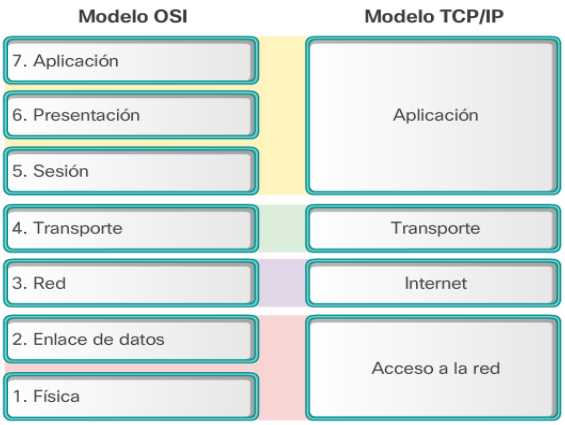
Programación de Comunicaciones en Red

1. Clases Java para Comunicaciones en Red

Las computadoras pueden conectarse a través de una variedad de medios físicos. En un aula de informática, por ejemplo, las computadoras están conectadas por redes de cable. Los impulsos eléctricos representan el flujo de información a través de los cables. Si se utiliza un módem DSL para conectar el ordenador a Internet, las señales viajan a través de la línea de cobre del teléfono codificadas como tonos o impulsos de alta frecuencia. En una red inalámbrica, las señales se envían transmitiendo una frecuencia de radio modulada (amplitud, fase, etc.). Pese a que las características físicas de estas transmisiones pueda diferir ampliamente, en última instancia consisten en enviar y recibir los flujos de ceros y unos a través de las conexiones de red.
Estos flujos representan dos tipos de información: los *datos de aplicación*, que corresponden con los datos de que un ordenador realmente se desea enviar otro, y los *datos del protocolo de red*, que comprenden los datos que describen cómo llegar al destinatario previsto y cómo comprobar los errores de la transmisión. Los protocolos siguen un conjunto de reglas establecidas por la *Suite del Protocolo de Internet*, también denominada TCP/IP, Estos protocolos se han convertido en la base para la conexión de ordenadores en todo el mundo a través de Internet.



El paquete **java.net** proporciona las clases para la implementación de las comunicaciones de red. Podemos dividir las funcionalidades en dos partes:

API de bajo nivel encargada de controlar las abstracciones relacionadas con:

- Las direcciones de red
- Los Sockets: mecanismos básicos de comunicación de datos
- Las Interfaces de red

Una API de más alto nivel que se ocupa de las abtracciones:

- URI: Identificador de Recurso Universal. Un URI identifica un recurso, ya sea por ubicación o un nombre, o ambos. Un URI tiene dos especializaciones conocidas como URL y URN
- URL: Localizador de Recurso Universal. Especifica que un recurso identificado está disponible y el mecanismo para recuperarlo
- Conexiones: identifica las conexiones realizadas al recurso especificado por un URL

1.1 La clase `InetAddress`

La clase `InetAddress` es la abstracción que representa una dirección de Internet (IP). Tiene dos subclases asociadas a cada uno de los protocolos `Inet4Address` (Ipv4) e `Inet6Addres.` Aunque en general `InetAddress` aporta la suficiente funcionalidad que no es necesario recurrir a estas subclases.

`http://download.java.net/jdk7/archive/b123/docs/api/java/net/InetAddress.html`

Lo primero es **crear un objeto** de la clase `InetAddress,` para ello podemos crearlo a partir de nuestro propio equipo.

```
equipo = InetAddress.getLocalHost();
```

Ahora, podemos acceder a algunas de las **propiedades** del objeto equipo.

```
System.out.println("Mi equipo es: "+equipo);
System.out.println("Mi dirección IP es: "+equipo.getHostAddress());
System.out.println("Mi nombre es: "+equipo.getHostName());
System.out.println("Mi nombre canónico es: "+equipo.getCanonicalHostName());
```

También podemos crear un objeto `InetAddress` a partir de una IP o de un nombre.

```
equipo = InetAddress.getByName("www.google.com");
equipo = InetAddress.getByName("2001:4860:4860::8888");
equipo = InetAddress.getByName("193.147.147.9");
```

Existe también la función `InetAddress.getByAddress(byte addr[])` pero el problema es que byte es un valor con signo, lo que limita su aplicación.

	Métodos
`byte[]`	**getAddress**`()` Devuelve la dirección IP del objeto `InetAddress`
`static InetAddress[]`	**getAllByName**`(String host)` Dado el nombre de un host, devuelve un array con sus direcciones IP
`static InetAddress`	**getByAddress**`(byte[] addr)` Devuelve un objeto `InetAddress dada una` dirección IP
`static InetAddress`	**getByAddress**`(String host, byte[] addr)` Crea un ub objeto `InetAddress` dado un nombre de host y una IP
`static InetAddress`	**getByName**`(String host)` Determina la IP dado un nombre de host
`String`	**getCanonicalHostName**`()` Obtiene el nombre FQDN
`String`	**getHostAddress**`()` Devuelve un string con la IP
`String`	**getHostName**`()` Obtiene el nombre del host
`static InetAddress`	**getLocalHost**`()` Devuelve la dirección local del host
`static InetAddress`	**getLoopbackAddress**`()` Devuelve la dirección del loopback

Ejercicio 1.1. Obtener la IP, el nombre de host y el nombre canónico de www.usal.es, 138.100.200.6 y de 2001:470:b53a::232

1.2 La clase `NetworkInterface`

Esta clase representa una interfaz de red compuesta por un nombre y un conjunto de IP asociadas a la interfaz.

http://download.java.net/jdk7/archive/b123/docs/api/java/net/NetworkInterface.html

