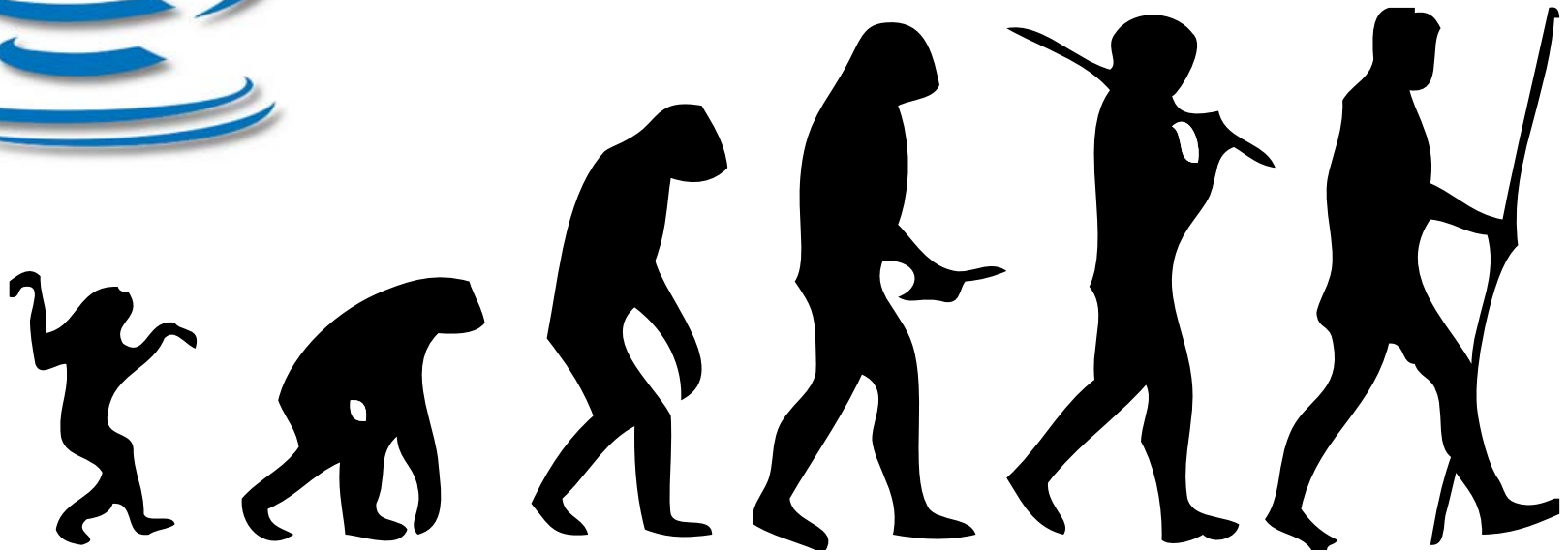
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA

Programando en Java

Raspberry Pi (RPi)



2015/05/08

Juan V. Capella

DISCA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contenido

- Objetivo
- Introducción
- Tecnología Java
- Clases y objetos. Herencia
- Aplicaciones y Applets
- Tipos de datos y estructuras
- Operadores
- Networking
- GPIO
- GUI

Objetivo

- Introducirse en la programación en Java con la Raspberry Pi
- Aprender a desarrollar aplicaciones distribuidas

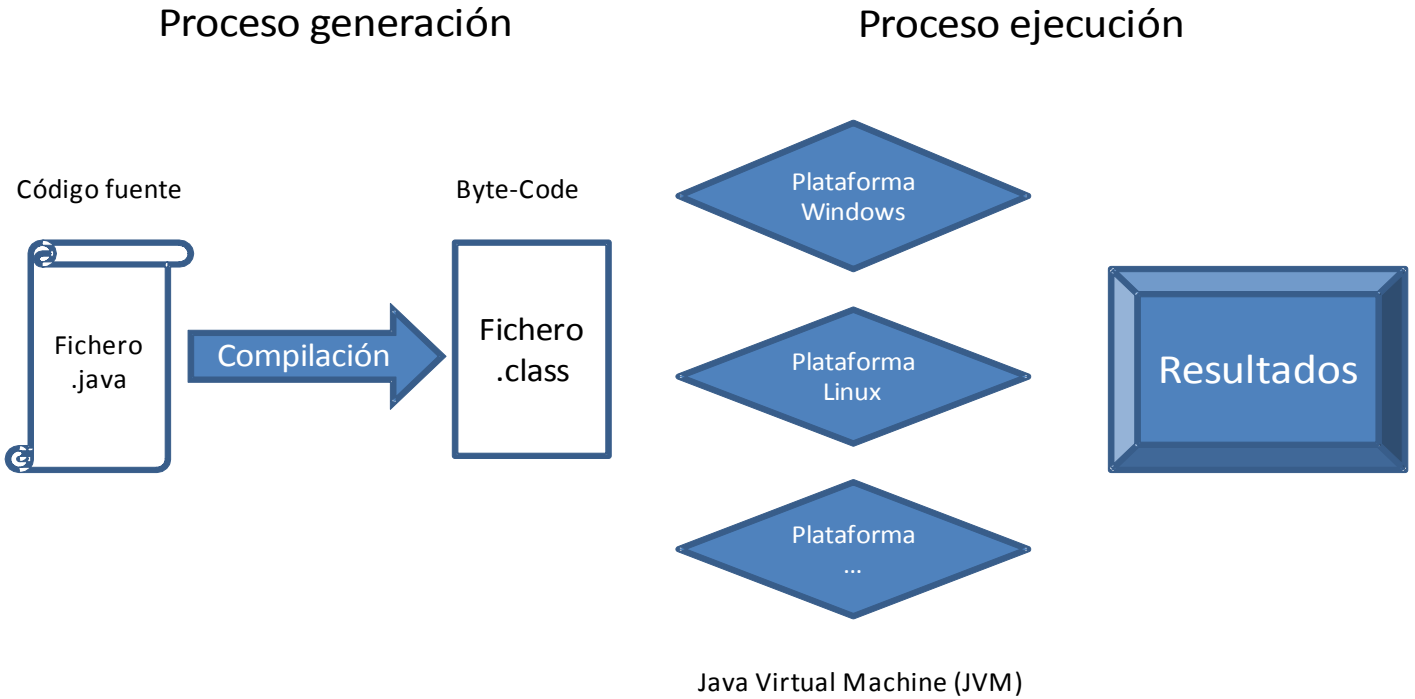
Introducción

- Lenguaje de programación desarrollado por Sun Microsystems a principios de los años 90
- La motivación principal Java: *Write Once, Run Anywhere*
  - Proporcionar un lenguaje independiente de la plataforma y un entorno de ejecución ligero y gratuito, para poder implementarlo en cualquier dispositivo
- Al tener que ser ejecutado mediante la JVM hace que no sea tan rápido como con otras opciones, por ejemplo C/C++.
- Java es Orientado a Objetos
- Java es Multi-hilo
- No hay punteros
- Recolección de Basura “Garbage Collection”
  - Responsable de liberar cualquier memoria que pueda ser liberada. Esto se realiza de manera automática durante la vida del programa Java.
  - El programador se libera de la carga de tener que liberar la memoria no utilizada.
- Seguridad
  - Java fue diseñado para hacer más fácil el desarrollo de código sin bugs

Tecnología Java

- Lenguaje programación (JAVA)
- Entorno de desarrollo (JDK). Herramientas:
  - Compilador (javac)
  - Intérprete (java)
  - Generador de documentación (javadoc)
  - Una herramienta para empaquetar los .class
  - Etc…
- Entorno en tiempo de ejecución de Java (JRE)
  - formado por una Máquina Virtual de Java (JVM), un conjunto de bibliotecas Java y otros componentes necesarios para que una aplicación escrita en lenguaje Java pueda ser ejecutada

Fases



Tarea	Herramienta a usar	Salida
Escribir el programa	Cualquier editor de texto o IDE	Fichero .java
Compilar el programa	Compilador Java (javac)	Fichero .class (Java bytecodes)
Ejecutar el programa	Intérprete Java (java)	Salida programa

Clases y objetos

- Las clases están compuestas por un conjunto de miembros (datos y funciones)
- Una clase se usa como el modelo que deben seguir los objetos
- Un objeto es una instancia de una clase
- Las clases y objetos son similares a los tipos de datos y a las variables

