

# Java Music Systems, Didkovsky

## JMSL MusicShapes, Dimension Name Spaces, and Instruments

You have been introduced to JMSL's polymorphic hierarchical scheduler, and by now, you might have created a silent composition built with MusicJobs, ParallelCollections, and Sequential Collections, defining their functionality by overriding methods like repeat(), adding Playables to their start playables, repeat playables, and stop playables, setting and changing scheduling data, possibly during the performance itself, using methods like setRepeatPause() and setRepeats(), and possibly even altering the hierarchical structure itself in realtime, by adding or removing Composables at various hierarchical levels.

This week we will get closer to sonifying your work, by introducing a JMSL class called Instrument which sonifies numerical data in a user-defined way, and a Composable called MusicShape, which holds and schedules numerical data. We will also investigate Dimension Name Spaces, which are mappings from numbers like 0, 1, 2, 3 to meaningful names like "duration", "pitch", "amplitude", and "hold time", useful for mediating between data sources like MusicShape and data interpreters like Instrument. Finally we will provide a jump start on how to play your JSyn SynthNotes with JMSL Instruments, so you can hear your JSyn sounds in a polyphonic, scheduled context.

## 1.0 Instruments and Interpreters

**Instrument** is a JMSL interface which is responsible for the interpretation of numerical data. It may use Midi, JSyn, printing, or anything else that is programmable. It all happens in its play() method which receives a timestamp, a "timestretch" value, and an array of double precision floating point numbers. It returns an updated timestamp which is understood to be the duration of that musical event.

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

The **playTime** sent to play is the timestamp the playing should take place.

The **timeStretch** is a time scaler which play() may use to multiply by the duration.

The **array of double** contains the numerical data the Instrument must sonify.

For example:

```
public double play(double playTime, double timeStretch, double dar[]) {
    double duration = dar[0];
    double timeStretchedDuration = duration * timeStretch;
    double fractureIndex = dar[1];
    System.out.println("fracture index=" + fractureIndex);
    return playtime + timeStretchedDuration;
}
```

### 1.1 Instruments included in JMSL

JMSL includes a number of useful Instrument classes, which may already do much of what you need for your own music. So you may not have to define your own Instrument unless your needs are more specialized than the following:

- **MidiInstrument** - interprets data as MIDI note-on's.
- **JSynInsFromClassName** - Plays JSyn SynthNotes polyphonically. Constructor takes in the class name of a SynthNote and max number of voices. Limited to duration, pitch, amplitude, and hold time, ie no control of SynthInputs other than frequency and amplitude.

- **SynthNoteAllPortsInstrument** - Like JSynInsFromClassName, with additional control over custom SynthInputs. Sniffs out all public SynthInput ports of SynthNote and creates a "SynthNoteDimensionNameSpace", mapping dimension numbers to SynthInput port names. Every call to play() gives you complete control over each SynthInput.
- **SynthNoteAllPortsInstrumentSP** - This Instrument behaves just like SynthNoteAllPortsInstrument except that it implements JSynSignalProcessingInstrument which defines addSignalSource(SynthOutput signalSource). So the output from another SynthCircuit or from some JSynInstrument.getoutput() can be sent into it for SignalProcessing. This means you can schedule and change signal processing over time.
- **SimpleSamplePlayingInstrument** - A JMSL Instrument that plays SynthSamples, mapped to integer pitch values. "One shot" style. No transposition, no loop points, plays all the way through
- **TransposingSampleSustainingInstrument** - A JMSL Instrument that loads a sequence of monophonic SynthSamples. Each sample must have exactly two cue points: loop start and release start (where release start == loop end). If there are gaps in the chromatic scale, a sample will be stretched up or down from the closest sample. While no more than a minor third between your samples is recommended for "believable" transposition, this is not enforced. Also, pitches may be fractional.
- **TransposingSamplePlayingInstrumentWithAmplitudeMap** – demo from jmsl testsuite package, this subclass of SimpleSamplePlayingInstrument overrides getAlternativeSampleIndex() to map note 60 to three different samples based on amplitude.
- **MaxInstrument** - JMSL Instrument that can be used to send scheduled data to Max/MSP

## 1.2 Defining your own Instrument

A convenience class called **InstrumentAdapter** is included in JMSL, which implements the Instrument interface, and allows the programmer to simply override the play() method to do things that may not already be done by one of the stock JMSL Instruments listed above.