Para conocer todas las interfaces y sus IP asociadas

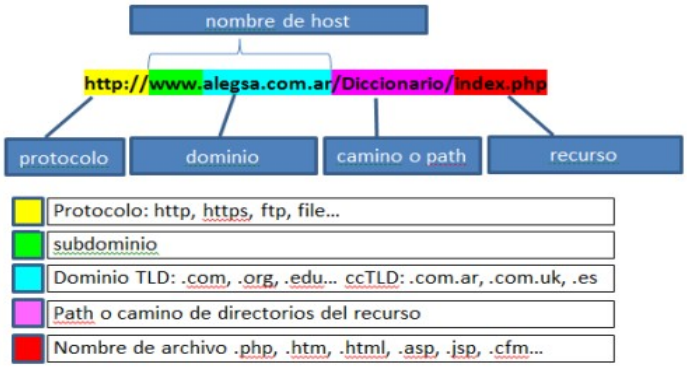
```
try {
Enumeration<NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();
for (NetworkInterface netint : Collections.list(nets))
           displayInterfaceInformation(netint);
}catch (SocketException e){}

static void displayInterfaceInformation(NetworkInterface netint) throws
SocketException {
        System.out.printf("Display name: %s\n", netint.getDisplayName());
        System.out.printf("Name: %s\n", netint.getName());
        Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();
        for (InetAddress inetAddress : Collections.list(inetAddresses)) {
            System.out.printf("InetAddress: %s\n", inetAddress);
        }
```

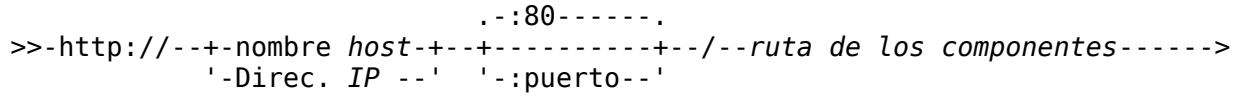
Ejercicio 1.2. Implementa una aplicación para obtener todas las IPs de un ordenador mediante `NetworkInterface`.

1.3 La clase `URL`

Esta clase representa un puntero a un recurso en la web. La URL se divide en varias partes, tal y como se muestra en la siguiente imagen.



Una URL puede también especificar un puerto, por defecto en función del protocolo elegido se elige automáticamente un puerto. Para http el puerto por defecto es el 80.




```
http://www.example.com/software/index.html
http://www.example.com:1030/software/index.html
```

La clase URL dispone de varios constructores

- **URL(String spec)**
  Crea un objeto URL a partir de la especificación
- **URL(String protocol, String host, int port, String file)**
  Crea un objeto URL a partir de la especificación de protocolo, host, numero de puerto, y archivo
- **URL(String protocol, String host, int port, String file, URLStreamHandler handler)**
  Crea un objeto URL a partir de la especificación de protocolo, host, numero de puerto, y handler
- **URL(String protocol, String host, String file)**
  Crea un objeto URL a partir de la especificación de protocolo, host y archivo
- **URL(URL context, String spec)**
  Crea un objeto URL a partir del análisis de una especificación en un contexto concreto
- **URL(URL context, String spec, URLStreamHandler handler)**
  Crea un objeto URL a partir del análisis de una especificación con un handler en un contexto concreto

Resumen de los principales métodos de esta clase.

https://docs.oracle.com/javase/7/docs/api/java/net/URL.html

	Métodos
`boolean`	**equals**`(Object obj)` Compara una URL con un objeto
`String`	**getAuthority**`()` Obtiene la parte de autoridad de una `URL`
`Object`	**getContent**`()` Obtiene el contenido de una `URL`
`Object`	**getContent**`(Class[] classes)` Obtiene el contenido de una URL.
`int`	**getDefaultPort**`()` Obtiene el puerto asociado por defecto a una `URL`.
`String`	**getFile**`()` Obtiene el nombre de archivo para una `URL`.

`String`	**getHost**`()` Obtiene el nombre del host
`String`	**getPath**`()` Obtiene el path de una `URL`.
`int`	**getPort**`()` Obtiene el puerto de una `URL`.
`String`	**getProtocol**`()` Obtiene el protocolo de una `URL`.
`String`	**getQuery**`()` Obtiene la query de una `URL`.
`String`	**getRef**`()` Obtiene el enlace de una `URL` (referencia)
`String`	**getUserInfo**`()` Obtiene la información de usuario de una `URL`.
`URLConnection`	**openConnection**`()` Devuelve una instancia al objeto `URLConnection` que representa una conexión a un objeto remoto referido por una `URL`.
`URLConnection`	**openConnection**`(Proxy proxy)` Devuelve una instancia al objeto `URLConnection` que representa una conexión a un objeto remoto mediante un proxy referido por una `URL`.
`InputStream`	**openStream**`()` Abre una conexión a una `URL` y retorna un `InputStream` para leer de la conexión
`protected void`	**set**`(String protocol, String host, int port, String file, String ref)` Configura los campos de una URL
`protected void`	**set**`(String protocol, String host, int port, String authority, String userInfo, String path, String query, String ref)` Configura los campos de una URL
`static void`	**setURLStreamHandlerFactory**`(URLStreamHandlerFactory fac)` Sets an application's `URLStreamHandlerFactory`.
`String`	**toExternalForm**`()` Construye una string que es una representación de una `URL`.
`String`	**toString**`()` Devuelve una string que es una representación de una `URL`.
`URI`	**toURI**`()` Retorna una `URI` equivalente a la URL.

