

Java Music Systems, Didkovsky

Introduction to JMSL

Then: Hierarchical Music Specification Language (HMSL)

Hierarchical Music Specification Language (HMSL) is a superset of the Forth programming language, and was designed for experimental music composition and performance. An open-ended, extensible music language, "there is no such thing as a typical HMSL piece, just as there is no such thing as a typical English novel or French poem." (HMSL article by ND in Ear Magazine). HMSL was co-authored by Phil Burk, Larry Polansky, and David Rosenboom at the Mills College Center for Contemporary Music.

HMSL runs on the Mac, and the Amiga (the machine that refuses to die). The Amiga version of HMSL can access this computer's audio hardware. HMSL can play any prerecorded Amiga audio sample. However, a number of pieces have been written which create and dynamically change their own Amiga audio samples (Robert Marsanyi's graduate recital composition at Mills used chaotic formulae to generate waveforms on-the-fly). MIDI and Amiga local sound are but two of the many ways that HMSL can interface to the outside world. David Mahler wrote an HMSL piece that controlled a variety of bells which are played by electronic solenoids activated via HMSL's VDI (Virtual Device Interface). Nick Didkovsky wrote a piece for sculptor Sara Garden Armstrong which controlled turning industrial air blowers on/off, which in turn created soundscapes and caused her paper sculptures to move.

HMSL is distributed by Frog Peak Music <http://www.frogpeak.org/>

Other HMSL-related links:

<http://www.softsynth.com/>

<http://www.doctornerve.org/nerve/pages/scan.shtml>

<http://www.doctornerve.org/nerve/pages/lottery/lottery.shtml>

<http://www.doctornerve.org/nerve/pages/hmslear.shtml>

Now: Java Music Specification Language (JMSL)

The key ideas in HMSL began to be ported to Java by Nick Didkovsky in 1997.

"In the late summer of 1997, I was reflecting on the uncertain future of the computer music pieces I've written over the years in the language HMSL, which runs on the Amiga computer. While my Amiga still faithfully runs my HMSL software, being committed to this platform has always been charged with worries concerning survival in a Microsoft Windows dominated world.

"About this same time, I became very interested in the programming language Java. Created and distributed freely by Sun Microsystems, Java promises to be a platform-independent language. That is, a program written in Java is supposed to run identically on a number of platforms for which Java "virtual machines" have been written: Amiga, Windows 95, MacIntosh, Silicon Graphics, etc. The significance of this hit me very hard. Writing for a "virtual" computer that crosses platform boundaries is understandably attractive to a software designer who has seen his favorite "real" computer gradually squeezed out of the mainstream market.

"In September of 1997, I began porting HMSL to Java. As a kind of early test flight, I built my first Java/HMSL piece, a real-time statistical deconstruction of a Schubert Impromptu. Since the realization of this work, I have been collaborating with HMSL co-author Phil Burk on a Java based successor to HMSL. Dubbed JMSL, this new computer music environment offers tools to create arbitrarily complex multi-user musical environments, using the Internet as a network, as well as non-networked, stand-alone works like the Impromptu performed here. Under JMSL, which because of its Java roots is both platform-independent and Web-capable, the hardware distinctions separating computer musicians into technological camps will mean as little as the political and geographical boundaries separating them." (Artist's Statement, Circuits Conference, March 28, 1998)

After the premiere of the Schubert JMSL piece, Didkovsky sent his source code to HMSL designer/programmer Phil Burk, who suggested and implemented extensive redesign. Burk both simplified

and clarified JMSL, taking it beyond a simple port of HMSL from Forth to Java. Since then Didkovsky has been primarily responsible for JMSL's evolution, adding a power notation package, for example. Robert Marsanyi has also contributed a portable MIDI I/O package to JMSL, which allows MIDI events to be read and written identically on both Mac and PC implementations.

The Notion of Hierarchies

A hierarchy is a tree-like network of parent/child relationships. A hierarchy can be very deep, that is, each child can in turn be a parent to other children, and very broad, that is, the number of children a parent can have is not limited.

A symphony orchestra can be thought of as a hierarchy where the conductor is the "parent". Below the conductor are the first hierarchy of children: the string section, percussion section, etc. Each of these sections, in turn, contains individual children musicians. A performance is executed by a "start command" from the conductor. Members of the string section, might in turn look to the first violinist for further scheduling details (ie to synchronize the minute timing of bow strokes). In this way, a musical "schedule" is passed down the hierarchy from conductor, to string section, to its individual members.