If none of the instruments listed above do what you need, you are encouraged to define your own Instrument. The easiest approach is to extend InstrumentAdapter and simply override its play() method, thereby doing the data interpretation directly within your InstrumentAdapter subclass. See below:

```
/** You can override this play() method with your own custom code */
public double play(double playTime, double timeStretch, double dar[]) {
    // do whatever you want here with the array of double in dar[]
    return playTime;      // return whatever interpretation of duration you want here
}
```

**IMPORTANT:** A typical return would be `return playTime + dar[0] * timeStretch` which follows a convention of putting duration value in `dar[0]`. If you don't add something to `playTime`, `play()` will be scheduled as though it took no time.

## 1.3 Interpreters

An Instrument may defer its treatment of play() data to an Interpreter which defines.

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

One reason you might use Interpreters is that they can be swapped out at runtime. But there has been little practical need for interpreters so we include them here only for completeness. Note that InstrumentAdapter is given a printing Interpreter by default, which is invoked in its play() method, whose source is shown below:

```
public double play(double playTime, double timeStretch, double dar[]) {
    if (interpreter != null)
```

```

        playTime = interpreter.interpret(playTime, timeStretch, dar, this);
    return playTime;
}

```

## 2.0 MusicShape

A JMSL MusicShape is an ordered, multidimensional list of numerical data. You can think of a MusicShape as having rows and columns. Its *elements* correspond to rows, each with a number of *dimensions*, or columns. For example, a 4-dimensional MusicShape would have 4 “columns”, and might use its dimensions to store duration (dim 0), pitch (dim 1), amplitude (dim 2), and Hold Time (dim 3). Each element of this shape would describe one note-on at a particular amplitude, pausing for a particular time, sustained for some amount of time.

Such a MusicShape might print itself out like so:

```

MusicShape_1
dim 0: Dur, dim 1: Pitch, dim 2: Vel, dim 3: Hold
Dur, Min=0.0, Max=1.0, Mean=0.38235294117647056, Limits={-Infinity, Infinity}
Pitch, Min=62.0, Max=68.0, Mean=65.29411764705883, Limits={-Infinity, Infinity}
Vel, Min=2.0, Max=117.0, Mean=59.11764705882353, Limits={-Infinity, Infinity}
Hold, Min=0.5, Max=2.5, Mean=1.5588235294117647, Limits={-Infinity, Infinity}
0: { 1.0 68.0 42.0 2.0 }
1: { 0.25 68.0 39.0 2.0 }
2: { 1.0 66.0 18.0 2.0 }
3: { 0.0 64.0 83.0 2.0 }
4: { 0.25 66.0 111.0 0.5 }
5: { 0.0 68.0 12.0 1.0 }
6: { 0.25 65.0 112.0 2.0 }
7: { 0.0 68.0 2.0 2.0 }
8: { 0.0 66.0 70.0 0.5 }
9: { 0.75 63.0 100.0 1.0 }
10: { 0.25 64.0 32.0 1.5 }

```

Q: How would you represent a chord, where two or more elements sound at the same time?

A: A triad would be stored as three elements, the first two with duration 0 and the third with the duration of the chord.

When a MusicShape is created, it gets an InstrumentAdapter by default, which interprets the 0<sup>th</sup> dimension as duration and prints the elements as they are performed. So if you launch a MusicShape without setting its Instrument yourself, you will see its elements printed to System.out as it is performed.

```
s.launch(JMSL.now())
```

```

Interpreter_1 called by ins_0 with { 1.0, 68.0, 42.0, 2.0 }
Interpreter_1 called by ins_0 with { 0.25, 68.0, 39.0, 2.0 }
Interpreter_1 called by ins_0 with { 1.0, 66.0, 18.0, 2.0 }
etc