Clases y objetos

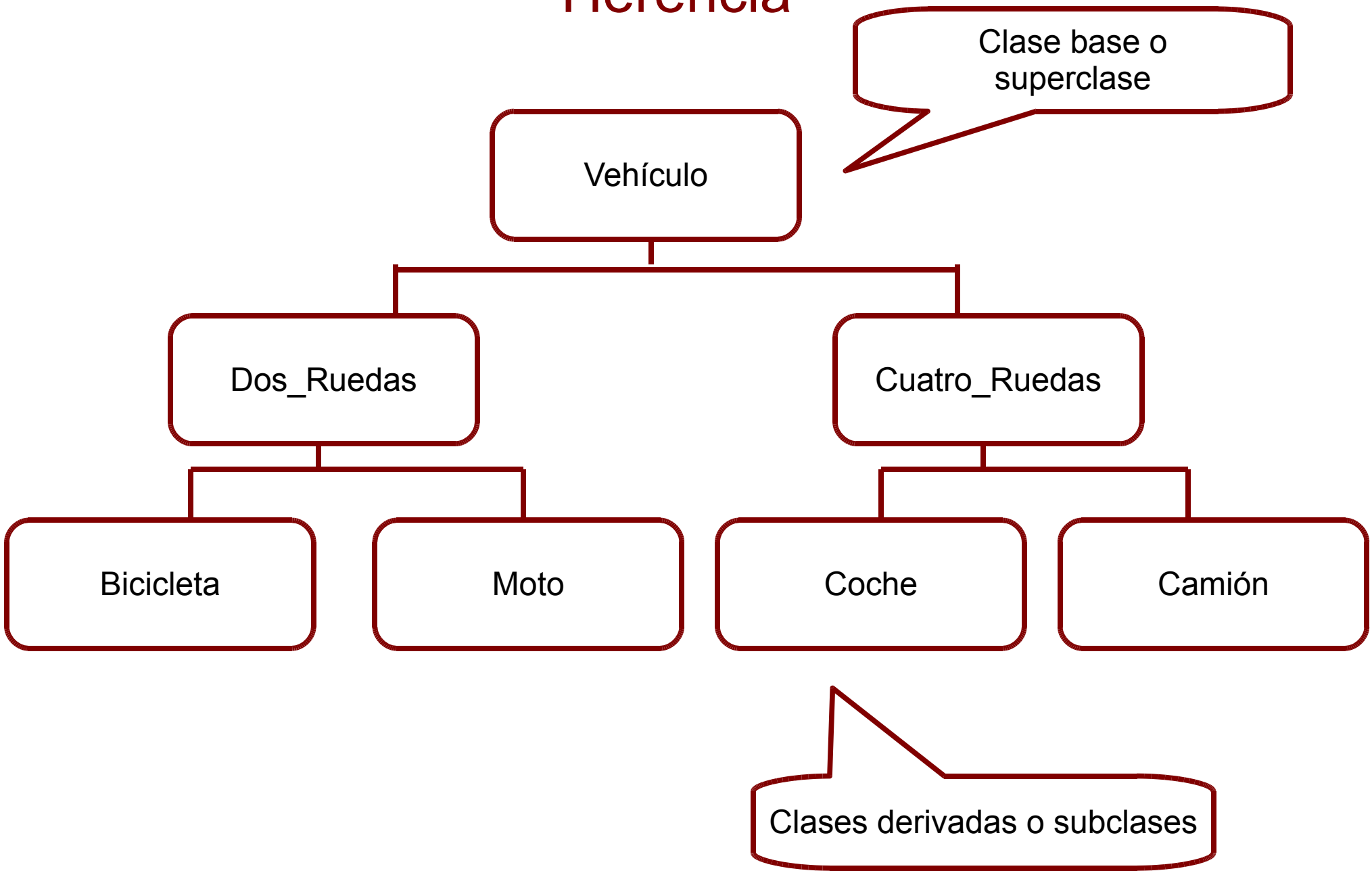
```
class Persona {
  int dni;
  String nombre;
  int peso;
  string profesion;
}
```

Definición de clase

Creación de objeto

```
Persona juan = new Persona();
```

Herencia


Clase base o superclase
Vehículo
Dos_Ruedas
Cuatro_Ruedas
Bicicleta
Moto
Coche
Camión
Clases derivadas o subclases

Una aplicación en Java

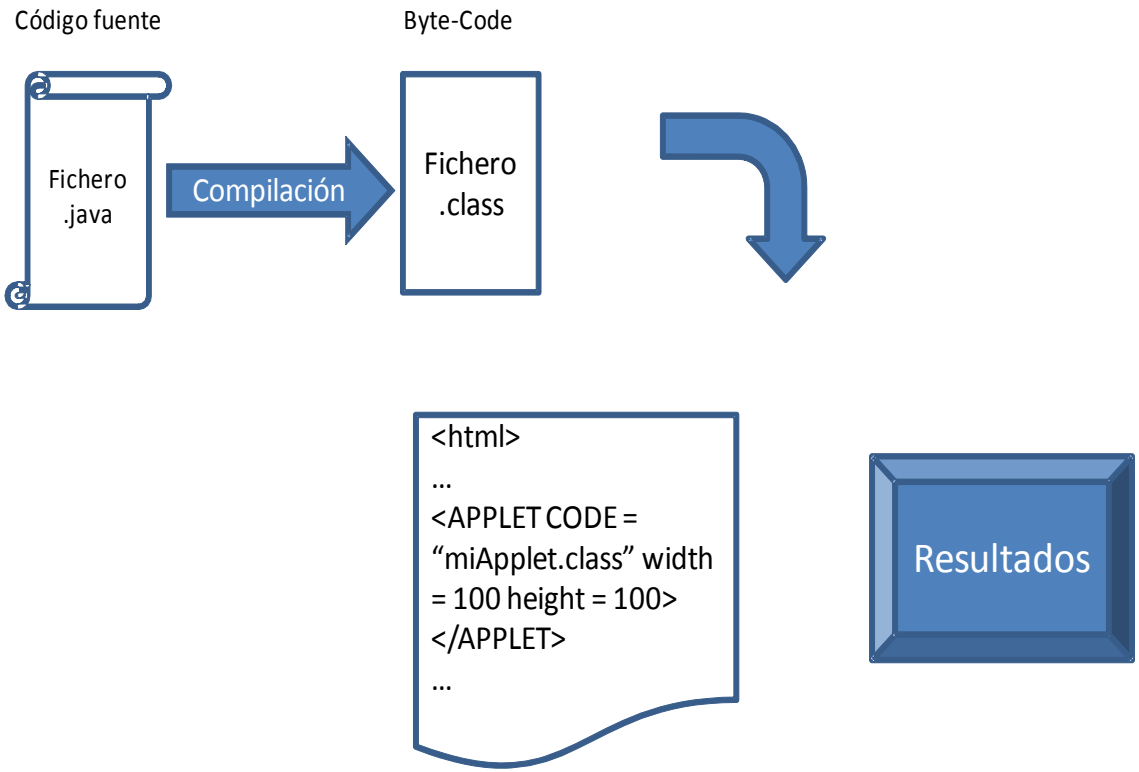
Toda aplicación java debe estar dentro de una clase

Método main()

```
public class HolaMundo {
      public static void main(String [] args) {
          System.out.println(”¡Hola Mundo!”);
        }
   }
```

Applets

- Programa escrito en Java que puede ejecutarse en un navegador web utilizando la Java Virtual Machine (JVM), o en el AppletViewer de Sun.



Un applet en Java

```
import java.awt.*;
import applet.Applet;
public class miApplet extends Applet {
      public void paint(Graphics g) {
           g.drawString(”¡Hola Mundo!”,50,50);
          }
   }
```

Importación de paquetes

Los Applets no tienen método main()

Tipos de datos

- byte, short, int, long
- float, double
- boolean

La clase String

- Se usa para almacenar cadenas de caracteres.
- Se pueden usar para: examinar los caracteres contenidos en un texto, buscar subcadenas, comparar cadenas, etc.

