J2ME | J2SE | J2EE | Home

MIDI & audio sequencing with java

by Mike Gorman

The Java Sound API, first introduced in J2SE 1.3, includes the package javax.sound.midi, which contains everything you need to be able to send and receive messages to and from any MIDI device visible to your operating system.

The Java Sound Programmer Guide and the Java Sound Demo, both available for download from Sun, are excellent references that illustrate all the "nuts and bolts" of sending and receiving messages. This article provides a brief overview of working with the MIDI and sampled audio primitives of the Java Sound API, and then explores using those primitives to construct a basic multi-track MIDI/audio sequencer in Java.

Programming with MIDI

Basically, every message you send to or receive from a MIDI device is one or more bytes. The first byte is referred to as the "message" or "status" byte. This is essentially the "command" (e.g., note on, note off, change patch, set volume, etc.). The specific command you are sending or receiving will determine what the next byte or bytes are, if any. Having an online MIDI specification reference available will be indispensable.

The first step is to obtain a reference to the specific MidiDevice you want to talk to. The javax.sound.midi. MidiSystem is your gateway to everything that Java Sound was able to detect was installed in your operating system. You might display a list of available devices to users and let them choose which device they want to use (see Listing 1).

Once you have a reference to your MidiDevice, you're ready to begin sending and receiving messages. The first step is to understand that there are Receivers and Transmitters. As you might suspect, Receivers receive MIDI messages and Transmitters transmit or are the source for MIDI messages.

Step one is finding out which MidiDevices the MidiSystem is reporting. The easiest way is to iterate through each one, try to play "middle C," and see what happens. You may find that some of the MidiDevices are configured as receive only, others are transmit only, and others might receive and transmit. In my setup, my MIDI interface reports two different MidiDevices for the same keyboard – one that is transmit only, the other that's receive only. So when I want to send a MIDI message, I send it to the MidiDevice that is receive only, and when I want to record MIDI messages that I trigger by playing the keyboard, I do it by listening to the MidiDevice that's configured to transmit only (see Listing 2).

Step two is to figure out which of the MidiDevices are transmitters. In this case, you'll need to create an implementation of the javax.sound.midi.Receiver interface (see Listing 3).

Next, figure out which of the available MidiDevices are acting as your keyboard's transmitter by trying them out one at a time. Obtain each MidiDevice's "transmitter," assign your receiver to it, play a few notes on your keyboard, and look for output in the console that indicates you've found the right "transmitter" device (see Listing 4).

Once you've figured out which MidiDevice is your receiver and which is your transmitter, and you know the basics of sending and receiving MidiMessages, you now have everything you need to create your own full-featured 16-track MIDI sequencer! Well, not quite…

Creating Your Own MIDI Sequencer

As I soon found out, there's a lot more to creating a sequencer than just knowing how to send and receive MIDI messages.

You may have noticed by now that the javax.sound.midi package already includes a class Sequencer. Unfortunately, this "built- in" sequencer is limited in its capabilities and is not extendable since it's an interface. But the biggest reason I was unable to use it is that it seems to be "hard wired" to use the internal Java synthesizer (Sun Bug ID 4783745). It also appears to have very bad timing problems (Sun Bug ID 4773012).

With the API it's easy to create your own sequencer. First we have to be able to record a single track and play it back in the exact same timing it was originally played in. Moreover, the performing artist (that's you) will want to have a four-bar metronome count off prior to recording start, then, to keep perfect time, you'll need to continue the metronome until the user clicks stop. Of course, the metronome sounds should not be part of the performance when it's played back.

You can have the computer emit a "system beep" for your metronome, but I prefer to listen to the hi-hat of a drumkit on the keyboard. A lot of sequencers use MIDI channel 10 (i.e., track 10) as a drumkit, but you can choose any one you like. A tempo of 120 beats per minute (bpm) means 2 beats/second or 1 tick of your metronome every 500ms.

As you may have guessed, you'll need one thread playing the ticks of your metronome (on Channel 10) while you record any MidiMessages you receive (via your Receiver implementation) on Channel 1. (Note that I am referring to the channels/tracks from the musician's perspective. The channel references in the API are zero-based.) Listing 5 shows what your metronome thread might look like.

Playing the metronome is simple enough, but you need to figure out a way to play it through just one time, then begin recording MidiMessages as they arrive at your receiver (discussed later). How you do that will be left for you to decide.