```

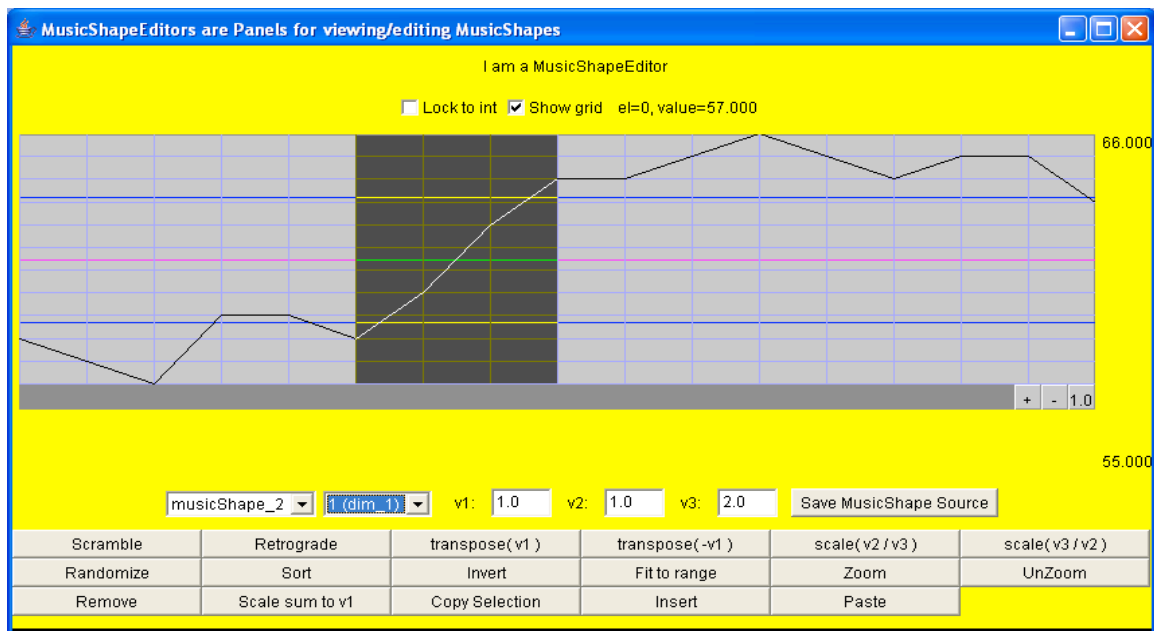
A MusicShape can represent any numerical ideas, not just note-like info. For example, it might store values that are used by a statistical process, frequencies and input port values for JSyn SynthCircuits. How the data is used depends on JMSL's Instruments.

MusicShapes are more than mere numeric containers, however. They are a subclass of MusicJob, so they are Composables. In other words, they “know” about time, and contain a custom Instrument that can perform MusicShape data on a schedule. The MusicShape above, for example, could contain a stock JMSL MidiInstrument. When this MusicShape is launch()’ed, it will iterate through its elements, each time handing one element of data to the MidiInstrument.

MusicShapes also have a number of useful mutating methods like `invert()`, `reverse()`, `scramble()`, and `sort()`, as well as `prefab()` which loads a MusicShape with some randomly generated data.

## 2.1 MusicShapeEditor

MusicShapeEditor is a Panel that contains any number of MusicShapes, displays their contents along any single dimension, and makes the data available for real-time mouse-controlled editing.



*JMSL's Shape Editor showing dimension 1 of musicShape\_2.*

Examine the screenshot above and you will notice a panel of buttons labeled with actions such as “Scramble”, “Retrograde”, etc. Clicking on one of these buttons will apply an operation to the selected subrange of MusicShape. You can add your own custom actions by implementing `MusicShapeEditorOperator`, then calling `myEditor.addMusicShapeEditorOperator(myOperator)`. Below is a code excerpt showing how `scramble` was implemented:

```
public void operate(MusicShapeEditor se, MusicShape s) {
    if (se.getRangeStartIndex() != -1 && se.getRangeEndIndex() != -1
        && Limits.within(se.getDimension(), 0, s.dimension() - 1)) {
        s.scramble(se.getRangeStartIndex(), se.getRangeEndIndex(),
            se.getDimension());
    }
}
```

## 3.0 Create a MusicShape, add and transform data

The number of elements in a MusicShape can change at runtime (ie you can add and remove elements during a performance). However, the number of dimensions is fixed from the moment it is declared in its constructor, and cannot change.

**Create:** Here we create a 4-dimensional MusicShape:

```
MusicShape s = new MusicShape(4);
```

**Add data:** To add elements to “s” above, you use MusicShape’s add() method. In this case, add() needs 4 parameters, one for each dimension declared above. Note that add() is overloaded to provide up to 12 parameters. For more than 12 parameters, create a double[] and pass this array to add()

```
s.add(1, 60, 0.5, 0.8);
s.add(1, 60.5, 0.6, 0.8);
s.add(1.5, 60.8, 0.7, 0.2);
```

You can see the data in a MusicShape, some of the statistics about it, and the name and limit definitions of its dimensions with:

```
s.print();

musicShape_1
dim 0: dim_0, dim 1: dim_1, dim 2: dim_2, dim 3: dim_3
dim_0, Min=1.0, Max=1.5, Mean=1.1666666666666667, Limits={-Infinity, Infinity}
dim_1, Min=60.0, Max=60.8, Mean=60.433333333333334, Limits={-Infinity, Infinity}
dim_2, Min=0.5, Max=0.7, Mean=0.6, Limits={-Infinity, Infinity}
dim_3, Min=0.2, Max=0.8, Mean=0.6, Limits={-Infinity, Infinity}
0: { 1.0, 60.0, 0.5, 0.8 }
1: { 1.0, 60.5, 0.6, 0.8 }
2: { 1.5, 60.8, 0.7, 0.2 }
```

**Transform data:** You can transform a MusicShape’s data with some useful methods like

```
public void scramble(int start, int end, int dim);
```

and

```
public void transpose(double val, int start, int end, int dim)
```

Consult MusicShape Javadocs to discover more.

