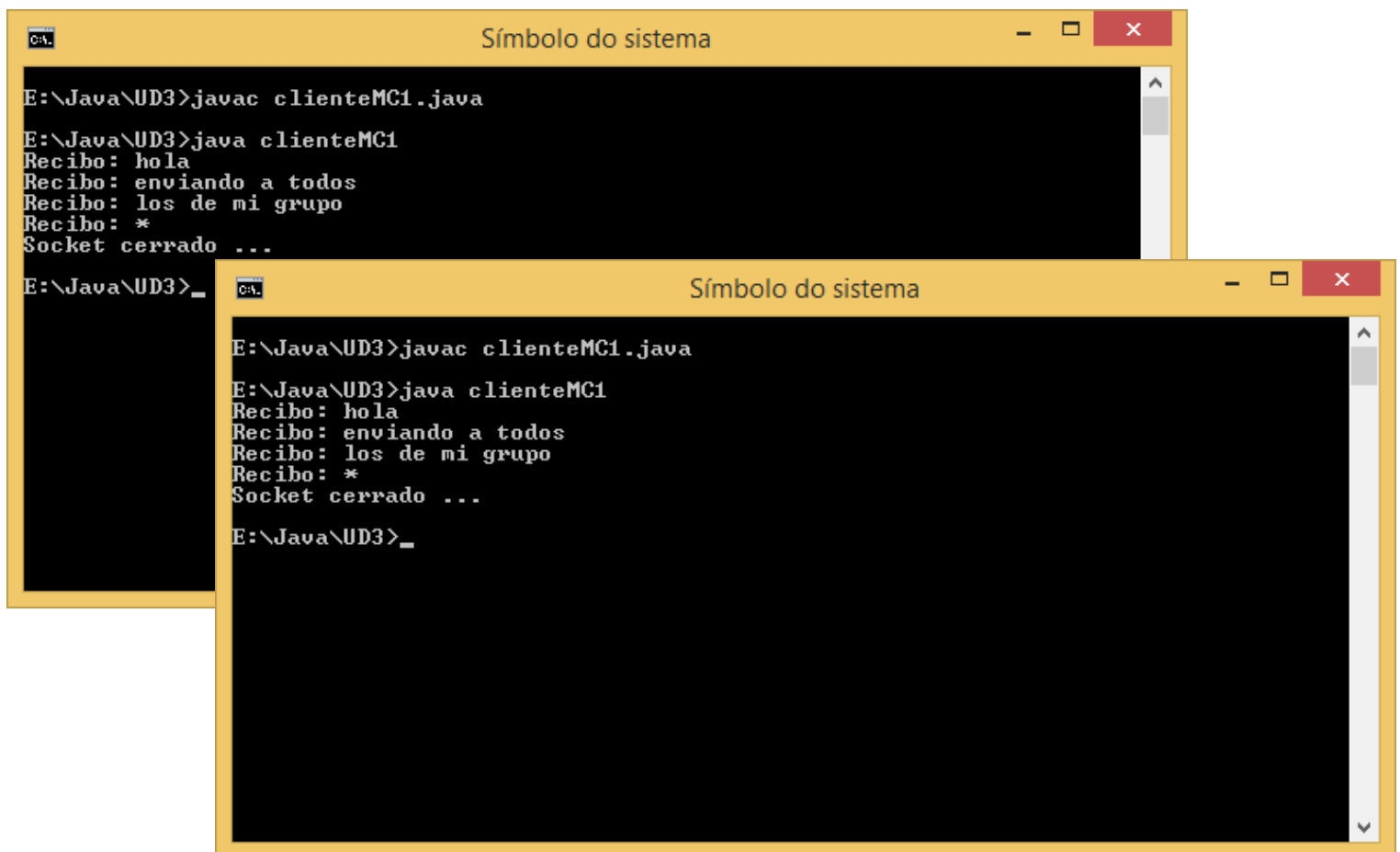
Para probarlo ejecutamos el programa servidor en una consola y a continuación ejecutamos diferentes instancias del programa cliente:



```

C:\>
E:\Java\UD3>javac servidorMC1.java
E:\Java\UD3>java servidorMC1
Datos a enviar al grupo: hola
Envío: hola
Datos a enviar al grupo: enviando a todos
Envío: enviando a todos
Datos a enviar al grupo: los de mi grupo
Envío: los de mi grupo
Datos a enviar al grupo: *
Envío: *
Socket cerrado ...
E:\Java\UD3>

