

Java Music Systems, Didkovsky

JMSL Instruments and JSyn integration

1. Instrument.play() and arrays

Recall that every call to `Instrument.play()` passes in a `playTime`, a `timeStretch`, and one `double[]` array of data to perform, and returns a delayed time (ex. 0.25 second later than the playtime passed in)

JMSL treats the notion of “playing” as a very general one. Playing an event may depend on only one value (turning an electric motor on or off), two values (pitch, velocity), or many values (setting the values of input ports on a JSyn circuit, or sending multiple parameters to external hardware through a native call). For this reason, an Instrument accepts performance data as an array of double, whose length can vary from class to class.

There are two ways of making and populating an array of double. One is a convenience which declares its contents and length at compile time using curly brackets, while the other allocates the array at run time. For example:

```
package misc;

public class ArrayFun {

    public static void main(String args[]) {
        double[] data1 = { 1.0, 2.0, 3.0 };
        data1[2] = 4.0;

        double[] data2 = new double[3]; // could be a variable calc'ed at run time
        data2[0] = 1.0;
        data2[1] = 2.0;
        data2[2] = 3.0;

        double a = 1.0;
        double b = 2.0;
        double c = 3.0;

        double[] data3 = { a, b, c };

        for (int i=0; i< data1.length; i++ ) {
            System.out.println(data1[i] + " " + data2[i] + " " + data3[i]);
        }
    }
}
```

2. Encapsulating JSyn SynthNotes within a JMSL Instrument

JMSL is intended to be a “device independent” compositional language. The output device could be graphical animation, MIDI sounds, robot controls, or JSyn.

The easiest way to play JSyn with JMSL is to use JMSL's `JSynInsFromClassName` which is in the package `com.softsynth.jmsl.jsyn`

`JSynInsFromClassName`'s constructor takes the class name of a `SynthNote` and the

maximum polyphony you want. It rolls this into a JMSL Instrument that plays the SynthNote.

JSynInsFromClassName will play its synthnote with a 4 dimensional DimensionNameSpace (duration, pitch, amplitude, hold). So a simple way to perform it is with a 4-dimensional MusicShape. Note that pitch is “midi-style” 0..127, and fractional pitches like 60.5 (quarter tone above middle C) are OK. Amplitude is 0..1

The most important code is shown here.

```
JMSLMixerContainer mixer = new JMSLMixerContainer();
mixer.start()
// play FilteredSawtoothBL with a max polyphony of 8 voices.
ins = new JSynInsFromClassName(8, "com.softsynth.jsyn.circuits.FilteredSawtoothBL");
mixer.addInstrument(ins);
myShape.setInstrument( ins );
```

3. Playing your JSynInsFromClassName in a MusicShape

We will build a JSynInsFromClassName using FilteredSawtoothBL, set it into a MusicShape, and launch the MusicShape. JMSL's convenient JSynInsFromClassName uses Midi style pitches and velocities (note that fractional pitches are OK). We will now create a tester class that builds a MusicShape and plays it with one of our FilteredSawtoothBL Instruments:

```
/*
 * Created on Mar 24, 2004
 *
 */
package com.didkovsky.javamusic;

/**
 * @author Nick Didkovsky, email: didkovn@mail.rockefeller.edu, (c) 2004 Nick Didkovsky,
 * all rights reserved.
 *
 */

import com.softsynth.jmsl.*;
import com.softsynth.jmsl.jsyn.*;
import com.softsynth.jmsl.view.*;
import java.awt.*;
import java.awt.event.*;

/** Test JSynInsFromClassName by putting it into a MusicShape
 * @author Nick Didkovsky, 4/1/03 10:43PM
 * */

public class TestFilteredSawtoothBLInstrument {

    public static void main(String args[]) {

        Frame myFrame = new Frame();
        myFrame.setLayout(new BorderLayout());
        myFrame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                JMSL.closeMusicDevices();
                System.exit(0);
            }
        });
    }
};
```

```

JSynMusicDevice dev = JSynMusicDevice.instance();
dev.edit(myFrame);
dev.open();

JSynInsFromClassName ins = new JSynInsFromClassName(8,
"com.softsynth.jsyn.circuits.FilteredSawtoothBL");

JMSLMixerContainer mixer = new JMSLMixerContainer();
mixer.start();
mixer.addInstrument(ins);

// Build a MusicShape standard dimension name space:
// duration, pitch, amplitude, hold

MusicShape s = new MusicShape(4);
s.useStandardDimensionNameSpace();
s.add(0.5, 66, 0.35, 1.66);
s.add(0.75, 77, 0.35, 2.66);
s.add(0.25, 48, 0.35, 3.66);
s.add(0.75, 77, 0.35, 1.66);
s.add(0.25, 48, 0.35, 2.66);
// put our instrument into the MusicShape
s.setInstrument(ins);
// make it repeat a long time, and launch it
s.setRepeats(1000);
s.launch(JMSL.now());

// create a graphical editor to play with the shape data in realtime
MusicShapeEditor se = new MusicShapeEditor();
se.addMusicShape(s);

// Build an awt Frame that exits gracefully when closed

// add the MusicShapeEditor and mixer panel to the Frame
myFrame.add(BorderLayout.NORTH, se.getComponent());
myFrame.add(BorderLayout.SOUTH, mixer.getPanAmpControlPanel());
myFrame.pack();
myFrame.setVisible(true);
}
}