You can also iterate through a MusicShape’s elements, and change their values however you like with set() and get(), as the following loop demonstrates. It adds a random number [0..1) to each value found in the MusicShape’s dimension 2:

```
for (int i = 0; i < s.size(); i++) {
    double dim2Value = s.get(i, 2); // element i, dimension 2
    s.set(dim2Value + JMSLRandom.choose(), i, 2);
}
```

**print():**

```
musicShape_1
dim 0: dim_0, dim 1: dim_1, dim 2: dim_2, dim 3: dim_3
dim_0, Min=1.0, Max=1.5, Mean=1.1666666666666667, Limits={-Infinity, Infinity}
dim_1, Min=60.0, Max=60.8, Mean=60.433333333333334, Limits={-Infinity, Infinity}
dim_2, Min=1.1800384521484375, Max=1.5184356689453125, Mean=1.3725830078124999,
Limits={-Infinity, Infinity}
dim_3, Min=0.2, Max=0.8, Mean=0.6, Limits={-Infinity, Infinity}
0: { 1.0, 60.0, 1.1800384521484375, 0.8 }
1: { 1.0, 60.5, 1.41927490234375, 0.8 }
2: { 1.5, 60.8, 1.5184356689453125, 0.2 }
```

### **3.1 Setting names, defaults, and limits to a MusicShape’s dimensions**

In the above print() output, you will see the dimensions are named “dim\_0”, “dim\_1”, “dim\_2”... and that their limits are from -infinity .. +infinity. You can set more meaningful names to your MusicShape’s dimensions, as well as meaningful min, max, and default values for each. To assign names, set limits, and assign default values to a MusicShape’s dimension, examine the code excerpt below, which sets these for dimensions 0..3.

```

    s.setDimensionName(0, "duration");
    s.setDefault(0, 1.0);
    s.setLimits(0, 0.0, 10.0);
    s.setDimensionName(1, "pitch");
    s.setDefault(1, 60.0);
    s.setLimits(1, 0.0, 120.0);
    s.setDimensionName(2, "amplitude");
    s.setDefault(2, 0.5);
    s.setLimits(2, 0.0, 1.0);
    s.setDimensionName(3, "hold");
    s.setDefault(3, 1.0);
    s.setLimits(3, 0.0, 20.0);

print():
musicShape_1
dim 0: duration, dim 1: pitch, dim 2: amplitude, dim 3: hold
duration, Min=1.0, Max=1.5, Mean=1.1666666666666667, Limits={0.0, 10.0}
pitch, Min=60.0, Max=60.8, Mean=60.433333333333334, Limits={0.0, 120.0}
amplitude, Min=1.1800384521484375, Max=1.5184356689453125, Mean=1.3725830078124999,
Limits={0.0, 1.0}
hold, Min=0.2, Max=0.8, Mean=0.6, Limits={0.0, 20.0}
0: { 1.0, 60.0, 1.1800384521484375, 0.8 }
1: { 1.0, 60.5, 1.41927490234375, 0.8 }
2: { 1.5, 60.8, 1.5184356689453125, 0.2 }

```

### 3.2 Using Dimension limits to generate data automatically

One of the niceties of specifying a MusicShape's dimension limits is that you can generate data algorithmically by iterating through its dimensions, and using the low/high limits of each to generate values that are in range. MusicShape's `prefab()` method does just this.

```

s.prefab(17); // add 17 self-fabricated elements to s

print():
musicShape_1
dim 0: duration, dim 1: pitch, dim 2: amplitude, dim 3: hold
duration, Min=0.0, Max=1.5, Mean=0.575, Limits={0.0, 10.0}
pitch, Min=60.0, Max=65.0, Mean=63.065, Limits={0.0, 120.0}
amplitude, Min=0.045989990234375, Max=1.5184356689453125, Mean=0.6409445190429688,
Limits={0.0, 1.0}
hold, Min=0.2, Max=2.5, Mean=1.115, Limits={0.0, 20.0}
0: { 1.0, 60.0, 1.1800384521484375, 0.8 }
1: { 1.0, 60.5, 1.41927490234375, 0.8 }
2: { 1.5, 60.8, 1.5184356689453125, 0.2 }
3: { 0.5, 65.0, 0.663055419921875, 1.0 }
4: { 0.0, 64.0, 0.779205322265625, 2.5 }
5: { 0.0, 62.0, 0.650238037109375, 0.5 }
6: { 1.0, 63.0, 0.135528564453125, 1.0 }
7: { 1.0, 65.0, 0.344451904296875, 0.5 }
8: { 0.0, 63.0, 0.636383056640625, 1.0 }
9: { 1.0, 65.0, 0.620697021484375, 0.5 }
10: { 0.25, 62.0, 0.156768798828125, 0.5 }
11: { 0.0, 63.0, 0.353973388671875, 2.0 }
12: { 0.5, 64.0, 0.571685791015625, 1.0 }
13: { 0.25, 63.0, 0.419281005859375, 1.5 }
14: { 0.75, 65.0, 0.756134033203125, 0.5 }
15: { 0.25, 64.0, 0.691619873046875, 1.5 }
16: { 0.75, 63.0, 0.585113525390625, 2.5 }
17: { 1.0, 62.0, 0.045989990234375, 1.5 }
18: { 0.5, 64.0, 0.933624267578125, 1.0 }
19: { 0.25, 63.0, 0.357391357421875, 1.5 }