1.3.1 Creando una URL

```
URL miURL = new URL("http://www.iescierva.net/");
```

Otra forma de crear una URL

```
new URL("http", "example.com", "/pages/page1.html");
```

```
URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");
```

```
http://example.com:80/pages/page1.html
```

Si necesitamos incluir caracteres especiales, podemos codificarlos

```
URL url = new URL("http://example.com/hello%20world");
```

Esta URL es equivalente a `http://example.com/hello world/`

Si la URL está mal formada se puede lanzar la excepción `MalformedURLException`

```
try {
    URL myURL = new URL(...);
}
catch (MalformedURLException e) {
    // exception handler code here
    // ...
}
```

Creando una URL relativa a otra

Por ejemplo, si en una página HTML `(JoesHomePage.html) tenemos enlaces a las páginas PicturesOfMe.html` y `MyKids.html`, dentro de la misma máquina y directorio que `JoesHomePage.html`. Los enlaces a las páginas `PicturesOfMe.html` y `MyKids.html` desde `JoesHomePage.html` pueden especificarse como:

```
<a href="PicturesOfMe.html">Pictures of Me</a>
<a href="MyKids.html">Pictures of My Kids</a>
```

Estas direcciones son relativas a la página desde donde se enlazan. Por ejemplo sean las páginas

`http://example.com/pages/page1.html`
`http://example.com/pages/page2.html`

Se pueden crear URL relativas a la dirección base `http://example.com/pages/`

```
URL miURL = new URL("http://example.com/pages/");
URL page1URL = new URL(miURL, "page1.html");
URL page2URL = new URL(miURL, "page2.html");
```

```
public class Test {

   public static void main(String [] args) {
      try {
         URL url = new URL("https://www.amrood.com/index.htm?language=en#j2se");

         System.out.println("URL        >" + url.toString());
         System.out.println("protocolo  >" + url.getProtocol());
         System.out.println("autoridad  > " + url.getAuthority());
         System.out.println("archivo    >" + url.getFile());
         System.out.println("host       >" + url.getHost());
         System.out.println("path       >" + url.getPath());
         System.out.println("puerto     >" + url.getPort());
         System.out.println("p. defecto >" + url.getDefaultPort());
         System.out.println("query      >" + url.getQuery());
         System.out.println("ref        >" + url.getRef());
      }catch(IOException e) { e.printStackTrace(); }
   }
}
```

Ejercicio 1.3.1. Implementa el ejemplo anterior y sustituye la URL por una como las que genera google cuando se realiza una consulta.

1.3.2 La clase `URLConnection`

Esta clase dispone de métodos que nos van a permitir comunicarnos con la URL mediante la red. La clase **`URLConnection`** es una clase orientada al HTTP, por lo que algunos de sus métodos únicamente serán válidos cuando se trabaje con HTTP URL. Sin embargo la mayoría de los protocolos nos van a permitir leer y escribir en la conexión.

1.3.2.1 Leyendo desde una URL

Una manera simple de leer desde una URL es crear un objeto URL e invocar al método `openStream()` que nos devuelve un `InputStream`, por lo que la lectura es como leer desde un `Input Stream`.

El siguiente ejemplo crea una URL y utiliza `openStream()` para leer su contenido.

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

public class URLReader {
    public static void main(String[] args) throws Exception {

        URL ies = new URL("http://moodle.iescierva.net/");
        BufferedReader in = new BufferedReader(
            new InputStreamReader(ies.openStream()) );

        String inputLine;

        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

Otra forma para leer el contenido de una URL es mediante la creación de una instancia de `URLConnection` y obtener un input stream de la misma. Al llamar al `getInputStream` la conexión se abre de modo implícito.

```
public class URLConnectionReader {
    public static void main(String[] args) throws Exception {

        // Lectura mediante URLConnectionReade
        URL iesC = new URL("http://moodle.iescierva.net/");
        URLConnection yc = iesC.openConnection();
        BufferedReader inC = new BufferedReader(new
            InputStreamReader(yc.getInputStream()));

        String inputLineC;

        while ((inputLineC = inC.readLine()) != null)
            System.out.println(inputLineC);
        inC.close();
    }
}
```

1.3.2.2 Escribiendo en una URL

Muchas páginas web incluyen formularios, campos de texto u otros objetos que permiten la introducción de datos para ser enviados al servidor. Cuando se pulsa el botón de envío, el navegador escribe los datos en la URL que son recibidos en el servidor vía red, los procesa y generalmente nos imprime una página de respuesta en formato HTML.

Los datos pueden remitirse mediante el método GET o POST. Así escribir en una URL muchas veces se define como posting a una URL. El servidor reconoce el POST y lee los datos enviados desde el cliente.

Para que nuestro programa en Java pueda interaccionar con el servidor debe realizar los siguientes pasos:

1. Crear una `URL`
2. Obtener el objeto `URLConnection`
3. Establecer la capacidad de salida sobre el objeto `URLConnection`
4. Abrir una conexión con el recurso
5. Obtener un output stream de la conexión
6. Escribir sobre el output stream
7. Cerrar el output stream

Para comprobar la escritura sobre la URL hay un pequeño **script en perl** que debemos guardar como backwards.pl en el directorio /cgi-bin/ de apache

```
!/usr/bin/perl
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
    # Stop people from using subshells to execute commands
    $value =~ s/~!/ ~!/g;
    $FORM{$name} = $value;
}

print "Content-type: text/plain\n\n";
print "$FORM{'string'} reversed is: ";
$foo=reverse($FORM{'string'});
print "$foo\n";
exit 0;
```

El código para escribir sobre la URL es el siguiente:

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

public class Reverse {
    public static void main(String[] args) throws Exception {


        String stringToReverse = URLEncoder.encode("Esto es una prueba", "UTF-
8");

        URL url = new URL("http://<ip.servidor>/cgi-bin/backwards.pl");
        URLConnection connection = url.openConnection();
        connection.setDoOutput(true);

        OutputStreamWriter out = new OutputStreamWriter(
                                        connection.getOutputStream());
        out.write("string=" + stringToReverse);
        out.close();

        BufferedReader in = new BufferedReader(
                                    new InputStreamReader(
                                    connection.getInputStream()));
        String decodedString;
        while ((decodedString = in.readLine()) != null) {
            System.out.println(decodedString);
        }
        in.close();
    }
}
```

El funcionamiento básico del programa es el siguiente:

1. El programa crea un objeto URL y establece la conexión para poder escribir. Se ha instalado en una máquina de ejemplo