```

4. More control over JSyn timbre: SynthNoteAllPortsInstrument

The previous sections showed you how to play SynthNote events by specifying duration, pitch, amplitude, and hold time. There's a lot more to sound than that! For example, FilteredSawtoothBL also has SynthInputs called "cutoff", "Rate", and "resonance". Here we show you how to control all these custom input ports.

A powerful subclass of JSynInsFromClassName, called SynthNoteAllPortsInstrument gives you this control.

When SynthNoteAllPortsInstrument is handed a SynthNote, it "sniffs out" all its input ports and builds a DimensionNameSpace, keeping dimensions 0..3 the standard duration, pitch, amplitude, hold, and assigning dimensions higher than 3 to these custom input ports.

DimensionNameSpace built by SynthNoteAllPortsInstrument for FilteredSawtoothBL:

0	duration
1	pitch
2	amplitude
3	hold
4	cutoff
5	resonance
6	Rate

Note that besides the four common names "duration", "pitch", "amplitude", and "hold", that three additional dimension names show up as well (in bold above). These are unique to FilteredSawtoothBL and were discovered by SynthNoteAllPortsInstrument. Now any MusicShape that has cutoff values in dimension 4, resonance values in dimension 5, and Rate values in dimension 6 can use this instrument to control these timbral parameters. You could add these values by hand like so...

```
// add an event with 1 second duration, pitch 60, amp = 0.5, hold = 0.8 sec,  
// cutoff freq= 1000, resonance = 0.5, Rate = 1.0  
myShape.add(1.0, 60, 0.5, 0.8, 1000.0, 0.5, 1.0);
```

... or generate these values algorithmically as the example below demonstrates.

Here we simply build one of these SynthNoteAllPortsInstruments using FilteredSawtoothBL, and create a MusicShape that will hold data that will affect all input ports. We will use MusicShape.prefab() to generate random data for each dimension. Dimensions 0..3 will be interpreted as standard duration, pitch, amplitude, hold. Higher dimensions will have their values randomly generated between the getLowLimit() and the getHighLimit() for that dimension.

```
/*  
    * Created on Mar 24, 2004  
    *  
    */  
package com.didkovsky.javamusic;  
import com.softsynth.jmsl.*;  
import com.softsynth.jmsl.jsyn.*;  
import com.softsynth.jmsl.view.*;  
import java.awt.*;  
import java.awt.event.*;  
  
/** Test SynthNoteAllPortsInstrument by putting it into a MusicShape  
    *  
    * @author Nick Didkovsky, 4/1/03 10:43PM  
    *  
    */  
public class TestSynthNoteAllPortsInstrument {  
  
    public static void main(String args[]) {  
  
        Frame myFrame = new Frame();  
  
    }  
}
```

```

myFrame.setLayout(new BorderLayout());
myFrame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        JMSL.closeMusicDevices();
        System.exit(0);
    }
});

JSynMusicDevice dev = JSynMusicDevice.instance();
dev.edit(myFrame);
dev.open();

SynthNoteAllPortsInstrument ins =
    new SynthNoteAllPortsInstrument(8,
"com.softsynth.jsyn.circuits.FilteredSawtoothBL");

JMSLMixerContainer mixer = new JMSLMixerContainer();
mixer.start();
mixer.addInstrument(ins);

// Build a music shape with random values for all ports
MusicShape s = new MusicShape(ins.getDimensionNameSpace());
s.prefab(); // generates random data
// put our instrument into the MusicShape
s.setInstrument(ins);
// make it repeat a long time, and launch it
s.setRepeats(1000);
s.launch(JMSL.now());

// create a graphical editor to play with the shape data in realtime
MusicShapeEditor se = new MusicShapeEditor();
se.addMusicShape(s);

// add the MusicShapeEditor and mixer panel to the Frame
myFrame.add(BorderLayout.NORTH, se.getComponent());
myFrame.add(BorderLayout.SOUTH, mixer.getPanAmpControlPanel());
myFrame.pack();
myFrame.setVisible(true);
}
}