```

The source code for `prefab()` is pasted below. Check out the powerful little loop at the end which generates data for dimensions higher than 3.

```

public void prefab(int numElements) {
    Limits noteRandomWalk = new Limits(40, 78, 3); // low note, high note,
    // range

```

```

for (int i = 0; i < numElements; i++) {
    noteRandomWalk.randomStep();
    int pitch = noteRandomWalk.getIntValue();
    double dur = JMSLRandom.choose(5) * 0.25;
    double hold = JMSLRandom.choose(5) * 0.5 + 0.5;
    double amp = JMSLRandom.choose(getLowLimit(2), getHighLimit(2));
    double[] data = new double[this.dimension()];
    data[0] = dur;
    if (data.length > 1)
        data[1] = pitch;
    if (data.length > 2)
        data[2] = amp;
    if (data.length > 3)
        data[3] = hold;
    for (int dim = 4; dim < dimension(); dim++) {
        double value = JMSLRandom.choose(getLowLimit(dim), getHighLimit(dim));
        data[dim] = value;
    }
    add(data);
}
}

```

## 4.0 DimensionNameSpace

DimensionNameSpace is an interface that manages the mapping from integers like 0, 1, 2... to names like “duration”, “pitch”, “amplitude”, as well these notions of defaults and limits. MusicShape implements DimensionNameSpace so you’ve already been introduced to this idea in the discussions above.

Let’s look more closely at JMSL’s **DimensionNameSpace** interface. A six dimensional DimensionNameSpace might assign names as follows.

dimension	name
0	"duration"
1	"pitch"
2	"amplitude"
3	"hold"
4	"modamp"
5	"cutoff"

NOTE: The first four names above (duration, pitch, amplitude, and hold) are considered standards by a number of JMSL tools. They are frequently identified as "invariants".

DimensionNameSpace also includes the notion of **upper limit**, **lower limit**, and **default value** per dimension.

For example:

dimension	name	low limit	high limit	default
2	"amplitude"	0.0	1.0	0.25

## 4.1 Getting and setting array values using names instead of indexes

One of the advantages DimensionNameSpace lends to the JMSL API is the ability to set and get values in an array by name instead of by array position. You can create a MusicShape compatible with the Instrument described above as follows:

```
MusicShape s = new MusicShape(ins.getDimensionNameSpace());
```

You could generate and add data without knowing the dimension indexes, as follows:

```
for (int i = 0; i < 8; i++) {
    double[] data = new double[s.dimension()];
    data[s.getDimension("duration")] = JMSLRandom.choose(); //0..1
    data[s.getDimension("modamp")] = 0;
    data[s.getDimension("cutoff")] = JMSLRandom.choose(100, 2048);
    s.add(data);
}
```

The instrument's play() method could extract data from the double[] parameter by name as well, as is shown here:

```
public double play(double playTime, double timeStretch, double[] dar) {
    // retrieve data values without knowing the dimension numbers

    DimensionNameSpace dns = getDimensionNameSpace();
    double duration = dar[dns.getDimension("duration")];
    int modAmp = (int) dar[dns.getDimension("modamp")];
    int cutoff = (int) dar[dns.getDimension("cutoff")];
    ...
}
```

## 4.2 Translating between DimensionNameSpaces

Recall that MusicShape uses arrays of double[] to store its elements, and implements DimensionNameSpace. Instrument.play() receives an array of double and may also have a DimensionNameSpace class variable. JMSL includes a **DimensionNameSpaceTranslator** class which translates an array of double[] from one DimensionNameSpace to another. It is assumed that the length of the array equals the number of dimensions, and that double[i] contains a value that corresponds to dimension i. One DimensionNameSpace may share names with another in some dimensions but not all (for example "rate" in one name space might be found in dimension 6 while it is found in dimension 5 of another space). The idea is to preserve the meaning of "rate" when an array of data is translated from one DimensionNameSpace to another. Let's examine two DimensionNameSpaces and see what translation would do.