Recording MIDI

How exactly do you record MidiMessages? There are basically two strategies: you can try to take note of what time each message arrives, or you can use the included timestamp of each message. In either strategy, your implementation of the Receiver interface will create an ArrayList and add each MidiMessage it receives to the ArrayList. Of course, you'll need to make sure you record only MidiMessages for the duration immediately following the four-bar Metronome count off until the user clicks stop.

Your first strategy might be to use System.currentTimeMillis() to take note of the current system time (in ms) at which each MidiMessage arrives. You'll need to know this when you play back these messages. The general idea is to play back the messages using a thread, that's sleeping between messages, according to the relative time they originally arrived. In my experience, the system clock was not reliable enough to deliver rock-solid timings during playback. You'll know what I mean if you try this strategy when you listen to the playback of messages based on the system clock.

The other strategy is to use the embedded timestamp that accompanies each MidiMessage. This timestamp is expressed in microseconds based on the time you first opened the MidiDevice. Unfortunately, by the time the four-bar metronome count off ends, it's difficult to say when the first message should be played back. That is you can't assume that the first message that arrives should be played back at time zero. Perhaps the musician's first note is played halfway through the first measure. Since the MidiDevice was opened long before your metronome began playing, it's difficult to determine from the timestamp alone how much time your playback thread should wait until it sends the very first message. Of course, all messages after that are easy, since you can just calculate the time to wait in between each message based on the relative differences of the message's timestamps.

The best solution I came up with was to just take note (by way of System.currentTimeMillis()) of when recording actually begins (that is, after the four-bar metronome count off), and then take note of when the first MidiMessage arrives. Then, during playback, the playback thread merely needs to wait the

calculated delay time before playing back the first message. Thereafter, it can simply use the relative differences between the MidiMessage timestamps for all subsequent messages.

It may surprise you to learn that what you think of as a chord (or several chords across multiple tracks) struck simultaneously is actually played back one note at a time, sent serially as a stream of MidiMessages, one at a time. You have to remember that the playback loop playing back the messages is so fast that the human ear will not be able to discern the difference between the original "three notes struck simultaneously" and "three notes played 1 ms apart."

You should now be able able to record and play back a single MIDI track at 120 bpm. If, when it plays back, it sounds just like you played it, you're halfway there. The next step is to be able record additional MIDI tracks while playing back previously recorded tracks.

Recording Multiple Tracks

You may have already begun to notice that, although you are receiving and recording the MIDI messages, it's hard to control what sound/voice/patch the keyboard is playing. This is why each of the 16 MIDI channels on the keyboard can have a different patch associated with it. Most keyboards allow you to change what MIDI channel they are transmitting on. Whatever MIDI channel you have selected on the keyboard should also change the patch selected as well.

The problem is that you don't want to constantly have to make sure your keyboard's selected channel matches the track you play to record in your sequencer. If they're not in sync, you'll think you're recording track/channel 2, but the keyboard still has channel 1 selected. Although you may have the "channel 2 ArrayList" full of the MidiMessages you received, those messages have one of their bytes indicating that they are channel 1 messages, and so playback of those "channel 2 messages" results in playback on channel 1, playing channel 1's patch instead of channel 2.

The solution seems tricky and not very efficient, but it seems to work just fine. The trick is to first turn off the keyboard's "keyboard" from triggering sounds internally; it will continue to transmit MIDI messages as usual:

```
ShortMessage msg = new ShortMessage();
msg.setMessage(
  ShortMessage.CONTROL_CHANGE, 122, 0);
_receiver.send(msg, -1);
```

Next, "route" all incoming MIDI messages to the keyboard, playing them back on the track the user thinks he is recording. For example, you may receive all your MIDI messages with the "channel 1 byte" set. If the user thinks she is recording track 2, then for each MIDI message received, in addition to recording it (by storing it in track 2's message ArrayList), change the "channel byte" to 2 and retransmit them back to the keyboard (see Listing 6).

Playing Back Multiple Tracks

Assuming you have several different tracks of MIDI data recorded, it's time to play them back. Your first approach might be to use a separate thread for each track (channel). While this is an intuitive programming model, you'll quickly find that although each track (thread) plays back in perfect time relative to itself, it's difficult to keep it perfectly in sync with the other tracks. If your tracks are short and you plan to loop them, you could use thread synchronization to make sure all tracks "sync up" with each other at the end of each iteration. However, you will soon find your clean sequencer code is getting cluttered up with complex thread synchronization all over the place, and it becomes harder and harder to manage and still achieve "rock solid" timing.