```

5. Signal Processing SynthNoteAllPortsInstrumentSP

SynthNoteAllPortsInstrumentSP is a subclass of SynthNoteAllPortsInstrument, which you saw in the previous section. SynthNoteAllPortsInstrumentSP behaves exactly the same way as its superclass by giving you control over all input ports. However, it requires that its SynthNote have a SynthInput called "input".

SynthNoteAllPortsInstrumentSP adds a method with the following signature...

```
public void addSignalSource(Object signalSource);
```

... permitting you to take any JSyn SynthOutput and process it. Playing a MusicShape with a signal processing instrument permits you to change its processing parameters over time, completely independently of the signal sources!

6. Tunings

All JSyn instruments that are subclasses of JMSL's **TunedSynthNoteInstrument** have a Tuning class variable which they use to convert pitch (dimension 1) to a frequency. By default, the tuning is 12 tone equal temperament. Here we show you that JMSL has flexible Tuning classes that let you explore other tunings. JMSL defines an abstract Tuning class with subclasses TuningET and TuningTable

1. Equal Tempered Tuning

An equal tempered tuning is defined by:

- reference frequency (for example 261.62Hz for middle C)
- reference pitch (the pitch to correspond to the reference frequency, say 60 for middle C)
- steps per octave (12 for western European music)
- octave stretch in cents (the number of cents that each octave is stretched beyond a 2:1 ratio)

Code example:

```
TuningET tuning = new TuningET(stepsPerOctave, referenceFrequency,
referencePitch, octaveStretchInCents);
tuning.setOctaveRatio(octaveRatio);
instrument.setTuning(tuning);
```

2. Tuning Table

JMSL also supports Tunings specified by an array of frequencies which define an octave

- reference frequency equals the first element in the frequency array
- reference pitch corresponds to the first frequency in the frequency array
- steps per octave equals the length of the array
- octave stretch in cents is supported

Code example:

```
double[] frequencies = { 274.8, 298.8, 316.6, 330.0, 358.4, 368.2,
421.1, 452.4, 482.6, 533.9 };

TuningTable tuning = new TuningTable(frequencies, referencePitch);
tuning.setOctaveStretchCents(octaveStretchInCents);
instrument.setTuning(tuning);
```

7. Weighted choosers

1. **WeightedIntegerSequence** The WeightedIntegerSequence is a SequenceGenerator whose next() method returns a randomly chosen integer 0..N-1 whose probabilities of being chosen are weighted by an array of length N. So with an array { 1, 1 } would return 0 roughly half the time, and 1 roughly half the time. The array { 2, 0, 1 } would return 0 roughly 2/3 of the time, would never return 1, and would return 2 roughly 1/3 of the time. The array of weights can be accessed and changed at runtime.

Code example

```
double[] weights = { 1, 1, 1 };
WeightedIntegerSequence weightedIntegerSequence = new
WeightedIntegerSequence(weights);
```

2. The **WeightedObjectChooser** supports a `next()` method which returns a randomly chosen Object whose probability of being chosen is weighted. You can

Code example

```
WeightedObjectChooser chooser = new WeightedObjectChooser();
chooser.addWeightedObject(new Object(), 1.0);
chooser.addWeightedObject(new Object(), 2.0);
Object obj = chooser.next();
```

```
public void setWeight(java.lang.Object object, double weight)
public double getWeight(java.lang.Object object)
```

