Utilisation de l'api javax.comm pour les ports séries

par Christophe Jollivet

Date de publication : 3/11/04

Dernière mise à jour : 02/01/05

Cet article vous présente l'utilisation de l'api javax.comm pour le contrôle des ports séries.

1 - Introduction

Attention cette API ne correspond pas à l'esprit « write once, run everywhere » de Java puisqu'une DLL est nécessaire. Pour les utilisateurs de Linux, il existe une API qui s'appelle RX/TX.

1.1 - Rappels sur le port série

Le port série utilise un fil pour l'émission et un autre fil pour la réception. Donc, dans un port série les bits de données sont envoyés les uns après les autres. L'interface série est orientée caractère et chaque caractère envoyé est délimité par un signal de début (un bits à 0) et par un signal de fin (un ou deux bits selon le paramétrage).Un caractère étant généralement composé d'un ensemble de huit bits, la conséquence de ce système de communication est une lenteur puisque pour un caractère de 8 bits, il faut ajouter 2 ou 3 bits supplémentaires qui représente 25% minimum de message en plus.

Pour plus d'informations sur le port série, veuillez consulter ces sites : **http://www.ctips.com/rs232.html** ou **http://www.google.fr**

1.2 - Contenu de l'archive javax.comm

Le zip est téléchargeable sur le site de SUN : **http://java.sun.com/products/javacomm/index.jsp**

Il est composé de :

- un jar : comm.jar
- un fichier de configuration : javax.comm.properties
- une dll : win32com.dll
- la documentation associée à l'API
- une série d'exemples

Attention, les exemples fournis avec l'API, sont écrits pour un environnement Unix, il faut remplacer le « /dev/term/a » par « COM1 » pour l'identifiant de port.

La dll est à mettre dans le répertoire Windows/system32, le fichier comm.jar et le fichier javax.comm.properties sont à mettre avec les autres librairies externes de votre application. Il ne faut pas toucher au fichier de configuration.

1.3 - Contenu de l'API

Cette API contient différentes classes et interfaces.

Pour la gestion des ports :

- class javax.comm.CommPort Abstraite
- class javax.comm.ParallelPort extends CommPort
- class javax.comm.SerialPort extends CommPort
- class javax.comm.CommPortIdentifier

Pour la gestion des événements :

- interface javax.comm.CommPortOwnershipListener (extends java.util.EventListener)
- class javax.comm.ParallelPortEvent
- class javax.comm.SerialPortEvent
- interface javax.comm.ParallelPortEventListener (extends java.util.EventListener)
- interface javax.comm.SerialPortEventListener (extends java.util.EventListener)

Pour les exceptions :

- class javax.comm.NoSuchPortException
- class javax.comm.PortInUseException
- class javax.comm.UnsupportedCommOperationException

Il existe deux niveaux pour la gestion des ports de communication :

- un niveau haut avec les deux classes CommPortIdentifier et CommPort
- un niveau bas avec les deux classes SerialPort et ParallelPort

La classe CommPortIdentifier est la classe centrale pour le contrôle des accès aux ports de communication.

Elle inclut les méthodes pour :

- déterminer les ports de communication disponibles
- ouvrir les ports de communications
- déterminer les possesseurs de ports et résoudre les conflits et changements

Une application utilise d'abord CommPortIdentifier pour négocier avec le driver et découvrir les ports qui existe sur la machine puis en sélectionner un pour l'ouverture. Ensuite elle utilise les méthodes des autres classes comme CommPort, ParallelPort et SerialPort pour communiquer par ce port.

La classe CommPort est une classe abstraite qui décrit les ports de communication rendus disponibles par le système. Elle inclut les méthodes pour le contrôle des entrées/sorties qui sont communes aux différents types de ports de communications. SerialPort et ParallelPort sont des sous-classes de CommPort qui incluent des méthodes additionnelles pour le contrôle bas-niveau des ports de communication.

Il n'y a pas de constructeur pour CommPort. Par conséquent une application doit utiliser la méthode CommPortIdentifier.getPortIdentifiers() pour générer une liste des ports disponibles. Elle choisit ensuite un port dans la liste et appelle CommPortIdentifier.open pour créer un objet CommPort qui est casté ensuite en SerialPort.

Après qu'un port de communication a été identifié et ouvert, il peut être configuré avec les méthodes de bas niveau de la classe SerialPort.

