

# API de comunicaciones

El API de comunicaciones es una extensión standard que nos permite realizar comunicaciones con los puertos serie RS-232 y el paralelo IEEE-1284, esto nos permitirá realizar aplicaciones de comunicaciones que utilizan los puertos de comunicaciones (tarjetas inteligentes, fax) independientes de la plataforma.

El API de comunicaciones no se encuentra incluido en el JDK y es una extensión de este, así que antes de empezar a trabajar deberemos instalar este nuevo API en las maquinas que vayamos a realizar el desarrollo y que vayan a ejecutar programas realizados con este API.

## Instalación del API de comunicaciones

Lo primero que haremos es obtener el API de comunicaciones, este se puede bajar fácilmente de Internet ya que no ocupa más de 300 Kb.

Una vez que desempaquemos el fichero procederemos a realizar los siguientes pasos:

Copiamos el fichero Win32com.dll a `\jre\bin` En el caso UNIX la librerías se deberían instalar en donde nos indicara la variable de entorno. Por ejemplo en una maquina Sun será donde nos marque la variable `LD_LIBRARY_PATH`. Copiamos el archivo `comm.jar` a `\jre\lib\ext`. Copiamos `javax.comm.properties` a `\jre\lib` este fichero contendrá los driver que soportara el API, esto es así ya que podremos creamos nuestros propios drivers de puertos.

La nomenclatura indica el directorio donde se ha instalado el paquete de desarrollo de Java.

## Características del API de comunicaciones

En el paquete de comunicaciones `javax.comm` tenemos una serie de clases que nos permiten tratar varios niveles de programación, estos niveles son los siguientes:

- Nivel alto: En este nivel tenemos las clases `CommPortIdentifier` y `CommPort` que nos permiten el acceso a los puertos de comunicación.
- Nivel medio: Con las clases `SerialPort` y `ParallelPort` que cubren el interfaces físico RS-232 para el puerto serie y IEEE-1284 para el puerto paralelo.
- Nivel bajo: Este nivel ya toca el sistema operativo y en el se encuentra el desarrollo de drivers.

Los servicios que nos proporciona este paquete son:

- Poder obtener los puertos disponibles así como sus características.
- Abrir y mantener una comunicación en los puertos.
- Resolver colisiones entre aplicaciones. Gracias a este servicio podremos tener varias aplicaciones Java funcionando y
- utilizando los mismos puertos y no solo sabremos que el puerto esta ocupado sino que podremos saber que aplicación lo esta utilizando.

Disponemos de métodos para el control de los puertos de entrada/salida a bajo nivel, de esta forma no solo nos limitamos a enviar y recibir datos sino que podemos saber en que estado

esta el puerto. Así en un puerto serie podremos no solo cambiar los estados sino que podemos programar un evento que nos notifique el cambio de cualquier estado.