```



```

C:\>
E:\Java\UD3>javac clienteMC1.java
E:\Java\UD3>java clienteMC1
Recibo: hola
Recibo: enviando a todos
Recibo: los de mi grupo
Recibo: *
Socket cerrado ...
E:\Java\UD3>_

```

```

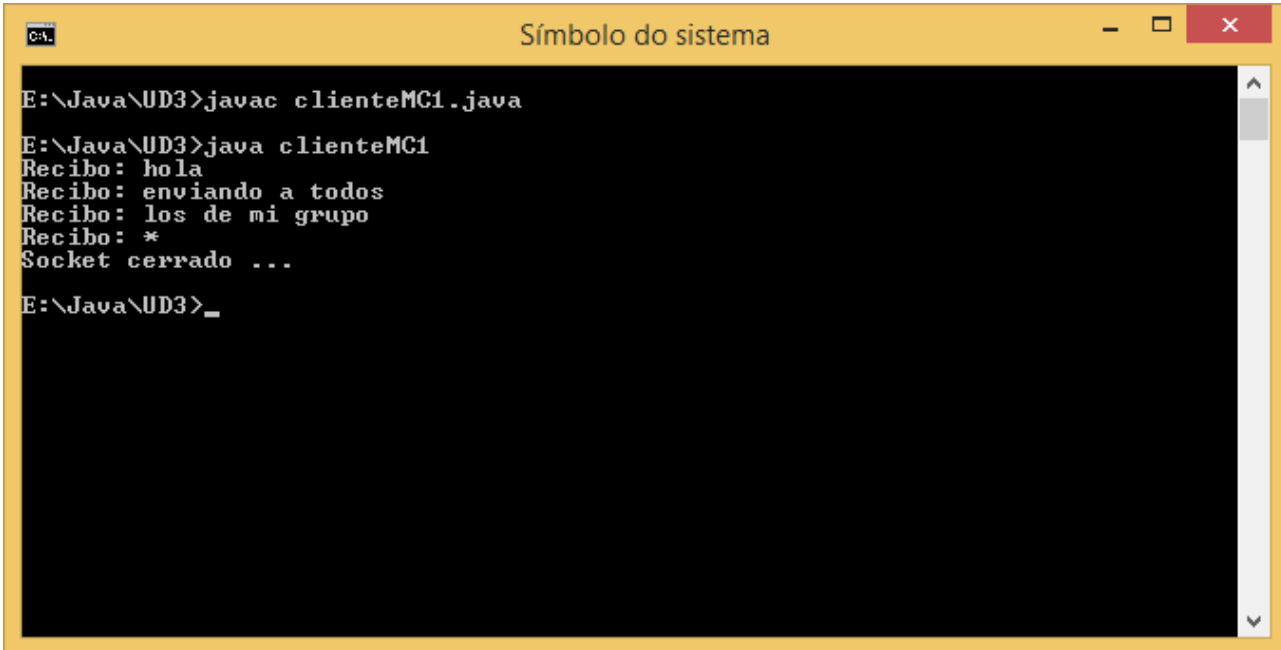
C:\>
E:\Java\UD3>javac clienteMC1.java
E:\Java\UD3>java clienteMC1
Recibo: hola
Recibo: enviando a todos
Recibo: los de mi grupo
Recibo: *
Socket cerrado ...
E:\Java\UD3>_

```

2. ENVÍO DE OBJETOS A TRAVÉS DE SOCKETS

Hasta ahora hemos estado intercambiando cadenas de caracteres entre programas cliente y servidor. Pero los stream soportan diversos tipos de datos como son los bytes, los tipos de datos primitivos, caracteres y objetos.