Enfin il est possible de travailler de 2 façons différentes avec ces ports :

- en utilisant les flux
- en mode événementiel

2 - Utilisation de l'API

2.1 - Importer les packages nécessaires

```
import javax.comm.*;
import com.sun.comm.Win32Driver;
```

2.2 - Obtenir une instance de SerialPort

Deux façons de travailler :

- **CommPortIdentifier.getPortIdentifiers()** renvoie l'énumération des ports de la machine.
- **CommPortIdentifier.getPortIdentifier(nom du port)** renvoie le CommPortIdentifier correspondant au nom du port donné. Elle est susceptible de lever une exception si le port associé au nom n'existe pas.

2.2.1 - Lister les ports de la machine

```
//initialisation du driver
Win32Driver w32Driver= new Win32Driver();
w32Driver.initialize();
//récupération de l'énumération
Enumeration portList=CommPortIdentifier.getPortIdentifiers();
//affichage des noms des ports
CommPortIdentifier portId;
while (portList.hasMoreElements()){
 portId=(CommPortIdentifier)portList.nextElement();
 System.out.println(portId.getName());
}
```

La sortie écran nous donne :

```
COM3
COM1
LPT1
LPT2
```

Attention cette liste de ports est susceptible de varier selon la configuration de votre machine.

2.2.2 - Obtenir un port

Maintenant nous allons essayer d'obtenir un objet SerialPort sur le port COM1. Pour cela on obtient d'abord l'identifiant puis on ouvre le port correspondant et le caste en SerialPort.

```
Win32Driver w32Driver= new Win32Driver();
w32Driver.initialize();
//récupération du port
CommPortIdentifier portId;
try{
 portId=CommPortIdentifier.getPortIdentifier("COM1");
}catch(NoSuchPortException ex){
//traitement de l'exception
}
SerialPort port;
try {
 port=(SerialPort)portId.open("Mon_Appli", 10000);
```

```
} catch (PortInUseException ex) {
 //traitement de l'exception
}
```

L'appel de la méthode open accepte deux paramètres :

- le nom de l'application qui demande le port
- le délai d'attente pour l'obtention du port en millisecondes

Si le port est déjà utilisé par une autre application, le système propage un PORT_OWNERSHIP_REQUESTED et donne alors le délai au système pour libérer le port. En cas d'échec une PortInUseException est levée.

2.2.3 - Rendre le port

Pendant l'exécution de votre application, une autre application est susceptible de vouloir utiliser le même port que vous. Si vous désirez en être informé pour éventuellement libérer le port, votre application qui ouvre le port doit implémenter l'interface CommPortOwnershipListener .

Si vous désirez libérer le port, il faut faire appel à la méthode CommPort.close()

2.3 - Utiliser cette instance de SerialPort

2.3.1 - Paramétrer le port

Après son ouverture, vous devez paramétrer le port. Les paramètres importants pour la communication par port série sont :

- le contrôle de flux
- le débit de la connexion (en Bauds)
- le nombre de bits de données
- le ou les bits stop
- la présence d'un bit de parité

```
try{
 port.setFlowControlMode(SerialPort.FLOWCONTROL_NONE);
 port.setSerialPortParams(9600, SerialPort.DATABITS_8, SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
}catch (UnsupportedCommOperationException ex){
 //gestion de l'exception
}
```

Ces méthodes de paramétrage sont susceptibles de lever une UnsupportedCommPortOperationException. Leur utilisation est simplifiée par l'usage des attributs statiques de SerialPort. Dans l'exemple précédent notre port est paramétré sans contrôle de flux, à 9600 bauds sur 8 bits avec 1 bit stop et sans parité ce qui correspond au paramétrage par défaut.

Enfin la classe SerialPort dispose aussi de méthodes permettant d'interrompre ou reprendre la propagation de certains types d'évènements selon vos besoins.

2.3.2 - Travailler avec les flux

On peut désirer travailler avec les flux des ports séries (dans le cas de communication avec un périphérique par exemple)

```
InputStream in;
OutputStream out;
try {
 out = port.getOutputStream();
 in = port.getInputStream();
} catch (IOException ex) {
 //gestion de l'exception
}
```

On travaille ensuite simplement avec les flux comme on le fait avec une socket par exemple (construction de BufferedReader, XMLEncoder, etc.).

2.3.3 - Travailler en événementiel

Il est possible de travailler en événementiel avec les ports séries. Pour cela il suffit d'implémenter l'interface SerialPortEventListener dans une classe et de l'ajouter en Listener.

Ajouter la classe en Listener

```
// ajout d'un Listener au port.
try {
 port.addEventListener(un SerialPortEventListener);
} catch (TooManyListenersException ex) {
 //traitement de l'exception
}
```

Les ports ne pouvant accepter qu'un Listener, une exception de type TooManyListenersException est levée si un listener est déjà enregistré auprès du port.

L'interface SerialPortEventListener présente une seule méthode : public abstract void serialEvent(SerialPortEvent ev). Il faut alors gérer l'évènement selon son type que l'on peut récupérer par la méthode ev.getEventType().