## Programación a alto nivel

Para esta programación contamos con la clase CommPortIdentifier. Nos encontramos ante la clase principal del paquete ya que lo primero que debemos hacer antes de empezar a utilizar un puerto es descubrir si esta libre para su utilización. El primer programa que nos servirá como ejemplo lo encontramos en el Listado 1, este programa nos enseña la disponibilidad de los puertos. En la Figura A podemos ver el resultado del mismo, el puerto COM2 se encuentra ocupado por el programa "Una aplicación ocupa".

```
/*
 * P1.java
 *
 * Descripción: Este programa nos permite analizar la disponibilidad de
 *             todos los puertos soportados en nuestra maquina. Tambien
 *             nos informara del tipo del puerto así como el propietario
 *             de este en caso de que se encuentre ocupado.
 *
 * Autor: Fco. Javier Rodriguez Navarro (c) Junio 2000
 */
import java.io.*;
import java.util.*;
import javax.comm.*;
public class P1
{
    static Enumeration listaPort;
    static CommPortIdentifier idPort;

    public static void main(String[] args)
    {
        listaPort = CommPortIdentifier.getPortIdentifiers();
        ListaPuertos();
    }

    private static void ListaPuertos()
    {
        System.out.println("Los puertos disponibles son:");
        while (listaPort.hasMoreElements())
        {
            idPort = (CommPortIdentifier) listaPort.nextElement();
            System.out.print("PUERTO: " + idPort.getName() + " ");

            if (idPort.getPortType() == CommPortIdentifier.PORT_SERIAL)
            {
                System.out.println("RS-232 (" + idPort.getPortType() + ")");
            } else if (idPort.getPortType() ==
CommPortIdentifier.PORT_PARALLEL)
            {
                System.out.println("IEEE 1284 (" + idPort.getPortType() +
")");
            } else System.out.println("Tipo de puerto desconocido");

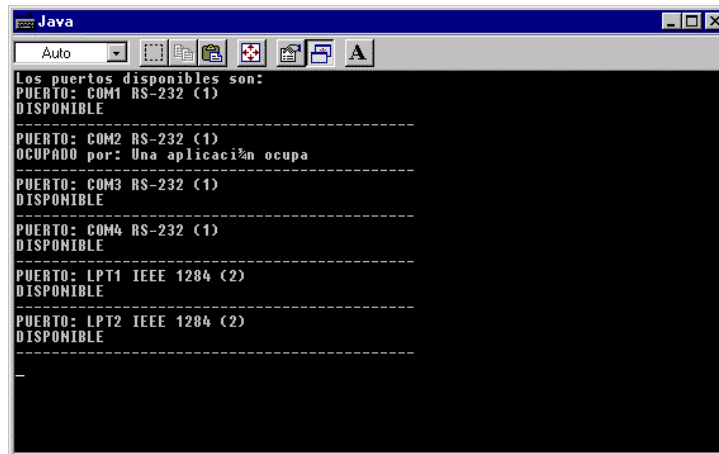
            // Describimos si esta disponible.
            if (idPort.isCurrentlyOwned())
```

```

        System.out.println("OCUPADO por: " +
                           idPort.getCurrentOwner());
    else
        System.out.println("DISPONIBLE");

    System.out.println("-----
--");
    }
}
}

```



Lo primero que debemos hacer antes de intentar abrir un puerto será ver si ya tiene o no un propietario y para obtener la información de esto debemos obtener el objeto de `CommPortIdentifier` correspondiente al puerto que se realizara con alguno de los siguientes métodos:

- `getPortIdentifiers()` es el método utilizado en nuestro programa y que nos entregará un enumerado con tantos objetos `CommPortIdentifier` como puertos dispongamos.
- `getPortIdentifier(String)` Nos dará el objeto correspondiente al puerto que se le pase como parámetro, este será el método que normalmente usaremos ya que lo normal es que siempre nos conectemos por el mismo puerto. Este método deberá saber tratar la excepción `NoSuchPortException` que saltara en el caso de que solicitemos un puerto inexistente. Una vez que tenemos el objeto del puerto tenemos una serie de métodos que nos permitirán obtener información del puerto y abrirlo para poder iniciar una comunicación. Estos métodos son los siguientes: `getName()` dará el nombre del puerto. En el caso de haber utilizado el método `getPortIdentifier(String)` será el mismo valor que pasamos como parámetro.
- `getPortType()` devuelve un entero que nos informa del tipo de puerto (serie o paralelo), para no tener que acordarnos del valor de cada tipo de puerto disponemos de las constantes `PORT_PARALLEL` y `PORT_SERIAL`.
- `isCurrentlyOwned()` nos informa si esta libre o no el puerto. En caso de que este ocupado podremos saber quien lo esta utilizando mediante el método `getCurrentOwner()`.
- `open(String, int)` abre y por lo tanto reserva un puerto, en el caso de que intentemos abrir un puerto que ya se encuentre en uso saltara la excepción `PortInUseException`. Los parámetros que le debemos pasar son un `String` con el nombre de la aplicación que reserva el puerto y un `int` que indica el tiempo de espera para abrir el puerto. Un ejemplo de este método se puede ver en el Listado 2. Este método nos da un objeto `CommPort`, realmente esto es una clase abstracta de la que heredan las clases

SerialPort y ParallelPort. Una vez que se tiene este objeto se deberán encaminar sus salidas a OutputStream y las entradas a InputStream una vez realizado esto la escritura y lectura de los puertos serán tan fácil como utilizar los métodos de estas clases que pertenecen al standard del JDK.

- addPortOwnershipListener(CommPortOwnershipListener) permite añadir una clase que escuche los cambios de propietarios en los puertos.