Vamos a ver cómo se pueden intercambiar objetos entre programas emisor y receptor (en sockets



```

C:\>
E:\Java\UD3>javac clienteMC1.java
E:\Java\UD3>java clienteMC1
Recibo: hola
Recibo: enviando a todos
Recibo: los de mi grupo
Recibo: *
Socket cerrado ...
E:\Java\UD3>_
  
```

UDP) o entre programas cliente y servidor (en sockets TCP) usando sockets.

2.1. OBJETOS EN SOCKETS TCP

Las clases **ObjectInputStream** y **ObjectOutputStream** nos permiten enviar objetos a través de sockets TCP. Utilizaremos los métodos *readObject()* para leer el objeto del stream y *writeObject()* para escribir el objeto al stream. Usaremos el constructor que admite un **InputStream** y un **OutputStream**, Para preparar el flujo de salida para escribir objetos escribimos:

```
ObjectOutputStream outObjeto = new ObjectOutputStream(socket.getOutputStream());
```

Para preparar el flujo de entrada para leer objetos escribimos:

```
ObjectInputStream inObjeto = new ObjectInputStream(socket.getInputStream());
```

Las clases a las que pertenecen estos objetos deben implementar la interfaz **Serializable**². Por

² Un objeto serializable es un objeto que se puede convertir en una secuencia de bytes. Para que un objeto sea serializable, su clase debe implementar la interfaz **java.io.Serializable**. Esta interfaz no define ningún método.

ejemplo, sea la clase *Persona* con 2 atributos, nombre y edad, constructor y los métodos *get* y *set* correspondientes:

```
import java.io.Serializable;
@SuppressWarnings("serial")3
public class Persona implements Serializable {
    String nombre;
    int edad;
    public Persona (String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    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;}
}
```

Podemos intercambiar objetos *Persona* entre un cliente y un servidor usando sockets TCP.

Por ejemplo el programa servidor crea un objeto *Persona*, dándole valores y se lo envía al programa cliente, el programa cliente realiza los cambios oportunos en el objeto y se lo devuelve modificado al servidor. El **programa servidor** es el siguiente:

Simplemente se usa para indicar que para aquellas clases, sus instancias pueden ser convertidas a secuencias de bytes (y posteriormente reconstruidas). Objetos tan comunes como String, Vector o ArrayList implementan Serializable, de modo que pueden ser serializados y reconstruidos más tarde.

3 Informa al compilador que no debe producir la advertencia serialVersionUID, que es un número de versión que posee cada clase Serializable, el cual es usado en la deserialización para verificar que el emisor y el receptor de un objeto serializado mantienen una compatibilidad en lo que a serialización se refiere con respecto a la clases que tienen cargadas (el emisor y el receptor).


```

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

public class ServidorObjeto {
    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
        ObjectOutputStream outObjeto = new ObjectOutputStream(cliente.getOutputStream());

        // Se prepara un objeto y se envia
        Persona per = new Persona("Juan", 20);
        outObjeto.writeObject(per); //enviando objeto
        System.out.println("Envio: " + per.getNombre() + "*" + per.getEdad());

        //Se obtiene un stream para leer objetos
        ObjectInputStream inObjeto = new ObjectInputStream(cliente.getInputStream()) ;
        Persona dato = (Persona)inObjeto.readObject();
        System.out.println("Recibo: " + dato.getNombre() + "*" + dato.getEdad());

        // CERRAR STREAMS y SOCKETS
        outObjeto.close() ;
        inObjeto.close();
        cliente.close();
        servidor.close();
    }
}