```
        URL url = new URL("http://79.109.156.94/cgi-bin/backwards.pl");
        URLConnection connection = url.openConnection();
        connection.setDoOutput(true);
```

2. Ahora crea un output stream sobre la conexión y establece un OutputStreamWriter

```
OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream());
```

3. El programa escribe la información y cierra el stream

```
      out.write("string=" + stringToReverse);
      out.close();
```

4. Ahora debemos leer la información que el servidor nos ha devuelto

```
   BufferedReader in = new BufferedReader(
   new InputStreamReader(connection.getInputStream()));
   String decodedString;
   while ((decodedString = in.readLine()) != null) {
            System.out.println(decodedString);
   }
   in.close();
```

1.4 Comunicación cliente/servidor con Socket TCP

El interfaz Java que da soporte a sockets TCP está constituida por las clases `ServerSocket` y `Socket.`

1. **`ServerSocket`**: es utilizada por un servidor para crear un socket en el puerto en el que escucha las peticiones de conexión de los clientes. Su método `accept` toma una petición de conexión de la cola, o si la cola está vacía, se bloquea hasta que llega una petición. El resultado de ejecutar `accept` es una instancia de *Socket*, a través del cual el servidor tiene acceso a los datos enviados por el cliente.
2. **`Socket`**: es utilizada tanto por el cliente como por el servidor. El cliente crea un socket especificando el nombre DNS del host y el puerto del servidor, así se crea el socket local y además se conecta con el servicio.
   Esta clase proporciona los métodos `getInputStream` y `getOutputStream` para acceder a los dos streams asociados a un socket (recordemos que son bidireccionales), y devuelve tipos de datos `InputStream` y `OutputStream`, respectivamente, a partir de los cuales podemos construir `BufferedReader` y `PrintWriter`, respectivamente, para poder procesar los datos de forma más sencilla.

Si nos centramos en la parte de comunicaciones, la **forma general de implementar un cliente** será:

3. Crear un objeto de la clase `Socket`, indicando host y puerto donde corre el servicio.
1. Obtener las referencias al stream de entrada y al de salida al socket.
2. Leer desde y escribir en el stream de acuerdo al protocolo del servicio. Para ello emplear alguna de las facilidades del paquete java.io.
3. Cerrar los streams.
4. Cerrar el socket.

La forma de implementar un servidor será:

4. Crear un objeto de la clase `ServerSocket` para escuchar peticiones en el puerto asignado al servicio.
1. Esperar solicitudes de clientes
2. Cuando se produce una solicitud:
   - Aceptar la conexión obteniendo un objeto de la clase Socket
   - Obtener las referencias al stream de entrada y al de salida al socket anterior.
   - Leer datos del socket, procesarlos y enviar respuestas al cliente, escribiendo en el stream del socket.Para ello emplear alguna de las facilidades del paquete java.io.
3. Cerrar los streams.
4. Cerrar los sockets.

1.4.1 Implementación de un servidor.

El ojetivo de este programa es implementar un servidor que gestione cuentas bancarias. Las órdenes soportadas son las siguientes:

Petición	Respuesta	Descripción
Balance n	n y el saldo	Muestra el saldo de la cuenta
Deposit n a	n y el nuevo saldo	Realiza un ingreso en una cuenta
Withdraw n a	n y el nuevo saldo	Realiza un reintegro en una cuenta
QUIT	Nada	Cierra la conexión

El servidor debe esperar conexiones de los clientes en un puerto específico.

```
ServerSocket server = new ServerSocket(8888);
```

El método `accept` de la clase ServerSocket espera una conexión de cliente. Cuando un Cliente se conecta, entonces el programa servidor obtiene un socket a través del cual se comunica con el cliente.

```
Socket s = server.accept();
BankService service = new BankService(s, bank);
```

La clase `BankService` realiza el servicio. Esta clase implementa la interfaz Runnable y su método run se ejecutará en cada subproceso que sirve una conexión de cliente. El método run obtiene un scanner y un writer del socket, de la misma manera que se ha visto en ejemplos anteriores.

```
public void doService() throws IOException
{
   while (true)
   {
      if (!in.hasNext()) { return; }
      String command = in.next();
      if (command.equals("QUIT")) { return; }
      executeCommand(command);
   }
}
```

El método executeCommand procesa un solo comando. Si el comando es DEPOSIT, entonces realiza el depósito:

```
int account = in.nextInt();
double amount = in.nextDouble();
bank.deposit(account, amount);
```

El comando WITHDRAW se maneja de la misma manera. Después de cada comando, el número de cuenta y el nuevo saldo se le envían al cliente:

```
out.println(account + " " + bank.getBalance(account));
```

El método `doService` vuelve al método `run` si el cliente cierra la conexión o el comando enviado es igual a "QUIT".
A continuación, el método run cierra el socket y sale. Cuando el socket del servidor acepta una conexión y construye el objeto BankService, en este punto, podríamos simplemente llamar al método run. Pero entonces nuestro programa del servidor tendría una limitación seria: solamente un cliente podría conectar con él en cualquier momento. Para superar esa limitación, los servidores generan un nuevo hilo por cada petición de conexión entrante.

Nuestra clase `BankService` implementa la interfaz `Runnable.` Por lo tanto, el programa de servidor `BankServer` simplemente inicia un subproceso con las siguientes instrucciones:

```
Thread t = new Thread(service);
t.start();
```

El hilo muere cuando el cliente se cierra o desconecta y el método run sale. Mientras tanto, el BankServer vuelve a aceptar la siguiente conexión:

```
while (true)
{
   Socket s = server.accept();
   BankService service = new BankService(s, bank);
   Thread t = new Thread(service);
   t.start();
}
```

El programa de servidor nunca se detiene. Para terminar de ejecutar el servidor, habrá que matarlo. Ahora podemos abrir una conexión con telnet y probar varios comandos.

Ejemplos comentados en http://www.binarytides.com/java-socket-programming-tutorial/

Ejercicio 1.4.1. Basándonos en el servicio bancario y el cliente implementados queremos crear un Sistema de Respuesta Instantánea o SRI. Para ello el servidor utilizará un array donde guardará las respuestas de cada cliente.

El cliente al conectarse enviará el comando "CLIENTE 0" y el servidor le remitirá un número de cliente, que será utilizado en las futuras comunicaciones con el servidor.
El cliente enviará el comando "MENUTXT <ncliente>" y recibirá las diferentes opciones a mostrar al usuario.
El cliente enviará el comando "MENUNUM <ncliente>" y recibirá el máximo de opciones a mostrar.
El cliente mediante el comando "RESPUESTA <ncliente> <numero>" Le enviará al servidor la respuesta seleccionada, por ejemplo "RESPUESTA 3 4.
El servidor dispondrá de alguna opción para ver las estadísticas de las respuestas. Por ejemplo
La pregunta "1. xxxxx" la han seleccionado 4
La pregunta "2. xxxxx" la han seleccionado 3

1.5 Comunicación cliente/servidor con Socket UDP

1.5.1 Servidor de echo con UDP

```
/**
Servidor con sockets UDP que realiza un Echo
*/

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

public class SRI
{
public static void main(String args[])
{
    DatagramSocket sock = null;

    try
    {
        //1. Crear un server socket, local port number
        sock = new DatagramSocket(7777);

        //buffer para incoming data
        byte[] buffer = new byte[65536];
        DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);

        //2. Esperar incoming data
        echo("Server socket creado. Esperando datos entrantes...");

        //communication loop
        while(true)
        {
            sock.receive(incoming);
            byte[] data = incoming.getData();
            String s = new String(data, 0, incoming.getLength());

            //echo incoming data - client ip : client port - client message
            echo(incoming.getAddress().getHostAddress() + " : " +
incoming.getPort() + " - " + s);

            s = "OK : " + s;
            DatagramPacket dp = new DatagramPacket(s.getBytes() ,
s.getBytes().length , incoming.getAddress() , incoming.getPort());
            sock.send(dp);
        }
    }

    catch(IOException e)
    {
        System.err.println("IOException " + e);
    }
}

//Función para echo data a terminal
public static void echo(String msg)
{
    System.out.println(msg);
}
}
```

Podemos verificar con netstat -u -ap que el servidor está a la escucha.

Para probar que funciona la conexión necesitamos conectarnos como con telnet, pero este programa no soporta las conexiones UDP, para ello vamos a utilizar **ncat**

```
ncat -vv localhost 7777 -u
```

Si la información que nos reporta es demasiada, podemos simplificar mediante

```
ncat localhost 7777 -u
```

1.5.2 Cliente UDP

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

public class SRI
{
public static void main(String args[])
{
	DatagramSocket sock = null;
	int port = 7777;
	String s;

	BufferedReader cin = new BufferedReader(new InputStreamReader(System.in));

	try
	{
		sock = new DatagramSocket();
		InetAddress host = InetAddress.getByName("localhost");

		while(true)
		{
			//Lee la entrada y envía el mensaje
			echo("Introduzca el mensaje a enviar : ");
			s = (String)cin.readLine();
			byte[] b = s.getBytes();

			DatagramPacket dp = new DatagramPacket(b , b.length , host ,
port);
			sock.send(dp);

			//recibimos la respuesta
			//se almacena en un buffer de entrada
			byte[] buffer = new byte[65536];
			DatagramPacket reply = new DatagramPacket(buffer,
buffer.length);
			sock.receive(reply);

			byte[] data = reply.getData();
			s = new String(data, 0, reply.getLength());

			//imprimir los detalles del input data - client ip : client
port - client message
			echo(reply.getAddress().getHostAddress() + " : " +
reply.getPort() + " - " + s);
```

```
            }
      }

      catch(IOException e)
      {
            System.err.println("IOException " + e);
      }
}

//función de echo data to terminal
public static void echo(String msg)
{
      System.out.println(msg);
}
}
```

Ejercicio 1.5.1 Modificar el servidor de echo en multihilo

Para **convertir el servidor en multihilo**

```
public class Server implements Runnable {
    public void run() {
        while (true) {
            DatagramPacket packet = socket.receive();
            new Thread(new Responder(socket, packet)).start();
        }
    }
}

public class Responder implements Runnable {

    Socket socket = null;
    DatagramPacket packet = null;

    public Responder(Socket socket, DatagramPacket packet) {
        this.socket = socket;
        this.packet = packet;
    }

    public void run() {
        byte[] data = makeResponse(); // code not shown
        DatagramPacket response = new DatagramPacket(data, data.length,
            packet.getAddress(), packet.getPort());
        socket.send(response);
    }
}
```

1.6 Comunicación cliente/servidor con Multicast

1.6.1 Servidor Multicast

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

public class UDPServerMulticast {
	//Dirección multicast seleccionada
	final static String INET_ADDR = "224.0.0.3";
	final static int PORT = 8889;


	public static void main(String[] args) throws UnknownHostException,
InterruptedException {

		InetAddress addr = InetAddress.getByName(INET_ADDR);

		try (DatagramSocket server = new DatagramSocket()){
			for (int i =0; i < 255; i++){
				String mensa = "Enviando mensaje n.-> "+i;
				DatagramPacket msgPacket = new
DatagramPacket(mensa.getBytes(),mensa.getBytes().length,addr,PORT);
				server.send(msgPacket);
				System.out.println("El Server ha enviado el
mensaje :"+mensa+i);
				Thread.sleep(1000);
			}
		}
		catch (IOException ex){
			ex.printStackTrace();
		}
	}