JMSL, like its Forth predecessor HMSL, builds a music system by putting intelligent musical objects into a hierarchy. JMSL hierarchies can be arbitrarily complex - far more complex than the orchestra example. A JMSL hierarchy must pass scheduling information down through all its children, as well as back up (ie children must inform their parents when their activity is completed). For example, a JMSL hierarchy might be a sequential collection (parent) containing a verse and a chorus (children). The verse starts when the parent collection tells it to. The verse must inform the parent collection when it is done playing, so that the parent collection can then command the chorus to begin. Thus, scheduling information needs to be a two-way street.

Classes Central to JMSL:

Playable

A very basic interface, which simply defines one method:

```
public double play( double time, Composable parent )
```

Implementations of play() receive the time they are scheduled, and return the updated time at the end of their task. Later we will see that we can define our own Playables and have them scheduled during the lifetime of a MusicJob.

Composable extends Playable

Any object that supports the Composable interface can be placed in a JMSL hierarchy. A class that implements Composable passes its incoming time to its children, and passes its completion time up to its parents. In this way, complex hierarchies can be scheduled. Composable adds methods to the Playable interface like the following.

```
void finishAll ()  
launch(double time, Composable parent)
```

...which upon inspection you can guess are helpful in starting and finishing execution, and imply parent/child relationships.

MusicJob

A MusicJob is very important. It is the first JMSL class that implements Composable. All other stock JMSL Composables extend MusicJob. MusicJob performs some user-defined activity on a repeated schedule. This activity may be printing a message to System.out, sending a MIDI note, calling setStage(0) on a SynthNote, drawing or updating an image, polling a sensing device, sampling data at regular intervals, etc. This action could include changing its own pause between repeats, with a call to setRepeatPause().

You can add functionality to a MusicJob by overriding its start(), repeat() and stop() methods. For example, the following repeat() method prints a message every time this MusicJob repeats:

```
// ex 1
```

```

public double repeat(double playtime) {
    System.out.println(getName() + "is repeating and the time is" + playtime);
    return playtime;
}

```

Note that the incoming playtime is being returned unchanged. This will cause this repeat() method to run as fast as possible. The following modification would cause it to pause for one second between repeats:

```

//ex 2
public double repeat(double playtime) {
    System.out.println(getName() + "is repeating and the time is" + playtime);
    return playtime + 1;
}

```

Another way of achieving this pause between repeats is by using setRepeatPause(). If you called myMusicJob.setRepeatPause(1.0) with example 1 above; it would have the same effect as ex. 2.

Q: What would happen if you called myMusicJob.setRepeatPause(1.0) with example 2?

A: It would pause two seconds between repeats.

Note that a MusicJob will only repeat once unless you call myMusicJob.setRepeats(100), for example.

But overriding repeat() is just one of the ways you can program a MusicJob's lifetime. Let's break down a MusicJob's schedule now:

MusicJob timeline (time delays enclosed in []):

```

[startDelay]
start()
callPlayables(startPlayables) (optional)
[startPause()]
<internalRepeat()> ( does nothing, overridden in subclasses of MusicJob )
repeat()
callPlayables(repeatPlayables) (optional)
[repeatPause ( )][loop back to repeat(), getRepeats() times]
[stopDelay()]
callPlayables(stopPlayables) (optional)
stop()

```

You can affect these pauses with methods setStartDelay(), setStartPause(), setRepeatPause(), and setStopDelay().

Note the reference to callPlayables. At these moments in a MusicJob's lifetime, you can insert your own Playables and they will be play()'ed. For example, examine the following code fragment

```

myMusicJob.addStartPlayable(new Playable() {
    public double play(double playTime, Composable parent) {
        System.out.println("I am a playable playing at " + playtime);
        return playtime;
    }
});

```

...then myMusicJob would call the play() method you defined above right after its start() method was called.

Why use addXXXPlayable() instead of overriding start(), repeat(), and stop() ? For one thing, Playables can be added at runtime, while you cannot redefine start() repeat(),and stop() at runtime. For example, you might have a Vector of Playables, and every time the MusicJob repeated, it might choose one to addRepeatPlayable(myRandomlyChosenPlayable)

To launch a MusicJob, call its launch method, passing in a time stamp.

```
myMusicJob.launch(JMSL.now());
```

Read more about JMSL.now() below in the section entitled “JMSL clock and advance time.”