You can have a `WeightedObjectChooser` implement the `Behavior` interface and so, use it to choose a random `Composable` every time a `SequentialCollection` repeats. Note that the `SequentialCollection` itself contains no children (!), as it depends entirely on its `Behavior` to provide it with the next `Child` to launch every time it repeats.

```
class WeightedObjectBehavior extends WeightedObjectChooser implements Behavior
{
    public Composable choose(SequentialCollection col) {
        return (Composable) next();
    }
}
```

8. More about JSyn and JMSL time

In order for JMSL to be able to hand JSyn the time in units JSyn understands, JMSL's clock must be a `SynthClock`. `SynthClock` is defined in `com.softsynth.jmsl.jsyn`. The following is done for you when you open a `JSynMusicDevice`:

```
JMSL.clock = new com.softsynth.jmsl.jsyn.SynthClock();
```

The clock is used to convert JMSL time to native JSyn time. The following code converts an absolute JMSL time to a JSyn time, and sets a JSyn port at that time.

```
int itime = (int) JMSL.clock.timeToNative(playTime);
myOsc.frequency.set( itime, 440.0 );
```

To convert a JMSL duration to native JSyn duration, use `JMSL clock's getNativeRate()` like so:

```
double someDur = 1.6; // seconds
int iDur = (int) (JMSL.clock.getNativeRate() * someDur); // JSyn ticks
```

When is time conversion necessary? Instruments with arbitrary `SynthInput` values like `SynthNoteAllPortsInstrument`, which does all this for you in its `play()` method!

If your instrument design does not fall into the classification of duration, pitch, amplitude, hold, upper synthsports... you will want to do custom work in your instrument's play() method. And to do that on time, you must convert JMSL time passed in to play() to JSyn time. Another use would be to convert incoming MIDI timestamps to JSyn timestamps so that MIDI could be used to control JSyn.

9. Interpreters

JMSL's Interpreter class adds another degree of flexibility to the interpretation of data. Rather than overriding play() in a subclass of Instrument, you could instead use a stock InstrumentAdapter, and simply setInterpreter(yourInterpreterSubclass);

Defining your own Interpreter is easy. Just extend Interpreter and override interpret()

```
public double interpret(double playTime,
                       double timeStretch,
                       double dar[],
                       Instrument ins);
```

Notice that interpret has access to the Instrument that called it, to grab a Midi channel for example.

Why would you ever do this when you can just as easily override Instrument.play() ? For one thing, you cannot override play() at runtime. An Instrument's play() method will always behave the same way once its overridden. An Interpreter, on the other hand, can be plugged into an Instrument on the fly. So a JSyn FMSynth interpreter may be swapped out for a JSyn Waveshaping interpreter, without the Instrument itself knowing about the change.

Another trick might be to have different Instruments who all sound dramatically different (various Instrument subclasses that play various JSyn circuits for example), all share the same interpreter that performs data-driven animation.

10. Scheduling future events without an event buffered sound engine

JSyn's event buffer is a very tight scheduler, which provides us with methods like noteOnFor(int playTime, int hildTime, double freq, double amp) ...which are guaranteed to fire on time. Later you will see that MidiShare allows you to similarly schedule future Midi events with great accuracy.

But what about non-event-buffered output like painting ? JMSL's notation package for example uses EventScheduler to manage the flashing of notes as music plays back. JMSL's EventScheduler allows one to schedule arbitrary events in the future. EventScheduler allows one to schedule any class that implements the ScheduledEvent interface. This interface requires the implementation of two methods:

```
public double getPlayTime()
    Returns:
        time that the event should occur.

public void play()
    Called by EventScheduler to cause event to happen.
```

Here's an example of a class that simply prints a custom message at some future time. It also prints the playtime and the error between the expected scheduling time and the actual time it fires. Note that unlike Instrument's play() method, this play() does not return an updated playTime. We don't want to schedule events according to delays caused by noteOff events!

```
package com.didkovsky.javamusic;

import com.softsynth.jmsl.*;

public class EventExample implements ScheduledEvent {

    private String msg;
    private double playTime;

    public EventExample(double playTime, String msg) {
        this.msg = msg;
        this.playTime = playTime;
    }

    public double getPlayTime() {
        return playTime;
    }

    public void play() {
        JMSL.out.println(msg + ", playTime: " + playTime + ", error: " + (JMSL.now() -
playTime));
    }

    public static void main(String args[]) {
        EventScheduler scheduler = new EventScheduler();
        scheduler.post(new EventExample(JMSL.now() + 4.0, "Ho there"));
        scheduler.post(new EventExample(JMSL.now() + 3.0, "Hi there"));
        scheduler.post(new EventExample(JMSL.now() + 2.0, "Hey there"));
    }
}
```

Note also:

You need a new EventExample for every event.