Un ejemplo de esto esta en el Listado 2, en la Figura B podemos ver un prototipo de clase que realiza la escucha de cambio de propietarios en el puerto, como se puede ver son tres valores tipos: abrir un puerto, cerrar un puerto e intentar abrir un puerto en uso. Si una vez que tenemos registrado un oyente de eventos lo queremos eliminar deberemos de utilizar removePortOwnershipListener(CommPortOwnershipListener).

```
class evProp implements CommPortOwnershipListener
{
    public void ownershipChange(int tipo)
    {
        System.out.print("Valor: " + tipo);
        if (tipo == CommPortOwnershipListener.PORT_OWNED)
            System.out.println(" Se abre el puerto");
        else if (tipo == CommPortOwnershipListener.PORT_UNOWNED)
            System.out.println(" Se cierra el puerto");
        else if (tipo == CommPortOwnershipListener.PORT_OWNERSHIP_REQUESTED)
            System.out.println(" Requerido puerto usado");
    }
}

/*****
 * Escritor.java
 *
 * Descripción: Este programa escribe una frase por el puerto
 *              que le pasemos como parametro.
 *
 * Autor: Fco. Javier Rodriguez Navarro (c) Junio 2000
 *****/
import java.io.*;
import java.util.*;
import javax.comm.*;

public class Escritor
{
    static CommPortIdentifier idPort;
    static SerialPort sPort;
    static ParallelPort pPort;
    static OutputStream salida;
    static String datos = new String("HOLA! esto es una prueba");
    static evProp ev = new evProp();

    public static void main(String[] args)
    {
        // Lo primero que hacemos es abrir el puerto
        try
        {
            idPort = CommPortIdentifier.getPortIdentifier(args[0]);
            idPort.addPortOwnershipListener(ev);
        } catch (NoSuchPortException e)
        {
            System.err.println("ERROR al identificar puerto");
        }

        // Abre el puerto necesario.
        try
        {
```

```

        if (idPort.getPortType() == CommPortIdentifier.PORT_SERIAL)
        {
sPort = (SerialPort) idPort.open("Escritor en serie", 30000);
            try
            {
                salida = sPort.getOutputStream();
            } catch (IOException e) {}
        }
        else
        {
pPort = (ParallelPort) idPort.open("Escritor en paralelo", 30000);
            try
            {
                salida = pPort.getOutputStream();
            } catch (IOException e) { }
        }
    } catch (PortInUseException e)
        { System.err.println("ERROR al abrir puerto");}

        try
        {
            salida.write(datos.getBytes());
        } catch (IOException e) { System.err.println("Error escritura"); }

    }
}

class evProp implements CommPortOwnershipListener
{
    public void ownershipChange(int tipo)
    {
        System.out.print("Valor: " + tipo);
        if (tipo == CommPortOwnershipListener.PORT_OWNED)
            System.out.println(" Se abre el puerto");
        else if (tipo == CommPortOwnershipListener.PORT_UNOWNED)
            System.out.println(" Se cierra el puerto");
        else if (tipo == CommPortOwnershipListener.PORT_OWNERSHIP_REQUESTED)
            System.out.println(" Requerido puerto usado");
    }
}

```

## Clase CommPort

Esta es una clase abstracta que describe los métodos comunes de comunicación y serán las clases que hereden de ellas (SerialPort y ParallelPort) la que añadan métodos y variables propias del tipo del puerto. Entre los métodos que nos interesa conocer tenemos:

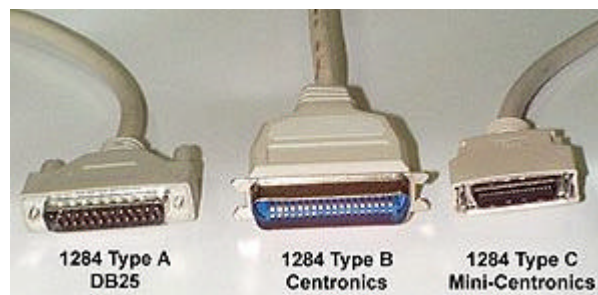
- close() nos permitirá liberar el puerto que se reservo con open, este notificara el cambio de dueño a las clases que se hubiesen registrado con el método addPortOwnershipListener que se explico anteriormente.
- getInputStream() nos permitirá enlazar la entrada del puerto al InputStream que nos devuelve para leer del puerto.
- getOutputStream() nos permitirá enlazar la salida del puerto al OutputStream que nos devuelve para poder escribir en el puerto de la misma forma que si se escribiera en un fichero.