ParallelCollection

Composables can contain other Composables, which allows a hierarchy to be created. A Parallel Collection launches all its children at the same time. Like MusicJobs which they extend, ParallelCollections have start, repeat, and stop methods, listeners, and delays.

The following code fragment will create a Parallel Collection with two children. When launched, it will simultaneously launch its two children, then wait five seconds, then launch its two children simultaneously again, then pause 5 seconds again ... repeating this 4 times. This ability to set repeats and repeat pauses does not surprise use because ParallelCollection extends MusicJob, and so, gains all the flexibility and the same timeline breakdown as MusicJob. This means you could addRepeatPlayable() to a ParallelCollection and have it rearrange, add, and delete children at run time! Hierarchical scheduling is implemented very flexibly in JMSL.

```
ParallelCollection col = new ParallelCollection();
col.add(musicJob1);
col.add(musicJob2);
col.setRepeats(4);
col.setRepeatPause(5);
col.launch(JMSL.now());
```

SequentialCollection

A SequentialCollection launches its children one after the other in sequence, waiting for each to finish before launching the next. The following code fragment creates a Sequential Collection which will first launch musicJob1, followed by musicJob2, followed by a 5 second pause, repeating this sequence 4 times.

```
SequentialCollection col = new SequentialCollection ();
col.add(musicJob1);
col.add(musicJob2);
col.setRepeats(4);
col.setRepeatPause(5);
col.launch(JMSL.now());
```

SequentialCollection can alternatively employ a user-defined behavior to choose which child to launch every repeat. The Behavior interface defines a choose() method that returns the next Composable to execute. It is up to the user to define how this is done. JMSL offers a stock UniformRandomBehavior which uses a uniform random distribution to choose the next child. Source shown below:

```
public class UniformRandomBehavior implements Behavior, Serializable {
    /** Choose a child to launch with a uniformly distributed random selection.
    @return next Composable child to launch.*/
    public Composable choose(SequentialCollection col) {
        return (Composable)col.get(JMSLRandom.choose(col.size()));
    }
}
```

To make a SequentialCollection act behaviorally, you need to define a class that implements the Behavior interface (or use the prefabricated UniformRandomBehavior), plug it in with setBehavior(Behavior b). To act sequentially again, simply call setBehavior(null).

The following code fragment illustrates. It creates a Sequential Collection which repeats 4 times, pausing 5 seconds each time. Every repeat, it will choose one of its two children to launch, uniformly randomly.

```
SequentialCollection col = new SequentialCollection ();
col.setBehavior(new UniformRandomBehavior());
col.add(musicJob1);
col.add(musicJob2);
col.setRepeats(4);
col.setRepeatPause(5);
col.launch(JMSL.now());
```

IMPORTANT: Since ParallelCollection and SequentialCollection implement Composable, they can be children of other Parallel and Sequential Collections and they can contain other Parallel and SequentialCollections! For example, the following code fragment puts a ParallelCollection and a MusicJob in a SequentialCollection

```
SequentialCollection colSeq = new SequentialCollection ();

ParallelCollection colPar = new ParallelCollection();
colPar.add(musicJob1);
colPar.add(musicJob2);
colPar.setRepeats(4);

colSeq.add(colPar);
colSeq.add(musicJob3);

colSeq.setRepeatPause(5);
colSeq.launch(JMSL.now());
```

See also <http://www.algomusic.com/jmsl/examples/BehavioralCollectionDemo.html>

JMSL clock and “advance time”

JMSL uses Java Threads to schedule events. Unfortunately, Java Threads do not wake up exactly when we want them to. You might ask a Thread to sleep for 1000 milliseconds but it may wake up after 995 millis, or after 1050 millis. But we want JMSL to be able to play rhythmically accurate music. So, to absorb the uncertainties in Java Thread scheduling, JMSL uses the notion of “advance time”. Advance Time is like a buffer which absorbs these inaccuracies. It’s like deciding to arrive 5 minutes early for a noon appointment. Your travel time might vary: you might show up 4 minutes early or 6 minutes early, but that variation is absorbed by the 5 minute advance time, and you’ll be on time for your appointment. That five minute window is like Advance Time.