Vectores en Java

- Almacenan valores de un mismo tipo
- El índice del vector identifica cada elemento

```
int vect[] = {2,7,1};
String[] s = {“hola”, “mundo”};
```

Estructuras de control en Java: if

- if(){ }
- if(){ }else{ }
- if(){ }else if(){ }

Estructuras de control en Java: Bucles

- for(;;){}
- while(){}
- do{}while();

Estructuras de control en Java: switch-case

```
int a = 2;
switch(a){
  case 1: a++;
  break;
  case 2: a = a + 4;
  break;
  default: a = a + 50;
}
```

Declaración de variables

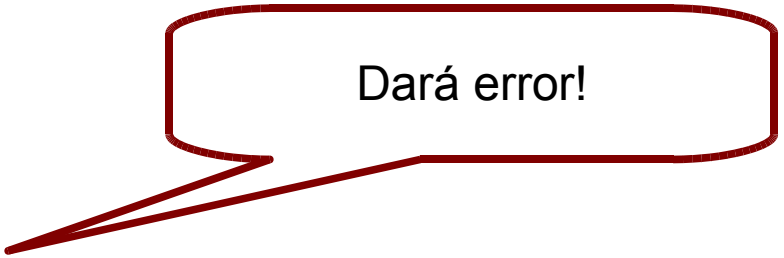
- Se pueden declarar variables en cualquier lugar del código
- Al declarar la variable podemos inicializar el contenido
- No se permite el uso de variables sin previa inicialización

```
int numCoches, identificador;
```

Alcance y bloques de código

- Los bloques se definen con {}
- Las variables declaradas en ese bloque solo existen en ese bloque

```
{
   int numCoches = 0;
}
{
   numCoches++
}
```



Conversión de tipos

- La conversión automática se realiza siempre que el tipo de la expresión a la derecha pueda ser transformado de forma segura al tipo de la expresión a la izquierda
  - Vamos, que no se puede convertir automáticamente un `float` a un `int` porque el primero requiere de más espacio de almacenamiento que el segundo, lo que puede resultar en pérdida de información
- Para forzar la conversión → conversión explicita:

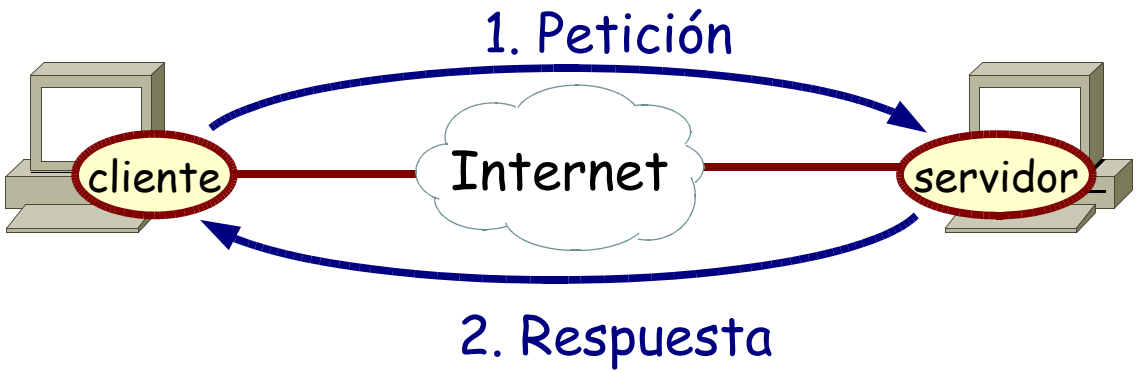
```
int v1;
float v2=3.47;
v1 = (int) v2;
```

Operadores

- ==
- !=
- >
- <
- <=
- >=
- &&
- ||
- !

Interfaz de programación en red: Sockets en JAVA

- Al comunicarse dos procesos siguen un modelo Cliente / Servidor:



Cliente:

- **Inicia la comunicación**
- Solicita un servicio al servidor
- Ejemplo:
  - Un cliente web solicita una página

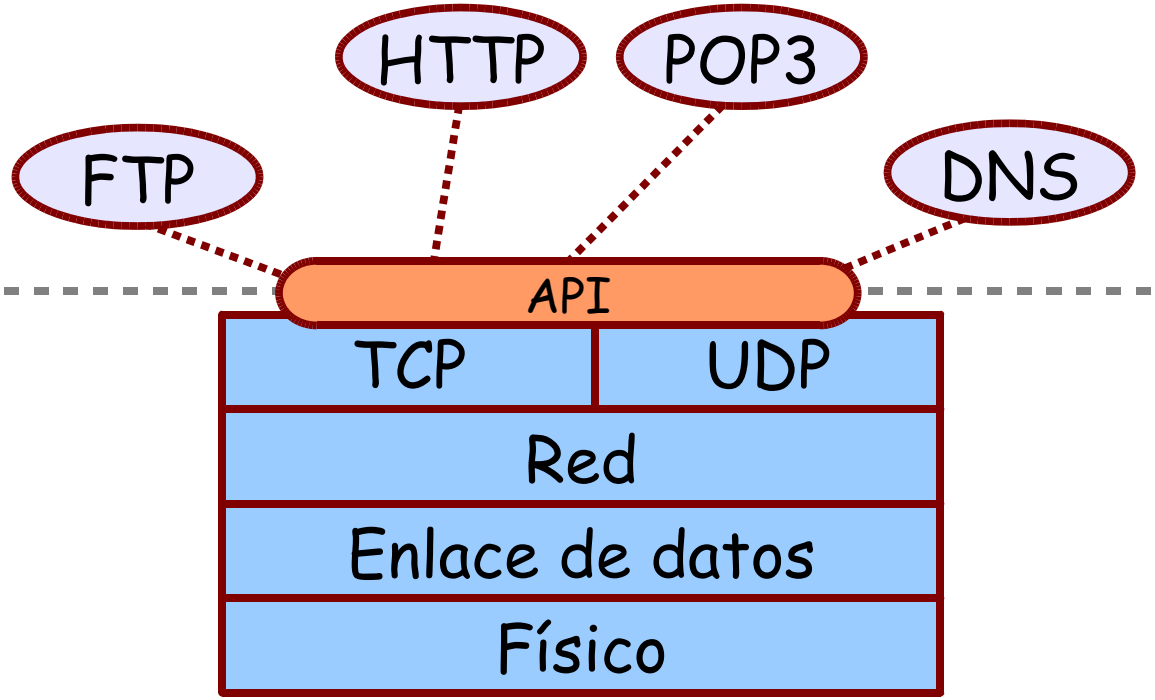
Servidor:

- **Espera peticiones**
- Proporciona el servicio solicitado
- Ejemplo:
  - El servidor web envía la página solicitada por el cliente

Interfaz de programación en red: Sockets en JAVA

- Clientes y servidores utilizan protocolos de transporte

Los procesos de las aplicaciones residen en el espacio de **usuario**



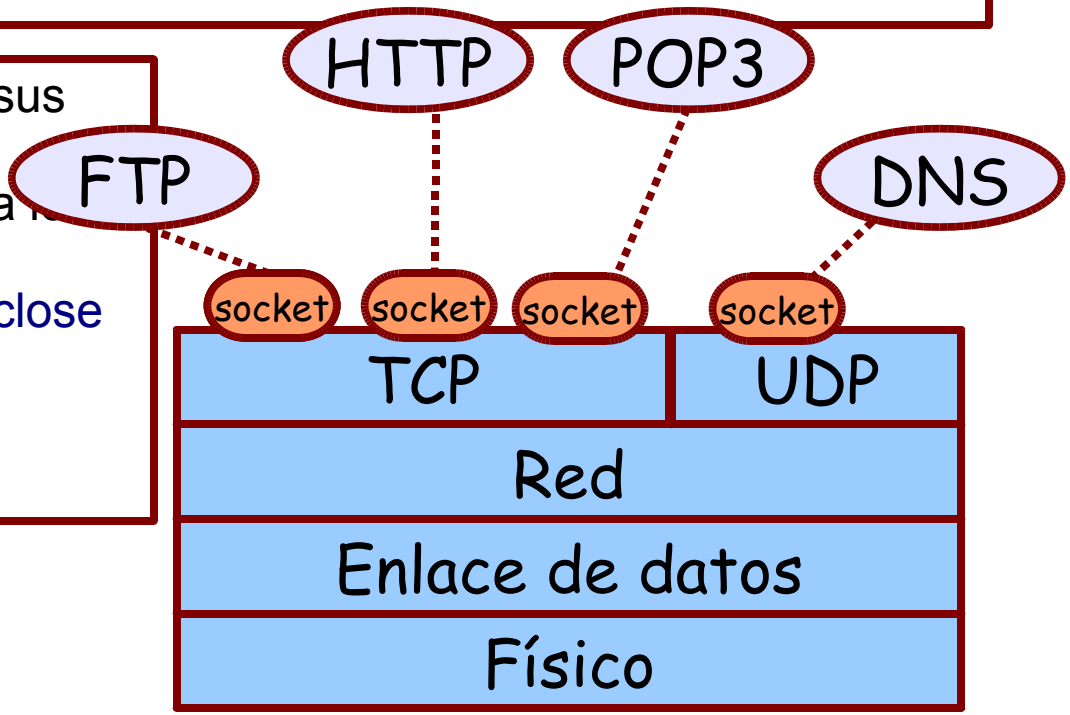
Los procesos de los protocolos de transporte forman parte del ***S.O.***

- Se necesita un mecanismo para ponerlos en contacto
  - API (*Application Programming Interface*)