What I found to be easier to manage and virtually guaranteed to stay "in time" was to collect all MidiMessages, regardless of track (channel), put them into a single ArrayList, sort them all based on their timestamp, and then play them all back using a single playback thread.

Adding Digital Audio

By now you should have a good instrumental recorded using multiple MIDI tracks, but you'll add more interest to your song by laying down a vocal track or two. Luckily, the Java Sound API includes the javax.sound.sampled package dedicated to recording and playing back digital audio.

Recording Audio

Ultimately, any recorded digital audio comes down to samples. A sample is a measurement at a point in time of what you might picture as the audio "waveform." The standard CD sampling rate is to take 44,100 measurements, or samples, each second. Each sample may be 8 bits, 16 bits, or more. There are a variety of sample formats in use today, and the Java Sound API supports about everything you'll encounter. Some useful constants for recording CD quality sound are:

```
AudioFormat.Encoding encoding =
  AudioFormat.Encoding.PCM_SIGNED;
int rate = 44100;
int sampleSize = 16;
int channels = 1;
boolean bigEndian = true;
```

An AudioFormat object will be needed later:

```
AudioFormat format = new AudioFormat(
encoding, rate, sampleSize, channels,
  (sampleSize / 8) * channels, rate,
  bigEndian);
```

Before you can begin recording, however, you'll need to obtain a TargetDataLine. The Java Sound API models its sampling API in terms of "lines." A line may be a microphone input, a previously recorded sample, the computer's "line out" or speaker, or any type of "input" or "output." To facilitate the playback of multiple samples at the same time, the interface Mixer is provided, which is itself a type of line. Lines may have controls that parallel what you'd find in a real mixer – gain, pan, volume, reverb, equalization, etc.

Like the MidiDevices returned from the MidiSystem, the class AudioSystem serves as your gateway into finding out and obtaining whatever Lines and Controls are installed and available to you. In general, the first step to recording an audio track is to obtain a TargetDataLine suitable for recording audio in the format requested, in this case an AudioFormat that is a single 16-bit channel recording 44,100 samples/second (see Listing 7).

As you may have suspected, you'll need a separate thread to capture the incoming sample data. Using the TargetDataLine and OutputStream created previously, you'll want to create a loop that reads a chunk of bytes at a time from the TargetDataLine, writing them out to the OutputStream until there's nothing left to read or until the user clicks stop (see Listing 8).

At this point, your ByteArrayOutputStream contains a ton of bytes. The average 3:30 minute song will require 9.3MB worth of samples for just a single mono track! FileOutput-

Stream might be a better choice if you're going to be recording lengthy samples and memory becomes scarce. Of course, recording the sample is just half of the story. Now we have to play it back.

Playing Back Audio

Playing back a previously recorded audio track is essentially the reverse of recording it. That is, the sample's bytes, originally stored in an OutputStream, are written out to a SourceDataLine one chunk at a time until there's nothing left or until the user clicks stop.

To read the bytes a chunk at a time, we'll need an InputStream. The Java Sound API provides the class AudioInputStream that has several convenience methods for working with samples. Again, we'll need to refer to the same AudioFormat that the sample was originally recorded in. In our case, we'll assume we're dealing with a completely in-memory sample, expressed as an array of bytes (see Listing 9).

Note that AudioInputStream's mark method is used to mark the beginning of the sample, while the reset method is used to "rewind" the sample to the beginning.

As has been the case, we'll need a separate thread to play back the sample. We'll use the AudioInputStream set up above to read sample bytes from it, a chunk at a time, writing them out to a SourceDataLine. Just as we obtained our TargetDataLine from the AudioSystem, we'll obtain a SourceDataLine suitable for playing back a sample in our AudioFormat through inquiry (see Listing 10).

Since we have a SourceDataLine that can handle our AudioFormat, we can start a thread to write out the sample bytes to it (see Listing 11).

Now that you have your audio track playing back – we're almost done!

Putting It All Together

At this point we have the main ingredients for a basic multi-track MIDI sequencer that can also record and play back audio. Although we can play back multiple tracks of MIDI using just one thread, it's much more difficult to play back multiple samples with a single thread. For simplicity, we'll continue to use one thread for all MIDI data, but create a different thread for each audio sample.

The basic trick for integrating MIDI and one or more samples is to simply synchronize the start of the MIDI tracks thread with the audio track thread(s) using normal thread sychronization techniques.