- `getInputBufferSize()` nos informa del tamaño que tiene el buffer de entrada del puerto. Este tamaño se puede modificar con el método `setInputBufferSize(int)`. Estos métodos no siempre dan el resultado esperado ya que será la memoria disponible y el sistema operativo quien al final decida si realizar o no correctamente la operación. `getOutputBufferSize()` nos informa del tamaño que tiene el buffer de salida, como en el caso anterior contamos con un método para modificar el tamaño que es `setOutputBufferSize(int)`. Al igual que pasaba con los métodos anteriores su correcto funcionamiento dependen del sistema operativo en si y del desarrollo del driver.

Una vez que hemos visto los principales métodos de la clase abstracta de la que heredan (`ParallelPort` y `SerialPort`) pasamos a ver con detenimiento las características de estas.

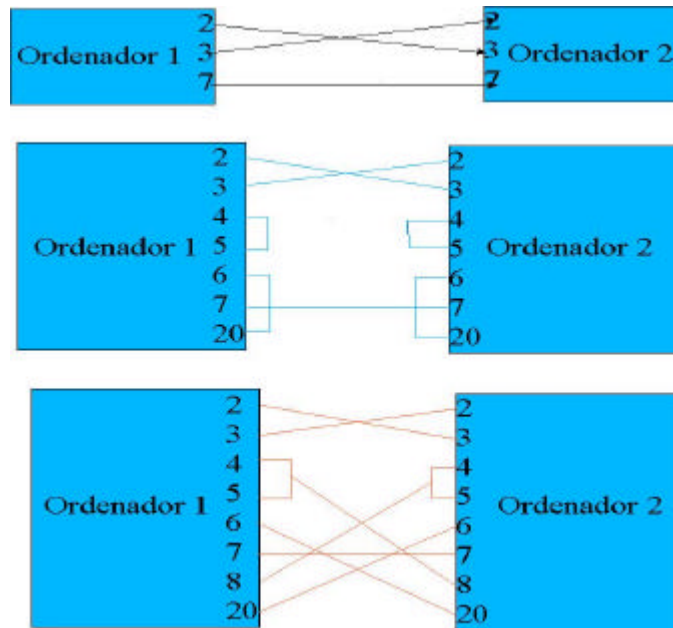
## Clase `SerialPort`

En esta clase encontramos el interfaces de bajo nivel del puerto paralelo que cumple el standard RS-232. En la Tabla A podemos ver la descripción de las patillas del conector de 25 contactos, ver la Figura C conector tipo A, y su equivalente con el de 9 contactos. En la emisión de un carácter se realizarán las siguientes comprobaciones:



1. Ponemos el DTR (Indicando que estamos preparados)
2. Ponemos RTS (Solicito permiso para emitir)
3. Comprueba que esta DSR (Mira si el destinatario esta listo)
4. Esperamos a CTS (Se espera a que me autoricen a enviar datos)
5. Enviamos datos.

En la figura D podemos ver distintas conexiones que nos podemos encontrar en los enlaces punto a punto.



### Conexiones DB25 y DB9 del interfaces RS-232

PIN (DB25)	PIN (DB9)	Descripción
2	2	TD: Canal de transmisión.
3	3	RD: Canal de recepción.
4	7	RTS: Petición para transmitir.
5	8	CTS: Preparado para emitir.
6	6	DSR: Equipo preparado.
7	5	Masa
8	1	DCD: Conexión establecida.
20	4	DTR: Terminal preparado.
22	9	RI: Indicador de llamada.

La clase SerialPort hereda de la clase abstracta CommPort y por lo tanto cuenta con sus métodos pero además de estos dispone de otros métodos y variables específicas para el tratamiento de los puertos serie. En el Listado 3 podemos ver el uso de los métodos y eventos que se exponen a continuación

1. `setSerialPortParams(int, int, int, int)` nos permite configurar los parámetros del puerto serie, este método debería tratar la excepción `UnsupportedCommOperationException` que saltara en el caso de que le introduzcamos valores no soportados. Los parámetros del método son:
  - o La velocidad en el caso de que le pasáramos un valor no válido. Si pasáramos 1210 se produciría la excepción y no se modificarían los datos.
  - o El segundo parámetro son los bits de datos. Para indicar el valor utilizaremos las constantes de la clase que se tiene para lo mismo (`DATABITS_5`, `DATABITS_6`, `DATABITS_7` o `DATABITS_8`).
  - o El bit o bits de stop que utilizaremos y que puede ser 1,2 o uno y medio. Las constantes que definen estas configuraciones son: `STOPBITS_1`, `STOPBITS_2` y `STOPBITS_1_5` respectivamente.
  - o Paridad que utilizaremos y que puede ser `PARITY_NONE` en el caso de no utilizar paridad, `PARITY_ODD` para la paridad impar, `PARITY_EVEN` paridad par, `PARITY_MARK` paridad por marca y `PARITY_SPACE` paridad por espacio.