API socket

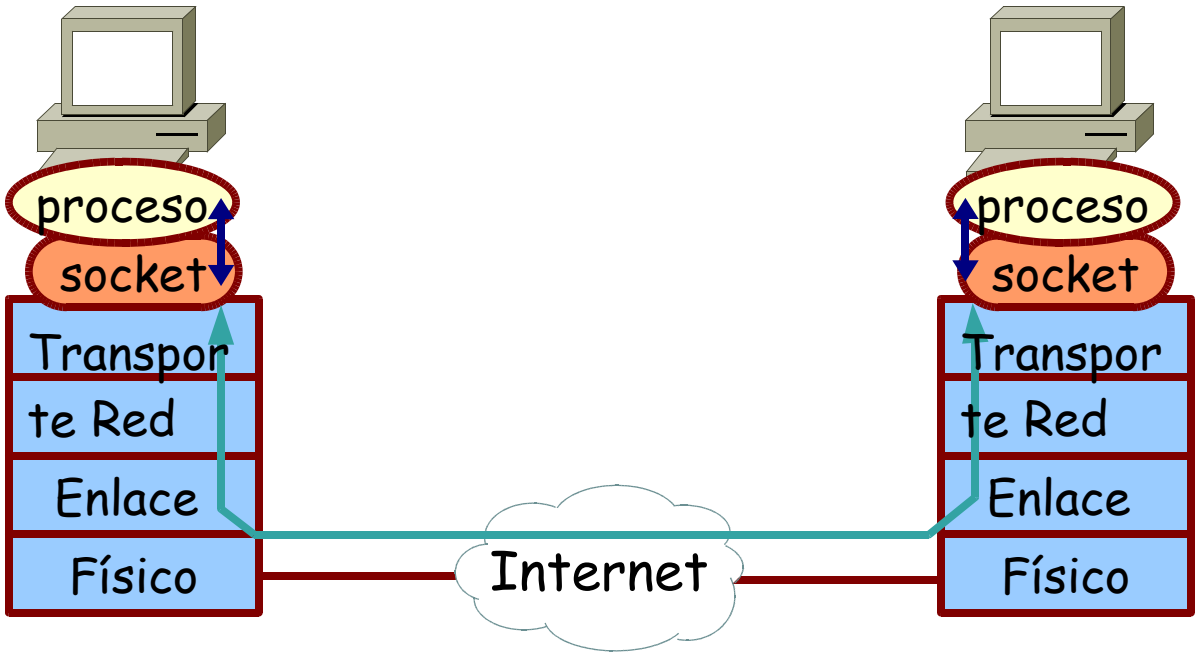
- Permite a las aplicaciones utilizar los protocolos de la pila TCP/IP

- Define las operaciones permitidas y sus argumentos
  - Parecido a la forma de acceder a los ficheros en Unix
  - Operaciones: open, read, write, close



Socket

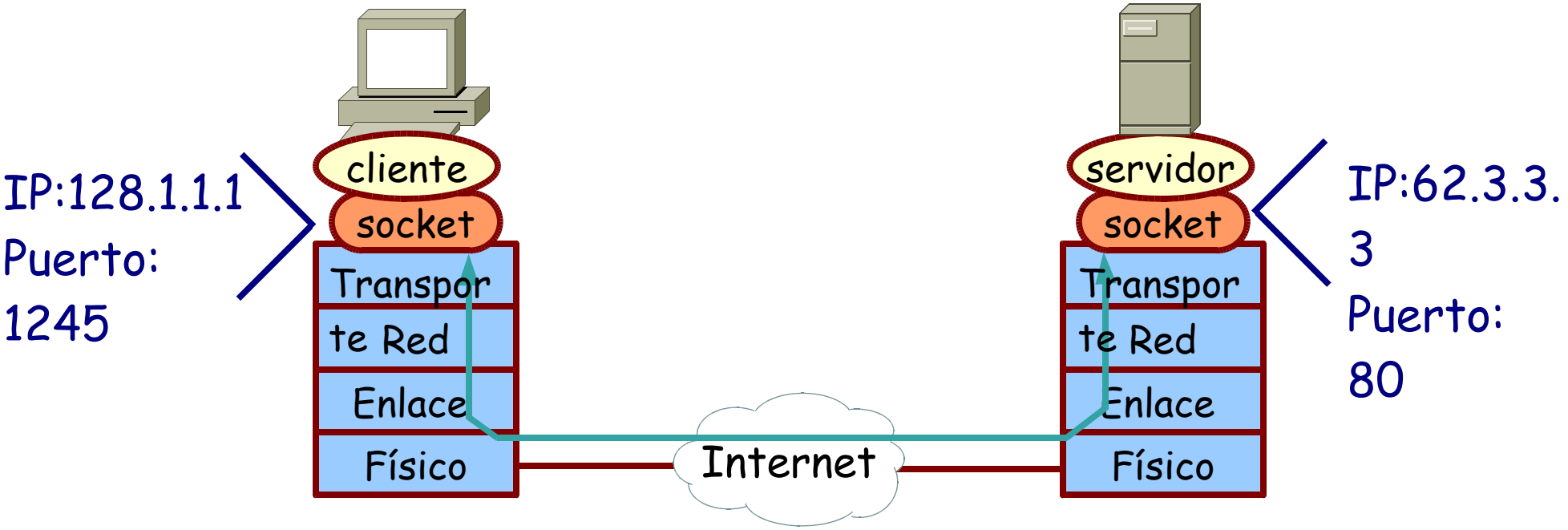
- Es una abstracción del sistema operativo
  - Las aplicaciones los crean, los utilizan y los cierran cuando ya no son necesarios
  - Su funcionamiento está controlado por el sistema operativo
- Comunicación entre procesos
  - Los procesos envían y reciben mensajes a través de sockets
  - Los mensajes fluyen entre sockets



Identificación de los sockets

- La comunicación en Internet es de socket a socket
- El proceso que está comunicándose se identifica en Internet por medio de su socket
- El socket tiene un identificador

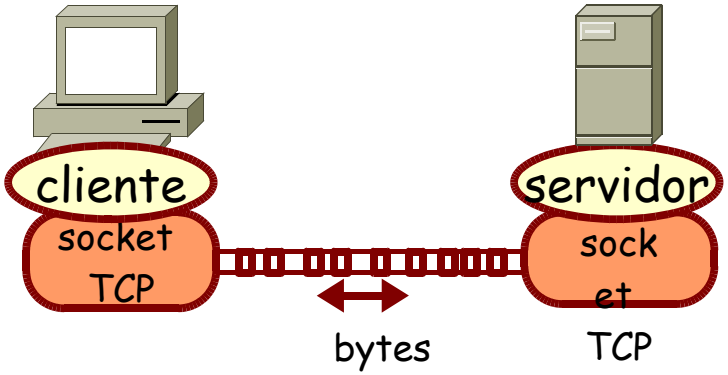
Identificador = dir. IP del computador + núm. puerto

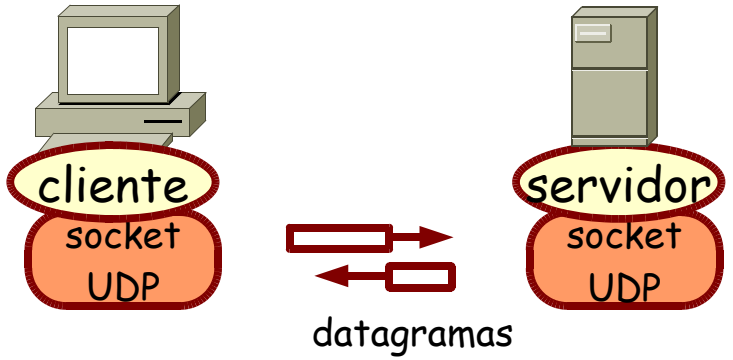


Tipos de sockets

- **Sockets TCP**
  - Las aplicaciones piden al S.O. una comunicación controlada por TCP:
    - Orientada a la conexión
    - Comunicación fiable y ordenada
- También se denominan sockets de tipo **Stream**

- **Sockets UDP**
  - Las aplicaciones piden al S.O. una comunicación controlada por UDP:
    - Transferencia de bloques de datos
    - Sin conexión ni fiabilidad ni entrega ordenada
    - Permite difusiones
- También se denominan sockets de tipo **Datagram**





Los sockets en Java

- Dentro del paquete **java.net** existen tres clases de sockets:
  - **Socket** Cliente **TCP**
  - **ServerSocket** Servidor **TCP**
  - **DatagramSocket** Cliente/Servidor **UDP**
- También hay otras clases auxiliares que facilitan la programación de aplicaciones en red

```
http://java.sun.com
```