1) DimensionNameSpace built by SynthNoteAllPortsInstrument for com.softsynth.jsyn.circuits.FilteredSawtoothBL :

0	duration
1	pitch
2	amplitude
3	hold
4	cutoff

5	resonance
<b>6</b>	<b>Rate</b>

2) DimensionNameSpace built by SynthNoteAllPortsInstrument for com.punosmusic.circuitgrabbag.NewPlucker :

0	duration
1	pitch
2	amplitude
3	hold
4	feedback
<b>5</b>	<b>rate</b>

*Translating between these two:*

Note that besides the four common names "duration", "pitch", "amplitude", and "hold", that the special name "rate" is found in both DimensionNameSpaces (name comparison is case **insensitive** so **Rate** == **rate**). DimensionNameSpaceTranslator will preserve these common names when translating a double[] from one name space to the other.

For example, if the following double[] were described in the first DimensionNameSpace:

```
{ 1, 60, 0.5, 0.5, 1000, 0.60, 0.1 }
// before translation, 0.1 "rate" is in dim 6
```

...and if it were translated to the latter DimensionNameSpace, the translation will preserve the first four dimensions, as well as the rate = 0.1 value. It will substitute the DimensionNameSpace's default value for "feedback", producing the array:

```
{1, 60, 0.5, 0.5, <defaultfeedbackvalue>, 0.1 }
// after translation, 0.1 "rate" is in dim 5, "feedback" is not common
so default is used
```

### ***4.3 Why is translating between DimensionNameSpaces useful?***

Translating arrays of double[] between DimensionNameSpaces is a very fluid and flexible way of dealing with musical data. It permits data to be described and interpreted without conforming to a strict specification, and permits the semantics of this data to be preserved wherever it can as it flows between interpreters.

For example, consider the action of copying and pasting musical notes from one staff to another. The notes in the source staff contain synthesis parameters appropriate for the instrument assigned to that staff. The destination staff may be assigned to a different instrument which may have all, some, or none of its synthesis parameters in common with the first instrument. By translating each note's data when pasting into the destination staff, we preserve as many semantics as possible, additionally ensuring that the order of the data in the array is in the order required by the destination instrument.

## 4.4 Invariants

Some dimensions may be so important that you don't want to risk mistranslations. It would be very unhelpful to penalize the choice of "dur" versus "duration" as a dimension name for dimension 0, and insist on matching names. To address this, `DimensionNameSpaceTranslator` supports the idea of a dimensional "invariant". If dimensions 0, 1, 2, and 3 are set as invariants, the `DimensionNameSpaceTranslator` will simply copy them in place without examining and matching their dimension names. For example, `myDimNameSpaceTranslator.addInvariant(0)` specifies that dimension 0 should be copied from position 0 of one array to position 0 of the other, regardless of naming. To give a working example, in JMSL's `Score`, when a user copies and pastes notes between staves with different instruments, its `DimensionNameSpaceTranslator` maintains dimensions 0..3 as invariants since duration, pitch, amplitude, and hold are required to be in dimensions 0..3 respectively.

**4.5 MusicShape uses DimensionNameSpaceTranslator** to translate its data from its own `DimensionNameSpace` to the `DimensionNameSpace` of its `Instrument`. Just before it calls `instrument.play()`, it queries the instrument for its `DimensionNameSpace`. If the `Instrument`'s `DimensionNameSpace` is not null, `MusicShape` first translates its element and passes the translated `double[]` to `instrument.play()` so the values conform to the instrument's `DimensionNameSpace`. To give a concrete example, consider building a `MusicShape` whose `Instrument`'s `DimensionNameSpace` is defined by the input ports on a `JSyn SynthNote` (see `com.softsynth.jmsl.jsyn.SynthNoteAllPortsInstrument`). You may know that "modfreq" is one of the input ports on the `SynthNote`, but you don't care whether this name is mapped to dimension 4, 5, 6 .... Since `MusicShape` will translate its data to conform to the instrument's `DimensionNameSpace` you can simply do the following...

```
myMusicShape.setDimensionName(4, "modfreq");
```

...and add data to `myMusicShape`, assured that dimension 4 will be interpreted as "modfreq". Even if the `Instrument`'s `DimensionNameSpace` assigns "modfreq" to dimension 5, the translator will relocate it from the `MusicShape`'s dimension 4 to the `Instrument`'s dimension 5, just in time to `play()` it.