While this toy example simply has a String as private data, you could rewrite it to do what you like.

Note also:

EventScheduler uses Object.wait() and Object.wait(timeout) to schedule events, and so, will not wakeup with the same accuracy as JSyn and MidiShare events. However, you can be assured that the design of EventScheduler will not permit errors to accumulate.

11.Using MusicJob to perform with an Instrument.

Instruments themselves are not scheduled. Only Composables can be scheduled in JMSL. Instruments return updated playTimes to Composables who care to schedule themselves with an Instrument's interpretation of data. One such Composable is MusicJob, with which you are already intimately familiar.

Let's subclass a MusicJob which generates double[] data on the fly, using some 1/F generators. Its repeat duration will be scheduled by the updated playTime returned by the

Instrument. We will plug in one of our PrintingInstruments defined at the beginning of this chapter.

```
package com.didkovsky.javamusic;

import com.softsynth.jmsl.*;
import com.softsynth.jmsl.util.Oof;

public class PrintingMusicJob extends MusicJob {

    double[] insData; // reusable array, stuffed with data on the fly
    Oof pitchGenerator; // 1/F sequence
    Oof velocityGenerator; // 1/F sequence
    Oof durationGenerator; // 1/F sequence, scaled to discrete durations

    public PrintingMusicJob() {
        super(); // important!
        setInstrument(new PrintingInstrument());
        insData = new double[3];
        pitchGenerator = new Oof(8); // 8 bit data -> 0..127
        pitchGenerator.randomize();
        velocityGenerator = new Oof(8); // 8 bit data -> 0..127
        velocityGenerator.randomize();
        durationGenerator = new Oof(3); // 3 bit data -> 0..7
        durationGenerator.randomize();
    }

    public double repeat(double playTime) {
        // generate some data
        int numThirtySeconds = (int) durationGenerator.next() + 1; // 1..8
        JMSL.out.println("32nds: " + numThirtySeconds);
        double dur = numThirtySeconds * 0.125;
        insData[0] = dur;
        insData[1] = pitchGenerator.next();
        insData[2] = velocityGenerator.next();
        return getInstrument().play(playTime, 1.0, insData);
    }

    public static void main(String args[]) {
        PrintingMusicJob job = new PrintingMusicJob();
        job.setRepeats(1000);
        job.launch(JMSL.now());

        // provide a Frame just to click and System.exit()
        java.awt.Frame f = new java.awt.Frame("Click to shut down");
        f.addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                System.exit(0);
            }
        });
        f.setSize(320, 200);
        f.setVisible(true);
    }
}
```

Note that the above example should not give you the impression that you may call `getInstrument.play()` only once in `musicJob.repeat()`. You could call it multiple times and return the maximum playtime returned by these numerous calls. Or you may accumulate the updated playtimes returned by each call to `instrument.play()` and return the accumulated time (how would the difference between these two approaches be heard?)

```
// the following example will schedule numerous instrument.play()'s where each new
// instrument.play() begins after the previous play(), and returns the accumulated time
public double repeat(double playTime) {
```

```

Instrument ins = getInstrument();
if (ins != null) {
    for (int i=0; i <NUMBER_OF_SOUND_EVENTS_YOU_WANT; i++) {
        double[] data = generateDataSomehow();
        playTime += ins.play(playTime, 1.0, data);
    }
}
return playTime;
}

// the following example will schedule each new instrument.play() at playTime and return
// the maximum delay caused by a single event
public double repeat(double playTime) {
    Instrument ins = getInstrument();
    if (ins != null) {
        double maxTime = playTime;
        for (int i=0; i <NUMBER_OF_SOUND_EVENTS_YOU_WANT; i++) {
            double[] data = generateDataSomehow();
            double timestamp = ins.play(playTime, 1.0, data);
            maxTime = Math.max(maxTime, timestamp)
        }
        playTime = maxTime;
    }
    return playTime;
}

```

HOMEWORK ASSIGNMENT (Extra Credit Only)

Nick Didkovsky's MandelMusic sonifies the Mandelbrot Set by mapping 5 values encountered during its iteration path to JSyn circuit parameters. For example, it may map x position to pitch, y to amplitude, etc... Any JMSL class that inherits from MandelInstrument can be added to MandelMusic's drop down list of instruments the user can play with.

