Serial Communication in Java with Example Program

Generally, communication with serial ports involves these steps (in no particular order):

- Searching for serial ports
- Connecting to the serial port
- Starting the input output streams
- Adding an event listener to listen for incoming data
- Disconnecting from the serial port
- Sending Data
- Receiving Data

I wrote an example program that includes all of those steps in it and are each in their own separate method within the class, but first I will go through my hardware set up.

Hardware Setup

My current hardware setup is as follows:

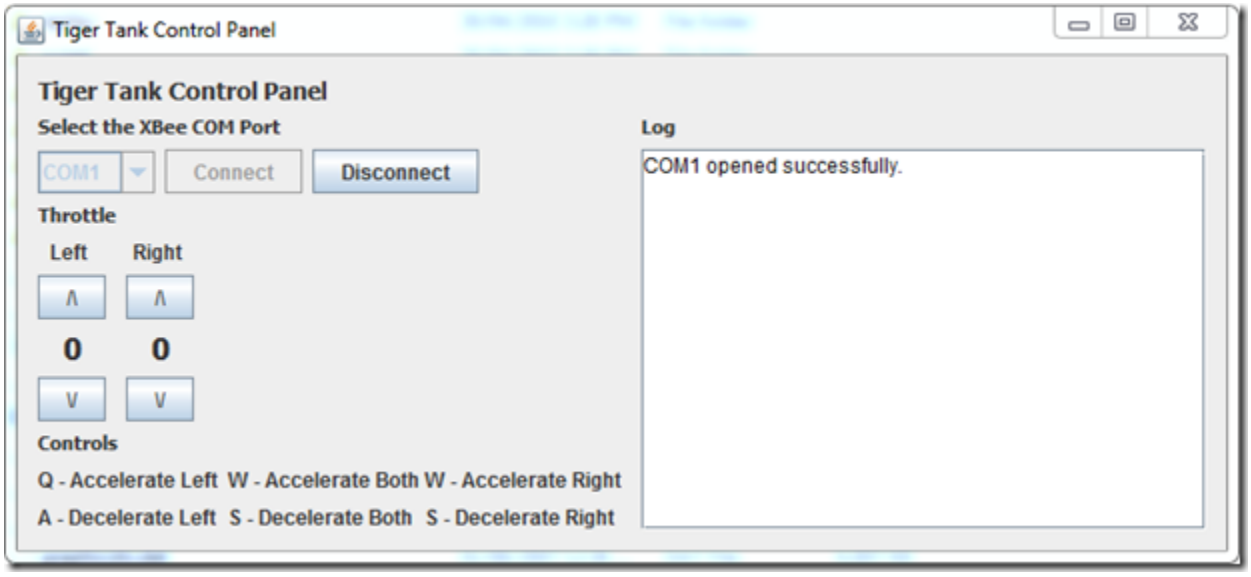
- PC connected to an XBee
- Arduino connected to an XBee

User input is given from the PC through the a Java GUI that contains code for serial communication, which is the code presented here.

The Arduino is responsible for reading this data. This set up is pretty much using my computer as a remote control for whatever device is on the Arduino end. It could be a motor control, on-off switch, etc.

Existing Code

The purpose of this post is to discuss serial programming in Java, and not GUI's. However, I did create a GUI for testing purposes. See the Code Downloads section for the actual files.



Above is the picture of the GUI complete with the buttons that I use to interact with the program. I also added key bindings which I can use to control the throttle.

When the program is first started, none of the GUI elements will work except for the combo box and the connect button. Once a successful connection is made the controls are enabled. This is done through the use of the *setConnected(true)* and the *toggleControls()* methods shown in the example code that follows.

Imports

The imports i used for this program were as follows:

```
import gnu.io.*;
import java.awt.Color;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.TooManyListenersException;
```

Depending on the Java IDE it might already know to tell you to use these imports except for the first one. That first import is specific to RXTX, and all its library methods/classes are in there.

Class Declaration

The code here reads:

```
public class Communicator implements SerialPortEventListener
```

I named my class *Communicator*, but the name is really up to the programmer. The name pretty much reflects its intended use.
The class should also *implement* the *SerialPortEventListener* class. This is a class in RXTX and is required in order to receive incoming data.
On some IDE's this may generate a public void method called *serialEvent()*. This method will be defined later.