Of course, real commercial MIDI/audio sequencers can do much more than record and play back multiple tracks. That's just the beginning. After all, a real sequencer can:

- Play back what was recorded at one tempo at a different tempo
- Import "instrument definitions" that specify the patch names mapped to patch numbers
- Select each track's "patch" by searching the available patches by name
- Provide a mixer with volume and pan sliders for each track
- Record and play back volume changes from the mixer in real time
- "Trigger" audio samples from the keyboard (a la a conventional sampler)
- Quantize recorded MIDI data to the nearest 1/4 note, 1/8th note, 1/16th note, etc.

I'm out of space, so for now, I'll have to leave that as an exercise for you, the reader. In the meantime, enjoy your new sequencer!

Mike Gorman is a senior software architect for J.D. Edwards, a PeopleSoft company, concentrating on J2EE distributed transaction systems. Mike has been coding in Java since 1997. In his spare time, Mike plays with MIDI, Swing, Web services, and JDO.

mike_gorman@jdedwards.com

References

- *Open source MIDI and audio projects: Audio Development System:* http://sourceforge.net/projects/adsystem
- *jMusic:* http://sourceforge.net/projects/jmusic
- *Sound Grid:* http://sourceforge.net/projects/soundgrid

API References

- *Java Sound Programmer Guide:* http://java.sun.com/j2se/1.4.1/docs/guide/sound/programmer_guide/contents.html
- *Java Sound Demo:* http://java.sun.com/products/java-media/sound/samples/JavaSoundDemo/

MIDI Specification

- *Official MIDI Specification:* www.midi.org
- *Online MIDI Specification (unofficial):* www.borg.com/~jglatt/tech/midispec.htm

Miscellaneous

- *Bug ID 4773012: RFE: Implement a new stand-alone sequencer:* http://developer.java.sun.com/developer/bugParade/bugs/4773012.html
- *Bug ID 4783745: Sequencer cannot access external MIDI devices:* http://developer.java.sun.com/developer/bugParade/bugs/4783745.html

Listing 1: Displaying the MIDI devices available

```
1MidiDevice.Info[] info =
2  MidiSystem.getMidiDeviceInfo();
3
4  for (int i=0; i < info.length; i++) {
5  log(i + ") " + info[i]);
6  log("Name: " + info[i].getName());
7  log("Description: " +
8    info[i].getDescription());
9
10  MidiDevice device =
11    MidiSystem.getMidiDevice(info[i]);
12  log("Device: " + device);
13}
```

Listing 2: Sending "middle C note on" and "note off"

```
1// For each MidiDevice, open it up,
2// obtain it's receiver, and try it out
3MidiDevice dev = getDevice();
4dev.open(); //(at program start)
5Receiver receiver = dev.getReciever();
6
7// Send middle C (60) "note on"
8// at maximum velocity (127)
9ShortMessage msg1 = new ShortMessage();
10msg1.setMessage(ShortMessage.NOTE_ON,
11  60, 127);
12receiver.send(msg1, -1);
13
14// Wait a second
15Thread.sleep(1000);
16
17// Send middle C "note off"
18ShortMessage msg2 = new ShortMessage();
19msg2.setMessage(ShortMessage.NOTE_OFF,
20  60, 0);
21receiver.send(msg2, -1);
22
23// Close the device (at program exit)
24dev.close();
```

Listing 3: Minimal Receiver implementation

```
1public class MyReceiver extends Object
2  implements Receiver {
3  public void send(MidiMessage msg,
4    long time) {
5    log("Received message " + msg);
6  }
7
8  public void close() {
9    log("Closing");
10  }
11}
```

J2ME | J2SE | J2EE | HOME

Listing 4: Listen to each device's transmitter

```
1// Listen for MIDI messages originating
2// from each MidiDevice
3MidiDevice device = getDevice();
4device.open(); // (at program start)
5
6// Hook up a receiver to the transmitter
7device.getTransmitter().setReceiver(
8  new MyReceiver());
9
10// Wait long enough to play a few notes
11// on the keyboard
12Thread.sleep(30000);
13
14// Close the device (at program exit)
15device.close();
```