}
```

1.6.2 Cliente Multicast

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

public class UDPClientMulticast {

	final static String INET_ADDR = "224.0.0.3";
	final static int PORT = 8889;


	public static void main(String[] args) throws UnknownHostException,
InterruptedException {

		InetAddress addr = InetAddress.getByName(INET_ADDR);
		//buffer para almacenar los datos que nos llegan
		byte[] buffer = new byte[512];

		try (MulticastSocket cliente = new MulticastSocket(PORT)){

			cliente.joinGroup(addr);

			while (true){
```

```
                        //Recibir paquete
                        DatagramPacket msgPacket = new
DatagramPacket(buffer,buffer.length);
                        cliente.receive(msgPacket);

                        //convertir bytes to string
                        String mensa = new String(buffer,0,buffer.length);

                        System.out.println("Mensaje recibido :"+mensa);
                  }
            }
            catch (IOException ex){
                  ex.printStackTrace();
            }
      }

}
```

Ejercicio 1.6.1 Realizar un servidor de hora multicast

2. Aplicaciones RMI

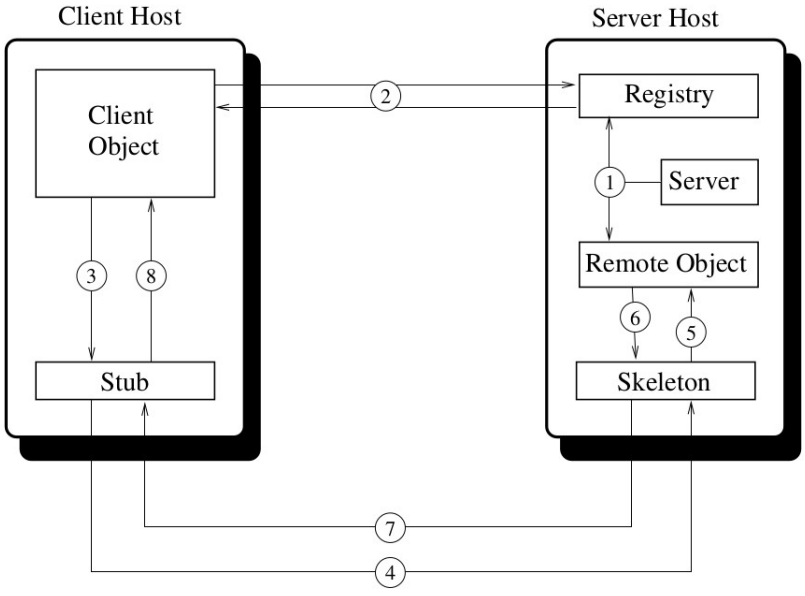
RMI es un sistema que nos permite el intercambio de objetos, el cual se realiza de manera transparente de un espacio de direcciones a otro, puesto que utiliza una técnica de serialización.

Además nos pemite el llamado de los métodos remotamente, sin tener la necesidad de tener los métodos localmente. Debido a que los sistemas necesitan manejo de datos en sistemas distribuidos cuando estos residen en direcciones de distintos hosts, los métodos de invocación remota son una buena alternativa para solucionar estas necesidades.

RMI es un sistema de programación para la distribución e intercambio de datos entre distintas aplicaciones existentes en un entorno distribuido. Se debe tener en cuenta que es más lento porque los objetos tienen que serializarse y luego deserializarse, se debe chequear la seguridad, los paquetes tienen que ser ruteados a través de switches. Esto trae como conclusión que, lamentablemente, el diseño de un sistema distribuido, no es solamente tomar un conjunto de objetos y ponerlos en otro proceso para balancear la carga.

En RMI intervienen dos progamas java que están ejecutándose en el mismo o distintos ordenadores. Uno de ellos, llamado **servidor**, "publica" algunos de los objetos/clases que tiene instanciadas, es decir, los pone visibles desde la red, de forma que otros programas java puedan llamar a sus métodos.

El otro programa, llamado **cliente**, localiza en el servidor el objeto publicado y llama a sus métodos. Estos métodos se ejecutan en el programa servidor y devuelven un resultado (por el return habitual) que recoge el cliente en su ordenador. El cliente se queda "colgado" en la llamada hasta que el servidor termina de ejecutar el código y devuelve un resultado. Los objetos que comparten el servidor se conocen como objetos remotos.



Arquitectura Java RMI

Objetos distribuidos
Elementos principales:

- Interfaces remotas
- Objetos remotos
- Objetos serializables
- Stubs
- Servicio de nombres

Interfaces remotas. Es una interfaz acordada entre el servidor y el cliente. Un método que el cliente puede invocar.
Las clases de los parámetros y del resultado han de ser serializables (en Java simplemente una interfaz) o remotos.
Un objeto se convierte en remoto implementando un interface remoto, que tenga estas características. Un interface remoto desciende del interface **java.rmi.Remote.**

Objetos remotos. Son objetos cuyos mensajes pueden ser invocados remotamente (desde objetos corriendo en otro proceso. En el caso de Java sería desde otra JVM).
Los objetos remotos deben implementar uno o varios interfaces remotos.
La clase del objeto remoto podría incluir implementaciones de otros interfaces (locales o remotos) y otros métodos (que sólo estarán disponibles localmente). Si alguna clase local va a ser utilizada como parámetro o cómo valor de retorno de alguno de esos métodos, también debe ser implementanda.

Objetos serializables. El RMI utiliza el mecanismo de serialización de objetos para transportar objetos entre máquinas virtuales. Implementar Serializable hace que la clase sea capaz de convertirse en un stream de bytes auto-descriptor que puede ser utilizado para reconstruir una copia exacta del objeto serializado cuando el objeto es leído desde el stream.

Stubs. Actúan como referencias a objetos remotos en el cliente. Es una clase usada por el cliente en sustitución de la remota.
Su clase es generada automáticamente a partir de la interfaz, e implementa la interfaz remota. La implementación de cada operación envía un mensaje a la máquina virtual que ejecuta el objeto remoto y recibe el resultado, retransmitiendo llamadas desde el cliente hacia el servidor y siendotransparente al código del cliente. Cuando un cliente invoca una operación remota que devuelve una referencia a un objeto remoto, obtiene una instancia del stub correspondiente.