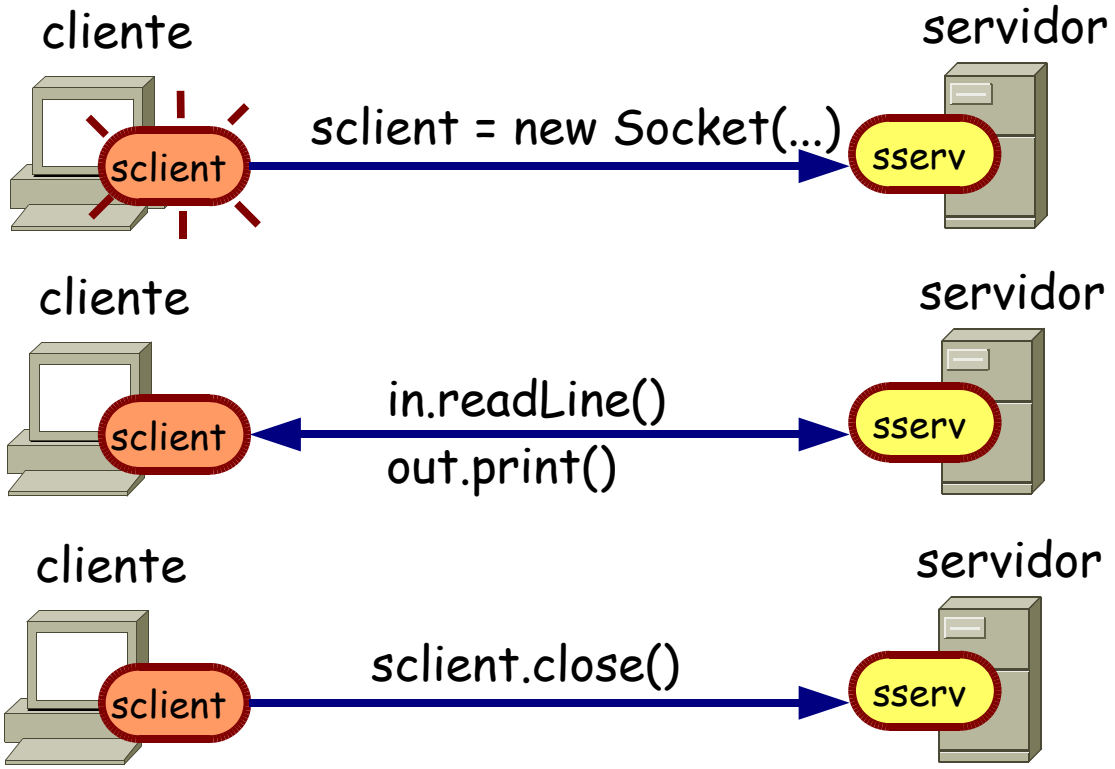
Direcciones IP en Java

Clase `InetAddress`

- `InetAddress` es la clase que se utiliza para almacenar direcciones IP
- **Algunos métodos importantes**
  - `InetAddress getByName(String nombre)`
    - Obtiene la dirección IP asociada a un nombre
  - `String getHostAddress()`
    - Devuelve la dirección IP en formato `"aa.bb.cc.dd"`
  - `String getHostName()`
    - Devuelve el nombre del host

Sockets TCP

- Cliente:
  - Crea un socket (sclient) y lo conecta con el del servidor
  - Transfiere información
  - Cierra el socket y la conexión (a veces lo hace el servidor)



- Servidor:
  - Ha de estar en ejecución
  - Debe haber creado un socket (sserv) donde recibir a los clientes que conectan con él

Clientes TCP

Clase `Socket`

- **Constructores**
  - `Socket(InetAddress dirIP, int puerto)`
  - `Socket(String nombre, int puerto)`
    - Crea un socket y lo conecta con el servidor indicado
  - `Socket(InetAddress dirIP, int puerto, InetAddress dirIPLocal, int puertoLocal)`
  - `Socket(String nombre, int puerto, InetAddress dirIPLocal, int puertoLocal)`
    - Crea un socket y lo conecta con el servidor indicado, ligándolo a una dirección IP y puerto locales concretos

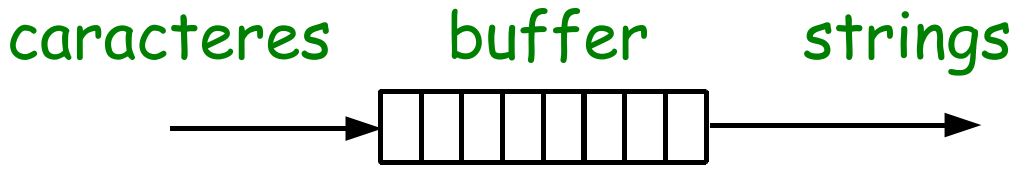
Clientes TCP

Clase `Socket`

- **Algunos métodos importantes**
  - `close()`: Cierra el socket
  - `InputStream getInputStream()`
    - Proporciona un descriptor para leer del socket
    - `InputStream` proporciona un flujo de bytes
      - Se puede leer un byte: `read()`
      - O un grupo de bytes: `read(byte[] b)`
  - `OutputStream getOutputStream()`
    - Proporciona un descriptor para escribir en el socket
    - `OutputStream` admite un flujo de bytes
      - Se puede escribir un byte: `write(int b)`
      - O un grupo de bytes: `write(byte[] b)`

Gestión de los flujos de entrada

- **`InputStreamReader`** convierte un flujo de bytes en un flujo de caracteres
  - Se puede leer un carácter: **`read()`**
  - O un grupo de caracteres: **`read(char[] text)`**
- **`BufferedReader`** añade un buffer al flujo



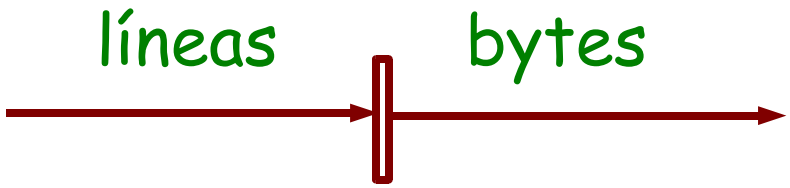
  - Se puede leer una línea de texto: **`String readLine()`**
- Ejemplo:

```
BufferedReader entrada=new BufferedReader(new

InputStreamReader(s.getInputStream()));
entrada.readLine()
```

Gestión de los flujos de salida

- **`PrintWriter`** permite enviar texto (caracteres)
  - Tiene métodos que permiten escribir una línea de texto:

    `print(String s)` y **`println(String s)`**



  - Ejemplo:

```
PrintWriter salida = new PrintWriter(s.getOutputStream());

salida.print("GET / HTTP/1.0" + "\r\n");
```

Cliente TCP básico

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

class ClienteTCP {

public static void main(String args[])
                        throws UnknownHostException, IOException {
  Socket s=new Socket("zoltar.redes.upv.es",21);
  BufferedReader entrada=new BufferedReader(new
                              InputStreamReader(s.getInputStream()));
  System.out.println(entrada.readLine());
  s.close();
 }

}
```

- Este cliente se conecta al servidor FTP (puerto 21) y visualiza la primera línea que recibe del servidor. Después cierra la conexión
- Genera una salida similar a:

```
220 (vsFTPd 1.2.1)
```

Segundo cliente TCP

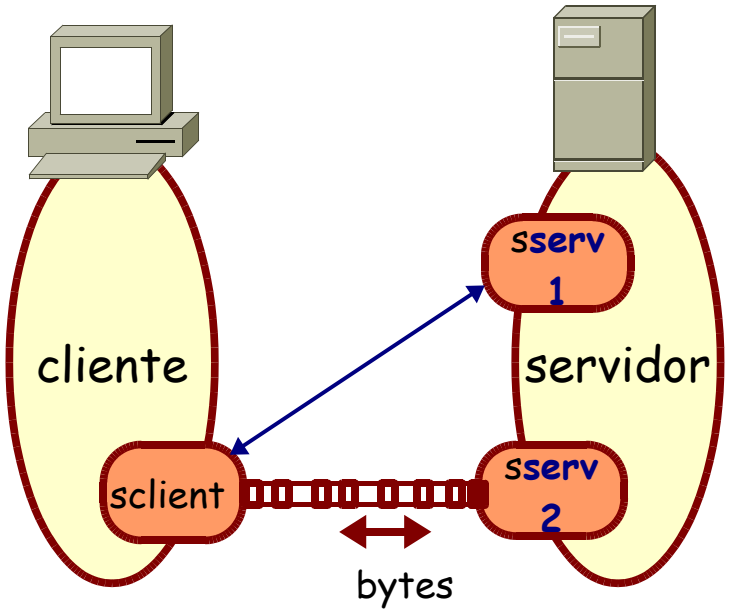
```
import java.net.*;
import java.io.*;

class ClienteTCP2 {

public static void main(String args[]) throws UnknownHostException,IOException {
 Socket s = new Socket("www.upv.es",80);
 BufferedReader entrada =
                new BufferedReader(new InputStreamReader(s.getInputStream()));
 PrintWriter salida = new PrintWriter(s.getOutputStream());
 salida.print("GET / HTTP/1.0" + "\r\n");
 salida.print("\r\n");
 salida.flush();
 System.out.println(entrada.readLine());
 s.close();
 }