Les ports séries disposent de toutes une batterie d'événements. Une gestion classique de ces événements s'opère par utilisation d'une structure switch et des attributs statiques de la classe SerialPortEvent

```
switch (ev.getEventType()) {
// Réception de données
 case SerialPortEvent.DATA_AVAILABLE:
  // Récupération des données
  break;
// Réception d'un signal BREAK.
 case SerialPortEvent.BI:
  // Gestion de l'évènement
  break;
}
```

2.4 - Cas des applets

Il faut savoir que le chargement du driver avec une applet lève une exception sur le :

```
w32Driver.initialize();
```

Le message dans la console est :

```
Caught java.lang.NullPointerException: name can't be null while loading driver com.sun.comm.Win32Driver
```

Cela n'empêche pas l'applet de fonctionner. Vous pouvez intercepter cette exception ou la laisser s'afficher dans la console java du navigateur.

3 - Exemples d'application

A la suite vous trouverez deux exemples d'utilisation du port série. Le premier travaille avec les flux, le second travaille en mode événementiel.

Ces exemples reprennent une partie du code des exemples disponibles avec l'API. Dans les exemples qui suivent, la section catch des blocs try-catch est laissée vide pour réduire la taille du code. De même, dans un souci de simplification, il n'y a pas de contrôle des arguments et la demande d'un port inexistant entraîne un plantage.

3.1 - Utilisation des flux (balance électronique)

Ce mode de fonctionnement peut aussi correspondre à celui de communication avec un modem, un routeur, un moteur pas à pas, etc.

Cet exemple est basé sur la demande de poids à une balance connectée sur le port série. Pour donner son poids, la balance attend un signal sous la forme du caractère '$'.

La ligne de commande attend en argument le nom du port où la balance est connectée (COM1 par exemple). Attention dans cet exemple, l'absence de périphérique sur le port entraîne un blocage de l'application sur la lecture de la réponse. Pour éviter ce blocage, la communication doit se faire dans un thread.

```
import javax.comm.*;
import com.sun.comm.Win32Driver;
import java.io.*;

public class UtilisationFlux {

 private BufferedReader bufRead; //flux de lecture du port
 private OutputStream outStream; //flux d'écriture du port
 private CommPortIdentifier portId; //identifiant du port
 private SerialPort sPort; //le port série
 /**
  * Constructeur
  */
 public UtilisationFlux(String port) {
  //initialisation du driver
  Win32Driver w32Driver = new Win32Driver();
  w32Driver.initialize();
  //récupération de l'identifiant du port
  try {
   portId = CommPortIdentifier.getPortIdentifier(port);
  } catch (NoSuchPortException e) {
  }
  //ouverture du port
  try {
   sPort = (SerialPort) portId.open("UtilisationFlux", 30000);
  } catch (PortInUseException e) {
  }
  //règle les paramètres de la connexion
  try {
   sPort.setSerialPortParams(
    9600,
    SerialPort.DATABITS_8,
    SerialPort.STOPBITS_1,
    SerialPort.PARITY_NONE);
  } catch (UnsupportedCommOperationException e) {
  }
  //récupération du flux de lecture et écriture du port
  try {
   outStream = sPort.getOutputStream();
   bufRead =
    new BufferedReader(
     new InputStreamReader(sPort.getInputStream()));
```

```
   } catch (IOException e) {
   }
  }
  /**
   * Méthode de communication.
   */
  public String communique(char envoie) {
   String poids = null;
   try {
    //demande de poids
    outStream.write((int) envoie);
    //lecture du poids
    poids = bufRead.readLine().trim();
   } catch (IOException e) {
   }
   return poids;
  }
  /**
   * Méthode de fermeture des flux et port.
   */
  public void close(){
   try {
    bufRead.close();
    outStream.close();
   } catch (IOException e) {
   }
   sPort.close();
  }
  /**
   * Méthode principale de l'exemple.
   */
  public static void main(String[] args) {
   //Récupération du port en argument
   String port = args[0];
   //Construction de l'interface à la balance
   UtilisationFlux utilFlux = new UtilisationFlux(port);
   //"interface utilisateur"
   System.out.println("taper q pour quitter, ou ENTER pour le poids");
   //construction flux lecture
   BufferedReader clavier =
    new BufferedReader(new InputStreamReader(System.in));
   //lecture sur le flux entrée.
   try {
    String lu = clavier.readLine();
    while (!lu.equals("q")) {
     System.out.println(utilFlux.communique('$'));
     lu = clavier.readLine();
    }
   } catch (IOException e) {
   }
   utilFlux.close();
  }
}
```

3.2 - Utilisation du mode événementiel (lecteur de code barre)

Ce mode de fonctionnement correspond à celui des capteurs de façon générale.