A MandelInstrument's play() method is handed an array of five doubles. The interpretation is done when you override your MandelInstrument's play() method:

```
public double play(double playTime, double timeStretch, double dar[])
```

The double values in dar[] follow this convention:

```
dar[0] = az, real part of current iteration point, normalized to 0.0 .. 1.0
dar[1] = bz, imaginary part of current iteration point, normalized to 0.0 .. 1.0
dar[2] = magnitude, distance of point from 0+0i, normalized to 0.0 .. 1.0
dar[3] = distanceTravelled, normalized to 0.0 .. 1.0
dar[4] = iterationCount (how many times the  $z=z^2+c$  function has been iterated: 1..1000)
```

Your job is as follows:

1) Design any JSyn circuit of your own choosing (use Wire if you like!!!). This circuit should have 5 input ports whose values change the sound of the circuit significantly.

2) Design a subclass of MandelInstrument whose play() method passes values from the double[] to the circuit in some creative way to the five ports of your circuit. Since you will need to subclass MandelInstrument, and since the source is not available (and not needed), you may simply define it on your machine like so:

```
package MandelMusic;
import com.softsynth.jmsl.*;
public class MandelInstrument extends InstrumentAdapter { }
```

Now your task is relatively easy. Depending on your JSyn circuit you will fill in the body of a method like so:

```
public double play(double playTime, double timeStretch, double dar[]) {
    int itime = JMSL.clock.timeToNative(playTime);

    // Do whatever you want here with the data in dar[] to make noise
    // you will need to scale the values in the array to ranges that make sense for
    // your circuit

    // finish with the following three lines of code so all MandelInstruments
interpret
    // duration the same way. Then your instrument will be in synch with the
    // line drawing instruments
    double distance = dar[3]; // distance travelled
    double duration = Math.max(distance, 0.01); // Don't want durations faster
than 0.01
    return playTime + duration*timeStretch;
}
```

3) Finally, test your MandelInstrument's behavior in an applet (source provided below) which provides five TextFields into which the user can plug numbers in the ranges indicated above. The applet bundles these up into an array when you click "play" button, and fires the MandelInstrument's play() method. Post the applet and source on your web site. Make sure your MandelInstrument subclass is in a package following our usual convention.

If all goes well, I should be able (with your permission) to add your instrument to MandelMusic.

Source for this Applet, a Ring Modulating JSyn circuit built in Wire (later tweaked by hand), and a MandelInstrument subclass that uses this circuit follows.

```
package com.didkovsky.javamusic;

import com.softsynth.jmsl.*;
import com.softsynth.jmsl.jsyn.*;
import com.softsynth.jsyn.*;
import java.awt.*;
import java.awt.event.*;
import MandelMusic.MandelInstrument;

/** Test your MandelInstrument
 * @author Nick Didkovsky, copyright 2000 Nick Didkovsky
 */
```

```

public class MandelInstrumentTester extends java.applet.Applet implements
ActionListener {

    Button playButton;
    Button randomButton;
    TextField azField;
    TextField bzField;
    TextField magField;
    TextField distField;
    TextField iterField;
    MandelInstrument myMandelInstrument;
    double[] dar;

    public void init() {
        setLayout(new GridLayout(0, 2));
        add(new Label("az (0..1)"));
        add(azField = new TextField("0.00"));
        add(new Label("bz (0..1)"));
        add(bzField = new TextField("0.00"));
        add(new Label("magnitude (0..1)"));
        add(magField = new TextField("0.00"));
        add(new Label("distance travelled (0..1)"));
        add(distField = new TextField("0.00"));
        add(new Label("iteration count (0..1000)"));
        add(iterField = new TextField("0"));
        add(randomButton = new Button("Random Values"));
        add(playButton = new Button("Play"));
        randomButton.addActionListener(this);
        playButton.addActionListener(this);
        dar = new double[5];
    }

    public void start() {
        Synth.startEngine(0);
        JMSL.clock = new SynthClock();
        myMandelInstrument = new RingBoyMandel(); // Change to yours!!!
        try {
            myMandelInstrument.open(JMSL.now());
        }
        catch(InterruptedException e) {}
    }

    public void stop() {
        try {
            myMandelInstrument.close(JMSL.now());
        }
        catch(InterruptedException e) {}
        Synth.stopEngine();
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == playButton) handlePlay();
        if (source == randomButton) handleRandom();
    }

    void handlePlay() {
        dar[0] = (new Double(azField.getText())).doubleValue();
        dar[1] = (new Double(bzField.getText())).doubleValue();
        dar[2] = (new Double(magField.getText())).doubleValue();
        dar[3] = (new Double(distField.getText())).doubleValue();
        dar[4] = (new Double(iterField.getText())).doubleValue();
        if (myMandelInstrument != null) {
            myMandelInstrument.play(JMSL.now(), 1.0, dar);
        }
    }
}