2. `getBaudrate()` nos da la velocidad a la que esta preparada para transmitir y que se puede cambiar con el método `setSerialPortParams(int, int, int, int)`.
3. `getDataBits()` nos da la configuración de número de datos y al igual que el método anterior se puede cambiar con el mismo método, los valores posibles son los mismos que utiliza el método de configuración de parámetros. `getStopBits()` nos indica la configuración de bits de parada y al igual que en los métodos anteriores se puede configurar con el mismo método.
4. `getParity()` indica la paridad que utiliza.
5. `getFlowControlMode()` nos da la configuración del control de flujo que se puede cambiar con el método `setFlowControlMode(int)` y los valores posibles son:  
FLOWCONTROL\_NONE no existe control de flujo,  
(FLOWCONTROL\_RTSDTS\_IN o FLOWCONTROL\_RTSDTS\_OUT) para el control de flujo por hardware y ( FLOWCONTROL\_XONXOFF\_IN o FLOWCONTROL\_XONXOFF\_OUT) para control de flujo por software. El método `setFlowControlMode(int)` al igual que pasaba con `setSerialPortParams(int, int, int, int)` deberá de saber capturar la excepción `UnsupportedCommOperationException`.
6. `addEventListener(SerialPortEventListener)` nos permite escuchar los cambios de estado del puerto, si quisiéramos quitar este oyente utilizaríamos el método `removeEventListener()`. Mediante los métodos `notifyOnXXX(boolean)` nos permitirá habilitar o deshabilitar la notificación de los diferentes cambios que pueden ser:
  - o `DataAvailable` existen datos en la entrada.
  - o `OutputEmpty` el buffer de salida esta vacio.
  - o `CTS` cambio de Clear To Send
  - o `DSR` cambio de Data Set Ready
  - o `RingIndicator` cambio en RI.
  - o `CarrierDetect` cambio en CD.
  - o `ParityError` se ha producido un error de paridad
  - o `BreakInterrupt` se detecta una rotura en la linea.

La clase oyente deberá de tener el método `serialEvent(SerialPortEvent)` que recibirá un objeto que trata los eventos del puerto serie con tres métodos importante:

- `getEventType()` nos informara del evento que se ha producido.
- `getNewValue()` que devuelve el nuevo estado.
- `getoldValue()` que nos dará el estado anterior al cambio.
- `isDTR()` nos informara del estado terminal que se podra cambiar con `setDTR(boolean)`.
- `isRTS()` nos dice si ha solicitado permiso para transmitir o no y para cambiar su estado tenemos el método `setRTS(boolean)`,
- `isCTS()` nos informa si podemos transmitir, `isDSR()` dará el estado en el que se encuentra el pin DSR. `isRI()` informa de si esta el indicador de timbre, `isCD()` nos indicara si tenemos portadora.

```
/*
 * Serie.java
 * Descripción: Ejemplo de información de puerto con
 *              tratamiento de eventos.
 * Autor: Fco. Javier Rodriguez Navarro (c) Julio 2000
 */
import java.io.*;
import java.util.*;
```

```

import javax.comm.*;

public class Serie
{
    static CommPortIdentifier idPort;
    static SerialPort sPort;
    static OutputStream salida;
    static String datos = new String("Una linea que deseamos escribir ");
    static evSerie ev = new evSerie();

    public static void main(String[] args)
    {
        try
        {
            idPort = CommPortIdentifier.getPortIdentifier("COM2");
        } catch (NoSuchPortException e)
        { System.err.println("ERROR al identificar puerto");}

        try
        {
            sPort = (SerialPort) idPort.open("Observador1(COM2)", 30000);
            informa();
            try
            {
                salida = sPort.getOutputStream();
            } catch (IOException e) { }
            try
            {
                sPort.addEventListener(ev);
                sPort.notifyOnDataAvailable(true);
                sPort.notifyOnOutputEmpty(true);
                sPort.notifyOnCTS(true);
                sPort.notifyOnDSR(true);
                sPort.notifyOnRingIndicator(true);
                sPort.notifyOnCarrierDetect(true);
                sPort.notifyOnParityError(true);
                sPort.notifyOnOverrunError(true);
                sPort.notifyOnBreakInterrupt(true);
            } catch (TooManyListenersException e) { }
        } catch (PortInUseException e)
        { System.err.println("ERROR al abrir puerto");}

        try
        {
            while (true) {salida.write(datos.getBytes());}

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

    // Metodo informa: Nos dara información del estado del puerto
    private static void informa()
    {
        System.out.println("Información del puerto: " + sPort.getName());
        System.out.println("Buffer entrada: " + sPort.getInputBufferSize());
        System.out.println("Buffer salida: " + sPort.getOutputBufferSize());
        System.out.println("Baudios: " + sPort.getBaudRate());
        System.out.print("Bits de datos: ");
        switch (sPort.getDataBits())
        {
            case SerialPort.DATABITS_5:
                System.out.println("cinco ");
                break;
        }
    }
}