Cet exemple est basé sur l'utilisation avec un lecteur de code à barre. Lorsqu'un code à barre est flashé, sa valeur s'affiche à l'écran. L'écoute du port série se fait dans un thread. La ligne de commande attend en argument le nom du port où le lecteur est connecté (COM1 par exemple).

```
import javax.comm.*;
import com.sun.comm.Win32Driver;
import java.io.*;
import java.util.*;
```

```
public class ModeEvenement extends Thread implements SerialPortEventListener {

 private CommPortIdentifier portId;
 private SerialPort serialPort;
 private BufferedReader fluxLecture;
 private boolean running;

 /**
  * Constructeur qui récupère l'identifiant du port et lance l'ouverture.
  */
 public ModeEvenement(String port) {
  //initialisation du driver
  Win32Driver w32Driver = new Win32Driver();
  w32Driver.initialize();
  //récupération de l'identifiant du port
  try {
   portId = CommPortIdentifier.getPortIdentifier(port);
  } catch (NoSuchPortException e) {
  }

  //ouverture du port
  try {
   serialPort = (SerialPort) portId.open("ModeEvenement", 2000);
  } catch (PortInUseException e) {
  }
  //récupération du flux
  try {
   fluxLecture =
    new BufferedReader(
     new InputStreamReader(serialPort.getInputStream()));
  } catch (IOException e) {
  }
  //ajout du listener
  try {
   serialPort.addEventListener(this);
  } catch (TooManyListenersException e) {
  }
  //paramétrage du port
  serialPort.notifyOnDataAvailable(true);
  try {
   serialPort.setSerialPortParams(
    9600,
    SerialPort.DATABITS_7,
    SerialPort.STOPBITS_1,
    SerialPort.PARITY_EVEN);
  } catch (UnsupportedCommOperationException e) {
  }
  System.out.println("port ouvert, attente de lecture");
 }
 public void run() {
  running = true;
  while (running) {
   try {
    Thread.sleep(2000);
   } catch (InterruptedException e) {
   }
  }
  //fermeture du flux et port
  try {
   fluxLecture.close();
  } catch (IOException e) {
  }
  serialPort.close();
 }
 /**
  * Méthode de gestion des événements.
  */
 public void serialEvent(SerialPortEvent event) {
  //gestion des événements sur le port :
  //on ne fait rien sauf quand les données sont disponibles
  switch (event.getEventType()) {
```

```
    case SerialPortEvent.BI :
    case SerialPortEvent.OE :
    case SerialPortEvent.FE :
    case SerialPortEvent.PE :
    case SerialPortEvent.CD :
    case SerialPortEvent.CTS :
    case SerialPortEvent.DSR :
    case SerialPortEvent.RI :
    case SerialPortEvent.OUTPUT_BUFFER_EMPTY :
     break;
    case SerialPortEvent.DATA_AVAILABLE :
     String codeBarre = new String();
     try {
      //lecture du buffer et affichage
      codeBarre = (String) fluxLecture.readLine();
      System.out.println(codeBarre);
     } catch (IOException e) {
     }
     break;
   }
  }
  /**
   * Permet l'arrêt du thread
   */
  public void stopThread() {
   running = false;
  }
  /**
   * Méthode principale de l'exemple.
   */
  public static void main(String[] args) {
   //Récuperation du port en argument
   String port = args[0];
   //lancement de l'appli
   ModeEvenement modeEve=new ModeEvenement(port);
   modeEve.start();
   //"interface utilisateur"
   System.out.println("taper q pour quitter");
   //construction flux lecture
   BufferedReader clavier =
    new BufferedReader(new InputStreamReader(System.in));
   //lecture sur le flux entrée.
   try {
    String lu = clavier.readLine();
    while (!lu.equals("q")) {
    }
   } catch (IOException e) {
   }
   modeEve.stopThread();
  }
 }
```

4 - Conclusion

Dans ce tutoriel, j'aborde l'utilisation du port série. Le problème du port série est qu'il n'existe pas d'API fonctionnant sur tous les systèmes. Cela fait perdre le caractère multiplateforme de votre application en JAVA. De plus l'API javax.comm utilisée dans ce tutorial et qui fonctionne sous Windows nécessite l'installation d'une dll sur le poste. Ce second point est encore plus gênant dans le cas d'une utilisation sous forme d'applet. Toutefois son utilisation reste très souple puisque tous les paramètres sont facilement accessibles. De plus cette API permet une utilisation des ports selon un mode événementiel ou par utilisation des flux, en fonction du type d'application que vous développez.