## 5.0 Summary idea and the Player class

*An Instrument may have its own DimensionNameSpace. MusicShape implements DimensionNameSpace. Right before a MusicShape hands an element to its Instrument's play() method, the Instrument uses DimensionNameSpaceTranslator to compare its own DimensionNameSpace with the Instrument's DimensionNameSpace and translates the order of the parameters by name, to ensure maximum compatibility. The MusicShape receives the updates playtime from its Instrument's play() method, and pauses before proceeding to the next element.*

Before we leave the subject of `MusicShapes` and `Instruments`, we mention here JMSL's `Player` class which is a `Composable` that contains an `Instrument` and a number of `MusicShapes`. It uses

the same Instrument to perform the data contained in all MusicShapes. For example, a Player might have a MIDI instrument, and two MusicShapes: the first MusicShape contains MIDI information for a verse, the second MusicShape contains the chorus. Players are a subclass of SequentialCollection and provide you with a convenience.

## 6.0 JSyn & JMSL Jump Start

Here we show you (with minimal explanation) how you can get started immediately by wrapping your JSyn SynthNotes up as a JMSL Instrument, and use it in a MusicShape. The following source creates a polyphonic JMSL/JSyn Instrument simply by being handed the number of voices and the class name. It uses a SynthNote that ships with JSyn, but you can substitute your own class names here. The user can edit the MusicShape data in a MusicShapeEditor panel. Also unique to this example, every time the MusicShape repeats, some subrange of the duration and pitch dimensions are scrambled.

This example uses JSynInsFromClassName which only plays duration, pitch, amplitude, and hold time. For more control over your SynthNote's input ports, use SynthNoteAllPortsInstrument (but this will be covered later)

The following applet source will perform a new algorithmically generated melody every time you click the start button.

```
/*
 * Created on Feb 18, 2004
 *
 */
package com.didkovsky.javamusic;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import com.softsynth.jmsl.*;
import com.softsynth.jmsl.jsyn.JSynInsFromClassName;
import com.softsynth.jmsl.jsyn.JSynMusicDevice;
import com.softsynth.jmsl.util.EventDistributions;
import com.softsynth.jmsl.util.Oof;
import com.softsynth.jmsl.view.MusicShapeEditor;

/**
 * JMSLJSynApplet.java
 *
 * Hit a button that generates and plays a MusicShape Use a JSyn SynthNote
 * wrapped in a polyphonic JMSL Instrument Every repeat, the MusicShape
 * scrambles a random subrange of data
 *
 * @author Nick Didkovsky, didkovn@mail.rockefeller.edu (C) 2003 Nick Didkovsky
 *
 * JSyn (C) Phil Burk, visit www.softsynth.com JMSL (c) Nick Didkovsky and Phil
 * Burk, visit www.algomusic.com
 *
 */
public class JMSLJSynApplet extends java.applet.Applet implements ActionListener {

    MusicShape myMusicShape;
    JMSLMixerContainer mixer;
    MusicShapeEditor musicShapeEditor;
    Button startButton;
    Button stopButton;
    JSynInsFromClassName ins;

    public void init() {
        JMSLRandom.randomize();
        JMSL.setIsApplet(true);
    }

    public void start() {
        synchronized (JMSL.class) {
            initJMSL();
            initMusicDevices();
        }
    }
}
```

```

        initMixer();
        initInstrument();
        initMusicShape();
        buildGUI();
    }
}

public void stop() {
    synchronized (JMSL.class) {
        removeAll();
        myMusicShape.finishAll();
        try {
            myMusicShape.waitForDone();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        JMSL.scheduler.stop();
        mixer.stop();
        JMSL.closeMusicDevices();
    }
}

private void initJMSL() {
    JMSL.scheduler = new EventScheduler();
    JMSL.scheduler.start();
    JMSL.clock.setAdvance(0.1);
}

void initMusicDevices() {
    JSynMusicDevice dev = JSynMusicDevice.instance();
    dev.open();
}

void initMixer() {
    mixer = new JMSLMixerContainer();
    mixer.start();
}

void initInstrument() {
    ins = new JSynInsFromClassName(8,
com.softsynth.jsyn.circuits.FilteredSawtoothBL.class.getName());
    mixer.addInstrument(ins);
}

void initMusicShape() {
    myMusicShape = new MusicShape(4);
    myMusicShape.useStandardDimensionNameSpace();
    myMusicShape.setInstrument(ins);
    myMusicShape.setRepeats(1000);
}

private void buildGUI() {
    setLayout(new BorderLayout());

    musicShapeEditor = new MusicShapeEditor();
    musicShapeEditor.addMusicShape(myMusicShape);
    add(BorderLayout.NORTH, musicShapeEditor.getComponent());

    add(BorderLayout.CENTER, mixer.getPanAmpControlPanel());

    Panel buttonPanel = new Panel();
    buttonPanel.setLayout(new FlowLayout());
    buttonPanel.add(startButton = new Button("Click to start"));
    buttonPanel.add(stopButton = new Button("Click to stop"));
    add(BorderLayout.SOUTH, buttonPanel);

    startButton.addActionListener(this);
    stopButton.addActionListener(this);
    stopButton.setEnabled(false);

    getParent().validate();
}