Let’s frame that as a JMSL example. If the advance time is 0.1 seconds, and a MusicJob is launched with a time stamp that is in the future by this advance time, then when the MusicJob wakes up to repeat(), the playTime parameter passed in to repeat() is actually about 1/10th second in the future. The MusicJob might wake up a few millis early or a few millis late. But as long as it does not wake up more than 0.1 second late, the playTime will be in the near future. Then, if the MusicJob uses this playtime to send a timestamped message to JSyn or MIDIShare, the sounds will be heard with rhythmic precision. This is because both JSyn and MIDIShare support highly accurate timestamped scheduling of their sound engines.

To set JMSL’s advance time make a call to:

```
JMSL.clock.setAdvance(0.1);
```

Then, to launch a Composable use **JMSL.now()** which is the true system time PLUS the advance time:
myMusicJob.launch(JMSL.now());

JMSL.realTime() returns the actual clock time (ie JMSL.now() = JMSL.realTime() + timeAdvance). So you might find it interesting to compare the real time against the scheduled time in a MusicJob’s repeat method:

```
public double repeat(double playtime) {  
    System.out.println(getName() + "is repeating and the time is" + playtime);  
    double diff = playtime - JMSL.realTime();  
    System.out.println("Difference between real time and playtime is " + diff);  
    return playtime;  
}
```

Q: What does it mean if `diff` in the example above is a negative number?

A: ?

Some useful stochastic tools

JMSLRandom

This is a useful class for generating random numbers. The `choose()` method is a heavily over-loaded method which behaves differently according to its arguments.

choose() with no arguments returns a random double from [0..1.0)

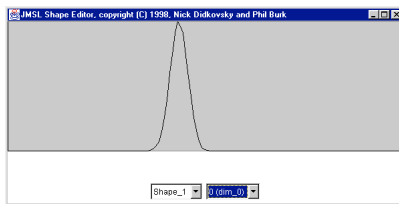
choose(int high) returns a random int [0..high) (analogous for double)

choose(int low, int high) returns a random int [low..high) (analogous for double)

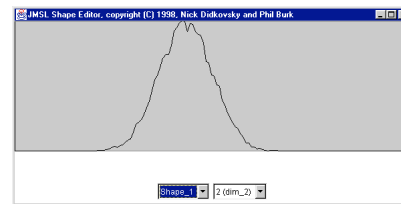
Other useful random number generators in the JMSLRandom class:

choosePlusMinus(int range) returns a random integer between +range and -range, inclusive.

gauss(double sigma, double xmu) returns a Gaussian distributed double. `sigma` controls narrowness of bell, `xmu` controls the centering of bell. Small sigmas make tight narrow bells.



sigma = 4



sigma=12

The effect of sigma on Shapes loaded with `jRandom.gauss(double sigma, double xmu)`

Event Distributions

JMSL offers logarithmic and Myhill distributions. These distributions can be used for placing events over time, for example.

EXPONENTIAL DISTRIBUTION

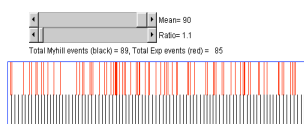
This gives you lots of short entry delays with an occasional long one. A nice balance, but with no control over standard deviation from the mean event density.

MYHILL DISTRIBUTION

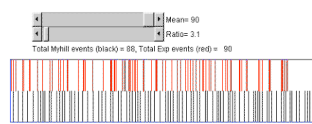
The Myhill distribution can be thought of as a more flexible exponential distribution; one whose variance can be controlled. This control is reflected in a ratio whose value causes the Myhill distribution to more or less approximate the exponential distribution. When the ratio is high (above 128), the two distributions are indistinguishable. When ratio is low, the Myhill transform generates increasingly regular, periodic patterns, as opposed to the clustering and leaps of the exponential transform.

Myhill distribution and additional comments from "A Catalog of Statistical Distributions" by Charles Ames. Leonardo Music Journal, Vol. 1 No. 1, 1991

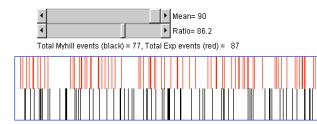
The applet pictured below allows the user to control the mean for both distributions, and another slider to control the Myhill ratio. Events distributed logarithmically are depicted on the top half of the rectangular display area. Myhill distributed events are in the bottom half.



A low Myhill ratio (1.1) distributes 90 events with regularity.



A higher Myhill ratio (3.1) distributes events with some "drunkenness".



A high Myhill ratio (86.2) approaches a distribution almost indistinguishable from logarithmic.