Servicio de nombres. Permite asociar nombres lógicos a objetos. El servidor asocia un nombre a un objeto, el cliente obtiene una referencia al objeto a partir del nombre (stub), y así se conseguiría el objetivo, tener transparencia de localización.

```
javac -d . Hello.java Server.java Client.java
rmiregistry &
java -classpath . -Djava.rmi.server.codebase=file:. example.hello.Server

javac -d . *.java
rmiregistry &
java -classpath . -Djava.rmi.server.codebase=file:. main.Servidor
java -classpath . -Djava.rmi.server.codebase=file:. main.Cliente
```

2.1 Un servicio básico: servicio de eco

Dividiremos el desarrollo de este servicio en las siguientes etapas:

- Definición del servicio
- Implementación del servicio
- Desarrollo del servidor
- Desarrollo del cliente
- Compilación
- Ejecución

Definición del servicio

En RMI para crear un servicio remoto es necesario **definir una interfaz que derive de la interfaz Remote** y que contenga los métodos requeridos por ese servicio, especificando en cada uno de ellos que pueden activar la excepción RemoteException, usada por RMI para notificar errores relacionados con la comunicación.

Este primer servicio ofrece únicamente un **método remoto que retorna la cadena de carácteres recibida como argumento pero pasándola a mayúscula**s.

A continuación, se muestra el código de esta definición de servicio (fichero ServicioEco.java):

```
import java.rmi.*;

interface ServicioEco extends Remote {
        String eco (String s) throws RemoteException;
}
```

Implementación del servicio

Es necesario desarrollar el código que implementa cada uno de los servicios remotos. Ese código debe estar incluido en **una clase que implemente la interfaz de servicio** definida en la etapa previa. **Para permitir** que los métodos remotos de esta clase puedan ser **invocados externamente**, la opción más sencilla es definir esta clase como **derivada** de la clase **UnicastRemoteObject**. La principal limitación de esta alternativa es que, debido al modelo de herencia simple de Java, nos impide que esta clase pueda derivar de otra relacionada con la propia esencia de la aplicación (así, por ejemplo, con esta solución no podría crearme una clase que deriva a la vez de Empleado y que implemente un cierto servicio remoto). En esta guía usaremos esta opción. Consulte la referencia previamente citada para estudiar la **otra alternativa** (basada en usar el método estático **exportObject de UnicastRemoteObject**).

Recapitulando, desarrollaremos una clase derivada de UnicastRemoteObject y que implemente la interfaz remota ServicioEco (fichero ServicioEcoImpl.java):

```
import java.rmi.*;
import java.rmi.server.*;

class ServicioEcoImpl extends UnicastRemoteObject implements ServicioEco {

    ServicioEcoImpl() throws RemoteException {
    }

    public String eco(String s) throws RemoteException {
        return s.toUpperCase();
    }
}
```

Observe la necesidad de hacer **explícito el constructor** para poder declarar que éste puede generar la excepción RemoteException.

Observación
Una interfaz es una clase abstracta que es utilizado para agrupar métodos relacionados sin implementar.
Para acceder a los métodos de la interfaz, esta debe ser implementada en otra clase.

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Es importante entender que **todos los objetos** especificados como parámetros de un **método remoto**, así como el retornado por el mismo, **se pasan por valor**, y no por referencia como ocurre cuando se realiza una invocación a un método local. Esta característica tiene como consecuencia que cualquier cambio que se haga en el servidor sobre un objeto recibido como parámetro no afecta al objeto original en el cliente. Por ejemplo, este método remoto no llevará a cabo la labor que se le supone, aunque sí lo haría en caso de haber usado ese mismo código (sin la excepción RemoteException, evidentemente) para definir un método local.

```
    public void vuelta(StringBuffer s) throws RemoteException {
        s.reverse();
    }
```

Un último aspecto que conviene resaltar es que la clase que implementa la interfaz remota es a todos los efectos una clase convencional y, por tanto, puede incluir otros métodos, además de los especificados en la interfaz. Sin embargo, esos métodos no podrán ser invocados directamente por los clientes del servicio.

Desarrollo del servidor

El programa que actúe como servidor debe iniciar el servicio remoto y hacerlo públicamente accesible usando, por ejemplo, el rmiregistry (el servicio básico de binding en Java RMI). Nótese que se podría optar por usar la misma clase para implementar el servicio y para activarlo pero se ha preferido mantenerlos en clases separadas por claridad.

A continuación, se muestra el código del servidor (fichero ServidorEco.java):

```
import java.rmi.*;
import java.rmi.server.*;

class ServidorEco  {
    static public void main (String args[]) {
       if (args.length!=1) {
            System.err.println("Uso: ServidorEco numPuertoRegistro");
            return;
        }
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            ServicioEcoImpl srv = new ServicioEcoImpl();
            Naming.rebind("rmi://localhost:" + args[0] + "/Eco", srv);
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString());
            System.exit(1);
        }
        catch (Exception e) {
            System.err.println("Excepcion en ServidorEco:");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Resaltamos los siguientes aspectos de ese código:

El programa asume que ya está arrancado el rmiregistry previamente (otra opción hubiera sido que lo arrancase el propio programa usando el método estático createRegistry de LocateRegistry). En la sección que explica cómo ejecutar el programa se muestra el procedimiento para arrancar el rmiregistry. Nótese que el programa espera recibir como único argumento el número de puerto por que el que está escuchando el rmiregistry.

Un aspecto clave en Java RMI es la seguridad. En el código del servidor se puede apreciar cómo éste instancia un gestor de seguridad (más sobre el tema en la sección dedicada a la ejecución del programa). Para ejemplos sencillos, podría eliminarse esta parte del código del servidor (y del cliente) pero es conveniente su uso para controlar mejor la seguridad y es un requisito en caso de que la aplicación requiera carga dinámica de clases.

La parte principal de este programa está incluida en la sentencia try y consiste en crear un objeto de la clase que implementa el servicio remoto y darle de alta en el rmiregistry usando el método estático rebind que permite especificar la operación usando un formato de tipo URL. Nótese que el rmiregistry sólo permite que se den de alta servicios que ejecutan en su misma máquina.

Desarrollo del cliente

El cliente debe obtener una referencia remota (es decir, una referencia que corresponda a un objeto remoto) asociada al servicio para luego simplemente invocar de forma convencional sus métodos, aunque teniendo en cuenta que pueden generar la excepción RemoteException. En este ejemplo, la referencia la obtiene a través del rmiregistry.

A continuación, se muestra el código del cliente (fichero ClienteEco.java):