}
```

- Envía una petición HTTP al servidor www.upv.es
- Esta es la salida del programa:

```
   HTTP/1.1 200 OK
```

Servidores TCP

- Cliente:
  - Cuando crea un socket (*sclient*) establece la conexión con el servidor
- Servidor:
  - Debe haber creado un socket (*sserv1*) donde espera a los clientes que conectan con él
- Cuando un cliente se conecta con un servidor:
  - El servidor crea un nuevo socket (*sserv2*) para que el proceso servidor se comunique con el cliente
  - De esta forma es posible que un servidor se comunique con varios clientes simultáneamente



Servidores TCP

Clase `ServerSocket`

- **Constructores**
  - `ServerSocket(int puerto)`
    - Abre un socket en el puerto indicado en modo de escucha
    - Si `port = 0`, entonces se elige cualquier puerto libre
  - `ServerSocket(int puerto, int backlog)`
    - Abre un socket en el puerto indicado en modo de escucha
    - `backlog` indica la longitud máxima de la cola de conexiones en espera
    - Cuando llega una solicitud de conexión y la cola está llena, se rechaza la conexión

Servidores TCP

Clase `ServerSocket`

- **Algunos métodos importantes**
  - `Socket accept()`
    - Acepta una conexión de un cliente y devuelve un socket asociado a ella
    - El proceso se **bloquea** hasta que se realiza una conexión
    - El diálogo con el cliente se hace por el nuevo socket
    - El ServerSocket puede atender nuevas conexiones
  - `close()`
    - Cierra el socket servidor

Primer servidor TCP

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

class ServidorTCP {

public static void main(String args[]) throws IOException {
  ServerSocket ss=new ServerSocket(7777);
  Socket s=ss.accept(); // espero a que llegue un cliente
  PrintWriter salida=new PrintWriter(s.getOutputStream(),true);
  salida.println("Bienvenido al servidor de prueba de Redes");
  s.close();
  ss.close();
  }
}
```

- El servidor espera un cliente. Cuando éste se conecta, el servidor le envía una cadena de bienvenida y acaba

```
zoltar:~/java> telnet localhost 7777
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Bienvenido al servidor de prueba de Redes
Connection closed by foreign host.
```

Segundo servidor TCP

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

class ServidorTCP2 {
public static void main(String args[]) throws IOException {
  ServerSocket ss = new ServerSocket(7777);
  int cliente=1;
  while(true) {
    Socket s = ss.accept(); // espera una conexión de un cliente
    PrintWriter salida=new PrintWriter(s.getOutputStream(),true);
    salida.println("Usted es el cliente " + cliente);
    System.out.println("Atendiendo al cliente " + cliente++);
    s.close();
  }
 }
}
```

- **Servidor iterativo**: continúa atendiendo a nuevos clientes
  - A cada uno le envía una cadena con su número de cliente y después cierra el socket

Sockets UDP

- Con UDP **no** se establece “conexión” entre cliente y servidor
  - El emisor indica explícitamente la dirección IP y el puerto del origen y del destino **en cada datagrama**
  - El receptor ha de extraer del datagrama recibido la dirección IP y el puerto del emisor
- Los datos transmitidos pueden llegar fuera de orden o incluso perderse

Datagrama UDP

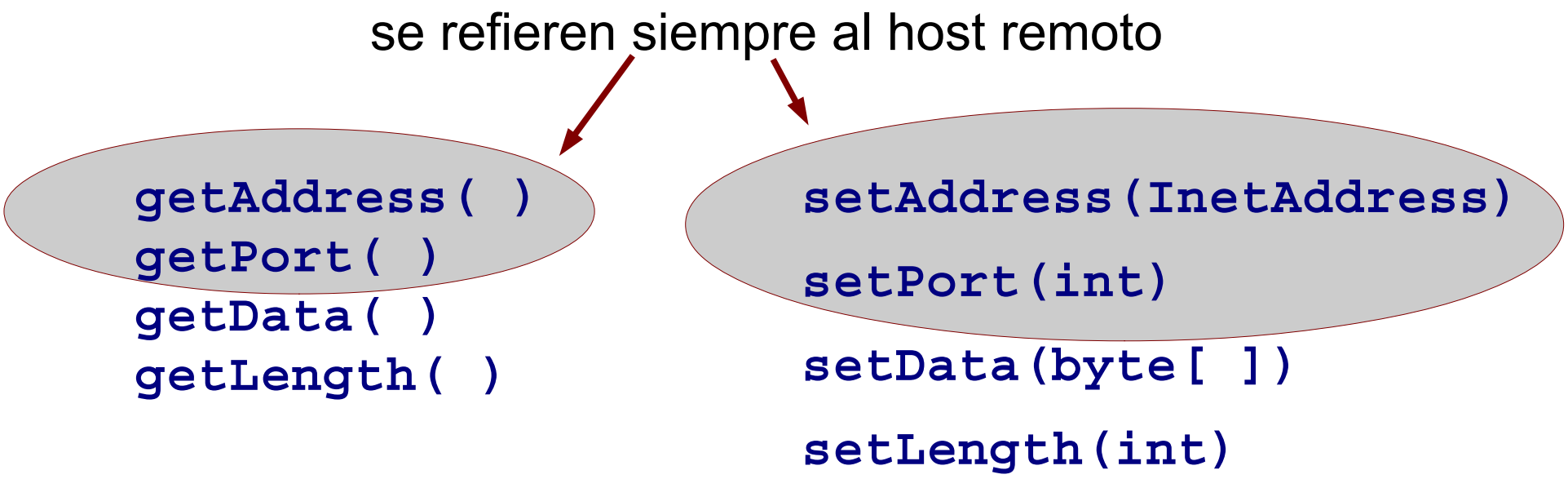
Clase `DatagramPacket`

- **Constructores**
  - **`DatagramPacket(byte buf[ ], int longitud)`**
    - Crea un datagrama UDP de esa longitud para recibir
  - **`DatagramPacket(byte buf[ ], int longitud, InetAddress dirIP, int puerto)`**
    - Crea un datagrama UDP con ese buffer y de esa longitud para enviarlo a la dirección IP y puerto que se indican

Datagrama UDP

Clase `DatagramPacket`

- **Algunos métodos importantes**



Socket UDP

Clase `DatagramSocket`

- **Constructores**
  - `DatagramSocket()`
    - Crea un socket UDP que escucha en un puerto libre
  - `DatagramSocket(int puerto)`
    - Crea un socket UDP que escucha en ese puerto

Socket UDP

Clase `DatagramSocket`

- **Algunos métodos importantes**
  - **`send(DatagramPacket p)`**
    - Envía un datagrama
    - El **`DatagramPacket`** incluye los datos a enviar, su longitud y la dirección IP y el puerto del destino
  - **`receive(DatagramPacket p)`**
    - Recibe datagramas. El método es bloqueante
    - Cuando el método retorna, el buffer **`DatagramPacket`** contiene los datos recibidos, la dirección IP y el puerto de quien envía el datagrama
  - **`close()`**

Cliente UDP

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

public class ClienteUDP{
  public static void main(String[] args) throws IOException {
    DatagramSocket s = new DatagramSocket();
    InetAddress dir = InetAddress.getByName("zoltar.redes.upv.es");
    String msg = "Hola, esto es un mensaje\n";
    byte[] buf = new byte[256];
    buf = msg.getBytes();
    DatagramPacket p = new DatagramPacket(buf, buf.length, dir, 7777);
    s.send(p);
    s.receive(p); // se bloquea hasta que recibe un datagrama
    System.out.write(p.getData());
    s.close();
  }
}
```

- El cliente envía un datagrama a un servidor y muestra la respuesta por pantalla