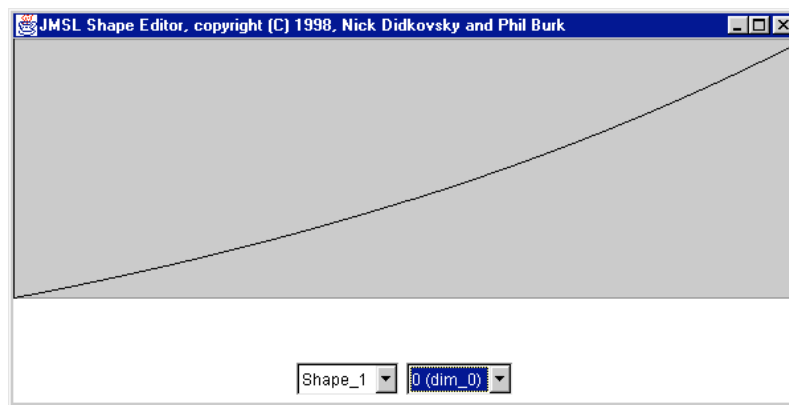
Getting a value from these distributions is easy, just make a call to these static methods:
EventDistributions.genEntryDelayLog (double meanEventDensity), or
EventDistributions.genEntryDelayMyhill (double meanEventDensity, double ratio)

See also <http://www.algomusic.com/jmsl/examples/DensityDemo.html>

See also <http://www.algomusic.com/jmsl/examples/EventDistributionsDemo.html>

Exponential Interpolator

Interpolate a value along an exponentially rising curve defined by two points (x_1, y_1) and (x_2, y_2) where $x_1 < x_2$, $y_1 < y_2$. The shape of $f(x) = \exp(x)$ on the interval $[0..1]$ is fitted to the user's endpoints. That is, (x_1, y_1) corresponds to $(0, 1)$ and (x_2, y_2) corresponds to $(1, e)$, and this new curve resembles a stretched or compressed version of the $\exp(x)$ function.



A MusicShape generated using JMSL's Exponential Interpolator

JMSL Installation

JMSL ships with a jar file called `jmsl.jar` and `jscore.jar`. Java projects need to add these to the classpath. **DO NOT PUT THESE IN YOUR JAVA EXTENSIONS FOLDER!**

Eclipse users select Project Properties -> Java Build Path -> Libraries -> Add External Jars, then browse for `jmsl.jar` and `jscore.jar`

JBUILDER users: add a new required libraries to your project, name them JMSL and JMSLScore, and browse to the location of `jmsl.jar` and `jscore.jar`

For more installation and testing notes, refer to the `README_FIRST.html` file that shipped with the JMSL distribution.

Deploying JMSL as an applet: Jar file and HTML

You must upload `jmsl.jar` your `public_html/classes` directory on your web server. **Additionally**, your applet tag needs to point to the `jmsl.jar` archive, as the following example tag demonstrates:

```
<applet code="mypackage.MyJMSLJSynApplet.class"
        codebase="classes"
        archive="jmsl.jar"
        WIDTH="400"
        HEIGHT="200">
```

You need a java enabled web browser
</applet>

IMPORTANT: Notice above that once the path to java classes is specified in the `codebase`, the `jmsl.jar` archive is automatically loaded from there. In other words you do **not** have to specify the path to `jmsl.jar` in the `archive` parameter!

JMSL related links:

<http://www.algomusic.com>
<http://midishare.sourceforge.net/>
<http://www.softsynth.com/javamidi/>
<http://www.softsynth.com/jsyn/>

HOMEWORK (deploy as applet):

Design a `MusicJob` that prints a message to `System.out` in its `repeat()` method. Instantiate a few of these, put them into `Parallel` and/or `Sequential` collections. You might add `RepeatPlayable()` to these collections to add new `MusicJobs` or remove them from the hierarchy in some interesting way. You might create a `Behavior` that chooses a child in an interesting way. You might set their repeats using random numbers or some other process. You might set their repeat pauses using some interesting random process or relate them to each other in some interesting way (for example, one phases against another over 10 seconds). While developing, make little incremental changes, test it frequently, and make sure you understand what it is doing before proceeding. Launch your creation at `JMSL.now()` when a Button is pressed. Post as an applet, and watch the Java console for output.

Think about form and process, think compositionally, make this an interesting silent piece of music.

For extra credit, have the `MusicJob` express its work in some way other than `System.out.println()`, like drawing to a canvas for example, or flashing button colors, or printing to a `TextArea`, or ...?