```
import java.rmi.*;
import java.rmi.server.*;

class ClienteEco {
    static public void main (String args[]) {
        if (args.length<2) {
            System.err.println("Uso: ClienteEco hostregistro
numPuertoRegistro ...");
            return;
        }

        if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        try {

            ServicioEco srv = (ServicioEco) Naming.lookup("//" + args[0] + ":" +
args[1] + "/Eco");

            for (int i=2; i<args.length; i++)
                System.out.println(srv.eco(args[i]));
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString());
        }
        catch (Exception e) {
            System.err.println("Excepcion en ClienteEco:");
            e.printStackTrace();
        }
    }
}
```

Resaltamos los siguientes aspectos de ese código:

El programa espera recibir como argumentos la máquina donde ejecuta rmiregistry, así como el puerto por el que escucha. El resto de los argumentos recibidos por el programa son las cadenas de caracteres que se quieren pasar a mayúsculas.

Como ocurre con el servidor, es conveniente, e incluso obligatorio en caso de descarga dinámica de clases, la activación de un gestor de seguridad.

El programa usa el método estático **lookup** para obtener del **rmiregistry** una referencia remota del servicio. Observe el uso de la operación de cast para adaptar la referencia devuelta por lookup, que corresponde a la interfaz Remote, al tipo de interfaz concreto, derivado de Remote, requerido por el programa (ServicioEco).

Una vez obtenida la referencia remota, la invocación del método es convencional, requiriendo el tratamiento de las excepciones que puede generar.

Compilación
El proceso de compilación tanto del cliente como del servidor es el habitual en Java. El único punto que conviene resaltar es que para generar el programa cliente, además de la(s) clase(s) requerida(s) por la funcionalidad del mismo, se debe disponer del fichero class que define la interfaz (en este caso, ServicioEco.class), tanto para la compilación como para la ejecución del cliente. Esto se ha resuelto en este ejemplo creando un enlace simbólico. Si quiere probar el ejemplo usando dos máquinas, lo que recomendamos, deberá copiar el fichero class a la máquina donde se ejecutará el cliente. Obsérvese que no es necesario, ni incluso conveniente, disponer en el cliente de las clases que implementan el servicio.

Hay que resaltar que en la versión actual de Java (realmente, desde la versión 1.5) no es necesario usar ninguna herramienta para generar resguardos ni para el cliente (proxy) ni para el servidor (skeleton). En versiones anteriores, había que utilizar la herramienta rmic para generar las clases que realizan esta labor, pero gracias a la capacidad de reflexión de Java, este proceso ya no es necesario.

En el ejemplo que nos ocupa, dado que, por simplicidad, no se han definido paquetes ni se usan ficheros JAR, para generar el programa cliente y el servidor, basta con entrar en los directorios respectivos y ejecutar directamente:

```
javac *.java
```

Ejecución
Antes de ejecutar el programa, hay que arrancar el registro de Java RMI (rmiregistry). Este proceso ejecuta por defecto usando el puerto 1099, pero puede especificarse como argumento al arrancarlo otro número de puerto, lo que puede ser lo más conveniente para evitar colisiones en un entorno donde puede haber varias personas probando aplicaciones Java RMI en la misma máquina.

Hay que tener en cuenta que el rmiregistry tiene que conocer la ubicación de las clases de servicio. Para ello, puede necesitarse definir la variable de entorno CLASSPATH para el rmiregistry de manera que haga referencia a la localización de dichas clases. En cualquier caso, si el rmiregistry se arranca en el mismo directorio donde ejecutará posteriormente el servidor y en la programación del mismo no se han definido nuevos paquetes (todas las clases involucradas se han definido en el paquete por defecto), no es necesario definir esa variable de entorno. Así ocurre en este ejemplo:

```
cd servidor
rmiregistry 54321 &
```

Ya estamos a punto de poder ejecutar el servidor y el cliente, y si puede ser, mejor en dos máquinas diferentes. Sin embargo, queda un último aspecto vinculado con la seguridad. Dado que tanto en el cliente como en el servidor se ha activado un gestor de seguridad, si queremos poder definir nuestra propia política de seguridad, con independencia de la que haya definida por defecto en el sistema, debemos crear nuestros propios ficheros de políticas de seguridad.

Dado que estamos trabajando en un entorno de pruebas, lo más razonable es **crear ficheros de políticas de seguridad que otorguen todos los permisos** posibles tanto para el cliente (fichero que hemos llamada cliente.permisos) como para el servidor (fichero servidor.permisos):

```
grant  {
    permission java.security.AllPermission;
};
```

Dada la importancia de esta cuestión, se recomienda que el lector revise más en detalle la misma en las numerosas fuentes disponibles en Internet que tratan este tema.

Procedemos finalmente a la ejecución del servidor:

```
cd servidor
java -Djava.security.policy=servidor.permisos  ServidorEco 54321
```

Y la del cliente:

```
cd cliente
java  -Djava.security.policy=cliente.permisos   ClienteEco  localhost  54321  hola
adios
HOLA
ADIOS
```