```

```

void handleRandom() {
    azField.setText(JMSLRandom.choose() + "");
    bzField.setText(JMSLRandom.choose() + "");
    magField.setText(JMSLRandom.choose() + "");
    distField.setText(JMSLRandom.choose() + "");
    iterField.setText(JMSLRandom.choose(1000) + "");
}

public static void main(String args[]) {
    JMSLRandom.randomize();
    MandelInstrumentTester applet = new MandelInstrumentTester();
    Frame f = new Frame("MandelInstrument tester");
    f.addWindowListener(
        new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                System.exit(0);
            }
        });
    f.add(applet, BorderLayout.CENTER);
    applet.init();
    applet.start();
    f.setSize(400,200);
    f.setVisible(true);
}
}

// ***** SYNTH CIRCUIT EXAMPLE (BUILT IN WIRE) *****

package com.didkovsky.javamusic;
import com.softsynth.jsyn.*;
/** Automatically generated source code.
** Do NOT edit unless you copy to another directory and change the name.
tweaked by Nick Didkovsky 11/6/0 12:38PM */
public class RingBoy extends SynthNote
{
    // Declare units and ports.
    com.softsynth.jsyn.SineOscillator sinOsc1;
    com.softsynth.jsyn.SineOscillator sinOsc2;
    com.softsynth.jsyn.MultiplyUnit mult1;
    com.softsynth.jsyn.PanUnit pan1;
    public com.softsynth.jsyn.SynthInput pan;
    com.softsynth.jsyn.EnvelopePlayer envPlay1;
    com.softsynth.jsyn.EnvelopePlayer envPlay2;
    SynthEnvelope enveloep1;
    SynthEnvelope enveloep2;
    public com.softsynth.jsyn.SynthInput freqA;
    public com.softsynth.jsyn.SynthInput freqB;
    public com.softsynth.jsyn.SynthInput ampA;
    public com.softsynth.jsyn.SynthInput ampB;

    public RingBoy() {
        // Create unit generators.
        add( sinOsc1 = new com.softsynth.jsyn.SineOscillator() );
        add( sinOsc2 = new com.softsynth.jsyn.SineOscillator() );
        add( mult1 = new com.softsynth.jsyn.MultiplyUnit() );
        add( pan1 = new com.softsynth.jsyn.PanUnit() );
        add( envPlay1 = new com.softsynth.jsyn.EnvelopePlayer() );
        add( envPlay2 = new com.softsynth.jsyn.EnvelopePlayer() );
        double[] enveloep1Data = {
            0.25484633770103227, 0.9957446808510638,
            0.0, 0.9744680851063829,
            0.3486075074637571, 0.13191489361702127,
            0.6322568993741555, 0.1276595744680851,
            0.09483018676714483, 0.0,
            0.005, 0.0,
        };
        enveloep1 = new SynthEnvelope( enveloep1Data );
    }
}