Class Variables and Constants

Below are the variables and constants that I defined in my class. What the variables are for is in the comments but a more detailed explanation will follow.

```
    //passed from main GUI
    GUI window = null;

    //for containing the ports that will be found
    private Enumeration ports = null;
    //map the port names to CommPortIdentifiers
    private HashMap portMap = new HashMap();

    //this is the object that contains the opened port
    private CommPortIdentifier selectedPortIdentifier = null;
    private SerialPort serialPort = null;

    //input and output streams for sending and receiving data
    private InputStream input = null;
    private OutputStream output = null;

    //just a boolean flag that i use for enabling
    //and disabling buttons depending on whether the program
    //is connected to a serial port or not
    private boolean bConnected = false;

    //the timeout value for connecting with the port
    final static int TIMEOUT = 2000;

    //some ascii values for for certain things
    final static int SPACE_ASCII = 32;
    final static int DASH_ASCII = 45;
    final static int NEW_LINE_ASCII = 10;

    //a string for recording what goes on in the program
    //this string is written to the GUI
```

```
String logText = "";
```

I could have easily put the variable definitions in the constructor but it wouldn't change anything.

The *GUI* object is another class that I wrote separate from this one that contains all the GUI elements. The GUI class extends *javax.swing.JFrame*.
The *Enumeration* in Java is used for storing a series of elements of objects. In this case, the objects are *CommPortIdentifiers* (see Enumeration in the Reference Material).
The *HashMap* is for mapping each of the ports' names to the actual object. What that means is that I can associate (*put()* method) the name of a serial port, say a string that says COM1, to an object in the code. Later, I can access the name COM1 from the *HashMap* by using the*get()* method and it will return the object that it was associated with previously.
The *CommPortIdentifiers* object is needed to gather the list of ports that are available for connection. by using its *getPortIdentifiers()* method.
The *SerialPort* object is for storing the data for the port once a successful connection is made.
The *InputStream* and *OutputStream* is the object that is required for sending and receiving data.
The Boolean variable *bConnected* is just a flag that I use for enabling and disabling elements on the GUI.
The constant *TIMEOUT* is a number required when opening the port so that it knows how long to try for before stopping.

The constants for the ASCII values are some values that I use to send through the output stream that act as delimiters for data.

The string *logText* is basically what the comment says. When stuff happens in the program, the program stores a string in this variable and it will be appended to a text area in the GUI.

Searching for Available Serial Ports

The method below is for searching for available serial ports on the computer. Code adapted from Discovering Available Comm Ports from the Reference Material.

```
//search for all the serial ports
//pre style="font-size: 11px;": none
//post: adds all the found ports to a combo box on the GUI
public void searchForPorts()
{
    ports = CommPortIdentifier.getPortIdentifiers();

    while (ports.hasMoreElements())
    {
        CommPortIdentifier curPort = (CommPortIdentifier)ports.nextElement();

        //get only serial ports
        if (curPort.getPortType() == CommPortIdentifier.PORT_SERIAL)
        {
            window.cboxPorts.addItem(curPort.getName());
            portMap.put(curPort.getName(), curPort);
        }
    }
}
```

The method *getPortIdentifiers()* returns an Enumeration of all the comm ports on the computer. The code can iterate through each element inside the Enumeration and determine whether or not it is a serial port. The method *getPortType()* can identify what kind of port it is. If it is a serial port, then the code will add its name to a combo box in the GUI (so that users can pick what port to connect to). The serial port that is found should also be mapped to the HashMap so we can identify the object later. This is helpful because the names listed in the combo box are the actual names of the object (COM1, COM2, etc), and so we can use these names to identify the actual object they are tied to.