Listing 5: Sample metronome

```
1public class Metronome extends Object
2  implements Runnable {
3  private Receiver _receiver;
4  private ShortMessage _accentOn;
5  private ShortMessage _accentOff;
6  private ShortMessage _nonAccentOn;
7  private ShortMessage _nonAccentOff;
8  private boolean _stopped = true;
9
10  public Metronome(MidiDevice rcvDev) {
11    super();
12    _receiver = rcvDev.getReceiver();
13    _accentOn = createNoteOnMsg(42,127);
14    _accentOff = createNoteOffMsg(42);
15    _nonAccentOn = createNoteOnMsg(42,90);
16    _nonAccentOff = createNoteOffMsg(42);
17  }
18
19  private ShortMessage createNoteOnMsg(
20    int note, int velocity) {
21    ShortMessage msg = new ShortMessage();
22    msg.setMessage(ShortMessage.NOTE_ON,
23      note, velocity);
24    return msg;
25  }
26
27  private ShortMessage createNoteOffMsg(
28    int note) {
29    ShortMessage msg = new ShortMessage();
30    msg.setMessage(ShortMessage.NOTE_OFF,
31      note, 0);
32    return msg;
33  }
34
35  public void startMetronome() {
36    _stopped = false;
37    new Thread(this).start();
38  }
39
40  public void stopMetronome() {
41    _stopped = true;
42  }
43
44  public void run() {
45    long startTime =
46      System.currentTimeMillis();
47    try {
48      while (_stopped == false) {
49        _receiver.send(_accentOn, -1);
50
51        Thread.sleep(100);
52        _receiver.send(_accentOff, -1);
53        Thread.sleep(
54          getTimeTillNextBeat(startTime));
55        for (int i=0; i < 3; i++) {
56          _receiver.send(_nonAccentOn,
57            -1);
58          Thread.sleep(100);
59          _receiver.send(_nonAccentOff,
60            -1);
61        Thread.sleep(getTimeTillNextBeat(
62          startTime));
63         }
64      }
65    } catch (InterruptedException e) {
66      e.printStackTrace();
67      _stopped = true;
68    }
69  }
70
71  // assumes 120 bpm (or 500ms per beat)
72    private static long getTimeTillNextBeat(
73    long startTime) {
74    long position =
75      System.currentTimeMillis() –
76      startTime;
77    long timeRemaining = position % 500;
78    return timeRemaining;
79  }
80}
```

Listing 6: Rebroadcasting incoming MIDI messages on the desired MIDI channel

```
1public void send(MidiMessage msg,
2  long time) {
3  try {
4    if (msg instanceof ShortMessage) {
5      // Play back the incoming msg on
6      // the desired channel
7      ShortMessage incomingMsg =
8        (ShortMessage) msg;
9      ShortMessage playbackMsg =
10        new ShortMessage();
11
12      // Change the incoming message
13      playbackMsg.setMessage(
14        incomingMsg.getCommand(),
15        _playbackChannel,
16        incomingMsg.getData1(),
17      incomingMsg.getData2());
18      _receiver.send(playbackMsg, -1);
19
20      // If the sequencer is currently
21      // recording, hold on to each msg
22      if (_recordEvents) {
23        // Take note of when the first
24        // msg arrives so we'll know
25        // when to start playback later
26        if (_firstMessageArrivedAt == 0) {
27          _firstMessageArrivedAt =
28            System.currentTimeMillis();
29        }
30        _recordedEvents.addElement(
31          new MyEvent(playbackMsg, time));
32      }
33    }
34  } catch (InvalidMidiDataException e) {
35    e.printStackTrace();
36  }
37}
```

Listing 7: Preparing to record audio

```
1DataLine.Info info = new DataLine.Info(
2  TargetDataLine.class, getAudioFormat());
3
4if (AudioSystem.isLineSupported(info) ==
5  false) {
6  log("Line matching " + info +
7    " not supported.");
8  return;
9}
10
11TargetDataLine targetLine =
12  (TargetDataLine) AudioSystem.getLine(
13  info);
14targetLine.open(getAudioFormat(),
15  targetLine.getBufferSize());
16
17// Create an in-memory output stream and
18// initial buffer to hold our samples
19ByteArrayOutputStream baos =
20  new ByteArrayOutputStream();
21int frameSizeInBytes =
22  getAudioFormat().getFrameSize();
23int bufferLengthInFrames =
24  targetLine.getBufferSize() / 8;
25int bufferLengthInBytes =
26  bufferLengthInFrames * frameSizeInBytes;
27byte[] data = new byte[bufferLengthInBytes];
```

Listing 8: Recording audio

```
1public void run() {
2  getTargetLine().start();
3
4  while (isRecording()) {
5    int numBytesRead =
6      getTargetLine().read(getData(), 0,
7      getBufferLengthInBytes());
8    if (numBytesRead == -1) {
9      break;
10    }
11    getOutputStream().write(getData(), 0,
12      numBytesRead);
13  }
14  getTargetLine().stop();
15
16  // flush and close the output stream
17  try {
18    getOutputStream().flush();
19    getOutputStream().close();
20  } catch (IOException e) {
21    e.printStackTrace();
22  }
23}
```