Servidor UDP

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

public class ServidorUDP{
  public static void main(String[] args) throws IOException {
    DatagramSocket s = new DatagramSocket(7777);
    DatagramPacket p = new DatagramPacket(new byte[256], 256);
    s.receive(p); // se bloquea hasta que recibe un datagrama
    p.setAddress(p.getAddress());
    p.setPort(p.getPort());
    s.send(p);
    s.close();
  }
}
```

- Envía de vuelta el datagrama recibido, sin modificarlo, a la dirección IP y puerto de origen
- Sólo procesa un cliente y acaba

Servidores concurrentes

- Normalmente, un servidor debe estar preparado para atender muchos clientes
- Se puede hacer de dos maneras:
  - **Secuencial**: un cliente detrás de otro
  - **Concurrente**: varios clientes al mismo tiempo

Threads en Java

En Java, la concurrencia la conseguimos usando **hilos de ejecución**

Clase `Thread`

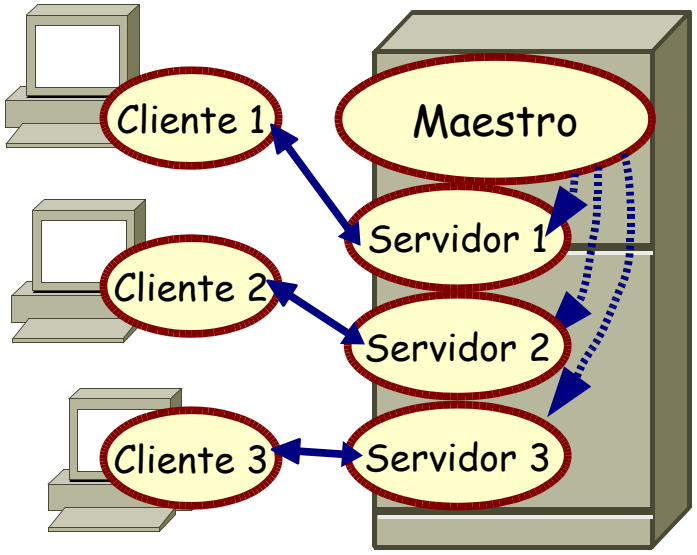
- Se define una clase derivada de **`Thread`**
- Código a ejecutar en cada hilo dentro del método **`run()`**
- Se lanza el hilo con **`start()`**

Ejemplo de uso:

```
class Hilos extends Thread {
 int id;
 public Hilos(int i) {id=i;}
 public void run() {
  for(int i=0;i<100;i++)  {
   System.out.print(id);
   try {sleep(100);}
   catch(InterruptedException e) {}
  }
 }
 public static void main(String args[]) {
  for(int i=0;i<3;i++) new Hilos(i).start();
 }
}
```

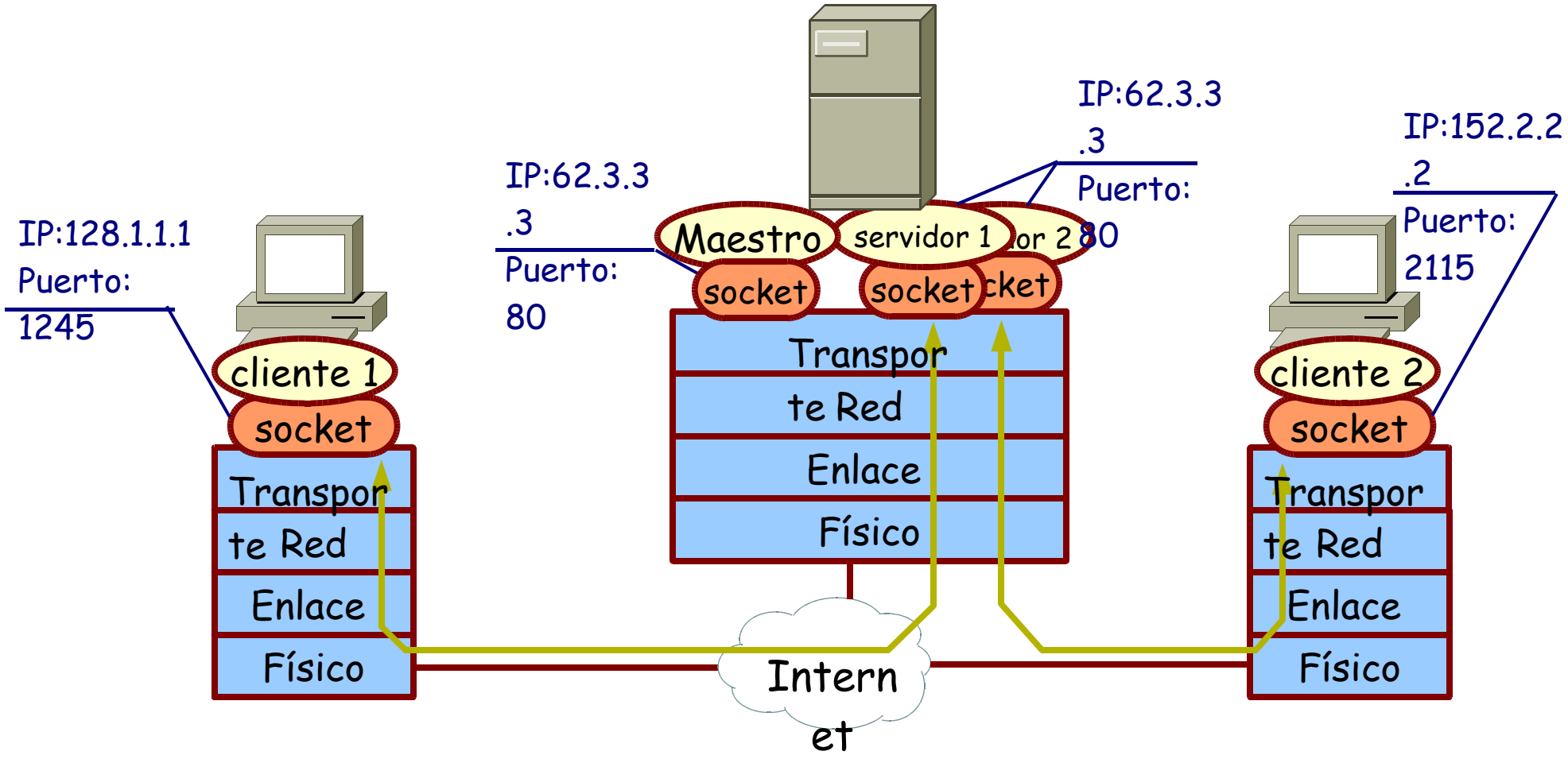
Servidores concurrentes (II)

- Diversos hilos de ejecución:
  - En el hilo principal se ejecuta permanentemente el método `accept()`
    - Espera el establecimiento de nuevas conexiones
  - Para cada cliente que se conecta, se lanza un nuevo hilo de ejecución para gestionar esa conexión



Identificación de los sockets

- Ahora tenemos varios sockets asociados al mismo puerto
- Para identificar al socket destino hay que tener en cuenta la dirección (dir. IP + puerto) del socket origen



Servidor concurrente TCP

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

class SCTCP extends Thread {
 Socket id;
 public SCTCP(Socket s) {id=s;}
 public void run() {
  try {
   PrintWriter salida=new PrintWriter(id.getOutputStream(),true);
   while(true){
   salida.println(System.currentTimeMillis());
   sleep(100);
   }
  } catch(Exception e) {}
 }

 public static void main(String args[]) throws IOException{
  ServerSocket ss=new ServerSocket(8888);
  while(true) {
    Socket s = ss.accept();
    SCTCP t = new SCTCP(s);
    t.start();
   }
 }
}
```

Más ejemplos

Micro-servidor web iterativo

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

class ServidorWeb {
public static void main(String args[]) throws Exception{
 byte[] buffer = new byte[1024];  int bytes;
 ServerSocket ss=new ServerSocket(7777);
 while(true) {
  Socket s=ss.accept(); // espero a que llegue un cliente
  BufferedReader in=new BufferedReader(new
                                InputStreamReader(s.getInputStream()));
  PrintWriter out=new PrintWriter(s.getOutputStream(),true);
  StringTokenizer tokens = new StringTokenizer(in.readLine());
  tokens.nextToken();  // esto debe ser el "GET"
  String archivo = "."+tokens.nextToken(); // esto es el archivo
  FileInputStream fis = null; boolean existe = true;
  try {fis = new FileInputStream(archivo);} // comprobamos si existe
  catch (FileNotFoundException e) {existe = false;}
  if (existe && archivo.length()>2)
    while((bytes = fis.read(buffer)) != -1 ) // enviar archivo solicitado
      s.getOutputStream().write(buffer, 0, bytes);
  else out.println("<html><body><h1>404 Not Found</h1></body></html>");
  s.close();
  }
 }
}
```

Acceso a bajo nivel

- Framboos
  - Tiene básicamente funciones para la GPIO y comunicaciones (UART)
  - Si necesitas PWM, SPI o I2C → Pi4J
  - https://github.com/jkransen/framboos

  - Ejemplo:

```
import framboos.InPin;
import framboos.OutPin;
InPin pulsador = new InPin(8);
boolean estaPulsado = pulsador.getValue();
OutPin led = new Outpin(0);
led.setValue(true); //enciendo el led
```

Acceso a bajo nivel

- Pi4J
  - basado en WiringPi
  - http://pi4j.com/
- Instalación:
  - Obtener el paquete. Desde consola de la Raspberry Pi:
    - `wget http://pi4j.googlecode.com/files/pi4j-0.0.5.deb`
  - Instalar:
    - `sudo dpkg -i pi4j-0.0.5.deb`
    - Instalará las librerías Pi4J y ejemplos en: /opt/pi4j/lib y /opt/pi4j/examples
- Al compilar incluir “Pi4J lib” en el “classpath”:
  - javac -classpath .:classes:/opt/pi4j/lib/'*' ...
- Al ejecutar el programa incluir “Pi4J lib” en el “classpath”:
  - sudo java -classpath .:classes:/opt/pi4j/lib/'*' ...

Acceso a bajo nivel

- Ejemplo:

```
import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioPinDigitalInput;
import com.pi4j.io.gpio.GpioPinDigitalOutput;

// creamos una instancia del controlador de gpio
final GpioController gpio = GpioFactory.getInstance();

// configuramos el pin 2 como entrada con su resistencia interna de pull down habilitada
GpioPinDigitalInput myPulsador = gpio.provisionDigitalInputPin(RaspiPin.GPIO_02,           // PIN #
                                                               "MiPulsador",               // PIN FRIENDLY NAME (opcional)
                                                               PinPullResistance.PULL_DOWN); // PIN RESISTANCE (opcional)


// configuramos el pin 4 como salida e indicamos que tras la inicialización debe estar a nivel bajo
GpioPinDigitalOutput miLed = gpio.provisionDigitalOutputPin(RaspiPin.GPIO_04,   // PIN #
                                                            "Mi LED",           // PIN FRIENDLY NAME (opcional)
                                                            PinState.LOW);      // PIN STARTUP STATE (opcional)

miLed.setState(PinState.HIGH);
miLed.low();
miLed.high();
miLed.toggle();
miLed.pulse(1000);

PinState estadoPulsador = miPulsador.getState();

boolean estaPulsado = miPulsador.isHigh();
```

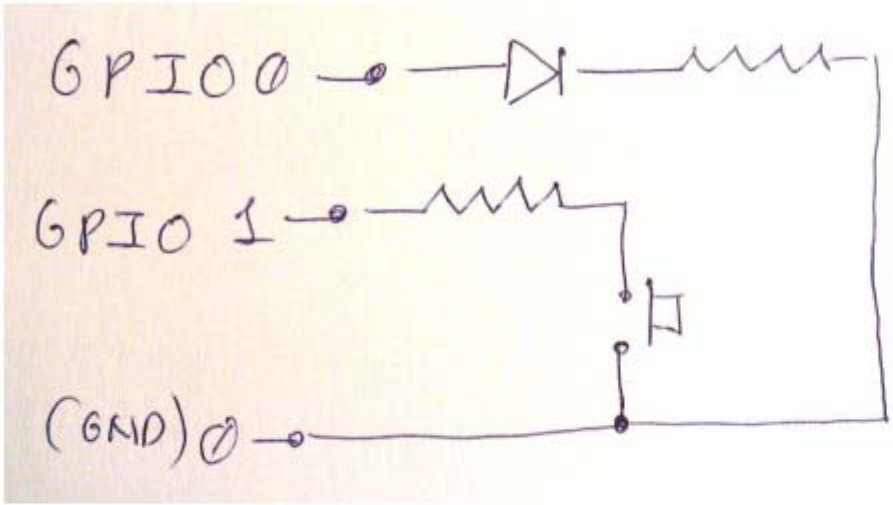
Acceso a bajo nivel

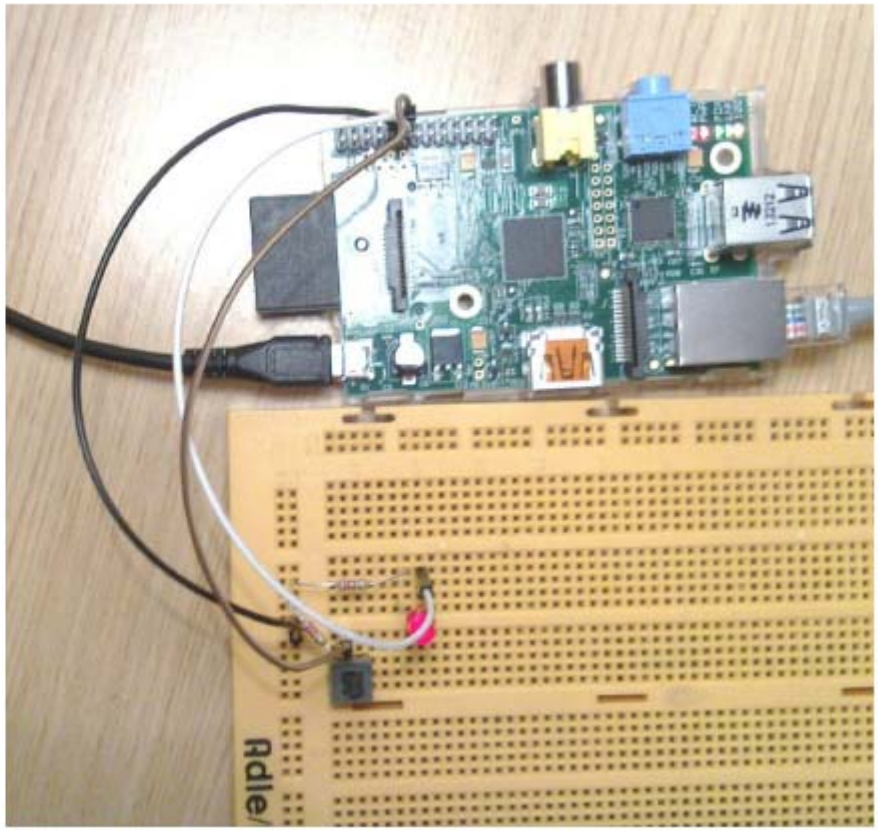
- Obteniendo información de sistema:

```
import java.io.IOException;
import java.text.ParseException;

import com.pi4j.system.SystemInfo;

// Ejemplo obtener informacion de sistema en la Raspberry Pi

public class SystemInfoExample {

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

        System.out.println("Serial Number     :  " + SystemInfo.getSerial());
        System.out.println("CPU Revision      :  " + SystemInfo.getCpuRevision());
        System.out.println("CPU Temperature   :  " + SystemInfo.getCpuTemperature());
        System.out.println("CPU Core Voltage  :  " + SystemInfo.getCpuVoltage());
         }
}
```

Un caso práctico: montaje

- Apagar la RPi, quitar la alimentación, montar con cuidado.





Programando GUIs con Java

- En programas en consola (basados en texto):
- el programa controla el flujo de la ejecución
- Los programas con GUI son “event-driven”
  - la secuencia de eventos controla el flujo de la ejecución

- Java proporciona las siguientes librerías para la creación de GUIs:
  - Java AWT (Abstract Window Toolkit)
  - Java Foundation Classes (JFC o Swing), a partir de Java2

- Pasos para crear una aplicación con GUI usando Tkinter:

  - Importar los paquetes que necesitemos
  - Crear la ventana principal de la aplicación
  - Añadir uno o más elementos gráficos.
  - Crear una instancia de la clase creada

Programando GUIs con Java

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JOptionPane;

public class EjGUI {
   //Declaramos los componentes que usaremos:
   JFrame ventana;
   JLabel etiqueta;
   JTextField campo;
   JButton boton;

   //constructor de la clase
   public EjGUI(){
         //Instanciamos los componentes que necesitamos:
         ventana = new JFrame("Ejemplo GUI #3");
         etiqueta = new JLabel("Etiqueta de ejemplo #3");
         campo = new JTextField(10);
         boton = new JButton("Botón De Ejemplo #3");
         //Configuramos los componentes:
           //Ponemos una acción de cerrado por default (salir en este caso):
           ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           //Ponemos el layout:
           ventana.setLayout(new FlowLayout());
           //Añadimos los complentes a la ventana:
           ventana.add(etiqueta);
           ventana.add(campo); //10 = Largo del campo
           ventana.add(boton);
           //Se llama a pack después de haber agregado componenetes a la ventana
           ventana.pack();
```

Programando GUIs con Java

```
            //Asociamos manejadores a eventos:
            boton.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e) {
                 accionesBoton();
                }});
              //Mostramos la ventana:
              ventana.setVisible(true);

} //del constructor

// Método manejador del boton:
private void accionesBoton(){
              JOptionPane.showMessageDialog(null, "Has hecho click en el boton");
            }

public void llenarCampo(String texto){
              campo.setText(texto);
             }

public static void main(String[] args){
            //Llamamos a una nueva instancia de la clase y a un método en la misma:
              new EjGUI().llenarCampo("Hola a todos!");
            }

}
```

IDEs

- Eclipse
- NetBeans
- BlueJ
- Jdeveloper
- Etc.

- sudo shutdown -h now