Connecting to the Serial Port

The method below is for connecting to the serial port once they have been found (see previous section). See How to Open A Serial Port in the Reference Material for more information.

```
//connect to the selected port in the combo box
```

```
//pre style="font-size: 11px;": ports are already found by using the searchForPorts
//method
//post: the connected comm port is stored in commPort, otherwise,
//an exception is generated
public void connect()
{
    String selectedPort = (String)window.cboxPorts.getSelectedItem();
    selectedPortIdentifier = (CommPortIdentifier)portMap.get(selectedPort);

    CommPort commPort = null;

    try
    {
        //the method below returns an object of type CommPort
        commPort = selectedPortIdentifier.open("TigerControlPanel", TIMEOUT);
        //the CommPort object can be casted to a SerialPort object
        serialPort = (SerialPort)commPort;

        //for controlling GUI elements
        setConnected(true);

        //logging
        logText = selectedPort + " opened successfully.";
        window.txtLog.setForeground(Color.black);
        window.txtLog.append(logText + "\n");

        //CODE ON SETTING BAUD RATE ETC OMITTED
        //XBEE PAIR ASSUMED TO HAVE SAME SETTINGS ALREADY

        //enables the controls on the GUI if a successful connection is made
        window.keybindingController.toggleControls();
    }
    catch (PortInUseException e)
    {
        logText = selectedPort + " is in use. (" + e.toString() + ")";

        window.txtLog.setForeground(Color.RED);
        window.txtLog.append(logText + "\n");
    }
    catch (Exception e)
    {
        logText = "Failed to open " + selectedPort + "(" + e.toString() + ")";
        window.txtLog.append(logText + "\n");
        window.txtLog.setForeground(Color.RED);
    }
}
```

Using the *HashMap* we can retrieve the *CommPortIdentifier* object from the string that was mapped earlier. This is achieved through the *HashMap's get()* method. The object must also be casted as a *CommPortIdentifier* because the *get()* method has a return type of *Object*.
The *setConnected* method just changes a boolean flag so that the program can store whether or not it is connected to a serial port or not. The *CommPort* must also be initialized because the port object will be stored here once a successful connection is made.
The main method of interest here is the *open()* method. This instructs the program to open the port, and this method will return the object for the opened port, which I store in in the previously initialized *CommPort* object. I then cast this object as a *SerialPort* and store it as well. This is helpful for accessing the methods and variables specific to the *SerialPort* class.
It should also be noted that the *open()* method requires the use of a try-catch block. So applying that, I catch two different exceptions. The *PortInUseException* is what the name says. If the port is in use, then this exception is thrown. The next catch block is just for the generic exceptions that occur. I never came across that during testing, nor do I know how to replicate that.

I neglected to include any code for setting the baud rate and other settings on the XBee's (see Hardware Configuration), because I already set those parameters previously using X-CTU (more on setting up XBee's here).

Initializing the Input and Output Streams

This method is pretty short and is pretty straightforward to read.

```
//open the input and output streams
//pre style="font-size: 11px;": an open port
//post: initialized input and output streams for use to communicate data
public boolean initIOStream()
{
    //return value for whether opening the streams is successful or not
    boolean successful = false;

    try {
        //
        input = serialPort.getInputStream();
```

```
            output = serialPort.getOutputStream();
            writeData(0, 0);

            successful = true;
            return successful;
        }
        catch (IOException e) {
            logText = "I/O Streams failed to open. (" + e.toString() + ")";
            window.txtLog.setForeground(Color.red);
            window.txtLog.append(logText + "\n");
            return successful;
        }
    }
```

The streams are initialized by returning the input and output streams of the open serial port. In order for the serial port object to not be null, it must store the object for the open serial port.

The *getInputStream()* and *getOutputStream()* methods both require a try-catch block. The important exception to catch here is the *IOException* to signify whether the streams failed to open or not. For more information see How to Open A Serial Port in the Reference Material.
I also call a method called *writeData(0, 0)*. This is the method that encapsulates the code required to write serial data. More on that later. Right now it is just used to set variables on the microcontroller side to zero.

Setting Up Event Listeners to Read Data

Once the port is open, the serial port must know whenever there is data to be read. This approach is event driven rather than by polling. When the event is hit, a special block of code will run. This is advantageous to polling because polling requires constantly asking if data is available. The code for the event driven approach is below. See Event Based Two Way Communication for more information.

```
    //starts the event listener that knows whenever data is available to be read
    //pre style="font-size: 11px;": an open serial port
    //post: an event listener for the serial port that knows when data is received
    public void initListener()
    {
        try
        {
            serialPort.addEventListener(this);
            serialPort.notifyOnDataAvailable(true);
        }
        catch (TooManyListenersException e)
        {
            logText = "Too many listeners. (" + e.toString() + ")";
            window.txtLog.setForeground(Color.red);
            window.txtLog.append(logText + "\n");
        }
    }
```

The code here is pretty much the whole reason why this class implements*SerialPortEventListener*. The parameter for adding the event listener is just the object itself. The base class has a method called *serialEvent(event)* that should be overridden before the event code will work. Overriding that method defines what happens when the event is hit.