```

El programa cliente es el siguiente:

```

import java.io.*;
import java.net.*;
public class Cliente1Objeto {
    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
        ObjectInputStream perEnt = new ObjectInputStream(cliente.getInputStream()) ;

        //Se recibe un objeto
        Persona dato = (Persona) perEnt.readObject() ;//recibo objeto
        System.out.println ("Recibo: "+dato.getNombre () +"*"+dato.getEdad()) ;

        //Modifico el objeto
        dato.setNombre (" Juan Ramos");
        dato.setEdad(22);

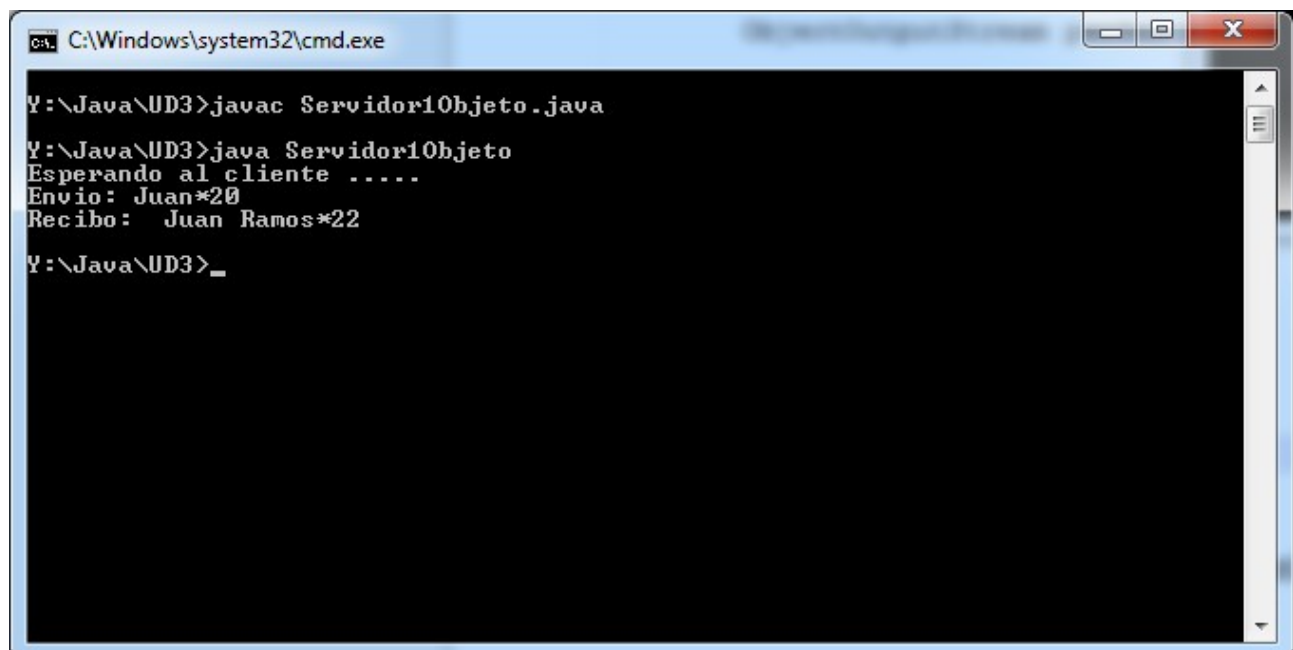
        //FLUJO DE salida para objetos
        ObjectOutputStream perSal = new ObjectOutputStream(cliente.getOutputStream()) ;

        // Se envía el objeto
        perSal.writeObject(dato) ;
        System.out.println ("Envio: "+dato. getNombre () +" * "+dato. getEdad ()) ;

        //CERRAR STREAMS y SOCKETS
        perEnt.close();
        perSal.close();
        cliente.close();
    }
}

```

La compilación y ejecución se muestra a continuación:



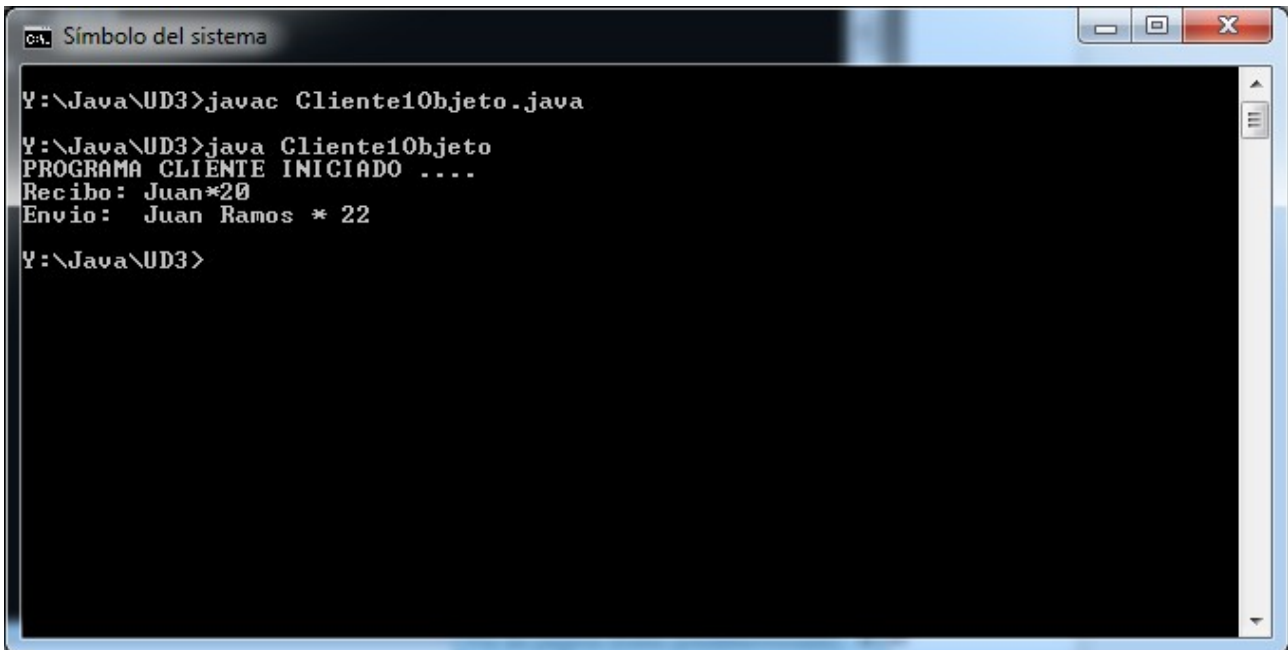
```

C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac Servidor1Objeto.java

Y:\Java\UD3>java Servidor1Objeto
Esperando al cliente .....
Envio: Juan*20
Recibo: Juan Ramos*22
Y:\Java\UD3>_

```



```

C:\> Símbolo del sistema

Y:\Java\UD3>javac Cliente1Objeto.java

Y:\Java\UD3>java Cliente1Objeto
PROGRAMA CLIENTE INICIADO ....
Recibo: Juan*20
Envio: Juan Ramos * 22

Y:\Java\UD3>

```

2.2. OBJETOS EN SOCKETS UDP

Para intercambiar objetos en sockets UDP utilizaremos las clases **ByteArrayOutputStream** y **ByteArrayInputStream**. Se necesita convertir el objeto serializable a un array de bytes, ya que en el **DatagramPacket** se espera un array de bytes. Por ejemplo, para convertir un objeto *Persona* a un array de bytes escribimos las siguientes líneas:

```

Persona persona = new Persona("Maria", 22);
//CONVERTIMOS OBJETO A BYTES
ByteArrayOutputStream bs= new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream (bs);
out.writeObject(persona);//escribir objeto Persona en el stream
out.close(); //cerrar stream
byte[] bytes = bs.toByteArray(); // objeto en bytes

```

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);
socket.receive(paqRecibido);//recibo el datagrama
// CONVERTIMOS BYTES A OBJETO
ByteArrayInputStream bais = new ByteArrayInputStream(recibidos);
ObjectInputStream in = new ObjectInputStream(bais);
Persona persona = (Persona) in.readObject();//obtengo objeto
in.close();

```

PSP 20-21

Este código es aplicable para convertir cualquier objeto serializable a array de bytes y viceversa.