```

```
        case SerialPort.DATABITS_6:
            System.out.println("seis ");
            break;
        case SerialPort.DATABITS_7:
            System.out.println("siete ");
            break;
        case SerialPort.DATABITS_8:
            System.out.println("ocho ");
            break;
    }

    System.out.print("Bits de parada: ");
    switch (sPort.getStopBits())
    {
        case SerialPort.STOPBITS_1:
            System.out.println("uno ");
            break;
        case SerialPort.STOPBITS_2:
            System.out.println("dos ");
            break;
        case SerialPort.STOPBITS_1_5:
            System.out.println("uno y medio ");
            break;
    }

    System.out.print("Paridad: ");
    switch (sPort.getParity())
    {
        case SerialPort.PARITY_NONE:
            System.out.println("ninguna ");
            break;
        case SerialPort.PARITY_ODD:
            System.out.println("impar ");
            break;
        case SerialPort.PARITY_EVEN:
            System.out.println("par ");
            break;
        case SerialPort.PARITY_MARK:
            System.out.println("por marca ");
            break;
        case SerialPort.PARITY_SPACE:
            System.out.println("por espacio ");
            break;
    }

    System.out.print("Control de flujo: ");
    switch (sPort.getFlowControlMode())
    {
        case SerialPort.FLOWCONTROL_NONE:
            System.out.println("ninguno ");
            break;
        case SerialPort.FLOWCONTROL_RTSCCTS_IN:
        case SerialPort.FLOWCONTROL_RTSCCTS_OUT:
            System.out.println("hardware ");
            break;
        case SerialPort.FLOWCONTROL_XONXOFF_IN:
        case SerialPort.FLOWCONTROL_XONXOFF_OUT:
            System.out.println("software ");
            break;
    }
}

}

class evSerie implements SerialPortEventListener
{
```

```

public void serialEvent(SerialPortEvent evento)
{
    switch (evento.getEventType())
    {
        case SerialPortEvent.DATA_AVAILABLE:
            System.out.println("Datos disponibles");
            break;
        case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
            System.out.println("Buffer vacio");
            break;
        case SerialPortEvent.CTS:
            System.out.print("CTS pasa a ");
            if (evento.getNewValue()) System.out.println("cierto");
            else System.out.println("falso");
            break;
        case SerialPortEvent.DSR:
            System.out.print("DSR pasa a ");
            if (evento.getNewValue()) System.out.println("cierto");
            else System.out.println("falso");
            break;
        case SerialPortEvent.RI:
            System.out.print("RI pasa a ");
            if (evento.getNewValue()) System.out.println("cierto");
            else System.out.println("falso");
            break;
        case SerialPortEvent.CD:
            System.out.print("CD pasa a ");
            if (evento.getNewValue()) System.out.println("cierto");
            else System.out.println("falso");
            break;
    }
}

```

## Clase ParallelPort

En esta clase tenemos el interfaces de bajo nivel del puerto paralelo que cumple la norma IEEE-1284. El standard IEEE-1284 cuenta con 8 líneas de datos, 5 entrada de estado y 4 salidas de control, en la figura C se pueden ver los tres tipos de conectores que podemos encontrar en este tipo de interfaces. Los contactos del conector tipo A es el más utilizado en los ordenadores y se definen en la Tabla B con sus funciones en el modo SPP que es el modo compatible de trabajo para la impresora.