The method *addEventListener(this)* is the event that is always checking for *SerialEvents*. It isn't really important to know what the events actually are, just that the one we're interested in is the one that tells us whenever data is received. Adding the event listener is complemented by the method *notifyOnDataAvailable(true)*, which definitely helps us achieve what I mentioned previously.

This code requires a try-catch box. The exception here is that there may be too many event listeners. I only use one in the code, so this exception should never be hit.

Disconnecting from the Serial Port

Once all the communication is completed, the serial port must be disconnected. In some instances, some ports stay stuck open, which isn't exactly a good thing.

```
    //disconnect the serial port
    //pre style="font-size: 11px;": an open serial port
```

```
//post: closed serial port
public void disconnect()
{
    //close the serial port
    try
    {
        writeData(0, 0);

        serialPort.removeEventListener();
        serialPort.close();
        input.close();
        output.close();
        setConnected(false);
        window.keybindingController.toggleControls();

        logText = "Disconnected.";
        window.txtLog.setForeground(Color.red);
        window.txtLog.append(logText + "\n");
    }
    catch (Exception e)
    {
        logText = "Failed to close " + serialPort.getName()
                          + "(" + e.toString() + ")";
        window.txtLog.setForeground(Color.red);
        window.txtLog.append(logText + "\n");
    }
}
```

The code here requires a try-catch block so there it is. The exception to catch here is pretty generic.

Before closing the port, I reset the data on the microcontroller. That step is optional depending on the application. For example, if the value sent to the microcontroller is controlling the speed of a motor, I'd want the motor to turn off if I turn off the remote control.

The event listener should be removed as it is no longer needed.

The method to close the port is the *close()* method.

The input and output streams should also be closed via the same method as above.

See How to Close A Serial Port in the Reference section for more information.

Reading Data – The *serialEvent* Method

This is a follow-up to the section on preparing to Read Data. It has information on how to process the data that the program reads.

```
//what happens when data is received
//pre style="font-size: 11px;": serial event is triggered
//post: processing on the data it reads
public void serialEvent(SerialPortEvent evt) {
    if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE)
    {
        try
        {
            byte singleData = (byte)input.read();

            if (singleData != NEW_LINE_ASCII)
            {
                logText = new String(new byte[] {singleData});
                window.txtLog.append(logText);
            }
            else
            {
                window.txtLog.append("\n");
            }
        }
        catch (Exception e)
        {
            logText = "Failed to read data. (" + e.toString() + ")";
            window.txtLog.setForeground(Color.red);
            window.txtLog.append(logText + "\n");
        }
```

```
        }
    }
```

Since we only want to read data if it exists, we test the condition in the if statement above. This could be redundant because we wouldn't be in this method if the event never got triggered.

Reading the data requires a try-catch block. The *read()* method returns an integer, but the value in it is actually a byte value so this is casted to a byte, in which I store in a variable called *singleData*. When I was testing this code before, I noticed that the data being read would always have random new line characters in it, and I never knew why this happened. I never wrote any new line characters. Since I don't want these, I add an if condition saying that if it encounters that, don't display that data on the screen.
Java also has a nice way of converting bytes to a string. A byte array can be used as a parameter when initializing a string, which is what I've done when I assign a value to *logText*. See Converting byte to String for more information.

The exception here is just the usual generic one.