```

```

envelope1Data = null;
envelope1.setSustainLoop( 2, 5 );
envelope1.setReleaseLoop( 5, 6 );
double[] envelope2Data = {
    0.022033440091855386, 0.3574468085106383,
    0.06353415754961451, 0.9957446808510638,
    0.08472425198273545, 0.3404255319148936,
    0.33889700793094196, 0.3404255319148936,
    0.2511658023155797, 0.0,
    0.005, 0.0,
};
envelope2 = new SynthEnvelope( envelope2Data );
envelope2Data = null;
envelope2.setSustainLoop( 2, 4 );
envelope2.setReleaseLoop( 5, 6 );
// Connect units and ports.
sinOsc1.output.connect( 0, mult1.inputA, 0 );
sinOsc2.output.connect( 0, mult1.inputB, 0 );
mult1.output.connect( 0, pan1.input, 0 );
addPort( pan = pan1.pan, "pan" );
pan.setup( -1.0, 0.472, 1.0 );
addPort( output = pan1.output );
envPlay1.output.connect( 0, sinOsc1.amplitude, 0 );
envPlay2.output.connect( 0, sinOsc2.amplitude, 0 );
addPort( freqA = sinOsc1.frequency, "freqA" );
freqA.setup( 10.0, 309.0, 1000.0 );
addPort( freqB = sinOsc2.frequency, "freqB" );
freqB.setup( 10.0, 54.5, 1000.0 );
addPort( ampA = envPlay1.amplitude, "ampA" );
ampA.setup( 0.0, 0.0, 1.0 );
addPort( ampB = envPlay2.amplitude, "ampB" );
ampB.setup( 0.0, 0.0, 1.0 );
}

public void setStage( int time, int stage ) {
    switch( stage )
    {
    case 0:
        // stop( time );
        envPlay1.envelopePort.clear( time );
        envPlay1.envelopePort.queueOn( time, envelope1 );
        envPlay2.envelopePort.clear( time );
        envPlay2.envelopePort.queueOn( time, envelope2 );
        // start( time );
        break;
    case 1:
        envPlay1.envelopePort.queueOff( time, envelope1 );
        envPlay2.envelopePort.queueOff( time, envelope2 );
        break;
    case 2: // added by hand
        envPlay1.envelopePort.clear( time );
        envPlay1.envelopePort.queue( time, envelope1, envelope1.getNumFrames() -
1, 1 );

        envPlay2.envelopePort.clear( time );
        envPlay2.envelopePort.queue( time, envelope2,
envelope2.getNumFrames() - 1, 1 );
        break;
    default:
        break;
    }
}
}

// ***** MANDELINSTRUMENT EXAMPLE *****
package com.didkovsky.javamusic;
import MandelMusic.MandelInstrument;
import com.softsynth.jsyn.*;
import com.softsynth.jmsl.*;

```

```

public class RingBoyMandel extends MandelInstrument {

    LineOut out;
    RingBoy circuit;

    public RingBoyMandel() {
        out = new LineOut();
        circuit = new RingBoy();
        circuit.output.connect(0, out.input, 0);
        circuit.output.connect(1, out.input, 1);
    }

    public double open(double startTime) {
        int itime = (int)JMSL.clock.timeToNative(startTime);
        out.start(itime);
        circuit.start(itime);
        // System.out.println("OPENING");
        return startTime;
    }

    public double close(double stopTime) {
        int itime = (int)JMSL.clock.timeToNative(stopTime);
        out.stop(itime);
        circuit.stop(itime);
        return stopTime;
    }

    public double play(double playTime, double timeStretch, double[] dar) {
        double duration = Math.max(dar[3], 0.01); // No durations faster than 0.01
        int itime = (int)JMSL.clock.timeToNative(playTime);
        int reltime=(int) (JMSL.clock.getNativeRate()*duration*timeStretch * 0.5);
        int hardreltime=(int) (JMSL.clock.getNativeRate()*duration*timeStretch*0.9);
        double freq1 = dar[0] * 1000;
        double freq2 = dar[1] * 1000;
        double amp1 = dar[2];
        double amp2 = dar[3];
        double pan = (dar[4] / 500.0) - 1.0;
        // System.out.println(pan + " ");

        circuit.freqA.set(itime, freq1);
        circuit.freqB.set(itime, freq2);
        circuit.ampA.set(itime, Math.max(0.5, amp1));
        circuit.ampB.set(itime, Math.max(0.5, amp2));
        circuit.pan.set(itime, pan);
        circuit.setStage(itime, 0);
        circuit.setStage(itime + reltime, 1);
        circuit.setStage(itime + hardreltime, 2);

        return playTime + duration * timeStretch;
    }
}

```