```

```

        getToolkit().sync();
    }

/**
 * Stuff a MusicShape with 16 algorithmically generated musical events.
 * Duration is Myhill distributed. Pitch is 1/F Amplitude is uniformly
 * random 0 .. 0.25 Hold time is in randomly chosen range 0.5 .. 2.0 times
 * duration
 *
 */
void buildMusicShape() {
    myMusicShape.removeAll();
    Oof oof = new Oof(7);
    oof.randomize();
    for (int i = 0; i < 16; i++) {
        double dur = EventDistributions.genEntryDelayMyhill(4.0, 20.0);
        double pitch = 60 + (oof.next() / 2);
        double amp = JMSLRandom.choose(0.25);
        double hold = dur * JMSLRandom.choose(0.5, 2.0);
        myMusicShape.add(dur, pitch, amp, hold);
    }
    myMusicShape.print();
    musicShapeEditor.refresh();
    // add a Playable that scrambles some data every time this MusicShape
    // repeats
    myMusicShape.addRepeatPlayable(new MusicShapeScrambler(musicShapeEditor));
}

public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if (source == startButton) {
        buildMusicShape();
        myMusicShape.launch(JMSL.now());
        startButton.setEnabled(false);
        stopButton.setEnabled(true);
    }
    if (source == stopButton) {
        myMusicShape.finishAll();
        startButton.setEnabled(true);
        stopButton.setEnabled(false);
    }
}

}

/**
 * This class implements Playable and can be added to a MusicShape's
 * RepeatPlayables. It scrambles a randomly chosen subset of data every time the
 * MusicShape repeats
 *
 * @author Nick Didkovsky, email: didkovn@mail.rockefeller.edu, (c) 2004 Nick
 *         Didkovsky, all rights reserved.
 *
 */

class MusicShapeScrambler implements Playable {

    MusicShapeEditor editor;

/**
 * Pass in the MusicShapeEditor just so we can call refresh() on it after
 * scarmalbing. Pass in null if you don't care
 *
 */

    MusicShapeScrambler(MusicShapeEditor editor) {
        this.editor = editor;
    }

    public double play(double playTime, Composable parent) {
        MusicShape s = (MusicShape) parent;

```

```

        int randomIndex1 = JMSLRandom.choose(s.size());
        int randomIndex2 = JMSLRandom.choose(s.size());
        // sort them so start less than end
        int startIndex = Math.min(randomIndex1, randomIndex2);
        int endIndex = Math.max(randomIndex1, randomIndex2);
        System.out.println("Scrambling from " + startIndex + " to " + endIndex);
        // scramble some subrange of durations
        s.scramble(startIndex, endIndex, 0);
        // scramble some subrange of pitches
        s.scramble(startIndex, endIndex, 1);
        if (editor != null) {
            editor.refresh();
        }
        return playTime;
    }
}

```

## 7.0 Deploying JMSL as an applet: Jar file and HTML

You must upload `jmsl.jar` your `public_html/classes` directory on your web server. Then follow the normal procedure for deploying JSyn applets (using `smart_embed_jsyn.js`, etc ... see <http://www.softsynth.com/jsyn/docs/usersguide.html#WebPage>). **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!

## 8.0 JMSL related links:

<http://www.algomusic.com>

<http://midishare.sourceforge.net/>

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

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

## HOMEWORK Extra Credit (but do it anyway):

Design a SynthNote in Wire. Make sure you have "frequency" and "amplitude" input ports defined! Wrap it up as a JMSL instrument following the model above.

Define a MusicShape that performs with data that is algorithmically generated. You may follow the spec below or **come up with your own way of generating duration, pitch, amplitude, and hold** data. If you choose the latter, make sure your web page includes an explanation of the algorithms used.

For example:

Dimension 0: duration generated by `genEntryDelayMyhill` (double `meanEventDensity`, double `ratio`), where the `meanEventDensity` and `ratio` are entered by the user into TextFields

Dimension 1: pitch index generated by a Oof generator.

Dimension 2: amplitude uniformly random 0..1

Dimension 3: hold time, calculated to be 2 times the duration.

Note that the algorithm to get a double from a String (from TextField) is as follows:

```

double value = (new Double(someString)).doubleValue();

```