### Conexiones DB25 del interfaces IEEE-1284 en modo SPP

PIN	NOMBRE	Descripción
1	STROBE	Indica que existen datos validos en la lineas de datos D0-7
2	D0	Dato bit 0
3	D1	Dato bit 1
4	D2	Dato bit 2
5	D3	Dato bit 3
6	D4	Dato bit 4
7	D5	Dato bit 5
8	D6	Dato bit 6
9	D7	Dato bit 7
10	ACK*	Indica que el ultimo carácter se recibió.
11	BUSY*	Indica que la impresora esta ocupada y no puede recoger datos.

```
12 PE* Sin papel.
13 SLCT* Indica que la impresora esta en linea.
14 AUTO FD Indica a la impresora que realice una alimentaci3n de linea.
15 ERROR* Existe alg3n error en la impresora.
16 INIT* Pide a la impresora que se inicie.
17 SLCT* IN Indica a la impresora que esta seleccionada.
```

**Nota:** Los nombres con \* indican se1al negada.

Este standard podemos trabajar con 5 modos de funcionamiento:

- Compatible o SPP.
- Modo Nibble, este modo reserva el canal de datos y se combina con el modo SPP.
- Byte Mode, este modo es bidireccional utilizado por IBM y tiene la capacidad de deshabilitar los drivers usando la linea de datos.
- EPP (Extended Parallel Port) es el modo extendido y se utiliza para perif3ricos que principalmente no son impresoras.
- ECP (Enhanced Capabilities Port) al igual que el modo EPP es bidireccional y se utiliza para comunicar diversos tipos de perif3ricos, este modo es una propuesta de Hewlett Packard y Microsoft.

La clase ParallelPort como dijimos es una clase que hereda de CommPort. Esta clase cuenta con una serie de m3todos que nos facilitan el uso del puerto paralelo y los exponemos a continuaci3n:

- addEventListener(ParallelPortEventListener ev) mediante este m3todo podremos asignar un oyente para cuando aparezca un error en el puerto no lo notifique, exactamente nos informara de cuando el buffer de salida este lleno o cuando la impresora indique error. Los m3todos que nos permiten habilitar o deshabilitar estos avisos son: notifyOnError(boolean) y notifyOnBuffer(boolean). Si deseamos quitar el oyente podemos utilizar el m3todo removeEventListener().
- getOutputBufferFree() nos informa de los bytes que tenemos disponible en el buffer de salida.
- isPaperOut() nos devolver3 un valor true en el caso de que nos quedemos sin papel en la impresora, esto es lo mismo que preguntar por el estado de la se1a PE (ver tabla B).
- isPrinterBusy() nos devolver3 un valor true en el caso de que la impresora este ocupada y se corresponde con el estado BUSY.
- isPrinterSelected() nos informa si la impresora esta seleccionada.
- isPrinterError() chequea si ocurre un error en la impresora y esto se refleja con el estado de ERROR que esta en el pin 15 (ver Tabla B).
- restart() realiza un reset en la impresora.
- suspend() suspende la salida.
- getMode() nos dice el modo configurado del puerto paralelo y los valores posibles son:
  - LPT\_MODE\_SPP modo compatible unidireccional.
  - LPT\_MODE\_PS2 modo Byte.
  - LPT\_MODE\_EPP modo extendido.
  - LPT\_MODE\_ECP modo mejorado.
  - LPT\_MODE\_NIBBLE modo nibble.
- setMode(int) configuramos el modo del puerto paralelo, los valores que puede tomar son los anteriores y a dem3s LPT\_MODE\_ANY que ser3 el mejor modo disponible. Este m3todo debera ser capaz de tratar la excepci3n

UnsupportedCommOperationException que saltara en el caso de que introduzcamos un modo no soportado.

Aunque de momento solo se dispone de esta API para unix (de Sun) y Windows, se esta desarrollando una versión para linux.

**Fco. Javier Rodriguez Navarro** trabaja como responsable de desarrollos en **Tms** con el objetivo de crear sistemas abiertos y escalables, de ahí la afición por Java. Cada vez que tiene un rato libre se lanza a viajar y pasear para conocer nuevas gentes, paisajes y costumbres. Para cualquier duda o tirón de orejas, e-mail a: [frodrigu@idecnet.com](mailto:frodrigu@idecnet.com)