Writing Data

Here is the method for writing data. In my application, I am writing two integers sequentially to my microcontroller. Data that is sent to the MCU can be used for such things such as controlling motor speeds, which is what I am trying to do here. I hope to use the data sent do control the PWM duty cycle of the motor for speed control.

```
    //method that can be called to send data
    //pre style="font-size: 11px;": open serial port
    //post: data sent to the other device
    public void writeData(int leftThrottle, int rightThrottle)
    {
        try
        {
            output.write(leftThrottle);
            output.flush();
            //this is a delimiter for the data
            output.write(DASH_ASCII);
            output.flush();

            output.write(rightThrottle);
            output.flush();
            //will be read as a byte so it is a space key
            output.write(SPACE_ASCII);
            output.flush();
        }
        catch (Exception e)
        {
            logText = "Failed to write data. (" + e.toString() + ")";
            window.txtLog.setForeground(Color.red);
            window.txtLog.append(logText + "\n");
        }
    }
```

The *write()* method here requires a try-catch block just like many of the other methods used for serial communication. Using the output stream, I call that method to write an integer value. I also write a dash and a space to separate the actual data being read. See the ASCII Table for the values of what these characters should be. This helps me differentiate the sets of data that is sent back and forth. The dash separates the two integers being sent and the space separates the entire set. On the Arduino end, four called to *read()* can separate the data without mixing things up. Example of that later.

Putting All the Java Code Together

Once all these methods are setup, they need to be called somewhere in the program so we can put them to use. When the program starts, the code for searching for serial ports will run so that it can populate the combo box on the GUI. The GUI buttons also need to be disabled to reflect the fact they should be disabled when there is no connection to a port. When the connect button is pressed, the input and output streams are started and the event listener is added. The code for that is shown below.

```
    private void btnConnectActionPerformed(java.awt.event.ActionEvent evt) {
        communicator.connect();
        if (communicator.getConnected() == true)
        {
```

```
            if (communicator.initIOStream() == true)
            {
                communicator.initListener();
            }
        }
    }
```

The if statement is needed such that the code will not run if a successful connection was not made. This is the precondition for the *initIOStream()* and the *initLIstener()* methods.
When the user decides to disconnect from the serial port, the user only needs to call the*disconnect()* method when the disconnect button is pressed.

The up/down arrows on my GUI are set to write data to the serial port when they are pressed. When data is written to the Arduino, the MCU will send the data back to the Java program, which will signal the event listener to display that data back to the screen.

Arduino Code

The Arduino code is pretty straight forward. There are just a few calls to the *read()* method.

Update 6 May 2012: the syntax for the Arduino code has updated since the writing of this article. Apparently, the lines

```
Serial.print(separator, BYTE);
Serial.print(space, BYTE);
```

are no longer valid and have since been replaced by the following:

```
Serial.write(byte(separator));
Serial.write(byte(space));
```

The code below hasn't been edited with this change, but basically the code required for writing a byte value through serial communication has changed.

```
// this is where we will put our data
int left = 0;
int right = 0;
byte space = 0;
byte separator = 0;

void setup(){
  // Start up our serial port, we configured our XBEE devices for 38400 bps.
  Serial.begin(9200);
}

void loop(){
  // handle serial data, if any
  if (Serial.available() >= 4){
    left = Serial.read();
    separator = Serial.read();
    right = Serial.read();
    space = Serial.read();
    Serial.flush();

    Serial.print(left);
    Serial.print(separator, BYTE);
    Serial.print(right);
    Serial.print(space, BYTE);
    Serial.print("\n");
  }
}
```

I first declare four variables to reflect the fact that I'm writing four different variables to the serial port.

In the loop method, I added a condition which will check whether there are four pieces of data available before reading them. This again reflects the fact I'm writing four values.

The four variables initially declared store the values in the order that they are written, and then are just printed to the screen.

The *print()* method is able to convert the byte value to an actual character by adding a second parameter called *BYTE*. The left and right variables don't need it because the values written were integers to begin with, however, the dash and space characters were written as ASCII values.

For more information about Arduino and Java, and Arduino's Serial library, please refer to those in the Reference Section.

Code Downloads

Here are the code downloads for the GUI, key bindings and the serial communication. WordPress doesn't allow uploads of zip or java files so I uploaded it all to Megaupload.

Link to download: http://www.mediafire.com/?z2l26ncypmzn20z

Other Examples

There are also other sites I looked at which had example code for serial port communication in Java. The sites are Programming Serial and Parallel Ports and Serial Port Access Using RXTX in Java, which are both in the Reference section.

Reference Material

Below are a bunch of links to reference sites that I used to find some of the information written here. I mostly referred to bits and pieces of each article to understand the serial code and to come up with the example I present here. The list is in the order that I referenced them.

Source:http://henrypoon.wordpress.com/2011/01/01/serial-communication-in-java-with-example-program/