Listing 9: Preparing for audio playback

```
1byte[] data = getSampleBytes();
2
3int frameSizeInBytes =
4  getAudioFormat().getFrameSize();
5AudioInputStream audioInputStream =
```

```
6  new AudioInputStream(
7new ByteArrayInputStream(data),
8  getAudioFormat(), data.length /
9  frameSizeInBytes);
10
11try {
12  audioInputStream.mark(2000000000);
13  audioInputStream.reset();
14} catch (IOException e) {
15  e.printStackTrace();
16  return;
17}
18
19long duration = (long)
20  ((audioInputStream.getFrameLength() *
21  1000) / getAudioFormat().getFrameRate());
```

Listing 10: Initializing a SourceDataLine

```
1// Define the required attributes for
2// our line, and make sure a compatible
3// line is supported.
4DataLine.Info dlInfo = new DataLine.Info(
5  SourceDataLine.class, getAudioFormat());
6if (AudioSystem.isLineSupported(dlInfo)
7  == false) {
8  throw new Exception("Line matching " +
9    dlInfo + " not supported.");
10}
11
12getAudioInputStream().reset();
13
14// Get and open the source data line for
15// playback.
16SourceDataLine sourceLine =
17  (SourceDataLine) AudioSystem.getLine(
18  dlInfo);
19int bufSize = 16384;
20sourceLine.open(getAudioFormat(),
21  bufSize);
```

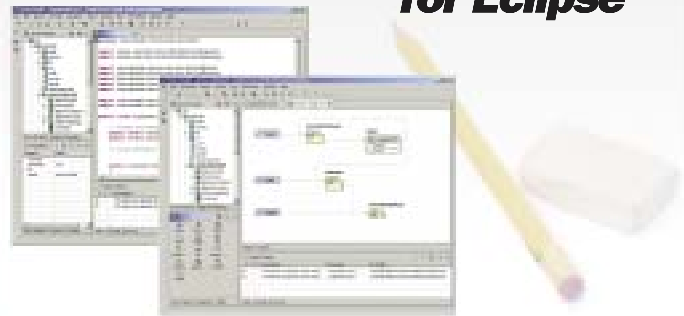
Listing 11: Playing back audio

```
1public synchronized void run() {
2  try {
3    // play back the captured audio data
4    int frameSizeInBytes =
5      getAudioFormat().getFrameSize();
6    int bufferLengthInFrames =
7      getSourceLine().getBufferSize() / 8
8    int bufferLengthInBytes =
9      bufferLengthInFrames *
10      frameSizeInBytes;
11    byte[] data = new byte[
12      bufferLengthInBytes];
13
14    // start the source data line
15    sourceLine.start();
16
17    // main playback loop
18    while (isPlaying()) {
19      // rewind at start of each loop
20      getAudioInputStream().reset();
21      while (true) {
22        int numBytesRead =
23          getAudioInputStream().read(
24          data);
25
26        if (numBytesRead == -1 ||
27          isPlaying() == false) {
28          break;
29        }
30
31        int numBytesRemaining =
32          numBytesRead;
33
34        while (numBytesRemaining > 0) {
35          numBytesRemaining -=
36            sourceLine.write(data, 0,
37            numBytesRemaining);
38        }
39      }
40
41      // We've reached the end of the
42      // stream.  Let the data play out,
43      // then stop and close the line.
44      sourceLine.drain();
45    }
46    sourceLine.stop();
47    sourceLine.close();
48  } catch (LineUnavailableException e) {
49    e.printStackTrace();
50  } catch (IOException e) {
51    e.printStackTrace();
52  } catch (InterruptedException e) {
53    e.printStackTrace();
54  } catch (JStudioException e) {
55    e.printStackTrace();
56  }
57}
```


You can use any tool...

But why not the right one...

Struts Studio for Eclipse



Design, build, test, and deliver Struts Web applications, all with ***Struts Studio 5*** – comprehensive, extensible, and easy-to-use. Struts Studio is the ideal tool for all your Web development projects.

As an Eclipse plug-in, ***Struts Studio 5*** supercharges the Eclipse IDE into a WIDE (Web Integrated Development Environment).

WWW.EXADEL.COM